Static Checking of Regular Language Types via Abstract Interpretation

FENGMIN ZHU, CISPA Helmholtz Center for Information Security, Germany
MICHAEL SCHRÖDER, TU Wien, Austria
ANDREAS ZELLER, CISPA Helmholtz Center for Information Security, Germany

Type systems in programming languages are among the most effective techniques to discover bugs well before production. However, there is one important subdomain of data types in which errors are hardly caught: strings. Indeed, if a string can have unexpected or invalid contents, type checkers can hardly detect this, leading to potential bugs such as runtime substring-not-found errors or vulnerabilities such as SQL injection. In principle, the concept of *refinement types* can help to express and check string properties. However, in existing refinement type systems, strings are *understudied* and *undersupported.*

In this paper, we present the first *static* type checker for *regular language types.* Our FLAT-Checker prototype targets string-manipulating programs in Python. It allows, for every string variable, argument, or return value to specify its *language* (= the set of all possible strings) as a regular expression. FLAT-Checker then *statically* checks these types, ensuring, for instance, that a well-typed program is free of substring-not-found errors. At the heart of FLAT-Checker are rules that infer the result types for common string operations via abstract interpretation, together with type narrowing for higher precision of type inference.

On a dataset consisting of 204 Python ad hoc parsers, with a reasonable amount of user annotations, FLAT-Checker successfully type-checked all of them. It also outperformed four SMT solvers, including two state-of-the-art string solvers, in terms of solvability and efficiency.

CCS Concepts: • **Theory of computation → Regular languages**; *Constraint and logic programming*; • **Software and its engineering → Formal software verification**; **Semantics**.

Additional Key Words and Phrases: Refinement types, type inference, regular expressions, abstract interpretation, SMT solving, string constraints, Rocq

ACM Reference Format:
Fengmin Zhu, Michael Schröder, and Andreas Zeller. 2025. Static Checking of Regular Language Types via Abstract Interpretation. *Proc. ACM Program. Lang.* 1, 1, Article 1 (January 2025), 26 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Type systems in programming languages are among the most effective and efficient techniques to discover bugs well before production. While typical type checkers would quickly identify type mismatches between types such as Booleans, numbers, or data types, modern refinement type systems [Freeman and Pfenning 1991] allow programmers to encode additional semantic constraints in types, making type checking play the role of a lightweight program verifier. Notably, the concept of *Liquid Types* [Rondon et al. 2008] in Haskell [Jhala 2014], TypeScript [Vekris et al. 2016], Java [Gamboa et al. 2023], Rust [Lehmann et al. 2023], and more, shines in reasoning about programs manipulating integers, bit vectors, arrays, algebraic data types (ADTs), and even Rust traits.

Authors' Contact Information: Fengmin Zhu, fengmin.zhu@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Saarland, Germany; Michael Schröder, michael.schroeder@tuwien.ac.at, TU Wien, Vienna, Austria; Andreas Zeller, zeller@cispa.de, CISPA Helmholtz Center for Information Security, Saarbrücken, Saarland, Germany.

However, there is one important subset of data types where errors are hardly caught: *strings.* While strings are ubiquitous in modern software, errors related to malformed or invalid string contents are hard to catch, especially before production. As an example, consider a Python function `extract_minor` that takes a semantic version number string, say `'1.2.3'`, to extract the minor version number (`'3'`):

```
1 def extract_minor(s: str) -> str:
2   i1 = s.index('.') + 1
3   i2 = s.index('.', i1)
4   return s[i1:i2]
```

The annotated type signature, "`str` to `str`", is rather *coarse*: if we feed a string that is *not* a semantic version number, say `'123'`, then the string operation `s.index('.')` in Line 2 will trigger an exception "`ValueError: substring not found`" because the separator `'.'` is not found in the input `'123'`. In a bigger context, such string errors can lead to denial of service attacks, but also to vulnerabilities such as SQL injection, and (in more low-level languages) to buffer overflows and other serious issues.

We pose that the fundamental problem underlying these issues is that traditional type systems, which use a unified string type for all kinds of string values, *fail to capture the latent structure of strings.* This is a twofold problem—we need to be able to *specify* the content of strings, and we need to *check* the correctness of a program with respect to these specifications.

The first problem, specifying string contents, has long been addressed by the field of formal languages. Most programmers are familiar with the concept of *regular expressions,* formally denoting sets of possible string values. The recently published FLAT framework [Zhu and Zeller 2025] (*"Formal Languages As Types"*) provides developers with *language types*—string types refined by a formal language, say a *regular expression* (regex) or a context-free grammar. The inhabitants of a language type are strings in that formal language. Using FLAT, we can provide a refined type signature for `extract_minor`, where both the input and output types are refined by regexes:

```
def extract_minor(s: lang(r'[0-9]+\.[0-9]+\.[0-9]+')) -> lang(r'[0-9]+')
```

FLAT can type-check such language types, but only *dynamically*, namely, a type error such as `extract_minor('123')` is only detected *at runtime.* The problem of *static* checking remains.

In this paper, we present the *first* (to the best of our knowledge) *static* type checker for *regular* language types in Python. Our type checker can catch runtime errors like `extract_minor('123')` *at compile type* (it simply reports a type error "input type mismatch"). This indicates that it can also prove, for example, that a string function is free of SQL injection by specifying the input type to only allow safe SQL statements. Meanwhile, if the program type-checks, we can conclude that it is correct *w.r.t.* the specified types.

To focus on *string-manipulating* programs, and to lower the amount of engineering work, we only consider a tiny fragment of Python: without object-oriented and dynamic features, but with basic control flows and some commonly used string operations (see Table 1 for a full list). We offer annotations for programmers to define regular language types (like the `lang` annotation shown in the `extract_minor` example) and to specify loop invariants. Like in many Liquid Type systems and program verifies, through a process of the input source along with the annotations, we generate *verification conditions* (VCs).

To discharge VCs, one idea is to simply encode them as SMT formulae and resort to *SMT solvers,* as Liquid Typing does. SMT solving is efficient and mature for linear integer arithmetic, but *not for strings.* We noticed that even on a problem that seems simple and stupid to humans (see §2.1 for an example later in this paper), mature SMT solvers such as Z3 [de Moura and Bjørner 2008] and cvc5 [Barbosa et al. 2022] *timed out.* As string solving is an emerging field, new solvers such as

Z3-Noodler-pos [Chen et al. 2024] and OSTRICH [Chen et al. 2019; Hague et al. 2025] have been built, and their benchmarking shows better results. However, they still time out and even reported *wrong answers* (see §7 for the details) on some of the VCs generated from the dataset [Schröder and Cito 2025a] we use for evaluation. In short, we see SMT string solving as not ready for production.

As SMT solving has both strong (for integers) and weak points (for strings), we apply a *hybrid* approach. For VCs that only need basic integer arithmetic, we resort to SMT solving. For others that need reasoning about string operations, we take over ourselves. We rely on *type inference* to infer the result types for expressions applying string operations as precise as we can. We borrow ideas from abstract interpretation through the observation that *abstract domains are types:* by defining the abstract versions that overapproximate the concrete string operations, the output of the abstract operations give rise to the inferred types. We also introduce *type narrowing* to further improve the precision of type inference, taking control-flow sensitive information into account. We turn the inferred types into logical formulae as *lemmas*, which are derivable from the premises in the VC. If the VC is indeed valid, these lemmas are likely to be helpful for SMT solvers to eventually prove the goal by adding them as new premises. We implement these ideas as a prototype tool FLAT-Checker. We mechanize in the Rocq interactive theorem prover the formal definitions of strings, string operations, regular expressions, and the soundness theorems of our type inference and narrowing rules.

We evaluated FLAT-Checker on a dataset consisting of 204 Python *ad hoc parsers* [Schröder and Cito 2025a]. These parsers use string operations extensively; some involve tricky string manipulation with loops; all are restricted in the Python language features they use. Given a reasonable amount of type annotations, FLAT-Checker successfully type-checked all of them. On the generated VCs, we compared with four SMT solvers that support string theory (two mature and two new solvers targeting strings) and found that FLAT-Checker outperformed all of them.

Contributions. Our main contribution is FLAT-Checker, the first static type checker for regular language types. After starting with an overview of FLAT-Checker using Python examples (§2), we provide necessary preliminaries in §3. We then make the following additional contributions:

- Type inference (§4) and type narrowing (§5), the underlying methods for type-checking string operations intertwined with regular language types;
- A mechanization (§6) in the Rocq interactive theorem prover of the *soundness* of our type inference and narrowing rules;
- An evaluation (§7) of FLAT-Checker on 204 Python ad hoc parsers, including a comparison with four SMT solvers.

After discussing related work (§8), §9 closes the paper with conclusion and future work.

2 A Tour of FLAT-Checker by Examples

We present an overview of the workflow and key ideas behind FLAT-Checker using four example Python programs, including two examples (§2.3, §2.4) taken from the dataset we use for evaluation. We showcase how users attach type annotations and loop invariants (if needed) directly in Python code, and how FLAT-Checker type-checks them.

2.1 The Appetizer

We start with a simple Python function that aims to prove a trivial property:

```python
def a_star(s: lang(r'a*')):
  i = 0
  while i < len(s):
```

```
4       assert s[i] == 'a'
5       i += 1
```

Above all, at Line 1, we specify the type of the input string `s`: a *regular language type* of the regex `r'a*'` (repeating `'a'` zero or many times). FLAT-Checker offers a special type constructor `lang` to define regular language types. This constructor takes a regex literal as the argument: it follows the same syntax of Python's regexes defined in the `re` module. FLAT-Checker supports a subset of Python's regexes; see §3.2 for a full set. In the function body, we iterate all characters of `s` in a loop at Lines 3–5, and assert that any character of `s` is `'a'` at Line 4.

Although `a_star` involves a loop, we do not have to specify loop invariants manually, as what we need here, `0 <= i <= len(s)`, is rather naive and FLAT-Checker can figure it out automatically. For nontrivial loop invariants, FLAT-Checker offers special annotations for users to specify them, as we will see later in §2.3 and §2.4. We focus on *partial* correctness (*i.e.,* correctness assuming termination) in the entire framework.

Workflow. So far we have completed necessary annotations; it is time for FLAT-Checker to type-check the program. We summarize the workflow as follows:

(1) A *frontend* reads the Python code along with special annotations, performs "basic" type checking (by erasing regular language types as plain string types), and transpiles it into a program of our *core language*, which is essentially the *IMP language* [Pierce et al. 2025] extended with string operations (see §3.1 for the list).
(2) On the core program, FLAT-Checker applies standard *weakest-pre* transformation to extract *verification conditions* (VCs), *i.e.,* the obligations that ensure type correctness.
(3) FLAT-Checker discharges VCs via a combination of *type inference* and SMT solving; if any VC is unproved, type errors will be reported accordingly.

Extracted VCs. For the `a_star` function, FLAT-Checker extracts four VCs:

$$\varphi_1 : s \in \texttt{r'a*'} \Rightarrow 0 \le 0 \le |s|$$
$$\varphi_2 : s \in \texttt{r'a*'} \land 0 \le i \le |s| \land i < |s| \Rightarrow 0 \le i < |s|$$
$$\varphi_3 : s \in \texttt{r'a*'} \land 0 \le i \le |s| \land i < |s| \Rightarrow s[i] = \texttt{'a'}$$
$$\varphi_4 : s \in \texttt{r'a*'} \land 0 \le i \le |s| \land i < |s| \Rightarrow 0 \le i + 1 \le |s|$$

In all, the type requirement `s: lang(r'a*')` is represented as the regex membership test $s \in \texttt{r'a*'}$. Two VCs, φ_1 and φ_4, come from the (guessed) loop invariant $P_I \doteq 0 \le i \le |s|$: φ_1 checks that P_I holds before the execution of the loop, *i.e.,* before Line 3; φ_4 checks that P_I holds after one iteration, during which the loop variable i has been increased by 1. Another VC, φ_2, is a side condition that comes from the char-at operation `s[i]` occurred in Line 4: it checks that the string index i is in range $0 \le i < |s|$. Finally, the last VC, φ_4, comes from the user assertion in Line 4.

When SMT solving succeeds. Expect φ_3, the other three VCs only require basic integer arithmetic, *without* the need to know $s \in \texttt{r'a*'}$. For VCs like them, FLAT-Checker gives SMT solvers a try. It elides all regex membership tests such as $s \in a$, encodes VCs as corresponding SMT formulae, and invokes the backend solver cvc5 [Barbosa et al. 2022] to check their validity. cvc5 proves φ_1, φ_2, and φ_4.

When SMT solving fails. The VC φ_3 involves string operations, but it describes a simple and stupid property for strings in `r'a*'`: any of its character must be `'a'`. We tried mature SMT solvers including cvc5 and Z3: unfortunately, both *timed out* on this query after a minute.

Type inference and lemmas. To discharge this, FLAT-Checker relies on *type inference* to generate useful *lemmas* about the string operations used in the VC, hoping these lemmas either trivially imply the goal or guide SMT solving to prove the goal. Here, given that s has type `r'a*'`, $s[i]$ is inferred to have type `r'a'`, from which a lemma $s[i]$ = `'a'` is generated, which is trivially the goal we wish to prove in φ_3. So far, all VCs are discharged and FLAT-Checker concludes the program is well-typed.

Under the hood, we formulate the type inference problem as an abstract interpretation problem: given a string s and some string operation f in the concrete domain, find an abstract operation $f^\#$ that overapproximates the concrete semantics of f, *i.e.,* $s \in r \Rightarrow f(s) \in f^\#(r)$. To infer the type of $s[i]$, where i can be any valid index, *i.e.,* $0 \leq i < |s|$. Concretely, $s[i]$ could be any character in s. Abstractly, the set of all characters occurred in r, called the *alphabet* of r, denoted by $\alpha(r)$, overapproximates $s[i]$. Thus, $s[i]$ has type $\alpha($`r'a*'`$)$ = `r'a'`. If the type of s were `r'(a|b)*'`, then $s[i]$ would have type `r'(a|b)'`, from which φ_3 would become unprovable so FLAT-Checker would report a type error.

Key takeaways. From this short appetizer example, we see that FLAT-Checker:

- offers a regular language type constructor `lang` that supports Python's regex syntax;
- guesses naive loop invariants;
- discharges VCs via a combination of type inference based on abstract string operations and SMT solving.

2.2 Substring Extraction in Version Numbers

In the second example, we study the `extract_minor` function introduced in §1:

```
1 def extract_minor(s: lang(r'[0-9]+\.[0-9]+\.[0-9]+')) -> lang(r'[0-9]+'):
2   i1 = s.index('.') + 1
3   i2 = s.index('.', i1)
4   return s[i1:i2]
```

Recall that we aim to extract the minor version from a three-part semantic version number, where the major, minor, and patch parts are separated by a dot (`'.'`). The minor version is the substring in between the first and second `'.'` occurred in `s`. To locate these two indices, we use `s.index(t, i)`: it evaluates to the index of the first occurrence of the pattern `t` in `s` starting from index `i` (it will raise a runtime error if `t` is not found). At Line 2, we compute `i1` as the position *to the right of* the first `'.'` in `s`. At Line 3, we compute `i2` as the position *of* the second `'.'` in `s`, which is the first `'.'` starting from `i1`. At Line 4, the substring operation `s[i1:i2]` extracts the needed minor version, where the starting index `i1` is inclusive and the ending index `i2` is exclusive.

FLAT-Checker type-checks `extract_minor` using the same workflow summarized in §2.1. During transpilation, `s.index('.', i1)` is desugared to `s[i1:].index('.')`, because the corresponding find operation $s.\mathit{find}(t)$ defined in our core language does not take the optional argument `i1` as the starting index. For simplicity, we will only show VCs of great importance and interest. Here the most important one checks that the return value has the annotated return type:

$$s \in \texttt{r'[0-9]+\textbackslash.[0-9]+\textbackslash.[0-9]+'} \Rightarrow s[i_1 : i_2] \in \texttt{r'[0-9]+'}$$
$$\text{where } i_1 \doteq s.\mathit{find}(\texttt{'.'}) + 1, i_2 \doteq s[i_1 :].\mathit{find}(\texttt{'.'})$$

This is indeed a type *checking* task. But FLAT-Checker does not have specific type checking rules. Instead, it relies on the type inference to obtain an inferred type r_i first and then checks that r_i is a *subtype* of the expected type r_e, *i.e.,* $r_i \subseteq r_e$. The type inference problem here is complex as the expression $s' \doteq s[i_1 : i_2]$ has a complex form (once i_1 and i_2 are unfolded). It *decomposes* this complex task into smaller and simpler ones via *simplification*, namely s' is simplified into $s_2[: s_2.\mathit{find}(\texttt{'.'})]$ where $s_2 \doteq s_1[1 :]$ and $s_1 \doteq s[s.\mathit{find}(\texttt{'.'}) :]$.

Infer the type of s_1. Concretely, the pattern `'.'` splits s into two parts where $s_1 = s[s.\mathit{find}(\texttt{'.'}) :]$ is the suffix starting from the first `'.'`. Abstractly, this pattern splits $r \doteq$ `r'[0-9]+\.[0-9]+\.[0-9]+'` into two parts where $r_1 \doteq$ `r'\.[0-9]+\.[0-9]+'`, the suffix starting from the first `'.'` in r, overapproximates s_1. Thus, s_1 has type r_1.

Infer the type of s_2. Concretely, $s_2 = s_1[1 :]$ drops the first character from s_1. Abstractly, dropping the first character from $r_1 =$ `r'\.[0-9]+\.[0-9]+'` gives $r_2 \doteq$ `r'[0-9]+\.[0-9]+'`, a suffix of r_1, that overapproximates s_2. Thus, s_2 has type r_2.

Infer the type of s'. This is dual to the type inference of s_1. Concretely, the pattern `'.'` splits s_2 into two parts where $s' = s_2[: s_2.\mathit{find}(\texttt{'.'})]$ is the prefix to the left of the first `'.'`. Abstractly, this pattern splits $r_2 =$ `r'[0-9]+\.[0-9]+'` into two parts where $r' \doteq$ `r'[0-9]+'`, the prefix to the left of the first `'.'`, overapproximates s'. Thus, s' has type r'. See §4.2.1 for the formal definitions of the abstract operations shown above.

Subtyping. The inferred type r' happens to be the expected `r'[0-9]+'` (*i.e.*, they are syntactically equivalent), thus the subtype check passes trivially. In general cases where the inferred and expected types differ in syntax, we apply a symbolic approach proposed by Keil and Thiemann [2014].

Local type inference. Due to the nature that FLAT-Checker only infers the result types of string operations, it indeed supports *local type inference*. This feature brings two benefits to users. First, users are allowed to query the type of any string-typed expression. To use it, FLAT-Checker offers a special `show_type(e)` command that will display the inferred type for the string-typed expression `e` specified by the user, for example:

```
def extract_minor(s: VersionNumber):
  i1 = s.index('.') + 1
  show_type(s[i1:])
  i2 = s.index('.', i1)
  return s[i1:i2]
```

It will output the type r_2 for `s[i1:]`. Second, users do not need to explicitly specify return types for non-recursive functions, but FLAT-Checker automatically infers it. In the code shown above (where the return type is not specified), it will infer the return type to be r'.

Key takeaways. From this example, we see that FLAT-Checker:

- decomposes complex type inference tasks into smaller and simpler ones via expression simplification;
- allows local type inference, and offers a `show_type` command to display the inferred type of a given string-typed expression;
- relies on subtype checking for type checking.

2.3 Reasoning About Loops with Invariants

Starting from the third example, let us move from short functions to longer ones that involve non-trivial loops. To reason about them, a standard approach from the program verification community is to employ user-specified loop invariants. FLAT-Checker offers a special `inv` command to accept user-specified loop invariants described as Boolean-typed expressions, at the beginning of a loop body, as follows:

```
while cond:
  inv(invariant1)
  inv(invariant2)
  ... # original code of loop body
```

Let us consider an example taken from our evaluation dataset. The input type is specified (note that here we introduce a type alias in Line 1), but loop invariants are missing:

```
1  type Input = lang(r'a*|b*')
2
3  def f401(s: Input):
4    i = 0
5    ca = 0
6    cb = 0
7    while i < len(s):
8      if s[i] == "a":
9        ca += 1
10     if s[i] == "b":
11       cb += 1
12     i += 1
13   assert ca == len(s) or cb == len(s)
```

Inspecting the loop, we see that the local variable `ca` (*resp.* `cb`) maintains the number of `'a'`s (*resp.* `'b'`s) we have seen so far. Because the input string `s` contains either `'a'` or `'b'`, the variable `ca` (*resp.* `cb`) is always identical to `i` if `s` contains `'a'` (*resp.* `'b'`). Therefore, when the loop exits, `i` will become `len(s)`, so that the assertion in Line 13 will hold. Based on this analysis, we specify two loop invariants and insert them at Line 8:

```
inv(0 <= i <= len(s))
inv(ca == i if 'a' in s else cb == i)
```

The first one is usual for describing the range of `i`. The second one uncovers the relationship between `ca`, `cb`, and `i`, as discussed above. The Python expression `e1 if e else e2` is analogue to `if e then e1 else e2` in Haskell or OCaml, and the infix test (aka string contains) `'a' in s` is analogue to `s.contains("a")` in Java or Scala.

FLAT-Checker now takes over type checking. The most important and difficult VCs check that the second loop invariant holds after an iteration:

$$P \doteq s \in \texttt{r'a*|b*'} \land 0 \le i \le |s| \land \textbf{if } \texttt{'a'} \textbf{ in } s \textbf{ then } c_a = i \textbf{ else } c_b = i \land i < |s|$$

$$\varphi_1 : P \land s[i] = \texttt{'a'} \Rightarrow \textbf{if } \texttt{'a'} \textbf{ in } s \textbf{ then } c_a + 1 = i + 1 \textbf{ else } c_b = i + 1$$

$$\varphi_2 : P \land s[i] = \texttt{'b'} \Rightarrow \textbf{if } \texttt{'a'} \textbf{ in } s \textbf{ then } c_a = i + 1 \textbf{ else } c_b + 1 = i + 1$$

where φ_1 is for the case $s[i] = \texttt{'a'}$ (the condition in Line 8), and φ_2 is for the case $s[i] = \texttt{'b'}$ (the condition in Line 10).

Type inference, first attempt. To discharge them, FLAT-Checker generates lemmas through inferring the type of `'a'` **in** s. Noticing this infix test returns a Boolean value, the inferred type should not be a regular language type any more, but one of the truth values in three-valued logic:

- true if the test *always* evaluates to true;
- false if the test *always* evaluates to false;
- otherwise bool, indicating unknown or nondetermined.

If starting from the user-specified type `r'a*|b*'` for s, one can not determine whether `'a'` **in** s is true or false, as `'a'` may (if $s \in$ `r'a*'`) or may not (if $s \in$ `r'b*'`) occur in s. Thus `'a'` **in** s is only safe to have type bool, but this does not give a useful lemma to progress the VCs.

Type narrowing to rescue. The user-specified type for s is too coarse to infer precise types that lead to useful lemmas. But noticing that some **`if`**- and **`while`**-guards *implicitly* provide additional constraints on s, it is possible to compute a *more precise* type according to them. Here, the constraint $s[i]$ = `'a'` in φ_1 (*resp.* $s[i]$ = `'b'` in φ_2) indeed narrows down the possible values of s. FLAT-Checker applies *type narrowing* on `r'a*|b*'` according to the predicate $\lambda s, s[i]$ = `'a'` (*resp.* $\lambda s, s[i]$ = `'b'`), yielding a more precise type `r'a+'` (*resp.* `r'b+'`).

From $s \in$ `r'a+'` (*resp.* $s \in$ `r'b+'`), it will infer that `'a'` **in** s has type true (*resp.* false), which leads to a useful lemma `'a'` **in** s (*resp.* $\neg$(`'a'` **in** s)). With this lemma, the VC becomes simple enough for SMT solving, as it can be simplified into a trivial goal $c_a = i \Rightarrow c_a + 1 = i + 1$ (*resp.* $c_b = i \Rightarrow c_b + 1 = i + 1$).

Key takeaway. This example demonstrates that with user-specified loop invariants, type inference, type narrowing, and SMT solving, all in one framework FLAT-Checker, we are able to verify nontrivial properties about loops within a reasonable amount of human efforts.

2.4 Types as Loop Invariants

In the last example, we study a program that iterates multiple characters at a time. This program is also taken from the Panini benchmark suite:

```
1  type Input = lang(r'(a[^ab]b)*')
2
3  def f451(s: Input):
4    i = 0
5    while i < len(s):
6      a,x,b = s[i:i+3]
7      assert a == "a"
8      assert b == "b"
9      assert x != "a" and x != "b"
10     i += 3
```

The input type reveals that the input string `s` loops in units of every three characters, where the first character is `'a'`, the last is `'b'`, and the middle is any character other than `'a'` and `'b'`. The loop at Lines 5–10 also iterates `s` in units of every three characters. At Line 6, it extracts the three characters starting at `i` and store them as `a`, `x`, and `b` respectively. List unpacking is used in this assignment statement, which requires that the number of extracted values is equal to the the number of the variables on the left-hand side. FLAT-Checker desugars it into:

```
assert len(s[i:i+3]) == 3
a = s[i:i+3][0]
x = s[i:i+3][1]
b = s[i:i+3][2]
```

Then, the three assertions in Lines 7–9 test each extracted character.

Our key observation here is that in each iteration, the index `i` always points to a starting position of the three-character unit. This can be precised described as "`s[i]` is in regex `r'(a[^ab]b)*'`", which is expressed as `isinstance(s[i:], Input)` in Python, in a view of types. We thus specify it as a loop invariant, together with a usual one for describing the range of `i`, and insert them at Line 6:

```
inv(0 <= i < len(s) + 3)
inv(isinstance(s[i:], Input))
```

FLAT-Checker now takes over type checking. The VCs we are interested here check that the three assertions in Lines 7–9 are valid:

$$P \doteq s \in \texttt{r'(a[\^{}ab]b)*'} \wedge s[i:] \in \texttt{r'(a[\^{}ab]b)*'} \wedge 0 \leq i < |s| + 3 \wedge i < |s|$$
$$\varphi_1 : P \Rightarrow a \doteq s[i : i+3][0] = \texttt{'a'}$$
$$\varphi_2 : P \Rightarrow b \doteq s[i : i+3][2] = \texttt{'b'}$$
$$\varphi_3 : P \Rightarrow x \neq \texttt{'a'} \wedge x \neq \texttt{'b'} \text{ where } x \doteq s[i : i+3][1]$$

To discharge them, FLAT-Checker first decomposes these substrings similar to what we have seen in §2.2: $s[i : i+3][k]$ for $k = 0, 1, 2$ is simplified to $s'[k]$ where $s' \doteq s[i:][: 3]$.

Infer the type of s'. Concretely, $s' = s[i:][: 3]$ takes three character from $s[i:]$. Abstractly, taking three characters from `r'(a[^ab]b)*'` gives $r' \doteq$ `r'a[^ab]b'`, a prefix of $r \doteq$ `r'(a[^ab]b)*'`, that overapproximates s'. Thus, s' has type r'.

Infer the type of $s'[0]$. Concretely, $s'[0]$ is the first character of s'. Abstractly, taking the first character from $r' =$ `r'a[^ab]b'` gives `r'a'` that approximates $s'[0]$. Thus, $s'[0]$ has type `r'a'`, from which a lemma $s'[0] =$ `'a'` is derived, and φ_1 is done.

Infer the types of $s'[1]$ *and* $s'[2]$. Concretely, $s'[1]$ is the first character after dropping one character from s', *i.e.*, $s'[: 1][0]$. Abstractly, dropping one character from $r' =$ `r'a[^ab]b'` gives `r'[^ab]b'` that approximates $s'[: 1]$, and then taking the first character from it gives `r'[^ab]'` that approximates $s'[1]$. Thus, $s'[1]$ has type `r'[^ab]'`, from which a lemma $s'[1] \neq$ `'a'` $\wedge\, s'[1] \neq$ `'b'` is derived, and φ_3 is done. Similarly, $s'[2]$ has type `r'b'`, from which a lemma $s'[2] =$ `'b'` is derived, and φ_2 is done.

Key takeaway. This example shows that "(regular language) types as invariants" is a good idea for reasoning about loops like in `f451` that traverses strings in accordance with their underlying looping structure, as the types precisely encode the needed structural information for the suffixes being traversed.

3 Preliminaries

3.1 Strings and String Operations

Let Σ be the set of Unicode characters. A *string* $s \in \mathbb{S}$ is a list of Unicode characters. Using the list syntax, we denote an empty string by $[]$ and a nonempty string by $\sigma :: s$ where σ is its head character. We consider string operations listed in Table 1, with their equivalences represented in Python and SMT-LIB, where the meta-variables s, t range over strings, and i, j range over integers.

The concat, reverse, and length operations all have usual semantics. We use the reverse operation to better describe operations defined by duality, though it is not included in SMT-LIB.

The *char-at* operation returns a singleton string $[\sigma]$ if $0 \leq i < |s|$ and σ is the character of s at i; otherwise it returns $[]$. We do not support Python's negative index feature, where `s[-1]` is a shorthand of `s[len(s) - 1]`. In our setting, $s[i] = []$ if $i < 0$. Thus $s[i]$ is equivalent to Python's `s[i]` only if $0 \leq i < |s|$.

For the substring operation, if both i and j are nonnegative, it will extract all characters from s starting at i (inclusive) and ending until $\max\{j, |s|\}$ (exclusive) as a string. If $i < j$, or either i or j is negative, it will return $[]$. Thus $s[i : j]$ is equivalent to Python's `s[i:j]` if both i and j are nonnegative. The corresponding SMT-LIB term is `(str.substr s i (- j i))`, where the last argument is the length of the extracted substring.

The substring operation contains two special forms: $s[i:]$ as a shorthand of $s[i : |s|]$, and $s[: j]$ as a shorthand of $s[0 : j]$. They happen to be the drop and take operations on lists: $s.drop(i)$ and

Table 1. String operations.

Name	Notation	Python	SMT-LIB
concat	$s_1 \mathbin{+\!\!+} s_2$	`s1 + s2`	`(str.++ s1 s2)`
reverse	s^{-1}	`s[::-1]`	-
length	$\lvert s\rvert$	`len(s)`	`(str.len s)`
char-at	$s[i]$	`s[i] if 0 <= i < len(s) else ''`	`(str.at s i)`
substring	$s[i:j]$	`s[i:j] if i >= 0 and j >= 0 else ''`	`(str.substr s i (- j i))`
prefix	$t \sqsubseteq s$	`s.startswith(t)`	`(str.prefixof t s)`
suffix	$t \sqsupseteq s$	`s.endswith(t)`	`(str.suffixof t s)`
infix	t **in** s	`t in s`	`(str.contains s t)`
find	$s.\mathit{find}(t)$	`s.find(t)`	`(str.indexof s t 0)`

$s.\mathit{take}(j)$. Indeed, the substring operation can be defined by composing a take and a drop operation, in either order:

$$s[i:j] = s[:j][i:] \qquad\qquad s[i:j] = s[i:][j-i:]$$

The char-at operation can be defined by $s[i] = s[i:][1]$.

The prefix, suffix, and infix (or contains, we use the name "infix" for better consistency with the other two operations) operations all have usual semantics.

The find operation returns the index of the first occurrence of t in s if found, and -1 otherwise. In SMT-LIB, the last argument specifies the starting index. We desugar `(str.indexof s t i)` as $s[i:].\mathit{find}(t)$ in our setting. In Python, there is a safe version `s.index(t)` that will raise `ValueError` if `t` is not found.

3.2 Regular Expressions

We consider a kind of *extended regular expressions* presented in [Keil and Thiemann 2014]:

$$r ::= \varnothing \mid \varepsilon \mid C \mid r_1 \mathbin{+\!\!+} r_2 \mid r_1 \cup r_2 \mid r^*$$

Compare with standard regular expressions, the literals (Cs) are *charsets*, *i.e.*, any subset of Σ. The usual character literal σ is encoded as $\{\sigma\}$, and the all-char expression is encode as Σ.

We inductively define the membership relation "$s \in r$" (s is a member of r) by the following rules:

$$\frac{}{[] \in \varepsilon}\ \in_\varepsilon \qquad \frac{\sigma \in C}{[\sigma] \in C}\ \in_C \qquad \frac{s_1 \in r_1 \quad s_2 \in r_2}{s_1 \mathbin{+\!\!+} s_2 \in r_1 \mathbin{+\!\!+} r_2}\ \in_{+\!\!+}$$

$$\frac{s \in r_1}{s \in r_1 \cup r_2}\ \in_\cup^L \qquad \frac{s \in r_2}{s \in r_1 \cup r_2}\ \in_\cup^R \qquad \frac{}{[] \in r^*}\ \in_*^0 \qquad \frac{s_1 \neq [] \quad s_1 \in r \quad s_2 \in r^*}{s_1 \mathbin{+\!\!+} s_2 \in r^*}\ \in_*$$

The usual notion of "the language of r" is essentially $\{s \mid s \in r\}$.

Kleene plus r^+, optional $r^?$, power r^n (repeat r exactly n times), and loop $r^{\{n_1,n_2\}}$ (repeat r at least n_1 and at most n_2 times) can be desugared into basic constructs:

$$r^+ \doteq r \mathbin{+\!\!+} r^* \qquad r^? \doteq \varepsilon \cup r \qquad r^0 \doteq \varepsilon \qquad r^n \doteq r \mathbin{+\!\!+} r^{n-1}$$

$$r^{\{n,n+k\}} \doteq r^n \mathbin{+\!\!+} (r^0 \cup \cdots \cup r^k)$$

We lift the reverse operation from strings to regexes:

$$
\begin{aligned}
\ulcorner r_1 \mathbin{+\!\!+} r_2 \lrcorner^{-1} &\doteq \ulcorner r_2 \lrcorner^{-1} \mathbin{+\!\!+} \ulcorner r_1 \lrcorner^{-1} \\
\ulcorner r_1 \cup r_2 \lrcorner^{-1} &\doteq \ulcorner r_1 \lrcorner^{-1} \cup \ulcorner r_2 \lrcorner^{-1} \\
\ulcorner r^* \lrcorner^{-1} &\doteq \left(\ulcorner r \lrcorner^{-1}\right)^* \\
\ulcorner r \lrcorner^{-1} &\doteq r \qquad \text{otherwise}
\end{aligned}
$$

This operation overapproximates the semantics of the string reverse operation:

Lemma 3.1. *If $s \in r$, then $s^{-1} \in \ulcorner r \lrcorner^{-1}$.*

The Brzozowski derivative [Brzozowski 1964] $\partial_\sigma r$ computes the regex after consuming σ using r (if not possible, it returns $\varnothing$):

$$
\begin{aligned}
\partial_\sigma \varnothing &\doteq \varnothing \\
\partial_\sigma \varepsilon &\doteq \varnothing \\
\partial_\sigma C &\doteq \textbf{if } \sigma \in C \textbf{ then } \varepsilon \textbf{ else } \varnothing \\
\partial_\sigma (r_1 \mathbin{+\!\!+} r_2) &\doteq (\partial_\sigma r_1 \mathbin{+\!\!+} r_2) \cup (\textbf{if } \nu(r_1) \textbf{ then } \partial_\sigma r_2 \textbf{ else } \varnothing) \\
\partial_\sigma (r_1 \cup r_2) &\doteq \partial_\sigma r_1 \cup \partial_\sigma r_2 \\
\partial_\sigma (r^*) &\doteq \partial_\sigma r \mathbin{+\!\!+} r^*
\end{aligned}
$$

Lemma 3.2. *$s \in \partial_\sigma r \Leftrightarrow \sigma :: s \in r$.*

The Brzozowski derivative can be extended from characters to strings via chaining:

$$
\begin{aligned}
\partial_{[\,]} r &\doteq r \\
\partial_{\sigma::t} &\doteq \partial_t (\partial_\sigma r)
\end{aligned}
$$

Lemma 3.3. *$s \in \partial_t r \Leftrightarrow t \mathbin{+\!\!+} s \in r$.*

We say two regexes are *equivalent*, denoted by $r_1 \equiv r_2$, if for any string s, $s \in r_1 \Leftrightarrow s \in r_2$.

4 Type Inference via Abstract Operations

Type inference is the fundamental component of FLAT-Checker. The inferred types are used to immediately discharge the VC or derive useful lemmas that ideally make the VC simple enough for SMT solving. Type inference applies to all string operations listed in Table 1. For each, we define one or more *abstract operations* that overapproximate its concrete semantics. The return values of the abstract operations, which are elements in the *abstract domain* of the return values of that concrete operation, are regarded as the inferred *types* ("abstract domains as types"). The concrete operations can return strings, Booleans, and integers. Thus our type system includes three groups of refinement types:

- regexes as refinement types for strings (*i.e.,* regular language types),
- three-valued logic truth values as refinement types for Booleans (introduced in §2.3), and
- *integer intervals* from standard interval abstraction [Cousot and Cousot 1976] as refinement types for integers.

We summarize the terms and types (in our core language) relevant to type inference as follows:

$$
\begin{aligned}
\text{Term } e &::= s \mid n \mid x \mid e_1 + e_2 \mid e_1 - e_2 \\
&\mid e_1 \mathbin{+\!\!+} e_2 \mid e^{-1} \mid |e| \mid e[e_i] \mid e[e_i : e_j] \mid e.\mathit{find}(e') \mid e' \sqsubseteq e \mid e' \sqsupseteq e \mid e' \textbf{ in } e \\
\text{Type } \tau &::= r \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{bool} \mid I
\end{aligned}
$$

The meta variable s ranges over strings, n ranges over integers, x ranges over program variables, and I ranges over integer intervals. Terms constructing the string operations listed in Table 1 share the same notation.

We introduce the type inference judgement "$\Gamma \vdash e \triangleright \tau$": it states that, under the *proof context* Γ, the term e is inferred to have the (refinement) type τ. For a VC, the proof context is the set of all premises (or assumptions, constraints) on the left-hand side of "$\Rightarrow$". We first present several straightforward type inference rules:

$$\frac{}{\Gamma \vdash s \triangleright \{s\}}\ \tau\text{-str-const} \qquad \frac{}{\Gamma \vdash n \triangleright [n, n]}\ \tau\text{-int-const} \qquad \frac{(e \in r) \in \Gamma}{\Gamma \vdash e \triangleright r}\ \tau\text{-ctx}$$

$$\frac{\Gamma \vdash e_1 \triangleright r_1 \qquad \Gamma \vdash e_2 \triangleright r_2}{\Gamma \vdash e_1 +\!\!+\, e_2 \triangleright r_1 +\!\!+\, r_2}\ \tau\text{-concat} \qquad \frac{\Gamma \vdash e \triangleright r}{\Gamma \vdash e^{-1} \triangleright \ulcorner r \urcorner^{-1}}\ \tau\text{-reverse}$$

Semantically, $\Gamma \vdash e \triangleright \tau$ is interpreted as $\Gamma \Rightarrow e \in \tau$. We say a rule is *sound* if it is logically *valid* under such a semantic interpretation. For example, we formulate the soundness of the rule τ-concat as the following logical proposition, which is valid by $\in_{+\!\!+}$:

$$(\Gamma \Rightarrow e_1 \in r_1) \wedge (\Gamma \Rightarrow e_2 \in r_2) \Rightarrow (\Gamma \Rightarrow e_1 +\!\!+\, e_2 \in r_1 +\!\!+\, r_2).$$

Similarly, the soundness of the rule τ-reverse is given by Lemma 3.1. All the rules we will present in this section are proved sound in Rocq.

4.1 Prefix, Suffix, and Infix

As the suffix test is just the reverse of prefix test: $t \sqsupseteq s \Leftrightarrow t^{-1} \sqsubseteq s^{-1}$, we will only consider rules for prefix and infix test. Given constant strings t as patterns in $t \sqsubseteq e$ and t **in** e, we study under which conditions the tests are *determined* to be true or false. For any other cases including non-constant patterns, the inferred type is simply bool.

4.1.1 Rules for prefix. We start with the prefix test $t \sqsubseteq e$. Let r be the inferred type for e. Recall the property of Brzozowski derivatives. By Lemma 3.3, if $\partial_t r$ is empty, then t is *never* a prefix of e, which suggests the following rule:

$$\frac{\Gamma \vdash e \triangleright r \qquad \partial_t r \equiv \varnothing}{\Gamma \vdash t \sqsubseteq e \triangleright \mathsf{false}}\ \tau\text{-prefix-false}$$

Oppositely, to show that t is *always* a prefix of e, computing derivative alone is insufficient. We must also ensure that at all times a character $\sigma \in t$ is ready to be consumed, any string in that language must start with σ but nothing else. For this, we need a "time machine" that restores the regex used to consume σ. We define this as an abstract operation take1 that overapproximates the take-first-character operation:

$$\begin{aligned}
\mathsf{take1}\,\varnothing &\doteq \varepsilon \\
\mathsf{take1}\,\varepsilon &\doteq \varepsilon \\
\mathsf{take1}\,C &\doteq C \\
\mathsf{take1}(r_1 +\!\!+\, r_2) &\doteq \mathsf{take1}\,r_1 \cup (\textbf{if } \nu(r_1) \textbf{ then } \mathsf{take1}\,r_2 \textbf{ else } \varnothing) \\
\mathsf{take1}(r_1 \cup r_2) &\doteq \mathsf{take1}\,r_1 \cup \mathsf{take1}\,r_2 \\
\mathsf{take1}(r^*) &\doteq \mathsf{take1}\,r \cup \varepsilon
\end{aligned}$$

Like in Brzozowski derivative, we must consider the case when r_1 is nullable for $r_1 +\!\!+\, r_2$. For r^*, the possibility of repeating r zero times is also considered (by $\cup$-ing ε).

Lemma 4.1. *If $s \in r$, then $s.take(1) \in \mathsf{take1}\, r$.*

Then our approach is formulated as a procedure $\mathsf{prefix}^{\forall}{}_t$:

$$\begin{aligned}\mathsf{prefix}^{\forall}{}_{[]}\, r &\doteq \mathbf{true}\\ \mathsf{prefix}^{\forall}{}_{\sigma::t}\, r &\doteq \mathbf{if}\ \ \mathsf{take1}\, r \equiv \{\sigma\}\ \mathbf{then}\ \mathsf{prefix}^{\forall}{}_t(\partial_\sigma r)\ \mathbf{else\ false}\end{aligned}$$

Lemma 4.2. *If $\mathsf{prefix}^{\forall}{}_t\, r$, then $t \sqsubseteq s$ for any $s \in r$.*

This lemma suggests the following sound rule determining that t is a prefix of e:

$$\frac{\Gamma \vdash e \triangleright r \qquad \mathsf{prefix}^{\forall}{}_t\, r}{\Gamma \vdash t \sqsubseteq e \triangleright \mathsf{true}}\ \tau\text{-prefix-true}$$

4.1.2 Rules for infix. If t **in** s, then there must be a suffix s_2 of s, which starts with the first character, say σ, of t (note the case for $t = []$ is trivial so we elide it here), such that $t \sqsubseteq s_2$. This motivates us to decompose the infix tests into two steps: first split s into a prefix s_1 and s_2, where s_2 starts with σ, and then reapply the approaches we have introduced for prefix to test if $t \sqsubseteq s_2$.

For the first step, we define an abstract operation $\mathsf{split}_\sigma\, r$ to split r into a set of regex pairs, where each $\langle r_1, r_2\rangle$ is a possibility that overapproximates the prefix and suffix pair $\langle s_1, s_2\rangle$ after splitting s by σ:

$$\begin{aligned}\mathsf{split}_\sigma\, \varnothing &\doteq \varnothing\\ \mathsf{split}_\sigma\, \varepsilon &\doteq \varnothing\\ \mathsf{split}_\sigma\, C &\doteq \mathbf{if}\ \sigma \in C\ \mathbf{then}\ \{\langle \varepsilon, C\rangle\}\ \mathbf{else}\ \varnothing\\ \mathsf{split}_\sigma(r_1 +\!\!+\, r_2) &\doteq \{\langle r_L, r_R +\!\!+\, r_2\rangle \mid \langle r_L, r_R\rangle \in \mathsf{split}_\sigma\, r_1\} \cup \{\langle r_1 +\!\!+\, r_L, r_R\rangle \mid \langle r_L, r_R\rangle \in \mathsf{split}_\sigma\, r_2\}\\ \mathsf{split}_\sigma(r_1 \cup r_2) &\doteq \mathsf{split}_\sigma\, r_1 \cup \mathsf{split}_\sigma\, r_2\\ \mathsf{split}_\sigma(r^*) &\doteq \{\langle r^* +\!\!+\, r_L, r_R +\!\!+\, r^*\rangle \mid \langle r_L, r_R\rangle \in \mathsf{split}_\sigma\, r\}\end{aligned}$$

Lemma 4.3. *Let $s \in r$. If $s[i] = \sigma$, then there exist $\langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r$ such that $s[:i] \in r_1$ and $s[i:] \in r_2$.*

This lemma shows that split_σ covers all possible splits by σ. Thus, to show that t is *never* an infix, it suffices to show that t is not a prefix of any r_2 for $\langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r$, via Brzozowski derivative as done in τ-prefix-false. For convenience, we take the (big) union of all r_2s as

$$\mathsf{splitSuffix}_\sigma \doteq \bigcup \{r_2 \mid \langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r\}.$$

We present our rule in a compact form:

$$\frac{t = \sigma :: t' \qquad \Gamma \vdash e \triangleright r \qquad \partial_t(\mathsf{splitSuffix}_\sigma\, r) \equiv \varnothing}{\Gamma \vdash t\ \mathbf{in}\ e \triangleright \mathsf{false}}\ \tau\text{-infix-false}$$

To show that t is *always* an infix, it suffices to show that t is always a prefix of $\mathsf{splitSuffix}_\sigma\, r$, via $\mathsf{prefix}^{\forall}{}_t$ as done in τ-prefix-true, assuming that σ always exists. For this side condition, we introduce a simple procedure $\mathsf{have}^{\forall}{}_{\sigma}\, r$ to test if σ is a common character for all strings in r:

$$\begin{aligned}
\mathsf{have}^{\forall}{}_{\sigma}\, \varnothing &\doteq \textbf{false} \\
\mathsf{have}^{\forall}{}_{\sigma}\, \varepsilon &\doteq \textbf{false} \\
\mathsf{have}^{\forall}{}_{\sigma}\, C &\doteq \textbf{if } C = \{\sigma\} \textbf{ then } \top \textbf{ else false} \\
\mathsf{have}^{\forall}{}_{\sigma}(r_1 +\!\!\!+\, r_2) &\doteq \mathsf{have}^{\forall}{}_{\sigma}\, r_1 \lor \mathsf{have}^{\forall}{}_{\sigma}\, r_2 \\
\mathsf{have}^{\forall}{}_{\sigma}(r_1 \cup r_2) &\doteq \mathsf{have}^{\forall}{}_{\sigma}\, r_1 \land \mathsf{have}^{\forall}{}_{\sigma}\, r_2 \\
\mathsf{have}^{\forall}{}_{\sigma}(r^*) &\doteq \textbf{true}
\end{aligned}$$

Lemma 4.4. *If* $\mathsf{have}^{\forall}{}_{\sigma}\, r$*, then* $\sigma \in s$ *for any* $s \in r$*.*

Finally we present a sound rule determining that t is an infix of e:

$$\frac{t = \sigma :: t' \qquad \Gamma \vdash e \triangleright r \qquad \mathsf{have}^{\forall}{}_{\sigma}\, r \qquad \mathsf{prefix}^{\forall}{}_{t}(\mathsf{splitSuffix}_{\sigma}\, r)}{\Gamma \vdash t \textbf{ in } e \triangleright \mathsf{true}}\ \tau\text{-infix-true}$$

4.2 Char-At and Substring

The char-at operation can be desugared into more basic substring operations: $s[i] = s[:i][1:]$. To define rules for substring, which itself can be decomposed into drop and take operations, we first define two basic operations that abstract over the drop and take operations under restricted forms of indices (§4.2.1), and then generalize them to substring (§4.2.2).

4.2.1 Two basic operations. An *abstract index* κ is one of the following:

- $\overrightarrow{k}$ $(k \geq 0)$: an absolute position counting from left to right;
- $\overleftarrow{k}$ $(k \geq 0)$: the reversed version of the above that counts from right to left; $\overleftarrow{1}$ is the right-most index; $\overleftarrow{0}$ marks the end of the string;
- $@t$: a relative position at the first occurrence of the constant pattern t.

We give semantic interpretation $[\![\kappa]\!]_s$ that concretizes κ to a concrete index of a given string s:

$$[\![\overrightarrow{k}]\!]_s \doteq k \qquad\qquad [\![\overleftarrow{k}]\!]_s \doteq |s| - k \qquad\qquad [\![@t]\!]_s \doteq s.\mathit{find}(t)$$

Our purpose is to define abstract operations $\mathsf{dropI}_{\kappa}\, r$ and $\mathsf{takeI}_{\kappa}\, r$ that overapproximate $s.\mathit{drop}(n)$ and $s.\mathit{take}(n)$ when $s \in r$ and $[\![\kappa]\!]_s = n$.

Operations under absolute positions. We start by defining the dual version of $\mathsf{take1}$ (defined in §4.1.1) that overapproximates the drop-one-character operation:

$$\begin{aligned}
\mathsf{drop1}\, \varnothing &\doteq \varepsilon \\
\mathsf{drop1}\, \varepsilon &\doteq \varepsilon \\
\mathsf{drop1}\, C &\doteq \varepsilon \\
\mathsf{drop1}(r_1 +\!\!\!+\, r_2) &\doteq (\mathsf{drop1}\, r_1 +\!\!\!+\, r_2) \cup (\textbf{if } \nu(r_1) \textbf{ then } \mathsf{drop1}\, r_2 \textbf{ else } \varnothing) \\
\mathsf{drop1}(r_1 \cup r_2) &\doteq \mathsf{drop1}\, r_1 \cup \mathsf{drop1}\, r_2 \\
\mathsf{drop1}(r^*) &\doteq (\mathsf{drop1}\, r +\!\!\!+\, r^*) \cup \varepsilon
\end{aligned}$$

This is indeed an analogue to $\partial_{\sigma} r$, but with two differences: (a) any character can be consumed, thus no need to check $\sigma \in C$; (b) this operation does not return $\varnothing$ but ε instead, as the drop operation returns [] if no characters are left.

Lemma 4.5. *If* $s \in r$*, then* $s.\mathit{drop}(1) \in \mathsf{drop1}\, r$*.*

Just like one can extend $\partial_\sigma r$ to $\partial_t r$, we extend $\mathsf{drop1}$ to drop_n for $n \geq 0$ via chaining, which overapproximates the drop operation:

$$\begin{aligned} \mathsf{drop}_0\, r &\doteq r \\ \mathsf{drop}_n\, r &\doteq \mathsf{drop}_{n-1}(\mathsf{drop1}\, r) \end{aligned}$$

Lemma 4.6. *If $s \in r$, then $s.drop(n) \in \mathsf{drop}_n\, r$.*

We have defined the $\mathsf{take1}$ operation in §4.1.1. Similarly, we extend it to take_n for $n \geq 0$ via chaining, which overapproximates the take operation:

$$\begin{aligned} \mathsf{take}_0\, r &\doteq \varepsilon \\ \mathsf{take}_n\, r &\doteq \mathsf{take1}\, r \mathbin{+\!\!+} \mathsf{take}_{n-1}(\mathsf{drop1}\, r) \end{aligned}$$

Lemma 4.7. *If $s \in r$, then $s.take(n) \in \mathsf{take}_n\, r$.*

With reverse, we complete the first part definition of the two basic operations:

$$\begin{aligned} \mathsf{dropl}_{\overrightarrow{k}}\, r &\doteq \mathsf{drop}_k\, r & \qquad \mathsf{takel}_{\overrightarrow{k}}\, r &\doteq \mathsf{take}_k\, r \\ \mathsf{dropl}_{\overleftarrow{k}}\, r &\doteq \ulcorner \mathsf{take}_k \ulcorner r \urcorner^{-1} \urcorner^{-1} & \qquad \mathsf{takel}_{\overleftarrow{k}}\, r &\doteq \ulcorner \mathsf{drop}_k \ulcorner r \urcorner^{-1} \urcorner^{-1} \end{aligned}$$

Operations under relative positions. Our purpose is to define abstract operations that approximate $s.drop(i)$ and $s.take(i)$ where $i = s.find(t)$. A trivial but rare case is when $t = []$, it is immediate that $s.drop(i) = s$ and $s.take(i) = []$, thus their approximations are given by r and ε.

A simple but very common case is when $t = [\sigma]$ (singleton). Lemma 4.3 implies that split_σ already gives us sound approximations, but they are *imprecise*: split_σ considers *all* occurrences of σ, but here i is the *first* occurrence. Based on these results, we present a more precise version:

$$\mathsf{find}_\sigma\, r \doteq \{\langle r_1', \{\sigma\} \mathbin{+\!\!+} \mathsf{drop1}\, r_2\rangle \mid \langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r, r_1' \doteq r_1 \setminus \{\sigma\}, r_1' \not\equiv \varnothing\}$$

We denote $r \setminus C'$ the exclusion of a charset C' from r, by propagating the exclusion of C' (*i.e.,* set minus $C \setminus C'$) to all charsets in r. For each result $\langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r$, we refine r_1 to exclude σ as r_1', so that i will become the first occurrence, as σ does not occur in r_1'. If the refinement leads to an empty language, *i.e.,* $r_1' \equiv \varnothing$, this indicates there is no chance i will become the first occurrence, thus we prune this case from the output. This operation covers all possible splits at the first occurrence of σ:

Lemma 4.8. *Let $s \in r$. If $s.find(\sigma) = i$ and $i \geq 0$, then there exist $\langle r_1, r_2\rangle \in \mathsf{find}_\sigma\, r$ such that $s[:i] \in r_1$ and $s[i:] \in r_2$.*

The needed approximations are given by taking the (big) union of all r_2s and r_1s:

$$\mathsf{findSuffix}_\sigma\, r \doteq \bigcup\{r_2 \mid \langle r_1, r_2\rangle \in \mathsf{find}_\sigma\, r\} \qquad \mathsf{findPrefix}_\sigma\, r \doteq \bigcup\{r_1 \mid \langle r_1, r_2\rangle \in \mathsf{find}_\sigma\, r\}$$

Finally, $t = \sigma :: t'$ can contain multiple characters. To generalize the above idea here, we need to define $r \setminus t$, the exclusion of an arbitrary string t from r. This operation is harder and we did not find any cheap and pleasant realization (unless the traditional regular language difference algorithm in automata theory). In the end, we fallback to extend split_σ to split_t where $t = \sigma :: t'$:

$$\mathsf{split}_t\, r \doteq \{\langle r_1, \{t\} \mathbin{+\!\!+} r_2'\rangle \mid \langle r_1, r_2\rangle \in \mathsf{split}_\sigma\, r, r_2' \doteq \partial_t r_2, r_2' \not\equiv \varnothing\}$$

Likewise, the needed approximations are given by taking the (big) union of all r_2s and r_1s:

$$\mathsf{splitSuffix}_t\, r \doteq \bigcup\{r_2 \mid \langle r_1, r_2\rangle \in \mathsf{split}_t\, r\} \qquad \mathsf{splitPrefix}_t\, r \doteq \bigcup\{r_1 \mid \langle r_1, r_2\rangle \in \mathsf{split}_t\, r\}$$

So far, we have completed the missing part of defining the two basic operations:

$$\mathsf{dropl}_{@[]}\, r \doteq r \qquad \mathsf{takel}_{@[]}\, r \doteq \varepsilon$$
$$\mathsf{dropl}_{@[\sigma]}\, r \doteq \mathsf{findSuffix}_{\sigma}\, r \qquad \mathsf{takel}_{@[\sigma]}\, r \doteq \mathsf{findPrefix}_{\sigma}\, r$$
$$\mathsf{dropl}_{@t}\, r \doteq \mathsf{splitSuffix}_{t}\, r \qquad \mathsf{takel}_{@t}\, r \doteq \mathsf{splitPrefix}_{t}\, r$$

Rules. The two basic operations give inference rules for $e.drop(e_n)$ and $e.take(e_n)$ if their indices e_n can be overapproximated by some κ:

$$\frac{\Gamma \vdash e \triangleright r \qquad \Gamma \Rightarrow 0 \le e_n = [\![\kappa]\!]_e}{\Gamma \vdash e.drop(e_n) \triangleright \mathsf{dropl}_{\kappa}\, r}\ \tau\text{-drop-}\kappa \qquad \frac{\Gamma \vdash e \triangleright r \qquad \Gamma \Rightarrow 0 \le e_n = [\![\kappa]\!]_e}{\Gamma \vdash e.take(e_n) \triangleright \mathsf{takel}_{\kappa}\, r}\ \tau\text{-take-}\kappa$$

4.2.2 *Rules for substring.*

Noticing that the substring operation can be decomposed into a drop and a take operation (in either order), our idea is to define an abstract operation $\mathsf{substr}(r, \kappa_i, \kappa_j)$ in terms of the two basic operations defined in §4.2.1, as the following rule:

$$\frac{\Gamma \vdash e \triangleright r \qquad \Gamma \Rightarrow 0 \le e_i < e_j \land e_i = [\![\kappa_i]\!]_e \land e_j = [\![\kappa_j]\!]_e}{\Gamma \vdash e[e_i : e_j] \triangleright \mathsf{substr}(r, \kappa_i, \kappa_j)}\ \tau\text{-substr-}\kappa$$

We assume that indices e_i and e_j are overapproximated by some κ_i and κ_j. Depending on their constructors, the inferred type $\mathsf{substr}(r, \kappa_i, \kappa_j)$ is given by the 3×3 table below, which includes a case when this operation is "undefined":

	$\kappa_i = \overrightarrow{k_i}$	$\kappa_i = \overleftarrow{k_i}$	$\kappa_i = @t_i$
$\kappa_j = \overrightarrow{k_j}$	$\mathsf{dropl}_{\kappa_i}(\mathsf{take}_{k_j}\, r)$	undefined	$\mathsf{dropl}_{\kappa_i}(\mathsf{take}_{k_j}\, r)$
$\kappa_j = \overleftarrow{k_j}$	$\mathsf{takel}_{\kappa_j}(\mathsf{dropl}_{\kappa_i}\, r)$		
$\kappa_j = @t_j$			

The first row "$\kappa_j = \overrightarrow{k_j}$" shows the idea of decomposing $s[i : j]$ as $s[: j][i :]$ (take and then drop). To ensure the suggested type $\mathsf{dropl}_{\kappa_i}(\mathsf{take}_{k_j}\, r)$ is an overapproximation of $s[: j][i :]$, a crucial property we need is that the semantics of κ_i should be preserved after taking j characters. The following lemma tells when the property holds:

Lemma 4.9. *Let κ be an abstract index constructed by $\overrightarrow{\cdot}$ or $@\cdot$. For any $n > [\![\kappa]\!]_s$, if $[\![\kappa]\!]_s \ge 0$, then $[\![\kappa]\!]_{s[:n]} = [\![\kappa]\!]_s$.*

We see that except when $\kappa_i = \overleftarrow{k_i}$, the suggested type $\mathsf{dropl}_{\kappa_i}(\mathsf{take}_{k_j}\, r)$ is sound. The case $\kappa_i = \overleftarrow{k_i}$ indicates a weird and rarely seen means of extracting substring such as $s[|s| - 1 : 2]$, where we do not define a rule.

The last two rows show the idea of decomposing $s[i : j]$ as $s[i :][: j - i]$ (drop and then take). To ensure the suggested type $\mathsf{takel}_{\kappa_j}(\mathsf{dropl}_{\kappa_i}\, r)$ is an overapproximation of $s[: i][: j - i]$, a crucial property we need is that the semantics of κ_j should be properly shifted after dropping i characters:

Lemma 4.10. *Let κ be an abstract index constructed by $\overleftarrow{\cdot}$ or $@\cdot$. For any $0 \le n < [\![\kappa]\!]_s$, we have $[\![\kappa]\!]_{s[n:]} = [\![\kappa]\!]_s - n$.*

This lemma implies that the suggested type $\mathsf{takel}_{\kappa_j}(\mathsf{dropl}_{\kappa_i}\, r)$ is sound.

Sometimes, developers also shift a relative position such as $s[s.\mathit{find}(\sigma)+1 :]$ and $s[: s.\mathit{find}(\sigma)-1]$. We deal with these cases by decoupling such shifts (+1, −1) by simplification, as demonstrated in §2.2. The simplification rules and their soundness are shown in the following lemma:

Lemma 4.11. *Let* $i, j, k \geq 0$*:*

$$
\begin{aligned}
k \leq i \wedge i \leq j \leq |s| \Rightarrow s[i-k : j] &= (s[: i]^{-1}[: k])^{-1} +\!\!+\; s[i : j] \\
s[i+k : j] &= s[i : j][k :] \\
i \leq j-k \wedge j \leq |s| \Rightarrow s[i : j-k] &= (s[i : j]^{-1}[k :])^{-1} \\
i \leq j \Rightarrow s[i : j+k] &= s[i : j] +\!\!+\; s[j :][: k]
\end{aligned}
$$

So far, our rules assume that the values of the indices are determined. We finally present a rule that can apply to nondetermined indices:

$$
\frac{\Gamma \vdash e \triangleright r \qquad \Gamma \Rightarrow 0 \leq e_i < e_j \wedge e_i \geq [\![\kappa_1]\!]_e \wedge e_j \leq [\![\kappa_2]\!]_e}{\Gamma \vdash e[e_i : e_j] \triangleright \alpha(\mathsf{substr}(r, \kappa_1, \kappa_2))^+} \;\tau\text{-substr-}\alpha
$$

We find κ_1 and κ_2 as the lower bound of e_i and the upper bound of e_j. The abstract operation $\mathsf{substr}(r, \kappa_1, \kappa_2)$ approximates the longest substring we can extract by the index range $[e_i : e_j]$. Hence any $e[e_i : e_j]$ must only contain characters occurred in $\mathsf{substr}(r, \kappa_1, \kappa_2)$. Thus we approximate $e[e_i : e_j]$ by the universal language generated from them, *i.e.*, $\alpha(\mathsf{substr}(r, \kappa_1, \kappa_2))^+$. This rule deals with, for example, the VC φ_3 shown in §2.1.

4.3 Length and Find

We lift the string length operation to an abstract operation that approximates all possible lengths as integer intervals:

$$
\begin{aligned}
|\varnothing|^\# &\doteq [0, 0] \\
|\varepsilon|^\# &\doteq [0, 0] \\
|C|^\# &\doteq [1, 1] \\
|r_1 +\!\!+\; r_2|^\# &\doteq |r_1|^\# + |r_2|^\# \\
|r_1 \cup r_2|^\# &\doteq |r_1|^\# \sqcup |r_2|^\# \\
|r^*|^\# &\doteq [0, \infty]
\end{aligned}
$$

Interval addition is defined as $[a, b] + [c, d] = [a+c, b+d]$. Interval join is defined as $[a, b] \sqcup [c, d] = [\min\{a, c\}, \max\{b, d\}]$.

Lemma 4.12. *If* $s \in r$*, then* $|s| \in |r|^\#$.

With $|r|^\#$, we define rules for length and find operations:

$$
\frac{\Gamma \vdash e \triangleright r}{\Gamma \vdash |e| \triangleright |r|^\#} \;\tau\text{-length} \qquad\qquad \frac{\Gamma \vdash e \triangleright r}{\Gamma \vdash e.\mathit{find}(t) \triangleright |\,\mathsf{takel}_{@t}\, r|^\# \sqcup [-1, -1]} \;\tau\text{-find}
$$

In the rule τ-find, if we are sure that t must exist in e, then we will exclude $[-1, -1]$ in the inferred type.

5 Type Narrowing for Higher Precision

FLAT-Checker applies *type narrowing* to improve the precision of type inference. As demonstrated in §2.3, type narrowing is the key to discharge the two VCs: without it, the inferred type bool for `'a'` **in** s would be too coarse. We formulate the type narrowing problem as follows: Let r be an original regular language type and $p : \mathbb{S} \to \mathbb{B}$ be a predicate over strings. Find a subtype $r' \subseteq r$ such that it also overapproximates p.

Here we consider *specific* predicates which we can do *efficient* type narrowing. We denote a *narrowing rule* by $r \overset{p}{\hookrightarrow} r'$. It is said *sound* if $\forall s, s \in r \land p(s) \Rightarrow s \in r'$. We integrate type narrowing into our type inference system by the following type inference rule:

$$\frac{(e \in r) \in \Gamma \qquad \Gamma \Rightarrow p(e) \qquad r \overset{p}{\hookrightarrow} r'}{\Gamma \vdash e \triangleright r'}\ \tau\text{-narrow}$$

This rule is sound as long as $r \overset{p}{\hookrightarrow} r'$ is sound.

In the rest of this section, we will present selected narrowing rules used by FLAT-Checker. Their soundnesses have been proved in Rocq.

Length comparison. We first look into predicates that compare the length of a string with a constant integer $k \geq 0$. We define an operation $\mathsf{resByLen}_I\, r$ for restricting r by a length interval I:

$$\begin{aligned}
\mathsf{resByLen}_I\, \varnothing &\doteq \varnothing \\
\mathsf{resByLen}_I\, \varepsilon &\doteq \textbf{if } 0 \in I \textbf{ then } \varepsilon \textbf{ else } \varnothing \\
\mathsf{resByLen}_I\, C &\doteq \textbf{if } 1 \in I \textbf{ then } C \textbf{ else } \varnothing \\
\mathsf{resByLen}_I(r_1 \mathbin{+\!\!+} r_2) &\doteq \textbf{if } |r_1|^{\#} = [k,k] \textbf{ then } r_1 \mathbin{+\!\!+} (\mathsf{resByLen}_{I-k}\, r_2) \\
&\ \textbf{else if } |r_2|^{\#} = [k,k] \textbf{ then } (\mathsf{resByLen}_{I-k}\, r_1) \mathbin{+\!\!+} r_2 \\
&\ \textbf{else } r_1 \mathbin{+\!\!+} r_2 \\
\mathsf{resByLen}_I(r_1 \cup r_2) &\doteq \mathsf{resByLen}_I\, r_1 \cup \mathsf{resByLen}_I\, r_2 \\
\mathsf{resByLen}_I(r^*) &\doteq \textbf{if } |r|^{\#} = [k,k] \textbf{ then let}[m,M] \doteq \frac{I}{k} \textbf{ in } r^{\{m,M\}} \\
&\ \textbf{else if } 0 \notin I \textbf{ then } r^+ \textbf{ else } r^*
\end{aligned}$$

We make several overapproximations here. For $r_1 \mathbin{+\!\!+} r_2$, if one side has constant length, say k, we continue restricting the other side by the leftover length interval $I-k$ where $[a,b]-k \doteq [a-k, b-k]$. Otherwise, there may exist too many possibilities to split I into two parts, thus we give up. We use the same strategy for r^*: if r has constant length, say $k > 0$, applying a kind of division $\frac{I}{k}$ gives an interval that approximates the repeating times of r. This division is defined as $\frac{[a,b]}{k} \doteq [\lceil \frac{a}{k} \rceil, \lfloor \frac{b}{k} \rfloor]$. Otherwise, we only check, if 0 is not included in I, meaning [] is not allowed, then we will narrow down r^* to r^+.

Lemma 5.1. *Let $s \in r$. Let I be an interval. If $|s| \in I$, then $s \in \mathsf{resByLen}_I\, r$.*

We present narrowing rules where the comparison operators are converted to intervals:

$$r \xhookrightarrow{\lambda s, |s|=k} \mathsf{resByLen}_{[k,k]}\, r \qquad r \xhookrightarrow{\lambda s, |s| \neq k} (\mathsf{resByLen}_{[0,k-1]}\, r) \cup (\mathsf{resByLen}_{[k+1,\infty]}\, r)$$

$$r \xhookrightarrow{\lambda s, |s| \leq k} \mathsf{resByLen}_{[0,k]}\, r \qquad r \xhookrightarrow{\lambda s, |s| \geq k} \mathsf{resByLen}_{[k,\infty]}\, r$$

$$r \xhookrightarrow{\lambda s, |s| < k} \mathsf{resByLen}_{[0,k-1]}\, r \qquad r \xhookrightarrow{\lambda s, |s| > k} \mathsf{resByLen}_{[k+1,\infty]}\, r$$

Similarly, we present a group of narrowing rules for the comparison between an index identified by the find operation and a constant integer, such as:

$$r \xhookrightarrow{\lambda s, s.\mathit{find}(\sigma) \geq k} \mathsf{resByLen}_{[k,\infty]}(\mathsf{findPrefix}_\sigma\, r) \mathbin{+\!\!+} \mathsf{findSuffix}_\sigma\, r$$

Char-at equations. We then look into predicates that test chars at specific absolute positions like $s[i] = \sigma$. Our idea is to filter out cases when $s[i] \cap \{\sigma\} = \varnothing$. For this, we need an abstract split operation that breaks r into a set of triples, where each $\langle r_1, C, r_2\rangle$ consists of r_1 that approximates $s[: i]$, C that approximates $s[i]$, and r_2 that approximates $s[i + 1 :]$. We first define $\mathsf{splitAt0}\, r$ that splits r into a set of pairs, where each $\langle C, r\rangle$ consists of C that approximations the head character of s, and r that approximates the tails of s:

$$
\begin{aligned}
\mathsf{splitAt0}\, \varnothing &\doteq \varnothing \\
\mathsf{splitAt0}\, \varepsilon &\doteq \varnothing \\
\mathsf{splitAt0}\, C &\doteq \{\langle C, \varepsilon\rangle\} \\
\mathsf{splitAt0}(r_1 \mathbin{+\!\!+} r_2) &\doteq \{\langle C, r' \mathbin{+\!\!+} r_2\rangle \mid \langle C, r'\rangle \in \mathsf{splitAt0}\, r_1\} \cup (\textbf{if } \nu(r_1) \textbf{ then } \mathsf{splitAt0}\, r_2 \textbf{ else } \varnothing) \\
\mathsf{splitAt0}(r_1 \cup r_2) &\doteq \mathsf{splitAt0}\, r_1 \cup \mathsf{splitAt0}\, r_2 \\
\mathsf{splitAt0}(r^*) &\doteq \{\langle C, r' \mathbin{+\!\!+} r^*\rangle \mid \langle C, r'\rangle \in \mathsf{splitAt0}\, r\}
\end{aligned}
$$

We extend this basic operation to what we need:

$$
\begin{aligned}
\mathsf{splitAt}_0\, r &\doteq \{\langle \varepsilon, C, r'\rangle \mid \langle C, r'\rangle \in \mathsf{splitAt0}\, r_1\} \\
\mathsf{splitAt}_n\, r &\doteq \{\langle C' \mathbin{+\!\!+} r_1, C, r_2\rangle \mid \langle C', r'\rangle \in \mathsf{splitAt0}\, r_1, \langle r_1, C, r_2\rangle \in \mathsf{splitAt}_{n-1}\, r'\}
\end{aligned}
$$

Lemma 5.2. *Let $s \in r$. If $s[i] = \sigma$, then there exists $\langle r_1, C, r_2\rangle \in \mathsf{splitAt}_i\, r$ such that $\sigma \in C$, $s[: i] \in r_1$ and $s[i + 1 :] \in r_2$.*

We define $\mathsf{resByCharAt}_{i,C}\, r$ for restricting r by $\lambda s, s[i] \in C$:

$$
\mathsf{resByCharAt}_{i,C}\, r \doteq \bigcup\{r_1 \mathbin{+\!\!+} C'' \mathbin{+\!\!+} r_2 \mid \langle r_1, C, r_2\rangle \in \mathsf{splitAt}_i\, r, C'' \doteq C' \cap C, C'' \not\doteq \varnothing\}
$$

We present narrowing rules where the equality/inequality is converted to a charset, and consider cases of right-to-left indices:

$$
r \xhookrightarrow{\lambda s, s[i]=\sigma} \mathsf{resByCharAt}_{i,\{\sigma\}}\, r \qquad\qquad r \xhookrightarrow{\lambda s, s[i]\neq\sigma} \mathsf{resByCharAt}_{i,\Sigma\setminus\{\sigma\}}\, r
$$

$$
r \xhookrightarrow{\lambda s, s[|s|-i]=\sigma} \boxed{\mathsf{resByCharAt}_{i-1,\{\sigma\}}\, \boxed{r}^{-1}}^{-1} \qquad r \xhookrightarrow{\lambda s, s[|s|-i]\neq\sigma} \boxed{\mathsf{resByCharAt}_{i-1,\Sigma\setminus\{\sigma\}}\, \boxed{r}^{-1}}^{-1}
$$

Char occurrence test. Finally, we present narrowing rules on predicates that tell whether a specific character occurs in a string, based on operations we have defined in §4:

$$
r \xhookrightarrow{\lambda s, \sigma \,\mathbf{in}\, s} \mathsf{findPrefix}_\sigma\, r \mathbin{+\!\!+} \mathsf{findSuffix}_\sigma\, r \qquad\qquad r \xhookrightarrow{\lambda s, \neg(\sigma \,\mathbf{in}\, s)} r \setminus \{\sigma\}
$$

Remarks. Let $r \overset{p}{\hookrightarrow} r'$ be a sound narrowing rule. The narrowed type r' is an overapproximation of the intersection of r and the language implied by p, *i.e.*, $r' \supseteq r \cap \{s : \mathbb{S} \mid p(s)\}$. Computing language intersection is in general exponential. For specific predicates, our type narrowing rules provide imprecise but cheap ways to compute intersection.

Type narrowing also has the potential to simplify refinement types in our framework. In principle we only allow string types to be refined by regexes. But since all type annotations will be translated into VCs, it is possible to extend our frontend to accept types refined by SMT logical formulas (as in Liquid Typing). For example, the refinement of the type $\{s : \mathbb{S} \mid s \in$ `r'a*'` $\wedge |s| > 0\}$ (or, $\{s :$ `r'a*'` $\mid |s| > 0\}$) is translated as $s \in$ `r'a*'` $\wedge |s| > 0$. By type narrowing, `r'a*'` is narrowed down to `r'a+'` by the predicate $\lambda s, |s| > 0$.

6 Rocq Mechanization

In the Rocq interactive theorem prover, first, we define strings with their operations, regexes with their operations, abstract and narrowing operations presented in §3, §4, and §5. In particular, we define string operations in terms of basic list operations, and prove they are consistent with the usual semantics. Then, we prove the lemmas presented in those sections, from which we finally show the soundness of our type inference and narrowing rules, in a *shallow-embedded* manner. For instance, to show the soundness of the rule τ-concat:

$$(\Gamma \Rightarrow e_1 \in r_1) \wedge (\Gamma \Rightarrow e_2 \in r_2) \Rightarrow (\Gamma \Rightarrow e_1 +\!\!+ e_2 \in r_1 +\!\!+ r_2),$$

We show the following theorem:

$$s_1 \in r_1 \Rightarrow s_2 \in r_2 \Rightarrow s_1 +\!\!+ s_2 \in r_1 +\!\!+ r_2,$$

where we shallow-embed core terms as string operations and elide the proof context Γ in a logically equivalent way.

7 Evaluation

We have implemented FLAT-Checker with a mixture of Python 3.13 and Scala 3.6. We rely on Python's builtin `ast` module to parse Python sources. We implement annotation processing, VC generation, type inference, and type narrowing in Scala. We use cvc5 (version 1.3.0) [Barbosa et al. 2022] as the backend SMT solver and invoke it through its Java bindings. Our prototype implementation is available at (blinded for review): https://anonymous.4open.science/r/flat-checker/. This repository also includes a full replication package for our evaluation.

7.1 Experimental Setting

We evaluated FLAT-Checker on the benchmark dataset [Schröder and Cito 2025a] of the Panini [Schröder and Cito 2025b] grammar inference system. This dataset contains 204 ad hoc parsers written in Python, together with *golden grammars* (*i.e.,* ground truth) in the form of POSIX regular expressions describing the exact input that these parsers accept. The parsers in the dataset are structured into three categories based on the structural features of the Python programs: *Straight-Line Programs* exhibit only linear execution flow without branching; *Programs with Conditionals* incorporate conditional statements to make decisions based in input characteristics; and *Programs with Loops* contain iterative constructs alongside conditional statements. We have augmented the parsers in this dataset by adding loop invariant annotations where necessary (cf. §2.3).

To be able to compare FLAT-Checker with traditional SMT and string constraint solvers, we extracted the verification conditions generated by FLAT-Checker into standalone SMT-LIB queries. We compared FLAT-Checker with cvc5 (version 1.3.0) [Barbosa et al. 2022], Z3 (version 4.15.1) [de Moura and Bjørner 2008], Z3-Noodler-Pos [Chen et al. 2025], and OSTRICH2 [Hague et al. 2025]. All experiments were run on a 3 GHz 6-Core Intel iMac, with 24 GB RAM. All solvers were given a 60 second timeout and no memory limit.

7.2 Performance Results

The results of evaluating FLAT-Checker on the Panini benchmark dataset are shown in Table 2. FLAT-Checker was able to verify all 204 ad hoc parsers against their golden grammars, i.e., type-checked each parser program with its golden grammar regex as the input type. The average time for type checking a program was about 1 second end-to-end wall-clock time, including all overheads (such as parsing the input program and calling an external SMT solver, if necessary). Of the 966 verification conditions generated during type checking (across all programs), almost

Table 2. Results of evaluating FLAT-Checker on the Panini benchmark dataset.

			Discharged VCs			
	Programs	Invariants	Trivial	SMT	Lemmas	Time (s)
Straight-Line	89	0	0	21	229	0.78 ±0.05
+ Conditionals	51	0	11	55	187	0.82 ±0.06
+ Loops	64	45	45	227	191	1.43 ±4.19
	204	45	56	303	607	1.00 ±2.36

Table 3. Comparison of FLAT-Checker with SMT/String solvers. We report the average verification time for VCs *without timeouts*, as well as the total time spent verifying all programs *with timeouts*.

	VCs				Programs	
	Valid	Invalid	Timeout	Time (ms)	Verified	Time
FLAT-Checker	966	0	0	53.09 ±814.02	203	51.29 s
OSTRICH2	961	0	5	1107.21 ±1394.51	200	22.76 min
Z3-Noodler-Pos	957	8	1	28.03 ±27.25	198	1.45 min
cvc5	910	0	56	13.17 ±24.74	159	56.21 min
Z3	866	0	100	11.34 ±4.88	152	1.67 h

two thirds (607) were complex enough to require lemma synthesis; the rest could be discharged without lemmas (56 trivial cases and 303 solvable using an off-the-shelf SMT solver).

Of the 204 programs in the dataset, 24 programs required loop invariants. On average, we need two annotations for each of these programs, resulting in 45 user-provided invariant annotations in total. A typical invariant is `inv(0 <= i <= len(s))` and most invariants take the form $c_1 \leq i \leq |s|+c_2$. We hope to reduce this annotation burden further by automatically inferring more of these kinds of loop invariants in the future.

7.3 Comparison with SMT Solvers

Table 3 compares FLAT-Checker with state-of-the-art SMT and string constraint solvers. For an SMT solver to verify a parser program, it has to validate all verification conditions extracted from that program. In total, there are 966 VCs for 203 of the benchmark programs; one program is trivial enough to not yield any VCs.

Only FLAT-Checker is able to validate all VCs and verify all programs in the Panini benchmark dataset. It does so in under a minute of pure verification time.

The traditional solvers Z3 and cvc5, while very fast at discharging most VCs, time out for many cases involving simple regular string constraints similar to the VC φ_3 discussed in §2.1. Z3 is only able to validate 866 VCs (out of 966) and verify 152 programs (out of 203); due to timeouts, this takes more than one and a half hours. cvc5 is able to validate 910 VCs, timing out half as much as Z3, but this also only verifies 159 programs, taking almost one hour.

Z3-Noodler-Pos is a very recent string solver that explicitly aims to address the issues surrounding *position string constraints*, which include problems such as our `r'a*'` example. It does fairly well on the Panini benchmark, with only one timeout and otherwise very fast verification times. However, we encountered some soundness issues, where Z3-Noodler-Pos incorrectly determined

8 VCs to be invalid (reporting SAT when they should be UNSAT, with a clearly incorrect model). Thus, Z3-Noodler-Pos is only able to correctly verify 198 out of the 203 parser programs.

OSTRICH2, the latest evolution of the OSTRICH string solver [Chen et al. 2019], can verify almost all programs in the dataset, but is by far the slowest of all the solvers. While it only exhibited 5 timeouts—successfully validating 961 VCs and verifying 200 programs—it took over 20 minutes to verify all programs. OSTRICH2 takes the longest amount of time to validate any single VC and is more than 20× slower than FLAT-Checker.

Unsoundness in Z3-Noodler-Pos. We have unconvered 8 instances of unsound behaviour in Z3-Noodler-Pos (see our anonymous repository for more details). For example, the rather simple verification condition

$$\varphi_{161.1} : s \in \texttt{r'[\^{}a]?'} \Rightarrow (s = \texttt{''} \vee s \neq \texttt{'a'}) \wedge |s| \leq 1$$

is rejected with the spurious counterexample $s = \texttt{'a'}$. All other solvers are able to correctly validate this VC, including Z3. We do not know why Z3-Noodler-Pos behaves in this way, as there is no apparent pattern to the VCs for which it produces spurious counterexamples.

8 Related Work

8.1 Abstract Interpretation of strings

Various abstract domains have been invented to perform static analysis of strings. Some capture partial information of strings such as the set of the characters that must/may occur, the prefixes/suffixes, the length intervals, and the hash values in simple domains: they are used for great efficiency and scalability [Amadini et al. 2017b; Costantini et al. 2011]. Some capture more structural information through regex-like domains, such as *bricks* [Costantini et al. 2011] and *dashed strings* [Amadini et al. 2018a,b, 2017a], but they are still fragments of regexes. The most precise domain is the entire class of regular languages: Arceri et al. [2020] approximate strings as finite state automata over an alphabet of characters; Negrini et al. [2021] extend this idea to automata over an alphabet of strings.

Although automata and regexes are semantic equivalences, automata-based abstract interpretation is criticized [Søndergaard 2021] to have significant issues due to size explosion of automata minimization and other operations. We use regexes as our abstract domains and design a couple of efficient abstract semantics for string operations as the foundations of our type inference rules. Our abstract semantics for the substring operation is *more expressive* than Arceri et al. [2020] and Negrini et al. [2021]: we consider relative positions generated by find operations while they both do not.

To apply automata-based abstract interpretation, one needs to define the *widening* operator [Arceri et al. 2020; Bartzis and Bultan 2004; Choi et al. 2006; Negrini et al. 2021]. This is known to be a difficult art, especially for highly expressive domains like automata [Søndergaard 2021]. Fortunately, this hard problem is *irrelevant* to us as we rely on user-specified loop invariants as in the traditional Floyd–Hoare logic style.

Moreover, there are studies on how to combine a large number of domains via a *reference domain* as a medium for information exchange [Amadini et al. 2018c], and how to track relational information between string variables [Arceri et al. 2022].

8.2 String Types

More than a decade ago, the concept of *regular expression types* were invented in *text processing languages* [Hosoya et al. 2000; Tabuchi et al. 2002] and extended for context-free grammars [Thiemann 2005].

In recent years, researchers have noticed that the string type in many general-purpose programming languages is too coarse: it neglects the latent structure of strings. Kelly et al. [2019] define a special *SafeStrings* interface in TypeScript. Programmers subclass it to define new types for particular sets of strings with specific latent structure, such as XML and email addresses, expressed by grammars. A follow-up work, June [Bruce et al. 2023], enables automated test generation on top of *SafeStrings* via Java annotations, including built-in annotations that mark a string variable to be delimited strings, file paths, email addresses, dates, etc. Technically speaking, the *SafeStrings* types are not subtypes of the string type in the type systems of their host languages. A more recent framework FLAT [Zhu and Zeller 2025] tackles this problem through a refinement type system that is somewhat similar to Liquid Typing, but allows to refine string types by formal languages including regular and context-free languages, and logical predicates as well which indeed considers context-sensitive languages. However, all of these frameworks tests or validates string types at *runtime*. Our FLAT-Checker, based on the idea of FLAT, takes the first step towards *static* type checking at *compile time*.

8.3 Liquid Type Systems

Modern *Liquid Type* systems [Gamboa et al. 2023; Jhala 2014; Jhala and Vazou 2020; Lehmann et al. 2023; Vekris et al. 2016] enrich existing type systems with logical predicates that allows users to pose semantic constraints of values. They require the predicates to stay in *decidable* SMT fragments, such as linear arithmetic over integers, fixed-size bit vectors, quantifier-free equality with uninterpreted functions, etc., so that they can fully rely on mature SMT solvers to efficiently discharge VCs generated through type checking. Compare with dependent types, they require lighter manual annotations [Rondon et al. 2008]. As discussed in the end of §5, FLAT-Checker can be extended to support types refined by SMT formulae.

Liquid Type systems usually do not offer loop invariant annotations but rely on Horn clause constraint solving [Flanagan and Leino 2001] to synthesize them [Jhala 2014; Jhala and Vazou 2020; Lehmann et al. 2023]. Generally speaking, loop invariant synthesis is hard and when it comes to loops that manipulate strings, we have no idea if existing techniques generalize. Thus, for the maximal flexibility and expressiveness, we make the choice of asking users to specify loop invariants in FLAT-Checker, following the tradition of Floyd-Hoare logic-based program verification.

9 Conclusion and Future Work

This paper presents FLAT-Checker, the first static type checker for regular language types in Python. It can be used as a lightweight program verifier for ensuring correct contents of string types throughout the entire program code, notably detecting possible violations of string operations and potential attack surfaces such as SQL injections or other third-party string manipulations. As our evaluation results show, FLAT-Checker can effectively and efficiently type-check ad hoc parsers, surpassing state-of-the-art SMT string solvers in terms of provability and efficiency.

Our future work on FLAT-Checker will focus on extension and generalization:

More string operations. We want to support more string operations and more Python language features, so that we can apply FLAT-Checker to more complex string-manipulating programs, including programs that handle third-party inputs.

Context-free grammars. We want generalize the abstract operations shown in §4 from regexes to *context-free grammars*. With that, FLAT-Checker will be able to check (and infer) *context-free language types*.

Our FLAT-Checker prototype and all experimental data are available as open source at

https://anonymous.4open.science/r/flat-checker/.

References

Roberto Amadini, Graeme Gange, François Gauthier, Alexander Jordan, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. 2018c. Reference Abstract Domains and Applications to String Analysis. *Fundam. Informaticae* 158, 4 (2018), 297–326. doi:10.3233/FI-2018-1650

Roberto Amadini, Graeme Gange, and Peter J. Stuckey. 2018a. Propagating lex, find and replace with Dashed Strings. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10848)*, Willem Jan van Hoeve (Ed.). Springer, 18–34. doi:10.1007/978-3-319-93031-2_2

Roberto Amadini, Graeme Gange, and Peter J. Stuckey. 2018b. Propagating Regular Membership with Dashed Strings. In *Principles and Practice of Constraint Programming - 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11008)*, John N. Hooker (Ed.). Springer, 13–29. doi:10.1007/978-3-319-98334-9_2

Roberto Amadini, Graeme Gange, Peter J. Stuckey, and Guido Tack. 2017a. A Novel Approach to String Constraint Solving. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10416)*, J. Christopher Beck (Ed.). Springer, 3–20. doi:10.1007/978-3-319-66158-2_1

Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald Søndergaard, Peter J. Stuckey, and Chenyi Zhang. 2017b. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 10205)*, Axel Legay and Tiziana Margaria (Eds.). 41–57. doi:10.1007/978-3-662-54577-5_3

Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. 2020. Static Analysis for ECMAScript String Manipulation Programs. *Applied Sciences* 10, 10 (2020). doi:10.3390/app10103525

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 (Lecture Notes in Computer Science, Vol. 13182)*, Bernd Finkbeiner and Thomas Wies (Eds.). Springer, 20–42. doi:10.1007/978-3-030-94583-1_2

Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

Constantinos Bartzis and Tevfik Bultan. 2004. Widening Arithmetic Automata. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3114)*, Rajeev Alur and Doron A. Peled (Eds.). Springer, 321–333. doi:10.1007/978-3-540-27813-9_25

Dan Bruce, David Kelly, Héctor D. Menéndez, Earl T. Barr, and David Clark. 2023. June: A Type Testability Transformation for Improved ATG Performance. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023*, René Just and Gordon Fraser (Eds.). ACM, 274–284. doi:10.1145/3597926.3598055

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

Taolue Chen, Matthew Hague, Anthony W. Lin, Philipp Rümmer, and Zhilin Wu. 2019. Decision procedures for path feasibility of string-manipulating programs with complex operations. *Proc. ACM Program. Lang.* 3, POPL (2019), 49:1–49:30. doi:10.1145/3290362

Yu-Fang Chen, David Chocholatý, Vojtech Havlena, Lukás Holík, Ondrej Lengál, and Juraj Síc. 2024. Z3-Noodler: An Automata-based String Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 14570)*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer, 24–33. doi:10.1007/978-3-031-57246-3_2

Yu-Fang Chen, Vojtěch Havlena, Michal Hečko, Lukáš Holík, and Ondřej Lengál. 2025. A Uniform Framework for Handling Position Constraints in String Solving. *Proc. ACM Program. Lang.* 9, PLDI, Article 169 (June 2025), 26 pages. doi:10.1145/3729273

Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. 2006. A Practical String Analyzer by the Widening Approach. In *Programming Languages and Systems, 4th Asian Symposium, APLAS 2006, Sydney, Australia, November 8-10, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4279)*, Naoki Kobayashi (Ed.). Springer, 374–388. doi:10.1007/

11924661_23

Giulia Costantini, Pietro Ferrara, and Agostino Cortesi. 2011. Static Analysis of String Values. In *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6991)*, Shengchao Qin and Zongyan Qiu (Eds.). Springer, 505–521. doi:10.1007/978-3-642-24559-6_34

Patrick Cousot and Radhia Cousot. 1976. Static determination of dynamic properties of programs. (01 1976).

Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. doi:10.1007/978-3-540-78800-3_24

Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2021)*, José Nuno Oliveira and Pamela Zave (Eds.). Springer, 500–517. doi:10.1007/3-540-45251-6_29

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. doi:10.1145/113445.113468

Catarina Gamboa, Paulo Canelas, Christopher Steven Timperley, and Alcides Fonseca. 2023. Usability-Oriented Design of Liquid Types for Java. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1520–1532. doi:10.1109/ICSE48619.2023.00132

Matthew Hague, Denghang Hu, Artur Jeż, Anthony W. Lin, Oliver Markgraf, Philipp Rümmer, and Zhilin Wu. 2025. OSTRICH2: Solver for Complex String Constraints. arXiv:2506.14363 [cs.LO] https://arxiv.org/abs/2506.14363

Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. 2000. Regular expression types for XML. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 11–22. doi:10.1145/351240.351242

Ranjit Jhala. 2014. Refinement types for Haskell. In *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL '14*, Nils Anders Danielsson and Bart Jacobs (Eds.). ACM, 27–28. doi:10.1145/2541568.2541569

Ranjit Jhala and Niki Vazou. 2020. Refinement Types: A Tutorial. *CoRR* abs/2010.07763 (2020). arXiv:2010.07763 https://arxiv.org/abs/2010.07763

Matthias Keil and Peter Thiemann. 2014. Symbolic Solving of Extended Regular Expression Inequalities. In *34th International Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2014, December 15-17, 2014, New Delhi, India (LIPIcs, Vol. 29)*, Venkatesh Raman and S. P. Suresh (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 175–186. doi:10.4230/LIPICS.FSTTCS.2014.175

David Kelly, Mark Marron, David Clark, and Earl T. Barr. 2019. SafeStrings: Representing Strings as Structured Data. *CoRR* abs/1904.11254 (2019). arXiv:1904.11254 http://arxiv.org/abs/1904.11254

Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI (2023), 1533–1557. doi:10.1145/3591283

Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. 2021. Twinning Automata and Regular Expressions for String Static Analysis. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12597)*, Fritz Henglein, Sharon Shoham, and Yakir Vizel (Eds.). Springer, 267–290. doi:10.1007/978-3-030-67067-2_13

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2025. *Logical Foundations*. Software Foundations, Vol. 1. Electronic textbook. http://softwarefoundations.cis.upenn.edu Version 6.7.

Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. 2008. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 159–169. doi:10.1145/1375581.1375602

Michael Schröder and Jürgen Cito. 2025a. *Artifact for 'Static Inference of Regular Grammars for Ad Hoc Parsers'*. doi:10.5281/zenodo.15732005

Michael Schröder and Jürgen Cito. 2025b. Static Inference of Regular Grammars for Ad Hoc Parsers. *Proc. ACM Program. Lang.* 10, OOPSLA2 (Oct. 2025), to appear. https://mcschroeder.github.io/files/panini_preprint.pdf

Harald Søndergaard. 2021. String Abstract Domains and Their Combination. In *Logic-Based Program Synthesis and Transformation - 31st International Symposium, LOPSTR 2021, Tallinn, Estonia, September 7-8, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13290)*, Emanuele De Angelis and Wim Vanhoof (Eds.). Springer, 1–15. doi:10.1007/978-3-030-98869-2_1

Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. 2002. Regular Expression Types for Strings in a Text Processing Language. In *International Workshop in Types in Programming, TIP@MPC 2002, Dagstuhl, Germany, July 8, 2002 (Electronic Notes in Theoretical Computer Science, Vol. 75)*, Gilles Barthe and Peter Thiemann (Eds.). Elsevier, 95–113. doi:10.1016/S1571-0661(04)80781-3

Peter Thiemann. 2005. Grammar-based analysis of string expressions. In *Proceedings of TLDI'05: 2005 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Long Beach, CA, USA, January 10, 2005*, J. Gregory Morrisett and Manuel Fähndrich (Eds.). ACM, 59–70. doi:10.1145/1040294.1040300

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement Types for TypeScript. *CoRR* abs/1604.02480 (2016). arXiv:1604.02480 http://arxiv.org/abs/1604.02480

Fengmin Zhu and Andreas Zeller. 2025. FLAT: Formal Languages as Types. arXiv:2501.11501 [cs.SE] https://arxiv.org/abs/2501.11501