Grammar Inference for Ad Hoc Parsers

Research Proposal for PhD Proficiency Evaluation

MICHAEL SCHRÖDER, TU Wien, Austria

Ad hoc parsers are everywhere: any time we use common string functions like `split`, `trim`, or `slice`, we effectively perform parsing. Yet no one ever bothers to write down *grammars* for such ad hoc parsers. Grammars are finite sets of rules that describe the possibly infinite set of input strings that a program accepts without going wrong. They can serve as documentation, aid program comprehension, be used to generate test inputs, and allow reasoning about language-theoretic security. But like most forms of specification, grammars are tedious to write and difficult to get right.

In this proposal, I describe my ongoing research toward an automatic grammar inference system for ad hoc parsers, which will allow programmers to get input grammars from unannotated source code "for free," enabling a range of new possibilities, from interactive documentation to grammar-aware semantic change tracking. To this end, I introduce the Panini language, an intermediate representation that uses a novel refinement type system to synthesize grammar-like string constraints.

This document is partly based on two previous publications [75, 76] and an unpublished extended abstract [77].

1 INTRODUCTION

Parsing is one of the fundamental activities in software engineering. Defined as "the process of structuring a linear representation in accordance with a given grammar" [39], it is an activity so common that pretty much every program performs some kind of parsing at one point or another. In academia, parsing has been studied since the very early days of computer science [45] and *formal language theory*, which has its origin in linguistics [18], provides the foundation for an impressive amount of both theoretical results [43] and practical applications [39]. Yet in every-day software engineering, only a small minority of programs, mainly compilers and some protocol implementations, make explicit reference to these formal-theoretic underpinnings, documenting grammars of their input languages or making use of formalized parsing techniques such as combinator frameworks [53] or parser generators [48, 68, 82]. The vast majority of parsing code in software today is *ad hoc*.

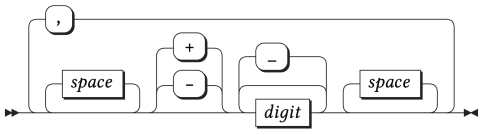
1.1 Ad Hoc Parsing

Consider the following Python expression, a typical example of ad hoc parsing:

```
xs = map(int, s.split(","))
```

Here, a string `s` is transformed into a list of integers `xs` using only the standard Python functions `split` and `map`, and the Python `int` constructor. A programmer writing this expression would probably not even think about the fact that they are actually writing a parser. Code like this 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* [62].

Figure 1 shows the above parser's implicit grammar, a finite but complete formal description of all values the input string can have without the program going wrong in some way. Parts of this grammar might be surprising and not at all obvious from looking at the code alone. Even an experienced Python programmer might be unaware, for example, that the `int` constructor, in addition to allowing an optional leading `+` or `-` sign, also permits leading zeroes, strips surrounding whitespace, and ignores single `_` characters that are used for grouping digits. Looking at the



$$
\begin{aligned}
s &\rightarrow int \mid int\ \texttt{,}\ s \\
int &\rightarrow space^{*}\ (\texttt{+} \mid \texttt{-})^{?}\ digit\ (\texttt{\_}^{?}\ digit)^{*}\ space^{*} \\
digit &\rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
space &\rightarrow \texttt{␣} \mid \texttt{\textbackslash t} \mid \texttt{\textbackslash n} \mid \texttt{\textbackslash v} \mid \texttt{\textbackslash f} \mid \texttt{\textbackslash r}
\end{aligned}
$$

Fig. 1. A formal grammar for the Python expression in Section 1.1, in both visual and textual representations. The notation used here denotes `terminals` with typewriter font and uses the common operators $*$, $+$, and $?$ for zero-or-more, one-or-more, and optional occurrences, respectively. The vertical bar $\mid$ separates alternative productions, without precedence. Parentheses are used for grouping in the usual way. This grammar assumes the semantics of Python 3.9.

grammar, we can determine that the strings `"1,2,3"` and `"+01_2,␣␣3␣"` will both be accepted by the parser, while the empty string `""` is not a valid input—it will in fact crash the program.[1]

1.2 The Benefits of Grammars

As the above example demonstrates, a grammar can elucidate the corresponding parsing code, revealing otherwise hidden features and potentially 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. There exist numerous notations for grammars, each suitable for different languages and in different contexts: regular expressions [80], Chomsky normal form [19], Augmented Backus-Naur Form (ABNF) [24], parsing expression grammars (PEGs) [33], to name a few. Graphic representations, like finite state machines [43] or railroad diagrams [15], can be particularly helpful in understanding abstract data and align with developers' appreciation of sketches and diagrams [5]. Augmenting regular documentation with formal grammars can increase program comprehension by providing alternative representations for a programming task [31, 35].

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.

Finally, as a formal description of an input language, a grammar allows us to reason about various language-theoretic properties, such as computability bounds. The *language-theoretic security* (LANGSEC) community[2] regards such reasoning as vital in assuring the correctness and safety of input handling routines. For example, if an input language is recursively enumerable, we can never guarantee that its parser behaves safely (i.e., halts) on inputs that are not in the language, because the parser must be equivalent to a Turing machine. Thus, input languages should be minimally powerful, and their parsers should match them in computational power [73]. Ad hoc parsers open themselves up to attack, because it is not clear what languages they implement, or if they implement them correctly, and variations among implementations are easily overlooked [74]. Grammars can help assure us that our input languages have favorable properties and that their parsers are implemented correctly.

1.3 The Vision of Grammar Inference

Despite providing all these benefits, hardly anyone ever bothers to write down a grammar. Grammars share the same fate as most other forms of specification: they are tedious to write, hard to get right,

[1] This is because the `split` function, when applied to an empty string, returns a singleton list also containing an empty string (rather than an empty list, as one might assume). The `int` constructor, applied to this empty string via `map`, will then throw a runtime exception.

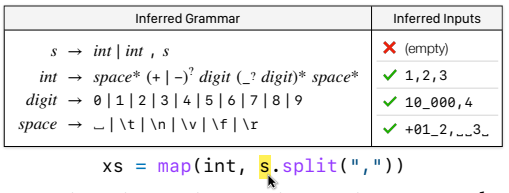
[2] https://langsec.org

and seem hardly worth the trouble—especially for such small pieces of code like ad hoc parsers. *If we are not building whole houses, why should we draw blueprints?* [52]

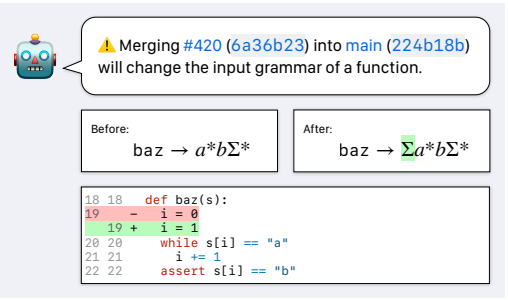
The only type of grammar that people actually routinely write down are *regular expressions* [80], probably the biggest and most widely known success story of applied formal language theory. But regular expressions are also brittle, non-portable, and as hard to master as full-featured grammars [25, 60]. They are often used within bigger pieces of ad hoc parsing code and thus usually only describe part of the actual input grammar of an ad hoc parser.

But there is 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 a 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.[3] If we could infer grammars like we can infer types, we could reap all the rewards of having a complete specification of our program's input language, without burdening the programmer with the full weight of formal language theory. This will not only let us enjoy all the benefits that formal grammars provide in general, it also enables some exciting new possibilities:

Interactive Documentation. A grammar that is automatically inferred will always be up-to-date—a significant advantage over manually written documentation, which tends to quickly drift from the object it documents [54]. Furthermore, an inferred grammar could be closely linked directly to the underlying source code, making productions traceable to their origins. One can imagine an interactive environment where hovering over parts of a grammar highlights the corresponding pieces of code—or even allows changing them by manipulating the high-level representation.



Mining & Learning. An inferred grammar abstracts over the underlying concrete parser implementation and can be viewed as an equivalence class, allowing us to group together different parser implementations with similar semantics.[4] This opens up new possibilities in mining software repositories, such as grammar-enhanced semantic code search [34, 61, 70] or detecting code clones [49, 84] of ad hoc parsers. By automatically inferring grammars for each code change, it also becomes possible to learn how (implicit) input specifications evolve over time, enabling a type of grammar-aware semantic change tracking [40, 71]. Augmenting the code review process with current as well as historical grammar information would allow developers to be alerted when a code change introduces a perhaps unexpected change in input grammar.



[3] For a good introduction to type inference and its history, see [56, § 4].

[4] While there are a number of theoretical bounds regarding the decidability of properties about grammars, it is in fact possible to efficiently decide equivalence for many types of grammars encountered in practice [57].

Bi-directional Parser Synthesis. Combining grammar inference with parser generation enables a framework of bi-directional parser synthesis. In the most basic case, starting from an existing complete parser implementation, the synthesizer can be used to generate different implementations according to certain criteria, e.g., performance or code style, by transformation via the inferred grammar—a specialized type of semantic program transformation [22]. If the initial parser is incomplete, a bi-directional parser synthesizer can be used for *program sketching* [55, 69, 79], wherein an initial implementation (a "sketch") is the basis of an initial grammar which can be manipulated by the user on a high level—perhaps graphically—to then in turn synthesize a completed or refined implementation. If the sketch-synth loop can be sufficiently shortened, it can be the basis for a direct manipulation bi-directional programming system [20, 59], although based on transformations of the (specification of) inputs to the program rather than its outputs.

2 BACKGROUND

I hope to realize automatic grammar inference based on the following intuition: Any parser is essentially a *machine* in the formal sense—it is a recognizer for its input language.

2.1 Languages & Machines

Formally, a *language* L is a possibly infinite set of sentences over a finite alphabet Σ. We can define languages very abstractly, as in $L = \{a^n b^n \mid n > 0\}$, a language over the alphabet $\Sigma = \{a, b\}$ that consists of all sentences with at least one a followed by the same number of bs. Usually, however, we define languages via generative devices called *grammars* or recognizing devices called *machines*.

A grammar $G = (V, \Sigma, P, S)$ is a finite description of a language and consists of a set of *variables* (or *nonterminals*) V; a *terminal* alphabet Σ; a set of *productions* P, which are rules of the form $\alpha \to \beta$ where α and β are from V and/or Σ; and a *start symbol* $S \in V$. By starting with S and applying a finite number of productions from P, we can generate sentences over Σ. The language $L(G)$ is the set of all sentences that can be generated by G. By putting various constraints on the *form* of a grammar, such as whether the left-hand side of a production can only include variables, or the right-hand side has to include at least one terminal symbol, and so on, we can limit the grammar's expressiveness, constraining the *family* of languages a grammar of this form can produce. The famous Chomsky hierarchy [19] partitions languages/grammars into four increasingly expressive levels: *regular*, *context-free*, *context-sensitive*, and *recursively enumerable*. Numerous additional language families and types of grammars have been discovered, within and beyond the classic hierarchy: *attribute grammars* [51], *boolean grammars* [65, 66], the *mildly context-sensitive* and *sub-regular* languages [46], *parsing expression grammars* (PEGs) [33], to name just a few.

A machine M, unlike a grammar, does not produce sentences but consumes them. Taking some sentence as input and moving through a finite number of internal states, it arrives at some halting configuration if and only if the sentence is part of the language $L(M)$. If the sentence is not part of the language, the machine either runs forever or gets stuck in a non-accepting state. Just like with grammars, the way that a machine is constructed determines its expressiveness. There is a natural correspondence between languages, grammars, and machines: regular languages correspond to *finite state machines*, which simply move from one internal state to another based on the next input character; context-free languages correspond to finite state machines equipped with a pushdown stack, also known as *pushdown automata*; context-sensitive languages correspond to *linearly bounded automata*, in essence Turing machines with a finite tape; and finally the recursively enumerable languages correspond to the well-known unbounded *Turing machines*. As with grammars, there are numerous additional and alternative constructions between and beyond these classic ones.

2.2 Parsers are Sub-Turing Islands

A parser, like the Python snippet in Section 1.1, which expects a string from some language L_1 as input, can be seen as a machine M_1 recognizing that language, so that $L_1 = L(M_1)$. This machine is however embedded within the more powerful machine M_0, the general-purpose programming language that the parser itself is written in. Any real world parser will do more than just recognize a language: it will allocate and transform data types, throw exceptions or handle parse errors, or perform side effects unrelated to the parsing process itself. Nevertheless, the control flow at the core of a parser will, in our experience, closely match that of the (hypothetical) machine M_1. While it is entirely possible that a parser written in a Turing-complete programming language exhibits exactly those traits that make it equivalent to a Turing machine, even though it might be parsing a "lesser" language, this seems very unlikely. In almost all practical situations, ad hoc parsing code will not significantly exceed the "power-level" of the language it is parsing. For example, unless it has been especially constructed to be confounding, the loops present in a parser will invariably be bounded by at most some linear factor of the length of its input, which corresponds to the expressiveness of a context-sensitive language. Essentially, parsers—and in particular ad hoc parsers—form *Sub-Turing islands* [7], regions of code for which interesting program analysis questions are decidable. Questions, such as: What is the language accepted by this machine?

2.3 Related Work

Grammatical Inference. Given the source code of a parser, we want to find a grammar describing the language that the parser recognizes. Note that this is different from the related problem of finding a grammar given a set of sentences that can be produced by that grammar, which is known as *grammar induction* or *grammatical inference* [41]. Early results in computational linguistics quickly established fundamental limits on what could be achieved by looking solely at outputs: it was shown that not even regular languages can be identified given only positive examples [37]. Nevertheless, with applications ranging from speech recognition to computational biology, grammatical inference is an active and vibrant field [26, 27].

Fuzzing. Obtaining input grammars of programs has been heavily pursued by the *fuzzing* community [58, 85] for use in *grammar-based fuzzing* [4, 42]. 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 [10, 36]. This has some theoretical limits [2, 3] 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 [44] or observing character accesses of input strings by tracking dynamic control flow [38]. 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 can not 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

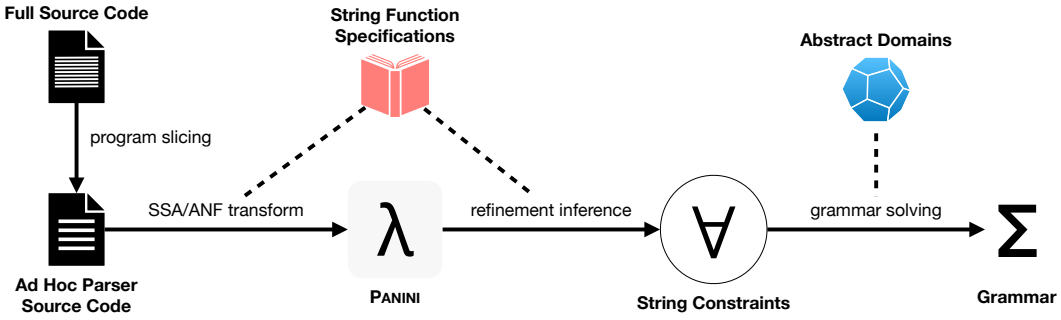


Fig. 2. Overview of our end-to-end grammar inference system.

and solving attendant combinatorial problems [1]. It is usually assumed that collecting string constraints requires some kind of (dynamic) symbolic execution [50], and practical SCS applications are generally concerned with the inverse of our problem: modeling the possible strings a function can return or express [16], 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 Section 3.1), which we then simplify/solve in ways not dissimilar but nonetheless different from traditional SCS techniques (Section 3.2). We do make use of traditional SCS behind the scenes, e.g., as part of the string theories embedded in the Z3 constraint solver [11].

Precondition Inference. Input grammars are essentially preconditions for parsers, thus grammar inference can be viewed as a special case of *precondition inference*. But despite a wide variety of approaches for computing preconditions [6, 23, 29, 67, 78], 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.

3 APPROACH

Figure 2 shows an overview of the envisioned end-to-end grammar inference system. In the first steps, ad hoc parsing code is transformed from a Turing-complete source language (e.g., Python) into an intermediate representation called PANINI, essentially a domain-specific language for parsing. This transformation can be seen as a simplification: we remove syntactic sugar, make control flow explicit, and throw away all parts of the source code that are not related to parsing. During this step, calls to known string processing functions are translated into applications of one or more equivalent functions of the intermediate language that precisely model the semantics of the source. The carefully designed typing rules admitted by the intermediate language, in conjunction with these string function specifications, then allow us to infer the parser's implicit string constraints (i.e., preconditions over its input argument). These constraints are initially big and unwieldy, and need to be simplified in a final step that we call "grammar solving." Here we use our knowledge of the common structure of parsing code (which tends to be written in a top-down, recursive descent, $LL(1)$ style), together with again carefully constructed abstract domains (to precisely but efficiently model the constraints' variables), to minimize the string constraints into a form that is useful for our downstream applications, i.e., into a grammar.

In the remainder of this section I will give a brief technical introduction to the PANINI language, sketch our grammar solving approach by presenting some simple examples, and end with a discussion of future work.

3.1 The Panini Language

Our approach is centered around Panini,[5] an intermediate representation of ad hoc parser code. It is a small λ-calculus in A-normal form (ANF) [13, 32] that is solely intended for type synthesis. Panini programs are neither meant to be executed nor written by hand. Ad hoc parser source code, written in a general-purpose programming language like Python, is first transformed into static single assignment (SSA) form [14] and then into a Panini program via an SSA-to-ANF transformation [17]. The only thing not auto-generated are specifications for source library functions, which have to be provided (once) for each source language in the form of axiomatic type signatures (cf. Figure 3).

Panini has a refinement type system in the *Liquid Types* tradition [72, 81]. The base types $\mathbb{1}$ (unit), $\mathbb{B}$ (Booleans), $\mathbb{Z}$ (integers), and $\mathbb{S}$ (strings) are decorated with predicates in a logic decidable using *satisfiability modulo theories* (SMT) [9], specifically quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) [8] extended with a theory of operations over strings [11]. For example, the type of natural numbers $\mathbb{N}$ can be defined as

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

Dependent function types allow output types to refer to input types and enable precise specification of, e.g., the length function for strings

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

Subtyping is possible, with function types decomposing into contra-variant inputs and co-variant outputs, and reduces into entailment constraints, e.g.,

$$\{x : \mathbb{Z} \mid x > 42\} \leqslant \mathbb{N} \dashv \forall x.\, x > 42 \Rightarrow x \geq 0,$$

which can be read as "the type of integers greater than forty-two is a subtype of the natural numbers if there is a proof that for all integers being greater than forty-two implies being greater than or equal to zero." These and other constraints produced in the course of type synthesis are called *verification conditions* (VCs) [64]. The validity of the VCs implies that the synthesized types are a correct specification of the program. VCs can be discharged by most off-the-shelf SMT solvers; we currently use Z3 [28].

Panini was heavily inspired by the Sprite tutorial language by Jhala and Vazou [47], and incorporates ideas from various other systems [21, 30, 63]. Notably, we use the Fusion algorithm by Cosman and Jhala [21] to enable 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.

Generating and discharging VCs is complicated by the presence of κ *variables* (also called "Horn variables" [47] or "liquid type variables" [72] in the literature) denoting unknown refinements. These arise naturally as part of type synthesis, e.g., to allow information to flow between intermediate terms, or if the user explicitly adds a *refinement hole* to a type signature. Before a VC can be discharged, all of its κ variables need to be replaced by concrete refinement predicates and it is generally desirable to find the strongest satisfying assignments for all κ variables given the overall constraints. The Fusion algorithm is state-of-the-art in this respect, and can usually find the strongest solutions for all κ variables in most programs.

Unfortunately, for κ variables denoting the input string arguments of parsers (as in the examples given below), Fusion falls short, at least for our purposes. It lacks the necessary domain knowledge of string operations to infer suitably precise grammar-like string constraints. In the next section, we describe our approach to fill this gap.

[5] Named after the ancient Indian grammarian Pāṇini [12], as well as the delicious Italian sandwiches.

$$\begin{aligned}
&\text{assert} : \{b : \mathbb{B} \mid b\} \to \mathbb{1} \\
&\text{equals} : (a : \mathbb{Z}) \to (b : \mathbb{Z}) \to \{c : \mathbb{B} \mid c \Leftrightarrow a = b\} \\
&\text{length} : (s : \mathbb{S}) \to \{n : \mathbb{N} \mid n = |s|\} \\
&\text{charAt} : (s : \mathbb{S}) \to \{i : \mathbb{N} \mid i < |s|\} \to \{t : \mathbb{S} \mid t = s[i]\} \\
&\text{match} : (s : \mathbb{S}) \to (t : \mathbb{S}) \to \{b : \mathbb{B} \mid b \Leftrightarrow s = t\}
\end{aligned}$$

Fig. 3. Panini specifications of standard functions.

3.2 Solving Grammar Constraints

Our premise is that the grammar of an ad hoc parser is equivalent to the most precise refinement type of the parser's input string argument and that by synthesizing the refinement type, we can infer the grammar. In order to synthesize this type, we ultimately need to find the strongest solution for the κ variable denoting the unknown string refinement.

Example 1: Assertion. To illustrate, consider the Python expression

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

We can transform it to the following Panini equivalent, assuming s to be the string whose grammar we wish to infer:

$$\begin{aligned}
&\boldsymbol{\lambda}(s : \mathbb{S}). \\
&\quad \textbf{let } x = \text{charAt } s\ 0 \textbf{ in} \\
&\quad \textbf{let } p = \text{match } x\ \texttt{"a"} \textbf{ in} \\
&\quad \text{assert } p
\end{aligned}$$

Given the specifications for charAt, match, and assert from Figure 3, we can infer the whole expression to have the top-level type

$$\{s : \mathbb{S} \mid \kappa(s)\} \to \mathbb{1}$$

under the incomplete verification condition

$$\begin{aligned}
&\forall s.\ \kappa(s) \Rightarrow \\
&\quad 0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \\
&\qquad \forall p.\ p \Leftrightarrow x = \texttt{"a"} \Rightarrow \\
&\qquad\quad p
\end{aligned}$$

In order to complete the VC, we have to find an appropriate assignment for κ, which must take the form of a single-argument function refining/constraining the string s. It is clear that choosing $\kappa(s) \mapsto \text{true}$, i.e., allowing *any* string for s, is not a valid solution because it does not satisfy the VC. On the other hand, choosing $\kappa(s) \mapsto \text{false}$ trivially validates the VC, but it implies that the function could never actually be called, as no string satisfies the predicate false.

One possible assignment could be $\kappa(s) \mapsto s = \texttt{"a"}$, 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 $\kappa(s) \mapsto s[0] = \texttt{"a"}$, which ensures that the first character of the string is "a" but leaves the rest of the string unconstrained. Translated into a grammar, this could be written as $s \to \texttt{a}\Sigma^*$, where Σ is any letter from the alphabet. Note that the solution is a minimized version of the consequent in the VC.

Example 2: Conditional. To hint at a general procedure for finding κ, based on systematically minimizing the VC consequent, let's look at a slightly more complex parser:

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

Here is the equivalent Panini program, alongside the top-level VC:

$$
\begin{array}{ll}
\boldsymbol{\lambda}(s : \mathbb{S}). & \forall s.\ \kappa(s) \Rightarrow \\
\quad \textbf{let } x = \text{charAt } s\ 0 \textbf{ in} & \quad 0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \\
\quad \textbf{let } p_1 = \text{match } x\ \texttt{"a"} \textbf{ in} & \quad\quad \forall p_1.\ p_1 \Leftrightarrow x = \texttt{"a"} \Rightarrow \\
\quad \textbf{if } p_1 \textbf{ then} & \quad\quad\quad (p_1 \Rightarrow \\
\quad\quad \textbf{let } n = \text{length } s \textbf{ in} & \quad\quad\quad\quad \forall n.\ n \geq 0 \wedge n = |s| \Rightarrow \\
\quad\quad \textbf{let } p_2 = \text{equals } n\ 1 \textbf{ in} & \quad\quad\quad\quad\quad \forall p_2.\ p_2 \Leftrightarrow n = 1 \Rightarrow \\
\quad\quad \text{assert } p_2 & \quad\quad\quad\quad\quad\quad p_2) \\
\quad \textbf{else} & \quad\quad\quad \wedge\ (\neg p_1 \Rightarrow \\
\quad\quad \textbf{let } y = \text{charAt } s\ 1 \textbf{ in} & \quad\quad\quad\quad 1 < |s| \wedge \forall y.\ y = s[1] \Rightarrow \\
\quad\quad \textbf{let } p_3 = \text{match } y\ \texttt{"b"} \textbf{ in} & \quad\quad\quad\quad\quad \forall p_3.\ p_3 \Leftrightarrow y = \texttt{"b"} \Rightarrow \\
\quad\quad \text{assert } p_3 & \quad\quad\quad\quad\quad\quad p_3)
\end{array}
$$

Our goal now is to find a precise assignment for κ that makes the VC valid. Since we are free to choose any predicate for κ, we could just take the entirety of the consequent following the top-level implication, i.e., $\kappa(s) \mapsto 0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \ldots$, creating a trivially valid tautology. But clearly this solution is practically useless: we want a succinct predicate in a grammar-like form, but the raw VC consequent 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 the consequent as a starting point and systematically apply truth-preserving transformations until we arrive at a predicate that is logically equivalent but syntactically smaller. To demonstrate, Figures 4a to 4f show a part of the consequent (corresponding to the else-branch in the Panini program) as an abstract syntax tree that is iteratively rewritten from the bottom up until a stable disjunctive normal form (DNF) is reached. The rewrite steps shown are:

- (4a → 4b) Simplify the $\forall p_3$ branch by applying the equivalence $(a \Leftrightarrow b) \Rightarrow a \equiv b$.
- (4b → 4c) Turn the implication into a disjunction by applying $a \Rightarrow b \equiv \neg a \vee (a \wedge b)$, an information-preserving version of the classic Boolean conditional equivalence.
- (4c → 4d) Eliminate the $\forall$-quantifier by resolving all equations involving the quantified variable, thereby eliminating it from the formula. Practically, this means replacing all uses of the quantified variable, separately in each branch, with the meet of all constant definitions of the variable in the respective branch.
- (4d → 4e) Distribute $\wedge$ over $\vee$.
- (4e → 4f) Cut the branch containing a contradiction.

By continuing this process further up the tree, we eventually reach a minimal DNF version of the whole VC consequent, and thus a valid assignment for κ,

$$\kappa(s) \mapsto s = \texttt{"a"} \vee (s[0] \neq \texttt{"a"} \wedge s[1] = \texttt{"b"}).$$

Since this solution is a simple quantifier-free predicate over the input string, we can translate it directly into grammar form:

$$s \to \texttt{a} \mid (\Sigma \backslash \texttt{a})\texttt{b}\Sigma^*$$

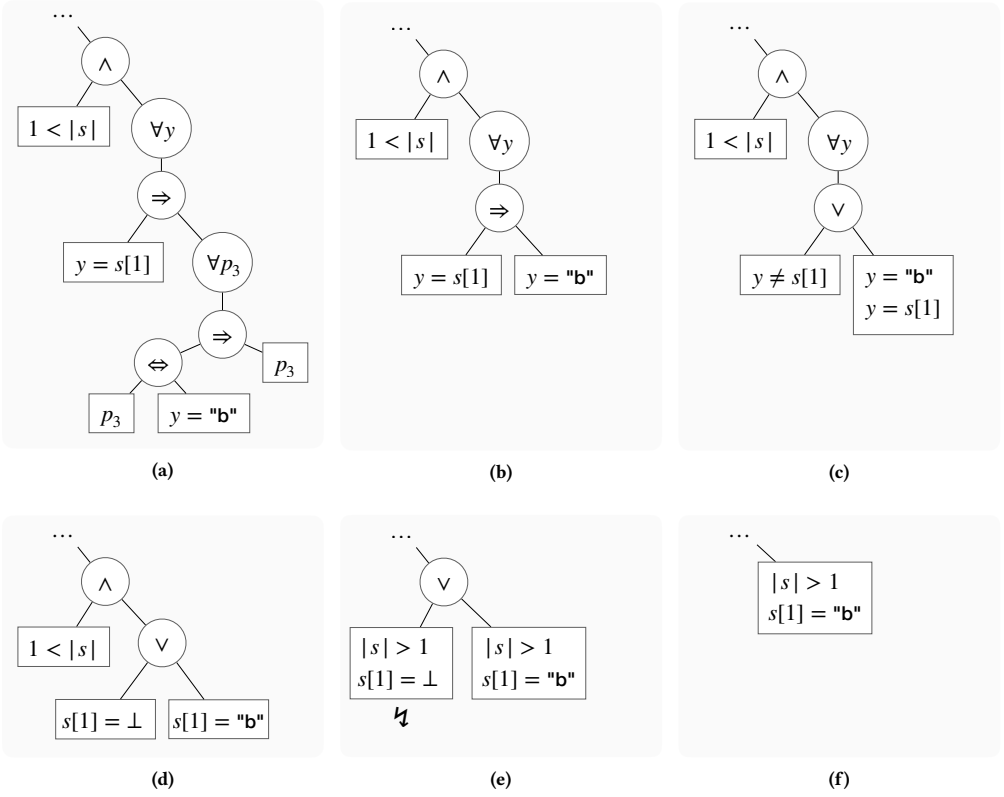


Fig. 4. Successive applications of rewrite rules to a part of the VC consequent from Example 2.

We have an operational definition of our κ-solving approach in our prototype implementation, based on a preliminary set of rewrite rules, and are currently working on a formal treatment. We are also working on a foundation of *abstract domains* to efficiently but precisely model the potentially infinite sets of numerical and string values that arise during constraint solving. For our purposes, we want to be able to cheaply switch between purely symbolic representations that allow easy syntactic manipulation (e.g., $s[0] = \texttt{"a"} \wedge |s| > 1$) and abstract models that enable algebraic operations over (infinite) values (e.g., $\mathsf{a}\Sigma^{[0..\infty]} \sqcap \Sigma^{[2..\infty]} = \mathsf{a}\Sigma^{[1..\infty]}$).

3.3 Future Work

My primary hypothesis is that we can infer accurate grammars for ad hoc parsers using a framework of syntax-driven refinement type synthesis that incorporates domain-specific knowledge of parsing. I intend to prove the general soundness of my approach and to demarcate its theoretical limits. More practically, I intend to provide an implementation of the Panini language together with an evaluation suite based on a corpus of curated ad hoc parser samples from the real world.

To reach the ultimate goal of automatic end-to-end grammar inference, a number of additional technical problems need to be solved:

- extracting the relevant parts of the initial source code (e.g., using program slicing [83]);
- ensuring source function specifications are correct (ideally in a mechanized way);
- preserving precise source location information to allow traceability of grammar productions;

- and transforming the simplified refinement predicates (the "grammar constraints") into representations that facilitate grammar comparisons [57] and can be shown in familiar form, e.g., ABNF [24] or railroad diagrams [15].

To demonstrate the practical viability of the system, I plan on building prototypes of at least some of the proposed applications (Section 1.3) and to conduct a large-scale mining study of inferred grammars.

4 ROADMAP

My research so far has resulted in the publication of two papers [75, 76] and the presentation of an extended abstract [77]. As I continue working on the implementation of the end-to-end grammar inference system described in this proposal, I am planning and preparing papers detailing the core technical contributions of my work (the Panini language and type system, and its novel approach to grammar solving); prototypical implementations of concrete useful applications arising from these contributions (cf. Section 1.3); and related empirical and user studies. A rough timeline of published and planned work is given in Table 1 below.

Additionally, we are currently preparing a DFG/FWF Joint Project grant application for a followup project building on and extending my research, with particular applications in the area of software testing. The targeted submission window is spring 2023.

Table 1. Timeline of past and future publications

ICSE-NIER 2022 Vision Paper	Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers Michael Schröder and Jürgen Cito [76]
TyDe 2022 Extended Abstract	Toward Grammar Inference via Refinement Types Michael Schröder and Jürgen Cito [77]
SPLASH 2022 Doctoral Symposium	Grammar Inference for Ad Hoc Parsers Michael Schröder [75]
MSR / EMSE 2023 Registered Report	*An Exploratory Study of Ad Hoc Parsers in the Wild* planned registration in February 2023
OOPSLA 2023 Research Paper	*Grammar Inference via Refinement Types* planned submission in April 2023
SCAM 2023 Library Paper	*Provenance-Preserving Source-To-ANF Translation* tentative submission in April 2023
ASE 2023 Tool Paper	*An IDE Plugin for Interactive Grammar Documentation* tentative submission in May 2023
ICSE 2024 Research Paper	*A Grammar-Aware Code Review Bot* planned submission in August 2023
Winter 2023	**PhD Thesis Submission**

REFERENCES

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

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

[3] 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

[4] 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

[5] Sebastian Baltes and Stephan Diehl. 2014. Sketches and Diagrams in Practice. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. 530–541.

[6] 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

[7] Earl T. Barr, David W. Binkley, Mark Harman, and Mohamed Nassim Seghir. 2019. Sub-Turing Islands in the Wild. (2019). arXiv:1905.12734 [cs.PL]

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

[9] 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.

[10] 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

[11] 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

[12] 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.

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

[14] 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

[15] Lisa M Braz. 1990. Visual syntax diagrams for programming language statements. *ACM SIGDOC Asterisk Journal of Computer Documentation* 14, 4 (1990), 23–27. https://doi.org/10.1145/97435.97987

[16] 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

[17] 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

[18] Noam Chomsky. 1957. *Syntactic Structures*. Mouton & Co. 117 pages.

[19] Noam Chomsky. 1959. On Certain Formal Properties of Grammars. *Information and Control* 2, 2 (1959), 137–167.

[20] Ravi Chugh, Brian Hempel, Mitchell Spradlin, and Jacob Albers. 2016. Programmatic and Direct Manipulation, Together at Last. 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, 341–354. https://doi.org/10.1145/2908080.2908103

[21] 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

[22] Patrick Cousot and Radhia Cousot. 2002. Systematic Design of Program Transformation Frameworks by Abstract Interpretation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL '02)*. ACM, New York, NY, USA, 178–190. https://doi.org/10.1145/503272.503290

[23] 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

[24] D. Crocker and P. Overell. 2008. *Augmented BNF for Syntax Specifications: ABNF*. STD 68. RFC Editor. http://www.rfc-editor.org/rfc/rfc5234.txt

[25] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why Aren't Regular Expressions a Lingua Franca? An Empirical Study on the Re-Use and Portability of Regular Expressions. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, 443–454. https://doi.org/10.1145/3338906.3338909

[26] Colin de la Higuera. 2005. A bibliographical study of grammatical inference. *Pattern Recognition* 38, 9 (2005), 1332–1348. https://doi.org/10.1016/j.patcog.2005.01.003

[27] Colin de la Higuera. 2010. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press. https://doi.org/10.1017/CBO9781139194655

[28] 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

[29] 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

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

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

[32] 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

[33] Bryan Ford. 2004. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Venice, Italy) *(POPL '04)*. 111–122.

[34] Isabel García-Contreras, José F. Morales, and Manuel V. Hermenegildo. 2016. Semantic code browsing. *Theory and Practice of Logic Programming* 16, 5-6 (2016), 721–737. https://doi.org/10.1017/S1471068416000417

[35] 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.

[36] 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

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

[38] 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

[39] Dick Grune and Ceriel J. H. Jacobs. 2008. *Parsing Techniques* (2nd ed.). Springer, New York, NY. https://doi.org/10.1007/978-0-387-68954-8

[40] Quinn Hanam, Ali Mesbah, and Reid Holmes. 2019. Aiding Code Change Understanding with Semantic Change Impact Analysis. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME 2019)*. 202–212. https://doi.org/10.1109/ICSME.2019.00031

[41] Jeffrey Heinz and Jos M. Sempere. 2016. *Topics in Grammatical Inference* (1st ed.). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-48395-4

[42] 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.

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

[44] 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

[45] Edgar T. Irons. 1983. A Syntax Directed Compiler for ALGOL 60. *Commun. ACM* 26, 1 (Jan. 1983), 14–16.

[46] Gerhard Jäger and James Rogers. 2012. Formal language theory: refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences* 367, 1598 (2012), 1956–1970.

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

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

[49] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. Do code clones matter?. In *Proceedings of the 31st International Conference on Software Engineering* (Vancouver, Canada) *(ICSE '09)*. 485–495. https://doi.org/10.1109/ICSE.2009.5070547

[50] 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

[51] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[52] Leslie Lamport. 2015. Who builds a house without drawing blueprints? *Commun. ACM* 58, 4 (2015), 38–41.

[53] 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

[54] T.C. Lethbridge, J. Singer, and A. Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software* 20, 6 (2003), 35–39. https://doi.org/10.1109/MS.2003.1241364

[55] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. *PACM on Programming Languages* 4, ICFP, Article 109 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408991

[56] 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

[57] Ravichandhran Madhavan, Mikaël Mayer, Sumit Gulwani, and Viktor Kuncak. 2015. Automating Grammar Comparison. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 183–200. https://doi.org/10.1145/2814270.2814304

[58] 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]

[59] Mikaël Mayer, Viktor Kuncak, and Ravi Chugh. 2018. Bidirectional Evaluation with Direct Manipulation. *PACM on Programming Languages* 2, OOPSLA, Article 127 (Oct. 2018), 28 pages. https://doi.org/10.1145/3276497

[60] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. 2020. Regexes Are Hard: Decision-Making, Difficulties, and Risks in Programming Regular Expressions. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 415–426. https://doi.org/10.1109/ASE.2019.00047

[61] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-Based Semantic Code Search over Partial Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Tucson, Arizona, USA) *(OOPSLA '12)*. ACM, New York, NY, USA, 997–1016. https://doi.org/10.1145/2384616.2384689

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

[63] 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

[64] 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.

[65] Alexander Okhotin. 2004. Boolean grammars. *Information and Computation* 194, 1 (2004), 19–48.

[66] Alexander Okhotin. 2013. Conjunctive and Boolean grammars: the true general case of the context-free grammars. *Computer Science Review* 9 (2013), 27–59.

[67] 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

[68] 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

[69] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. 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, 522–538. https://doi.org/10.1145/2908080.2908093

[70] Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. ACM, New York, NY, USA, 1066–1082. https://doi.org/10.1145/3385412.3386001

[71] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. 2004. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *Proceedings of the 20th IEEE International Conference on Software Maintenance* (Chicago, IL, USA) *(ICSM 2004)*. 188–197. https://doi.org/10.1109/ICSM.2004.1357803

[72] 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

[73] Len Sassaman, Meredith L. Patterson, Sergey Bratus, and Michael E. Locasto. 2013. Security Applications of Formal Language Theory. *IEEE Systems Journal* 7, 3 (2013), 489–500. https://doi.org/10.1109/JSYST.2012.2222000

[74] Joern Schneeweisz. 2020. *How to exploit parser differentials.* Retrieved July 16, 2021 from https://about.gitlab.com/blog/2020/03/30/how-to-exploit-parser-differentials/

[75] Michael Schröder. 2022. Grammar Inference for Ad Hoc Parsers. In *Companion Proceedings of the 2022 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Auckland, New Zealand) *(SPLASH Companion 2022)*. Association for Computing Machinery, New York, NY, USA, 38–42. https://doi.org/10.1145/3563768.3565550

[76] Michael Schröder and Jürgen Cito. 2022. Grammars for Free: Toward Grammar Inference for Ad Hoc Parsers. In *2022 IEEE/ACM 44th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)* (Pittsburgh, PA, USA). 41–45. https://doi.org/10.48550/arXiv.2202.01021

[77] Michael Schröder and Jürgen Cito. 2022. Toward Grammar Inference via Refinement Types (Extended Abstract). Unpublished. Presented at *TyDe '22: 7th ACM SIGPLAN International Workshop on Type-Driven Development* (Ljubljana, Slovenia).

[78] 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

[79] Armando Solar-Lezama. 2008. *Program Synthesis by Sketching.* Ph. D. Dissertation. UC Berkeley.

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

[81] 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

[82] 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.

[83] 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

[84] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural Detection of Semantic Code Clones via Tree-Based Convolution. In *Proceedings of the 27th International Conference on Program Comprehension* (Montreal, Quebec, Canada) *(ICPC '19)*. IEEE Press, 70–80. https://doi.org/10.1109/ICPC.2019.00021

[85] 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/