Static Grammar Inference for Ad Hoc Parsers

ANONYMOUS AUTHOR(S)

Parsing, the process of structuring a linear representation according to a given grammar, is a fundamental activity in software engineering. While formal language theory has provided theoretical foundations for parsing, the most common kind of parsers used in practice are written *ad hoc*. They use common string operations for parsing, without explicitly defining an input grammar. These ad hoc parsers are often intertwined with application logic and can result in subtle semantic bugs. Grammars, which are complete formal descriptions of input languages, can enhance program comprehension, facilitate testing and debugging, and provide formal guarantees for parsing code. Writing grammars, however, can be tedious and error-prone. Inspired by the success of type inference in programming languages, we propose a general approach for static inference of input string grammars from unannotated ad hoc parser source code. We approach this problem as an extension of refinement typing, synthesizing logical and string constraints that represent parsing operations, then simplifying and abstracting them into formal grammars. Our contributions include a core calculus λ_Σ for representing ad hoc parsers, a method for solving refinement variables that represent unknown input string constraints, an abstract interpretation framework for (in)equality predicates over string variables, and a set of abstract domains for efficient representation of the numerical and string values encountered during this process. We implement our approach in the Panini system and demonstrate its effectiveness in principle.

1 INTRODUCTION

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

Ad hoc parsers are pieces of code that use combinations of common string operations like `slice`, `index`, or `trim` to effectively perform parsing. A programmer manipulating strings in an ad hoc fashion would probably not even think about the fact that they are actually writing a parser. These string-manipulating programs can be found in functions handling command-line arguments, reading configuration files, or as part of any number of minor programming tasks involving strings, often deeply entangled with application logic—a phenomenon known as *shotgun parsing* [Momot et al. 2016a]. They have also been shown to produce subtle and difficult to identify semantic bugs [Eghbali and Pradel 2020; Kapugama et al. 2022].

A *grammar* is a complete formal description of all values an input string may assume. It can elucidate the corresponding parsing code, revealing otherwise hidden features and potentially subtle bugs or security issues. By focusing on *data* rather than code, grammars provide a high-level perspective, allowing programmers to grasp an input language directly, without being distracted by the mechanics of the parsing process and the intricacies of imperative string manipulation. Augmenting regular documentation with formal grammars can increase program comprehension by providing alternative representations for a programming task [Fitter and Green 1979; Gilmore and Green 1984]. Because a grammar is also a *generating device*, it is possible to construct any sentence of its language in a finite number of steps—manually or in an automated fashion. Being able to reliably generate concrete examples of possible inputs is invaluable during testing and debugging. But despite providing all these benefits, hardly anyone ever bothers to write down a

2018-2475-1421/2018/1-ART1 $15.00
https://doi.org/

grammar, even for more complex ad hoc parsers. Grammars share the same fate as most other forms of specification: they are tedious to write, hard to get right, and seem hardly worth the trouble—especially for such small pieces of code like ad hoc parsers.

But there is a form of specification, one wildly more successful than grammars, that we can draw inspiration from: *types*. Formal grammars are similar to types, in that an ad hoc parser without a grammar is very much like a function without a type signature—it might still work, but you will not have any guarantees about it before actually running the program. Types have one significant advantage over grammars, however: most type systems offer a form of *type inference*, allowing programmers to omit type annotations because they can be automatically recovered from the surrounding context [MacQueen et al. 2020, § 4]. If we could infer grammars like we can infer types, we would reap all the rewards of having a complete specification of our program's input language.

We present a general approach for static inference of string grammars from unannotated ad hoc parser source code by posing the problem of grammar inference as inferring constraints in a refinement type system. As part of type inference, we first synthesize logical constraints that declaratively represent the parsing operations performed on the input string. We then simplify this complex first-order verification condition into a minimal quantifier-free disjunctive normal form by systematically applying semantics-preserving transformations and abstract interpretations that ultimately lead to a logically equivalent minimal string constraint representing a formal grammar.

The contributions of our work are:

- A core calculus λ_Σ for representing ad hoc parsers that allows for complete refinement type inference given a library of string function specifications.
- A method for solving unknown refinement variables denoting input string refinements, a.k.a. *grammar variables*, by minimizing implication consequents.
- An abstract interpretation framework for (in)equality predicates over string variables.
- A set of abstract domains for exactly and efficiently representing infinite sets of integer and string values for the purposes of grammar inference.
- An implementation of our approach in the Panini system.

2 OVERVIEW

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

The Panini system can be separated into a front- and a back-end. In the front-end, ad hoc parsers written in a general-purpose programming language, e.g., Python, are translated into λ_Σ programs. In the back-end, those λ_Σ programs are statically analyzed and their input string constraints extracted. *This paper is about the back-end.* In short order, we will describe the λ_Σ calculus, its refinement type system, our string constraint solving algorithm, and the underlying abstract domains. To situate this work, we first want to briefly sketch the front-end process before giving a more detailed overview of the main contributions of this paper.

2.1 The Front End: From Source to λ_Σ

Slicing. To translate an ad hoc parser to a λ_Σ program from its original source code, we first have to locate it. Due to the nature of ad hoc parsers, we have to assume that they are entangled with other, non-parsing related code and might be spread throughout a program in a shotgun manner [Momot

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

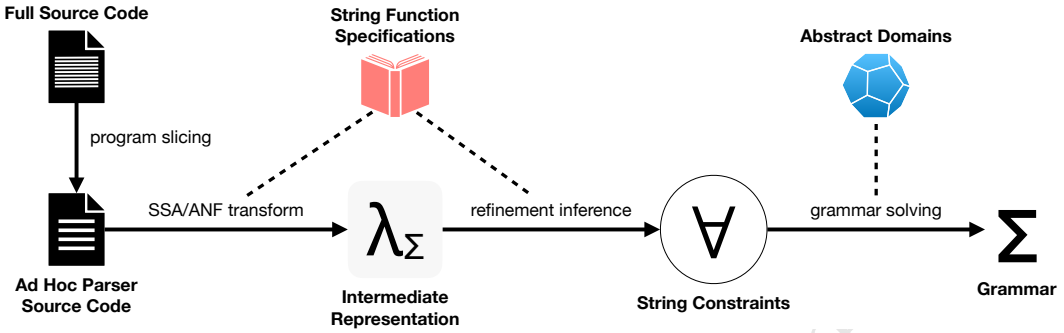


Fig. 1. The complete Panini system.

et al. 2016b; Underwood and Locasto 2016]. Locating and extracting ad hoc parser cores from a full program can either be done manually, by the user of the system annotating or highlighting pieces of parser code they want analyzed, or ideally in an automatic fashion, using program slicing techniques [Weiser 1984].

Translation. Once located, the ad hoc parser is transformed into static single assignment (SSA) form [Braun et al. 2013] and then, via an SSA-to-ANF transformation [Chakravarty et al. 2004], into a Panini program. It is important that precise source location information is preserved throughout this process, in order to enable the full range of envisioned applications.

Axioms. It is necessary to have a library of string function specifications that map the source language's string operations to equivalent λ_Σ functions. Note that it is not necessary to have actual λ_Σ *implementations* of these functions. We only need axiomatic specifications—in the form of type signatures—that capture those properties of the original functions that are necessary to ultimately synthesize precise string grammars. Below are some examples of such axioms, based on the semantics of certain equivalent Python functions. We will use these function signatures in code examples throughout this paper:

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

Axioms such as these most likely have to be manually defined, based on close reading of the source functions' implementations and documentation, but efforts could be made to establish trust in their correctness through automated means. For example, *dynamic* input grammar mining [Gopinath et al. 2020] could be used to generate partial or initial specifications, and parser-directed fuzzing techniques [Mathis et al. 2019] could demonstrate alignment of specification and implementation.

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

2.2 The Back End: From λ_Σ to Grammar

Refinement Types. λ_Σ is a simple λ-calculus with a refinement type system in the style of *Liquid Types* [Rondon et al. 2008; Vazou et al. 2014]. Refinement types allow us to extend base types with logical constraints. This is useful to precisely describe subsets of values, as well as track complex relationships between values, all on the type level. For example, the type of natural numbers can be defined as a subset of the integers,

$$\mathbb{N} \stackrel{\text{def}}{=} \{\nu : \mathbb{Z} \mid \nu \geq 0\},$$

and we can give a precise definition of the length function on strings using a dependent function type and the string length operator $|\square|$ of the refinement logic:

$$\text{length} \ : \ (s : \mathbb{S}) \to \{\nu : \mathbb{N} \mid \nu = |s|\}.$$

Verification Conditions. A particular aspect of refinement type systems is the generation of verification conditions (VCs) [Nelson 1980]. These are constraints in the refinement logic whose validity entails the correctness of the program's inferred or given types. For example, in order to check whether $\{x : \mathbb{Z} \mid x > 42\}$ (the type of all numbers greater than forty-two) is a subtype of $\mathbb{N}$, the constraint $\forall x.\ x > 42 \Rightarrow x \geq 0$ has to be verified. For verification to remain practical, the refinement logic is typically chosen to allow *satisfiability modulo theories* (SMT) [Barrett et al. 2021], which means VCs can be discharged using an off-the-shelf constraint solver such as Z3 [De Moura and Bjørner 2008]. Our system uses quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) [Barrett et al. 2016] for its refinement predicates, extended with a theory of operations over strings [Berzish et al. 2017].

Type Inference. Our goal is to infer ad hoc parsers' implicit input string constraints, i.e., their grammars. For an ad hoc parser expressed as a λ_Σ program, this essentially means inferring the most precise refinement type for the program's top-level input string. In order to infer the most precise refinement for any type, one must find a predicate that expresses the relationship of that type to all other (type) variables in scope. To facilitate this, refinement type systems typically first infer the basic shapes of all types in the program, using standard type inference à la Hindley-Damas-Milner [Damas and Milner 1982; Hindley 1969], with placeholder variables standing in for as-yet-unknown concrete refinement predicates. These placeholder variables—variously called "κ variables" [Cosman and Jhala 2017], "Horn variables" [Jhala and Vazou 2020], or "liquid type variables" [Rondon et al. 2008] in the literature—are also present in the VCs at this point and prevent them from being discharged. Our system needs to find the strongest satisfying assignments for all placeholder variables given the overall constraints expressed in the VCs. To do this efficiently, it uses the state-of-the-art Fusion algorithm by Cosman and Jhala [2017], which can find the strongest solutions for (almost) all κ variables in most programs and enables inference of the most precise local refinement type for all program statements without requiring any prior type annotations, except for library functions. Another advantage of the Fusion approach is the preservation of scoping structure, yielding VCs that more closely match the original program structurally. Unfortunately, Fusion falls short when it comes to κ variables denoting parser inputs—the very ones we are the most interested in! To infer suitably precise grammar-like string constraints for these *grammar variables*, we need a different approach.

Grammar Variables. Grammar variables are those κ variables that are placeholders for input string refinements. A concrete assignment for a grammar variable is a (finite) predicate that describes all possible values the input string can have without the program going wrong.

To illustrate, consider the following simple Python expression:

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

Assuming the function specifications from § 2.1, we can transform this expression to the following equivalent λ_Σ program (on the left) and infer its top-level type to be $\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}$ with an incomplete VC (on the right) that closely matches the program:

$$\begin{array}{ll} \boldsymbol{\lambda}(s : \mathbb{S}). & \forall(s : \mathbb{S}).\ \kappa(s) \Rightarrow \\ \quad \textbf{let } x = \text{charAt } s\ 0 \textbf{ in} & \quad 0 < |s| \wedge \forall(x : \mathbb{Z}).\ x = s[0] \Rightarrow \\ \quad \textbf{let } p = \text{match } x \text{ “a”} \textbf{ in} & \quad\quad \forall(p : \mathbb{B}).\ (p \Leftrightarrow x = \text{“a”}) \Rightarrow \\ \quad \text{assert } p & \quad\quad\quad p \end{array}$$

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

Grammar Solving. The key insight that allows us to find suitable assignments for grammar variables in a general manner is that the top-level VC for a parser will always be of the form $\forall s.\ \kappa(s) \Rightarrow \varphi$, where s is the input string and φ is a constraint that precisely captures all parsing operations the program performs on s. By simply taking $\kappa(s) \mapsto \varphi$, the VC becomes a trivially valid tautology and the string refinement captures exactly those inputs that the parser accepts. But clearly this solution is practically useless: we want a succinct predicate in a grammar-like form, but the top-level VC consequent φ is a complex term in first-order logic that is basically identical to the program code itself; it does not lead to any further insight about the input string's language. However, we can take φ as a starting point and simplify it into a minimal quantifier-reduced disjunctive normal form (DNF) by systematically applying truth-preserving transformations, ultimately arriving at a predicate that is logically equivalent but syntactically smaller. The details of this procedure are given in § 4 and involve a method of *abstract interpretation* of first-order constraints over string variables, based on abstract domains defined in § 5.

Example. Figure 2 shows all the steps of our approach on an example parser, including the intermediate representations synthesized and transformed along the way. First, Python source code ① is transformed to the λ_Σ intermediate language ②; then, refinement inference synthesizes a string constraint ③, shown here as an abstract syntax tree, which is rewritten and partially abstractly interpreted into a minimal DNF string constraint ④. Finally, the minimal constraint is further abstracted into a proper string grammar ⑤.

3 REFINEMENT INFERENCE

We now give a formalization of λ_Σ, our core abstraction for representing ad hoc parsers. The language and its type system were heavily inspired by the Sprite tutorial language by Jhala and Vazou [2020], and incorporate ideas from various other refinement type systems [Cosman and Jhala 2017; Dunfield and Krishnaswami 2021; Montenegro et al. 2020]. Our main contribution in

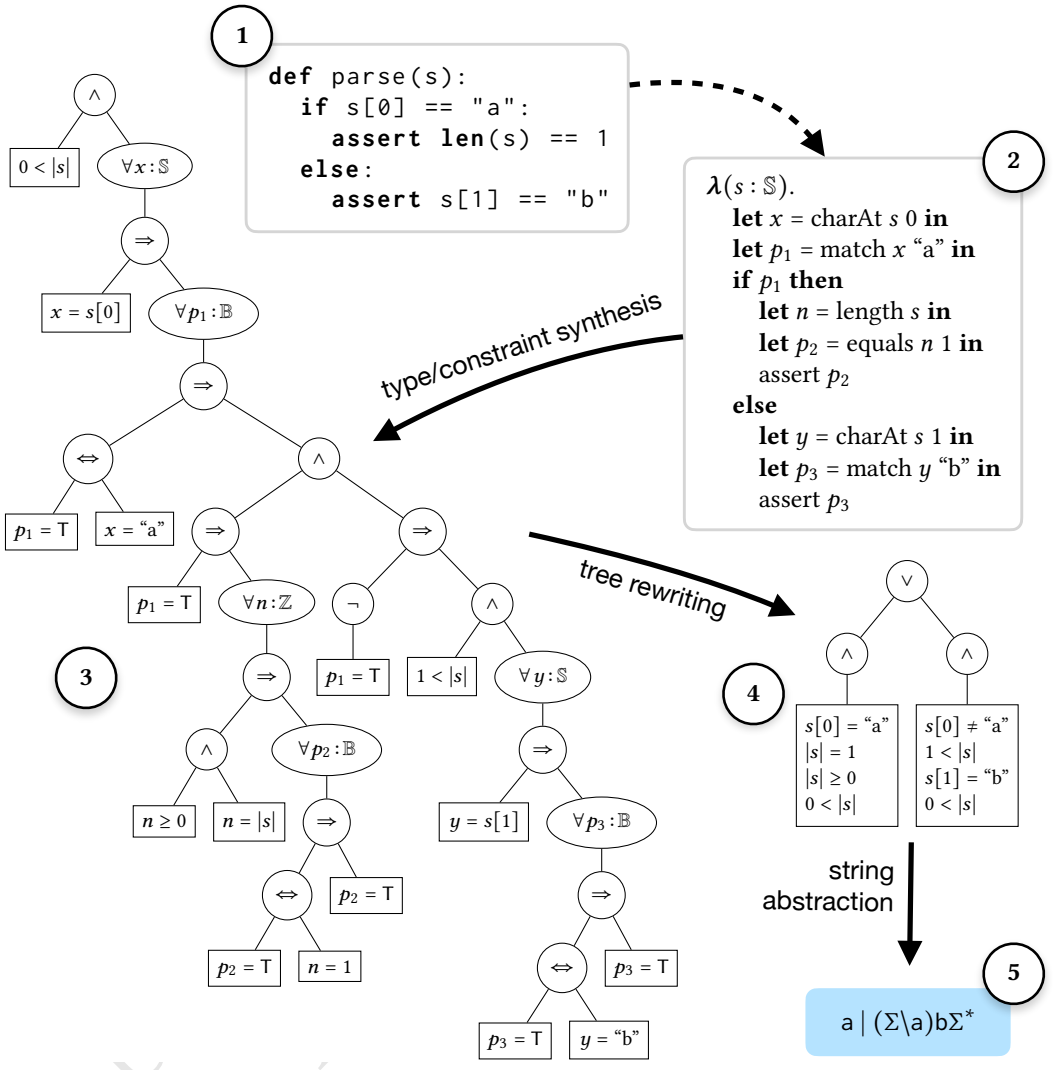


Fig. 2. An example of our approach, showing Python source code ①, the λ_Σ intermediate representation ②, a string constraint tree ③, its minimized DNF version ④, and the final inferred grammar ⑤.

this area is an extended refinement variable solving procedure that synthesizes precise grammars for input string constraints (§§ 3.3 and 4).

3.1 Syntax

λ_Σ is a small λ-calculus in A-normal form (ANF) [Bowman 2022; Flanagan et al. 1993]. It exists solely for type synthesis and its programs are neither meant to be executed nor written by hand. Its syntax, collected in Figure 3, is thus minimal and has few affordances.

Values are either variables or primitive constants. **Terms** are comprised of values and the usual constructs: function applications, function abstractions, bindings, recursive bindings, and branches. Applications are in ANF as a natural result of the SSA translation performed on the original source

Values	v	::=	unit	unit
		|	$\mathsf{T} \mid \mathsf{F}$	Booleans
		|	$0, -1, 1, \ldots$	integers
		|	"…"	strings
		|	$x, y, z, \ldots$	variables
Terms	e	::=	v	value
		|	$e\ v$	application
		|	$\boldsymbol{\lambda}(x : b).\ e$	abstraction
		|	**let** $x = e_1$ **in** e_2	binding
		|	**rec** $x : t = e_1$ **in** e_2	recursion
		|	**if** v **then** e_1 **else** e_2	branch
Base Types	b	::=	$\mathbb{1} \mid \mathbb{B} \mid \mathbb{Z} \mid \mathbb{S}$	unit, Boolean, integer, string
Types	t	::=	$\{x : b \mid p\}$	refined base
		|	$(x : t_1) \to t_2$	dependent function
Predicates	p	::=	$\mathsf{T} \mid \mathsf{F}$	Booleans
		|	$p_1 \wedge p_2 \mid p_1 \vee p_2$	connectives
		|	$p_1 \Rightarrow p_2 \mid p_1 \Leftrightarrow p_2$	implications
		|	$\neg p$	negation
		|	$w_1 = w_2 \mid w_1 < w_2 \mid \ldots$	(in)equality
		|	$v \in \mathrm{RE}$	regular language membership
		|	$\kappa(\overline{v})$	κ application
		|	$\exists(x : b).\ p$	existential quantification
Pred. Expressions	w	::=	v	value
		|	$w_1 + w_2 \mid w_1 - w_2 \mid \ldots$	integer arithmetic
		|	$\lvert w \rvert \mid w_1[w_2] \mid \ldots$	string functions
		|	$f(\overline{w})$	uninterpreted function
Constraints	c	::=	p	predicate
		|	$c_1 \wedge c_2$	conjunction
		|	$\forall(x : b).\ p \Rightarrow c$	universal implication

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

code (see § 2.1), but we also generally enforce this in the syntax to simplify the typing rules (§ 3.2). λ_Σ terms have no type annotations, except for $\boldsymbol{\lambda}$-binders and recursive **rec**-binders, whose (base) types we assume are inferred in the pre-processing phase or provided by the programmer.

Types are formed by decorating **Base Types** with refinements, or by constructing (dependent) function types, whose output types can refer to input types. For notational convenience, we also define the following syntactic abbreviations:

$$\{v : \mathbb{N} \mid p\} \stackrel{\text{def}}{=} \{v : \mathbb{Z} \mid v \geq 0 \wedge p\} \qquad \text{(natural numbers)}$$

$$b \stackrel{\text{def}}{=} \{v : b \mid \mathsf{T}\} \quad \text{with } v \text{ fresh} \qquad \text{(simple types)}$$

$$t_1 \to t_2 \stackrel{\text{def}}{=} (x : t_1) \to t_2 \quad \text{with } x \text{ fresh} \qquad \text{(simple functions)}$$

$$\{x : b \mid p\} \to t \stackrel{\text{def}}{=} (x : \{x : b \mid p\}) \to t \qquad \text{(name punning)}$$

Refinement **Predicates** are terms in a Boolean logic, with the usual Boolean connectives, plus (in)equality relations between predicate expressions, membership queries for regular expression matching, applications of unknown κ variables, and existential quantifiers, which arise during the Fusion phase of κ solving (§ 3.3). Both κ applications and existentials are not part of the user-visible surface syntax. **Expressions** within predicates are built from lifted values, linear integer arithmetic, functions over strings, and uninterpreted functions. To simplify the presentation, predicate expressions are not further syntactically stratified, but are assumed to always occur well-typed (which is assured by the implementation).

VC generation (§ 3.2) results in **Constraints** that are Horn clauses [Bjørner et al. 2015] in Negation Normal Form (NNF), basically tree-like conjunctions of refinement predicates, where each root-to-leaf path is a Constrained Horn Clause (CHC). This representation of VCs is due to Cosman and Jhala [2017], who cleverly employ the constraints' nested scoping structure to make κ solving tractable.

3.2 Type System

The main purpose of the type system of λ_Σ is to generate constraints, in particular input string constraints for parser functions. Thus, the type system is focused on inference/synthesis, rather than type checking. Terms need only be minimally annotated, at λ-abstractions and recursive bindings. The types of applied functions need to be known, however, and available in the typing context (see the discussion of axioms, § 2.1). Our system borrows heavily from Liquid Haskell [Vazou et al. 2014] and the expositions given by Cosman and Jhala [2017] and Jhala and Vazou [2020]. Our presentation is somewhat novel, in that we combine typing rules and VC generation into one syntax-driven declarative system of inference rules, given in Figure 4. This obviates the need for a separate algorithmic presentation of VC generation. Let us now discuss the typing judgements.

$\boxed{t_1 \leqslant t_2 \dashv c}$ ***Subtyping.*** A type t_1 is a subtype of t_2 (meaning, the values denoted by t_1 are subsumed by t_2), if the entailment constraint c is satisfied. In the Sub/Base case, this means the refinement predicate of t_1 must imply the predicate of t_2, for all possible values the types can have. In the Sub/Fun case, where the contra-variant input constraint is joined with the co-variant output constraint, we add an additional implication to the output constraint, strengthening it with the supertype's input predicate. This is done using a *generalized implication* operation, defined below, which simply ensures that only base types are bound to quantifiers in the refinement logic.

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

$\boxed{\Gamma \vdash t \triangleright \hat{t}}$ ***Template Generation.*** To enable complete type synthesis for all intermediate terms, it is sometimes necessary to turn a type t into a template $\hat{t}$, where the refinement predicate is denoted by a placeholder variable whose resolution is deferred (see §§ 2.2, 3.3, and 4). The rule Kap/Base introduces a fresh κ variable, representing an n-ary relation between the type itself and all variables in the current environment Γ. Usually this environment is empty, but if t is a function, Kap/Fun recursively generates input and output templates, extending the environment along the way.

$\boxed{\Gamma \vdash e \nearrow t \dashv c}$ ***Type/Constraint Synthesis.*** Given a typing context Γ mapping values to types, and a term e, we can synthesize a type t whose correctness is implied by the constraint c.

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

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

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

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

$$t \triangleright \hat{t} \stackrel{\text{def}}{=} \varnothing \vdash t \triangleright \hat{t}$$

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

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

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

$$\frac{\Gamma \vdash e \nearrow (y : t_1) \to t_2 \dashv c_e \qquad \Gamma \vdash x \nearrow t_x \qquad t_x \leqslant t_1 \dashv c_x}{\Gamma \vdash e\ x \nearrow t_2[y := x] \dashv c_e \wedge c_x}\ \textsc{Syn/App}$$

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

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

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

$$\frac{\Gamma \vdash x \nearrow \mathbb{B} \qquad \Gamma \vdash e_1 \nearrow t_1 \dashv c_1 \qquad \Gamma \vdash e_2 \nearrow t_2 \dashv c_2}{\Gamma \vdash \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \nearrow t_1 \sqcup t_2 \dashv (x \Rightarrow c_1) \wedge (\neg x \Rightarrow c_2)}\ \textsc{Syn/If}$$

Fig. 4. Typing rules for λ_Σ

Syn/Con synthesizes built-in primitive types, denoted by $\mathsf{prim}(c)$ in the obvious way:

$$\mathsf{prim}(0) \stackrel{\text{def}}{=} \{v : \mathbb{Z} \mid v = 0\}$$
$$\mathsf{prim}(1) \stackrel{\text{def}}{=} \{v : \mathbb{Z} \mid v = 1\}$$
$$\vdots$$

$$\text{prim}(\text{“abc”}) \stackrel{\text{def}}{=} \{\nu : \mathbb{S} \mid \nu = \text{“abc”}\}$$
$$\vdots$$

Syn/Var retrieves the type of a variable from the current context, using *selfification* to produce the most precise possible type [Ou et al. 2004]. The self function lifts the variable into the refinement, allowing each occurrence of the variable in different branches of the program to be precisely typed.

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

Syn/App synthesizes the result type of a function application, where the given input can be a subtype of the function's declared input type. In the result type, the declared input variable is replaced by the given input, using standard capture-avoiding substitution, where

$$e[x := y] \stackrel{\text{def}}{=} e \text{ with all free occurrences of } x \text{ replaced by } y.$$

Note that because our terms are in ANF, no arbitrary expressions are introduced into the type during this substitution. The synthesized VC is simply a conjunction of the function's VC and the constraint created by the subtyping judgement.

Syn/Lam produces a function type whose input refinement is a fresh κ variable, based on the annotation on the $\boldsymbol{\lambda}$-binder.

Syn/Let first synthesizes the type t_1 for the bound term, then the type t_2 for the expression's body under an environment where x is bound to t_1. To ensure that x does not escape its scope, the whole **let**-expression is given the templated type $\hat{t}_2$, a supertype of t_2 with a κ variable in place of its refinement.

Syn/Rec is similar to Syn/Let, with the addition that we first assume the bound term's type as a placeholder $\hat{t}_1$ based on the annotation $\tilde{t}_1$ on the binder, before synthesizing it as t_1, allowing for recursion in the bound term.

Syn/If deals with conditional branches with path-sensitive reasoning. In the VC, we imply the **then**-branch's constraint if the condition is true, and the **else**-branch's constraint if the condition is false. The synthesized type is simply the *join* of the types in the two branches, defined as

$$\{\nu_1 : b \mid p_1\} \sqcup \{\nu_2 : b \mid p_2\} \stackrel{\text{def}}{=} \{\nu_1 : b \mid p_1 \vee p_2[\nu_2 := \nu_1]\}.$$

3.3 Variable Solving

Before the synthesized VCs can be sent off to an SMT solver to be proven valid, we have to replace all κ variables with concrete refinement predicates. Algorithm 1 shows a high-level overview of this procedure, which uses Fusion [Cosman and Jhala 2017] and predicate abstraction [Rondon et al. 2008] to deal with non-grammar κ variables. For grammar variables, i.e., κ variables that appear in constraints of the form $\forall(s : \mathbb{S}).\ \kappa(s) \Rightarrow \varphi$, we use the approach described in § 4 to infer input string grammars by constraint tree rewriting and abstract interpretation.

4 GRAMMAR SOLVING

When presented with an incomplete refinement type $\{s : \mathbb{S} \mid \kappa(s)\}$, together with a verification condition of the form $\forall(s : \mathbb{S}).\ \kappa(s) \Rightarrow \varphi$, our goal is to find an assignment for κ that (1) validates the verification condition and (2) meaningfully refines the type of s.

Validating the verification condition means finding some ρ, such that

$$\mathfrak{M} \vDash \forall(s : \mathbb{S}).\ \rho \Rightarrow \varphi,$$

where $\mathfrak{M}$ is some model of the refinement logic, e.g., linear integer arithmetic and basic string operations, assigning the usual meaning to common structures and operations. While we could

Algorithm 1 Solve an incomplete VC and return its grammar assignments

```
1:  procedure Solve(c)
2:      c ← eliminate all acyclic non-grammar κ variables in c        [Cosman and Jhala 2017]
3:      σ ← ∅
4:      for all grammar constraints in c of the form ∀(s : 𝕊). κ(s) ⇒ φ do
5:          ρ ← Infer(s, φ)                                           [§ 4]
6:          σ ← extend σ with the assignment κ(s) ↦ ρ
7:      end for
8:      c ← apply σ to c
9:      c ← find approximate solutions for residual κ variables left in c   [Rondon et al. 2008]
10:     return c and σ
11: end procedure
```

take any ρ that validates the formula (trivially, we could take ρ to be the constant F), we want to find a *meaningful* refinement. Assuming that $\kappa(s)$ refines the input string of a parser, a meaningful refinement should represent exactly those strings that are accepted by the parser.

The VC's consequent φ technically fulfills this condition. Taken on its own, it is a first-order formula over a free string variable s and captures the semantics of the parsing operations performed on s. The parser P represented by φ accepts some particular string t if and only if φ is true under an assignment of s to t,

$$\mathfrak{M}, [s \mapsto t] \vDash \varphi \quad \iff \quad P \text{ successfully parses } t.$$

The (possibly infinite) set of all strings that satisfy φ is exactly the language $L(P)$ accepted by P. Thus, φ is technically a *grammar* for P, i.e., a finite description of $L(P)$. Practically, however, it makes for a rather poor grammar, being a first-order formula close in size and structure to P itself.

To obtain a better grammar for $L(P)$, we can take φ as a starting point and alternately rewrite and abstract parts of the first-order formula until we have reached a minimal, partially abstract but quantifier-free constraint in disjunctive normal form (DNF). This minimal DNF is then further abstracted to be practically usable as a grammar, and can finally be fully re-concretized into a lightly quantified formula to act as a valid assignment for κ. Algorithm 2 describes the complete procedure. The key components of our approach are

(1) a tree representation of constraints, admitting Boolean rewrite rules (§ 4.1),
(2) a procedure for eliminating quantified variables by abstract substitution (§ 4.2),
(3) a re-concretizable abstract semantics of predicate expressions (§ 4.3),
(4) abstract value representations of all base types (§ 5).

4.1 Constraint Trees

We can represent a first-order constraint φ from our refinement language (Figure 3) in the form of an abstract syntax trees (AST), whose interior nodes are Boolean connectives, including the universal quantifier, and whose leaves are either the Boolean constants or (in)equality relations between predicate expressions. Figure 2 shows an example of such a tree.

The formal tree syntax, given below, combines the constraint and predicate syntax of Figure 3 into tree nodes $\mathcal{T}$ but splits off (in)equality predicates $\mathcal{P}$ and predicate expressions $\mathcal{E}$. Note that existential quantifiers, which are part of the original predicate syntax, are both introduced and eliminated during the Fusion phase of variable solving (§ 3.3) and are not present in any constraints at this point. We also do not allow κ applications to occur in a grammar consequent, nor uninterpreted

Algorithm 2 Simplify a string constraint by inferring its grammar

```
1:  procedure Infer(s, φ)
2:      φ̂ ← rewrite φ into DNF                ▷ apply rewrite rules (§ 4.1) and VarElim (§ 4.2)
3:      G ← ∅
4:      for all disjuncts P ∈ φ̂ do
5:          ŝ ← Σ*                             ▷ initialize grammar for this branch
6:          for all p ∈ P do
7:              ŝ ← ŝ ⊓ ⟦p⟧↑_s                 ▷ abstract string constraint into grammar (§§ 4.3 and 5.4)
8:          end for
9:          G ← G ⊔ ŝ                          ▷ collect grammar alternations
10:     end for
11:     ρ ← ⟦G⟧↓_s                             ▷ re-concretize complete grammar into string constraint (§§ 4.3 and 5.4)
12:     return ρ
13: end procedure
```

functions. In addition to variables $\mathcal{V}$ and concrete values $\mathcal{C}$ (restricted here to Booleans, integers, and strings), we extend expressions with abstract values $\mathcal{A}$. These will be defined in §§ 4.3 and 5.

$$\begin{aligned}
\mathcal{T} &::= \mathcal{T} \wedge \mathcal{T} \mid \mathcal{T} \vee \mathcal{T} \mid \mathcal{T} \Rightarrow \mathcal{T} \mid \mathcal{T} \Leftrightarrow \mathcal{T} \mid \neg\mathcal{T} \mid \forall(\mathcal{V} : b).\, \mathcal{T} \mid \mathsf{T} \mid \mathsf{F} \mid \mathcal{P} \\
\mathcal{P} &::= \mathcal{E} \in \mathrm{RE} \mid \mathcal{E} \bowtie \mathcal{E} \quad \text{where } \bowtie \in \{=, \neq, <, \leq, >, \geq\} \\
\mathcal{E} &::= \mathcal{C} \mid \mathcal{A} \mid \mathcal{V} \mid \mathcal{F}(\overline{\mathcal{E}}) \quad \text{where } \mathcal{F} \in \{\square + \square, \ldots, \square[\square], \ldots\}
\end{aligned}$$

Rewriting. Our goal is to turn this AST into a flat tree with a single $\vee$-node at the root. To achieve this, we iteratively rewrite the AST from the bottom up [Mitchell and Runciman 2007], eliminating quantifiers along the way, until a stable DNF is reached. In addition to the standard identities

$$\begin{array}{lllllll}
a \wedge a \rightsquigarrow a & a \wedge \mathsf{T} \rightsquigarrow a & \mathsf{T} \wedge a \rightsquigarrow a & a \wedge \mathsf{F} \rightsquigarrow \mathsf{F} & \mathsf{F} \wedge a \rightsquigarrow \mathsf{F} & \neg\mathsf{T} \rightsquigarrow \mathsf{F} \\
a \vee a \rightsquigarrow a & a \vee \mathsf{T} \rightsquigarrow \mathsf{T} & \mathsf{T} \vee a \rightsquigarrow \mathsf{T} & a \vee \mathsf{F} \rightsquigarrow a & \mathsf{F} \vee a \rightsquigarrow a & \neg\mathsf{F} \rightsquigarrow \mathsf{T}
\end{array}$$

we apply the following rewrite rules:

$$a \wedge (b \vee c) \rightsquigarrow (a \wedge b) \vee (a \wedge c) \tag{1}$$
$$(a \vee b) \wedge c \rightsquigarrow (a \wedge c) \vee (b \wedge c) \tag{2}$$

Rules 1 and 2 distribute $\wedge$ over $\vee$; together with the lack of symmetrical distributivity rules for $\vee$ over $\wedge$, they guarantee that we stably end up in DNF.

$$\neg\neg a \rightsquigarrow a \tag{3}$$
$$\neg(a \wedge b) \rightsquigarrow \neg a \vee \neg b \tag{4}$$
$$\neg(a \vee b) \rightsquigarrow \neg a \wedge \neg b \tag{5}$$

Rule 3 is double-negation elimination and rules 4 and 5 are DeMorgan's laws.

$$a \Rightarrow b \rightsquigarrow \neg a \vee (a \wedge b) \tag{6}$$
$$a \Leftrightarrow b \rightsquigarrow (\neg a \wedge \neg b) \vee (a \wedge b) \tag{7}$$

Rule 6 is an information-preserving version of the classic material implication rule, where the fact that either a is false or *both* a and b are true is explicitly preserved. The related rule 7 rewrites material equivalences into DNF.

$$\neg(e_1 \bowtie e_2) \;\rightsquigarrow\; e_1 \bowtie^{\mathsf{C}} e_2 \tag{8}$$

Rule 8 distributes negation over (in)equality expressions by complementing the relational operator, i.e., turning $\neg(e_1 > e_2)$ into $e_1 \leq e_2$ and so on.

$$p\,\lightning \;\rightsquigarrow\; \mathsf{F} \tag{9}$$

Rule 9 eliminates branches containing unsolvable expressions. The $\lightning$ operator is defined recursively over tree nodes, predicates, and expressions, and returns true when it hits an abstract value that is $\bot$ in its domain.

$$\forall(x : b).\, \bigvee\bigwedge p \;\rightsquigarrow\; \bigvee \textsc{VarElim}(x, b, \bigwedge p) \tag{10}$$

Finally, rule 10 eliminates $\forall$-quantifiers by substituting the (abstract) definition of the quantified variable, determined via abstract interpretation, at all use sites. The details of this procedure, encapsulated in the VarElim algorithm, are given in § 4.2. Note that at this point in the rewriting, the tree beneath the $\forall$-quantifier is guaranteed to be in DNF and the substitution is carried out separately in each $\vee$-branch, with separately determined definitions.

Example. Figures 5a to 5c show successive applications of a number of rewrite rules on a part of the constraint tree from Figure 2. The rewrite steps shown are:

- (5a $\rightsquigarrow$ 5b) Apply the extended material implication rule 6, negating the left branch and copying the implication's antecedents $n \geq 0$ and $n = |s|$ to its consequent.
- (5b $\rightsquigarrow$ 5c) Apply rules 4 and 8 to push the negation all the way down and flip the (in)equality predicates' relational operators.

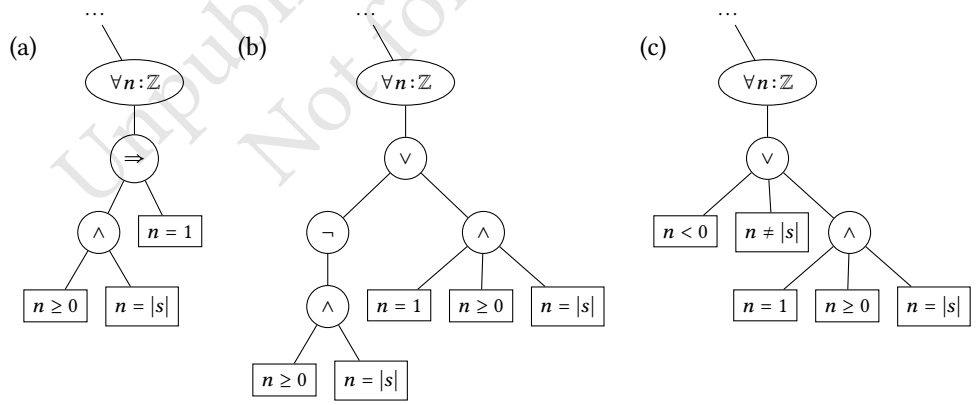


Fig. 5. Successive applications of rewrite rules to a constraint tree: $(a) \overset{6}{\rightsquigarrow} (b) \overset{4}{\rightsquigarrow}\overset{8}{\rightsquigarrow} (c)$.

4.2 Variable Elimination

A $\forall$-quantifier introduces the variable it binds and thereby restricts its scope. A quantified variable can not occur in the AST above its quantifier. Therefore, when bottom-up rewriting reaches a quantifier, all expressions involving that variable occur in the minimized DNF branches below, and

we can eliminate the variable from the tree by replacing all uses of the variable (separately in each disjunct branch) with its *abstract definition*.

The abstract definition of a variable represents all values the variable can assume according to the constraints imposed on it. For example, in the formula

$$x \geq 0 \wedge x < 5 \wedge y = f(x),$$

the abstract definition of x is the range of integers $[0, 4]$, which we can substitute in x's stead to obtain the more compact formula

$$y = f\big([0, 4]\big).$$

We will define efficient abstract representations of (possibly infinite) sets of integers and other base values in § 5. Note that every quantified variable has *some* abstract definition, even if it is not constrained by the formula's predicates—at the very least, the quantifier itself determines the variable to have "all possible" values.

When substituting quantified variables with their abstract definitions, care must be taken to preserve relationships between use sites. Take the formula

$$n = |s| \wedge n > 0 \wedge s[n-1] = \text{“a”},$$

which describes a string whose last character is "a", i.e., a string of the grammar $\Sigma^*\mathsf{a}$. If we naively substitute the abstract definition $n = [1, \infty]$, the resulting formula

$$[1, \infty] = |s| \wedge s\big[[0, \infty]\big] = \text{“a”}$$

is no longer semantically identical to the original: it now describes a string where *all* characters are "a", not just the last one, i.e., the grammar a^+. By replacing every occurrence of n with a set of possible values, the choice of a particular value for n has become independent at each use site. The connection between string length and character-at-index constraint has been severed.

To avoid this situation, we need to first determine if there are *reciprocal dependencies* between the variable to be eliminated and some other variable of the expression. A reciprocal dependency arises when a predicate relates two variables to each other such that each variable could be expressed in terms of the other. In the previous example, $n = |s|$ is a reciprocal dependency between n and s, while $s[n-1] = \text{“a”}$ is not a reciprocal dependency, because even though the latter is also a predicate involving both n and s, it does not relate them in a way that allows us to express one via the other.

After collecting its reciprocal dependencies, we can substitute the quantified variable with the *meet* of its reciprocal expressions. Continuing the previous example, the only reciprocal expression for n is $|s|$, so the formula with n eliminated correctly is

$$|s| > 0 \wedge s\big[|s| - 1\big] = \text{“a”}.$$

How does this work if there is more than one reciprocal expression for a variable? Take a variation of the previous example, describing a string whose last two characters are "a",

$$n < |s| \wedge n \geq |s| - 2 \wedge s[n] = \text{“a”}.$$

Here, the reciprocal dependencies for n are $n < |s|$ and $n \geq |s| - 2$. We can abstract these expressions and compute their meet (using the abstract interpretation techniques we will describe in § 4.3):

$$\begin{aligned} \big[\![n < |s|\big]\!]\!\uparrow_n &= |s| + [-\infty, -1] \\ \big[\![n \geq |s| - 2\big]\!]\!\uparrow_n &= |s| + [-2, \infty] \\ \hline n &= |s| + [-2, -1] \end{aligned}$$

Substituting with the meet of the abstract reciprocals, we obtain

$$s\big[|s| + [-2, -1]\big] = \text{“a”}.$$

In cases where there are multiple reciprocal variables, we can simply pick one at random, preferably the one resulting in the smallest meet. Since all reciprocal variables are by definition in relations with the variable to be eliminated, any substitution affects all variables.

Algorithm 3 describes the full procedure for eliminating quantified variables.

Algorithm 3 Eliminate a ∀-quantified variable

procedure VarElim(x, b, P)
  $\hat{X} \leftarrow \varnothing$ ▷ initialize map from reciprocal variables $\bar{v}$ to abstract definitions $\hat{x}$
  for all $p \in P$ **where** $\bar{v} = \text{vars}(p)$ **and** $x \in \bar{v}$ **do** ▷ for each predicate p with variables $\bar{v}$…
    $\hat{x} \leftarrow \hat{X}(\bar{v})$ or $\top_b$ ▷ …find previous definition of x via $\bar{v}$…
    $\hat{X} \leftarrow \hat{X}\big[\bar{v} \mapsto \hat{x} \sqcap [\![p]\!]\uparrow_x\big]$ ▷ …and refine it by abstracting p (§§ 4.3 and 5)
  end for
  $\bar{v}_{\min}, \hat{x}_{\min} \leftarrow \varnothing, \top_b$
  for all $\bar{v} \mapsto \hat{x} \in \hat{X}$ **where** $\bar{v} \neq \{x\}$ **do**
    if $|\hat{x}| < |\hat{x}_{\min}|$ **then** ▷ find the smallest definition of x via reciprocals
      $\bar{v}_{\min}, \hat{x}_{\min} \leftarrow \bar{v}, \hat{x}$
    end if
  end for
  if $\bar{v}_{\min} = \varnothing$ **then** ▷ if there were no reciprocal definitions…
    $\bar{v}_{\min}, \hat{x}_{\min} \leftarrow \{x\}, \hat{X}(\{x\})$ ▷ …use abstract self-definition of x
  end if
  $Q \leftarrow \varnothing$
  for all $p \in P$ **where** $\text{vars}(p) \neq \bar{v}_{\min}$ **do**
    $Q \leftarrow Q \cup \big\{p\big[x := \hat{x}_{\min}\big]\big\}$ ▷ substitute abstract definition for all uses of x
  end for
  return Q
end procedure

The helper function vars(□), returns the variables that occur in a given predicate or expression:

$$\text{vars}(e_1 \bowtie e_2) = \text{vars}(e_1) \cup \text{vars}(e_2)$$

$$\text{vars}(\mathcal{F}(e_1, e_2, \ldots, e_n)) = \bigcup_{i \le n} \text{vars}(e_i)$$

$$\text{vars}(x) = \begin{cases} \{x\} & \text{if } x \in \mathcal{V} \\ \varnothing & \text{otherwise} \end{cases}$$

Example. Figures 6a to 6c show an application of variable elimination as a continuation of the example from Figure 5. The rewrite steps shown are:

- (6a ⇝ 6b) Apply rule 10 to eliminate the ∀-quantified variable n. This invokes Algorithm 3 on each ∨-branch to determine and substitute abstract definitions for n.
- (6b) This intermediate step shows that the two left disjuncts have been emptied of predicates as a result of VarElim. While it is possible to find abstract definitions $[-\infty, -1]$ for $n < 0$ and $|s| + [-\infty, -1 \,|\, 1, \infty]$ for $n \neq |s|$, the variable n is never actually used by any other expressions in those branches (because there are none). Meanwhile, in the rightmost branch, we could determine that the reciprocal expression $|s|$ is the best abstraction for n.

- (6b ⇝ 6c) After substituting $|s|$ for n and pruning the empty branches, all that is left are the two conjuncts $|s| = 1$ (previously $n = 1$) and $|s| \geq 0$ (previously $n \geq 0$).

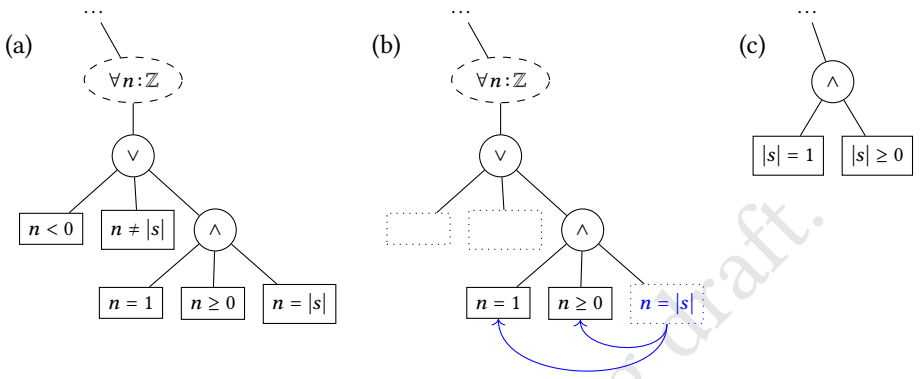


Fig. 6. Elimination of a quantified variable via Algorithm 3.

4.3 Abstract Interpretation

Our approach necessitates that we move between purely symbolic first-order constraints, e.g.,

$$\forall (x : \mathbb{Z}) \ldots \forall (y : \mathbb{Z}).\ y > 4 \wedge x = y,$$

and partially abstract representations of those constraints, e.g.,

$$\forall (x : \mathbb{Z}) \ldots\ x = [5, \infty],$$

where $[5, \infty]$ is an abstract value representing the infinite set $\{5, 6, \ldots\}$. Such abstract representations reify the semantics of first-order formulas. They succinctly model the potentially infinite sets of values prescribed by quantified constraints and allow us to manipulate them on a value level, e.g., to perform algebraic operations on whole expressions. To enable this transformation of predicates to abstract values (and back), we use abstract interpretation.

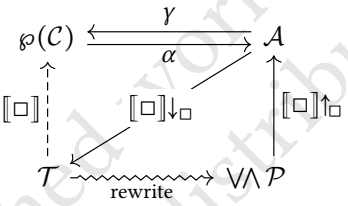
Abstract interpretation [Cousot and Cousot 1977, 1979] is a well-established framework for formalizing static program analyses. It involves the sound approximation of all possible states of a program, usually trading precision for efficiency. The concrete semantics of a program $\mathcal{P}$ are defined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

In our case, the programs we want to abstractly interpret are first-order formulas of (in)equality predicates (§ 4.1). We take the concrete semantics $[\![\varphi]\!]$ of some formula $\varphi \in \mathcal{T}$ to be an assignment σ of free variables to values, such that

$$\mathfrak{M}, \sigma \vDash \varphi \quad \Longleftrightarrow \quad \sigma \in [\![\varphi]\!].$$

Computing $[\![\varphi]\!]$ is generally intractable and would likely result in an unrepresentable infinite set of values that we can not simply abstract after the fact. But directly computing an abstract semantics $[\![\varphi]\!]^\sharp$ for the full first-order constraint is also not easy, especially since we desire an *exact* abstraction—we want to capture all and just those values that satisfy φ.

The trick is to only abstract quantifier-free DNF fragments of the formula, one free variable at a time. When rewriting $\varphi \rightsquigarrow \hat{\varphi}$ (§ 4.1), quantifier elimination (§ 4.2 and Algorithm 3) occurs at a point where all predicates involving the quantified variable are already in DNF. It proceeds by eliminating the variable through an abstract semantics $[\![\square]\!]\uparrow_\square$ in each disjunct, substituting abstract expressions representing the variable's values as prescribed by each conjunction of predicates. Later on (Algorithm 2), we recover a fully concrete constraint ρ from the partially abstract $\hat{\varphi}$ using a constraint re-concretization semantics $[\![\square]\!]\downarrow_\square$. The first-order constraint ρ is logically equivalent to φ but syntactically much simpler. The diagram below summarizes the relationships between the various objects involved in this process.



Abstract Semantics of Constrained Variables. To efficiently abstract the values represented by particular variables involved in relational constraints, we define a variable-focused abstract semantics of (in)equality predicates,

$$[\![\square]\!]\uparrow_\square : \mathcal{P} \times \mathcal{V} \to \mathcal{E}.$$

The semantic function $[\![p]\!]\uparrow_x$ returns an abstract expression that represents all possible values that could be substituted for the variable x in order for the predicate p to be satisfied. More formally,

$$\mathfrak{M}, \sigma \vDash p \quad \Longleftrightarrow \quad \mathfrak{M}, \sigma \backslash x \vDash p[x := y] \text{ where } y \in \gamma\big([\![p]\!]\uparrow_x\big).$$

Here are some examples:

$$[\![x > 5]\!]\uparrow_x \doteq [6, \infty] \qquad [\![x \neq 2]\!]\uparrow_x \doteq [-\infty, 1 \,|\, 3, \infty] \qquad [\![s[2] = \text{“a”}]\!]\uparrow_s \doteq \Sigma^2 \text{a} \Sigma^*$$
$$[\![x > y + 3]\!]\uparrow_x \doteq y + [4, \infty] \qquad [\![x \neq 2]\!]\uparrow_y \doteq \top \qquad [\![|s| \leq 5]\!]\uparrow_s \doteq \Sigma^5$$

Constraint Re-Concretization. To turn abstract values back into constraints over a given variable, we define a "reverse" abstract semantics

$$[\![\square]\!]\downarrow_\square : \mathcal{E} \times \mathcal{V} \to \mathcal{T},$$

mirroring the abstract semantics $[\![\square]\!]\uparrow_\square$. The re-concretization function $[\![\hat{e}]\!]\downarrow_x$ produces a constraint over a free variable x that is satisfied by any and only those values abstractly represented by $\hat{e}$. More formally,

$$\mathfrak{M}, \sigma \vDash [\![\hat{e}]\!]\downarrow_x \quad \Longleftrightarrow \quad \mathfrak{M}, \sigma \backslash x \vDash [\![\hat{e}]\!]\downarrow_x [x := y] \text{ where } y \in \gamma(\hat{e}).$$

And here again are some examples:

$$[\![[6,\infty]]\!]\downarrow_x \doteq x \geq 6 \qquad [\![\mathsf{a}^*\mathsf{b}^*\mathsf{c}]\!]\downarrow_s \doteq s \in \mathsf{a}^*\mathsf{b}^*\mathsf{c}$$
$$[\![[-\infty,1\,|\,3,\infty]]\!]\downarrow_x \doteq x \neq 2 \qquad [\![\Sigma^*]\!]\downarrow_s \doteq \mathsf{T}$$

Note that while the abstraction of a predicate followed by the re-concretization of that abstraction by definition results in an equivalent predicate, i.e., a predicate with the same semantics as the original, it is not necessarily the exact same predicate (e.g., the chain $x > 5 \to [6,\infty] \to x \geq 6$ in the two sets of examples above).

In § 5, we will give complete definitions of $[\![\square]\!]\uparrow_\square$ and $[\![\square]\!]\downarrow_\square$ for various abstract domains $\mathcal{A}$.

5 ABSTRACT DOMAINS

We now define abstract domains for each of the base types in our system (except the unit type $\mathbb{1}$). Each abstract domain efficiently captures (possibly infinite) sets of concrete values of the corresponding type, and defines the relevant abstract semantics $[\![\square]\!]\uparrow_\square$ and $[\![\square]\!]\downarrow_\square$ for variables of that type. To simplify those definitions and avoid boilerplate repetition, we generally assume that predicates have already been arranged in a uniform manner and that constant sub-terms have been maximally reduced via their abstract semantics.

5.1 Booleans

The subset lattice $\langle \wp(\mathbb{B}), \subseteq \rangle$ is a complete abstraction of the Boolean values, adding $\varnothing$ and $\{\mathsf{T},\mathsf{F}\}$ as bottom and top elements, respectively, and forming a complete complemented lattice via subset inclusion $\subseteq$ and set complement $\complement$. For consistency with the other definitions, we call this abstraction $\hat{\mathbf{B}}$ and will use the notation $\langle \hat{\mathbf{B}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$ for the lattice elements and operations. The abstraction and concretization functions are trivially defined as $\alpha = \gamma = \mathrm{id}$. The non-extrema definitions for predicate semantics are given below.

$$[\![x = b]\!]\uparrow_x \stackrel{\text{def}}{=} \{b\} \qquad [\![\{\mathsf{T}\}]\!]\downarrow_x \stackrel{\text{def}}{=} x = \mathsf{T}$$
$$[\![x \neq b]\!]\uparrow_x \stackrel{\text{def}}{=} \{\neg b\} \qquad [\![\{\mathsf{F}\}]\!]\downarrow_x \stackrel{\text{def}}{=} x = \mathsf{F}$$

5.2 Integers

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

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

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

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

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

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

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

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

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

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

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

$$(x:xs) \sqsubseteq (y:ys) \iff x \sqsubseteq_{\mathbb{IZ}} y \wedge xs \sqsubseteq ys$$

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

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

$$\neg[-\infty, b_1 \mid \dots \mid a_n, \infty] = [b_1+1, a_2-1 \mid \dots \mid b_{n-1}+1, a_n-1]$$
$$\neg[-\infty, b_1 \mid \dots \mid a_n, b_n] = [b_1+1, a_2-1 \mid \dots \mid b_{n-1}+1, a_n-1 \mid b_n+1, \infty]$$
$$\neg[a_1, b_1 \mid \dots \mid a_n, \infty] = [-\infty, a_1-1 \mid b_1+1, a_2-1 \mid \dots \mid b_{n-1}+1, a_n-1]$$
$$\neg[a_1, b_1 \mid \dots \mid a_n, b_n] = [-\infty, a_1-1 \mid b_1+1, a_2-1 \mid \dots \mid b_{n-1}+1, a_n-1 \mid b_n+1, \infty]$$

The standard arithmetic operations are lifted into $\hat{\mathbf{Z}}$ via pointwise mapping of the equivalent operations on $\mathbb{IZ}$. We assume an abstract semantics for integer expressions, defined in the standard way, allowing us to reduce and rewrite arithmetic expressions into a canonical form.

We can construct the abstraction function $\alpha : \wp(\mathbb{Z}) \to \hat{\mathbf{Z}}$ for any finite subset of integers $X \in \wp(\mathbb{Z})$ by first sorting all elements of X in ascending order and then identifying all non-overlapping

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

intervals of consecutive integers. This strategy does not work if X is infinite, and in any case is rather inefficient. Likewise, the concretization function $\gamma : \hat{\mathbf{Z}} \to \wp(\mathbb{Z})$, defined as

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

is not very practical. Fortunately, for our use case we only ever need to abstract and concretize sets of integers defined via relational predicates, which is easily tractable, even with infinite bounds. The corresponding functions are defined below. We assume that relations have been re-arranged into canonical form and have omitted definitions that can be easily derived.

$$[\![x = a]\!]\uparrow_x \stackrel{\text{def}}{=} [a, a] \qquad [\![x \ge a]\!]\uparrow_x \stackrel{\text{def}}{=} [a, \infty] \qquad [\![x \le a]\!]\uparrow_x \stackrel{\text{def}}{=} [-\infty, a]$$
$$[\![x \ne a]\!]\uparrow_x \stackrel{\text{def}}{=} [-\infty, a-1 \,|\, a+1, \infty] \qquad [\![x > a]\!]\uparrow_x \stackrel{\text{def}}{=} [a+1, \infty] \qquad [\![x < a]\!]\uparrow_x \stackrel{\text{def}}{=} [-\infty, a+1]$$

$$[\![\varnothing]\!]\downarrow_x \stackrel{\text{def}}{=} \mathsf{F} \qquad [\![[a, \infty]]\!]\downarrow_x \stackrel{\text{def}}{=} x \ge a \qquad [\![[a, a]]\!]\downarrow_x \stackrel{\text{def}}{=} x = a$$
$$[\![[-\infty, \infty]]\!]\downarrow_x \stackrel{\text{def}}{=} \mathsf{T} \qquad [\![[-\infty, b]]\!]\downarrow_x \stackrel{\text{def}}{=} x \le b \qquad [\![[a, b]]\!]\downarrow_x \stackrel{\text{def}}{=} x \ge a \wedge x \le b$$

$$[\![[-\infty, b_1 \,|\, \ldots \,|\, a_n, \infty]]\!]\downarrow_x \stackrel{\text{def}}{=} \overbrace{x \ne b_1 + 1 \wedge \cdots \wedge x \ne a_2 - 1 \wedge x \ne b_2 + 1 \wedge \cdots \wedge x \ne a_n - 1}^{\text{gaps}}$$
$$[\![[-\infty, b_1 \,|\, \ldots \,|\, a_n, b_n]]\!]\downarrow_x \stackrel{\text{def}}{=} x \le b_n \wedge \text{gaps}$$
$$[\![[a_1, b_1 \,|\, \ldots \,|\, a_n, \infty]]\!]\downarrow_x \stackrel{\text{def}}{=} x \ge a_1 \wedge \text{gaps}$$
$$[\![[a_1, b_1 \,|\, \ldots \,|\, a_n, b_n]]\!]\downarrow_x \stackrel{\text{def}}{=} x \ge a_1 \wedge x \le b_n \wedge \text{gaps}$$

5.3 Characters

While in the refinement logic individual characters are simply treated as single-element strings, it is useful to have an explicit notion of an abstract character, both in our formal treatment and as part of our implementation. Abstract characters, i.e., sets of possible characters, are a common component of string grammars—whitespace, for example, is usually defined as a set of certain invisible characters. While the size of any string alphabet is always bounded and thus the maximum number of possibilities for a single character is finite, these bounds can be quite large—the Unicode standard currently defines 149 186 characters. Additionally, it is often desirable to define elements of a string by what characters are *not* allowed to be there. Thus the need for an efficient abstract representation of large sets of (im)possible characters.

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

Practically, the machine representation of $\hat{\mathbf{C}}$ is based on complemented PATRICIA tries [Kmett 2012; Morrison 1968; Okasaki and Gill 1998].

5.4 Strings

To abstractly represent infinite sets of strings, we use a domain of regular expressions

$$\hat{\mathbf{S}} \stackrel{\text{def}}{=} \big\{ \text{RE} \;\big|\; \text{RE is a regular expression over abstract characters } \hat{\mathbf{C}} \big\},$$

which forms a complete complemented lattice $\langle \hat{\mathbf{S}}, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \neg \rangle$ with $\bot = \varnothing$ and $\top = \Sigma^*$, and the standard operations on regular sets [Hopcroft and Ullman 1979]. Regular expressions RE are formed in the usual way, with alternation $\square|\square$, the Kleene star $\square^*$, and superscripts to indicate repetition, e.g., $\mathsf{a}^3 \equiv \mathsf{aaa}$. Abstract characters directly translate into character literals for singleton sets; character classes for sets of consecutive characters in an ordered alphabet, e.g., $\{\mathsf{a}, \mathsf{b}, \ldots, \mathsf{z}\} \equiv \texttt{[a-z]}$; the wildcard . for Σ; or simply alternations of the characters they represent.

The language $L(\hat{s})$ describes all strings generated/recognized by the regular expression $\hat{s} \in \hat{\mathbf{S}}$, thus $\gamma(\hat{s}) = L(\hat{s})$. We can of course only abstract sets of strings that form a regular language, i.e., $\alpha(S) = \hat{s} \iff S = L(\hat{s})$, which includes all finite sets of strings, as well as every infinite set of strings expressible by singular predicates in our refinement language. While α is not generally realizable [Gold 1967], we can exploit the fact that, in the context of our work, sets of strings are described via a limited set of refinement constraints. We can thus define the abstract semantics for (in)equality relations over string expressions given below. Again we assume that relations have been re-arranged into canonical form and have omitted definitions that can be easily derived.

$$\begin{aligned}
[\![s \in \hat{s}]\!]\uparrow_s &\stackrel{\text{def}}{=} \hat{s} \\
[\![s[e] \neq c]\!]\uparrow_s &\stackrel{\text{def}}{=} [\![s[e] = \neg c]\!]\uparrow_s \\
\left[\!\!\left[s\big[[a,b]\big] = c \right]\!\!\right]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a-1} c^{b-a} \Sigma^* \\
\left[\!\!\left[s\big[[a,\infty]\big] = c \right]\!\!\right]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a-1} c^{+} \\
\left[\!\!\left[s\big[[a_1,b_1 \,|\, \ldots \,|\, a_n,b_n]\big] = c \right]\!\!\right]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a_1-1} c^{b_1-a_1} \Sigma^{a_2-b_2} \ldots c^{b_n-a_n} \Sigma^* \\
\left[\!\!\left[s\big[[a_1,b_1 \,|\, \ldots \,|\, a_n,\infty]\big] = c \right]\!\!\right]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a_1-1} c^{b_1-a_1} \Sigma^{a_2-b_2} \ldots c^{+} \\
[\![|s| = [a,b]]\!]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^a \mid \ldots \mid \Sigma^b \\
[\![|s| = [a,\infty]]\!]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^a \Sigma^* \\
[\![|s| = [a_1,b_1 \,|\, \ldots \,|\, a_n,b_n]]\!]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a_1} \mid \Sigma^{a_1+1} \mid \ldots \mid \Sigma^{b_1} \mid \Sigma^{a_2} \mid \Sigma^{a_2+1} \mid \ldots \mid \Sigma^{b_n} \\
[\![|s| = [a_1,b_1 \,|\, \ldots \,|\, a_n,\infty]]\!]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^{a_1} \mid \Sigma^{a_1+1} \mid \ldots \mid \Sigma^{b_1} \mid \Sigma^{a_2} \mid \Sigma^{a_2+1} \mid \ldots \mid \Sigma^{a_n} \Sigma^* \\
\left[\!\!\left[s\big[|s| - a\big] = c \right]\!\!\right]\uparrow_s &\stackrel{\text{def}}{=} \Sigma^* c \Sigma^{a-1}
\end{aligned}$$

We define re-concretization simply via the regular membership query of the refinement logic,

$$[\![\hat{s}]\!]\downarrow_s \stackrel{\text{def}}{=} s \in \hat{s}.$$

While it is possible to express abstract strings without involving regular membership queries, this usually comes at the cost of re-introducing quantified variables, e.g.,

$$[\![\mathsf{ab}^*]\!]\downarrow_s \doteq s[0] = \text{“a”} \wedge \forall (i : \mathbb{Z}).\ i \geq 1 \Rightarrow s[i] = \text{“b”}.$$

6 RELATED WORK

Dynamic Grammar Inference for Fuzzing. Obtaining input grammars of programs has been heavily pursued by the *fuzzing* community [Manes et al. 2019; Zeller et al. 2021] for use in *grammar-based fuzzing* [Aschermann et al. 2019; Holler et al. 2012]. The idea behind fuzzing is simple: test programs by bombarding them with (systematically generated) random inputs and see if anything breaks. But generating good fuzz inputs is hard, because in order to penetrate into deep program states, one generally needs valid or near-valid inputs, meaning inputs that pass at least the various

syntactic checks and transformations—i.e., ad hoc parsers—scattered throughout a typical program. In grammar-based fuzzing, valid inputs are specified with the help of language grammars, shifting the problem to that of obtaining accurate grammars or language models. Black-box approaches try to infer a language model by poking the program with seed inputs and monitoring its runtime behavior [Bastani et al. 2017; Godefroid et al. 2017]. This has some theoretical limits [Angluin 1987; Angluin and Kharitonov 1995] and the amount of necessary poking (i.e., membership queries) grows exponentially with the size of the grammar. White-box approaches make use of the program code and can thus use more sophisticated techniques like taint tracking to monitor data flow between variables [Höschele and Zeller 2016] or observing character accesses of input strings by tracking dynamic control flow [Gopinath et al. 2020]. These approaches can produce fairly accurate and human-readable grammars, at least in test settings, but they rely on dynamic execution and thus require complete runnable programs. Furthermore, the accuracy of such dynamically inferred grammars cannot be guaranteed.

String Constraint Solving. Precise formal reasoning over strings can be accomplished using *string constraint solving* (SCS), a declarative paradigm of modeling relations between string variables and solving attendant combinatorial problems [Amadini 2021]. It is usually assumed that collecting string constraints requires some kind of (dynamic) symbolic execution [Kausler and Sherman 2014], and practical SCS applications are generally concerned with the inverse of our problem: modeling the possible strings a function can return or express [Bultan et al. 2018], instead of the strings a function can accept. In our approach, we use purely static means (viz. refinement type inference) to essentially collect input string constraints (see § 4.1), which we then simplify/solve in ways not dissimilar but nonetheless different from traditional SCS techniques (see § 5). There have been many recent advances in SCS for SMT [Abdulla et al. 2015; Kan et al. 2022; Kiezun et al. 2009; Trinh et al. 2014, 2020; Zheng et al. 2013]. We particularly make use of string theories embedded in the Z3 constraint solver [Berzish et al. 2017] in our implementation (§ 7).

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

Abstract Domains. Abstract string domains approximate strings to track information precisely enough to analyze particular behaviors of interest while only preserving relevant information. Most of the existing work in string domains differs in what kind of behavior is of interest and how the approximation is achieved efficiently.

Costantini et al. [2015] introduces a suite of different abstract semantics for concatenation, character inclusion, and substring extraction (particularly pre- and suffixes). In their work, they explicitly discuss the trade-off between precision and efficiency. Amadini et al. [2020] review the dashed string abstraction, an approach that considers strings as blocks of characters and the constraints on these blocks, which has shown good performance on benchmarks involving constraints on string length, equality, concatenation, and regular expression membership. M-String [Cortesi and Olliaro 2018] considers a parametric abstract domain for strings in the C programming language by leveraging abstract domains for the content of a string and for expressions to infer when a string index position corresponds to an expression of interest. While most of the existing work has

focused on approximating a single variable, very recent work by Arceri et al. [2022] focuses on relational string domains that try to capture the relation between string variables and expressions for which we cannot compute static values, such as user input.

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

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

7 IMPLEMENTATION

We have implemented our approach in the Panini system, available at [link to repository removed for double blind review]. It includes a small default set of λ_Σ specifications for common string functions. Our implementation is written in Haskell and uses the Z3 theorem solver [De Moura and Bjørner 2008] and is modular with respect to the abstract domains used during grammar solving. Panini can be used as a library, a standalone batch-mode command-line application, or interactively via a read-eval-print loop.

REFERENCES

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

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

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

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

Roberto Amadini, Alexander Jordan, Graeme Gange, François Gauthier, Peter Schachte, Harald SØndergaard, Peter J. Stuckey, and Chenyi Zhang. 2017. Combining String Abstract Domains for JavaScript Analysis: An Evaluation. In *Proceedings, Part I, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10205*. Springer-Verlag, Berlin, Heidelberg, 41–57. https://doi.org/10.1007/978-3-662-54577-5_3

Dana Angluin. 1987. Queries and Concept Learning. *Machine Learning* 2, 4 (1987), 319–342. https://doi.org/10.1007/BF00116828

D. Angluin and M. Kharitonov. 1995. When Won't Membership Queries Help? *J. Comput. System Sci.* 50, 2 (April 1995), 336–355. https://doi.org/10.1006/jcss.1995.1026

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

Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium* (San Diego, California, USA) *(NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23412

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

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

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

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

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

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

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Springer, 24–51.

Nikolaj Bjørner, Nikolai Tillmann, and Andrei Voronkov. 2009. Path feasibility analysis for string-manipulating programs. In *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer, 307–321.

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

Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leißa, Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In *Proceedings of the 22nd International Conference on Compiler Construction* (Rome, Italy) *(CC'13)*. Springer-Verlag, Berlin, Heidelberg, 102–122. https://doi.org/10.1007/978-3-642-37051-9_6

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

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

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

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

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

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

Patrick Cousot. 1997. Types as Abstract Interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 316–331. https://doi.org/10.1145/263699.263744

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

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

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

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

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

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

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

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

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

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

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

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

Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, 50–59. https://doi.org/10.1109/ASE.2017.8115618

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

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

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

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

Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with Code Fragments. In *Proceedings of the 21st USENIX Conference on Security Symposium* (Bellevue, WA) *(Security'12)*. USENIX Association, USA, 38.

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

Matthias Höschele and Andreas Zeller. 2016. Mining Input Grammars from Dynamic Taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. ACM, New York, NY, USA, 720–725. https://doi.org/10.1145/2970276.2970321

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

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

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

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

Charaka Geethal Kapugama, Van-Thuan Pham, Aldeida Aleti, and Marcel Böhme. 2022. Human-in-the-loop oracle learning for semantic bugs in string processing programs. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 215–226.

Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.

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

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

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

Daan Leijen and Erik Meijer. 2001. *Parsec: Direct Style Monadic Parser Combinators for the Real World.* Technical Report UU-CS-2001-35. Department of Information and Computing Sciences, Utrecht University. http://www.cs.uu.nl/research/techreps/repo/CS-2001/2001-35.pdf

Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. 2011. KLOVER: A symbolic execution and automatic test generation tool for C++ programs. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 609–615.

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

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

Valentin J. M. Manes, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. 2019. The Art, Science, and Engineering of Fuzzing: A Survey. arXiv:1812.00140 [cs.CR]

Björn Mathis, Rahul Gopinath, Michaël Mera, Alexander Kampmann, Matthias Höschele, and Andreas Zeller. 2019. Parser-Directed Fuzzing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. 548–560.

Neil Mitchell and Colin Runciman. 2007. Uniform Boilerplate and List Processing. In *Haskell '07: Proceedings of the ACM SIGPLAN workshop on Haskell* (Freiburg, Germany). ACM, 49–60. https://doi.org/10.1145/1291201.1291208

Falcon Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. 2016a. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 45–52.

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

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

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

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

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

Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *Exploring New Frontiers of Theoretical Informatics*, Jean-Jacques Levy, Ernst W. Mayr, and John C. Mitchell (Eds.). 437–450. https://doi.org/10.1007/1-4020-8141-3_34

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

Changhee Park, Hyeonseung Im, and Sukyoung Ryu. 2016. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages*. 25–36.

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

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. ACM, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. 2010. A symbolic execution framework for javascript. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 513–528.

Mohamed Nassim Seghir and Daniel Kroening. 2013. Counterexample-Guided Precondition Inference. In *Proceedings of the 22nd European Conference on Programming Languages and Systems* (Rome, Italy) *(ESOP'13)*. Springer-Verlag, Berlin, Heidelberg, 451–471. https://doi.org/10.1007/978-3-642-37036-6_25

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

Minh-Thai Trinh, Duc-Hiep Chu, and Joxan Jaffar. 2020. Inter-theory dependency analysis for SMT string solvers. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27.

Katherine Underwood and Michael E Locasto. 2016. In Search of Shotgun Parsers in Android Applications. In *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, 140–155.

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

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

Mark Weiser. 1984. Program Slicing. *IEEE Transactions on Software Engineering* SE-10, 4 (July 1984), 352–357. https://doi.org/10.1109/TSE.1984.5010248

Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. 2021. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security. https://www.fuzzingbook.org/

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