Grammar Inference for Ad Hoc Parsers

Michael Schröder
TU Wien
Vienna, Austria
michael.schroeder@tuwien.ac.at

Abstract
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. We propose a grammar inference system that allows 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, we introduce PANINI, an intermediate representation with a novel refinement type system that incorporates domain knowledge of ad hoc parsing.

CCS Concepts: • Theory of computation → Grammars and context-free languages; Program analysis.

Keywords: grammars, ad hoc parsers, refinement types

ACM Reference Format:

1 Motivation
Ad hoc parsers are pieces of code that use common string functions like split, trim, or slice to effectively perform parsing: “the process of structuring a linear representation in accordance with a given grammar” [29]. But they do so without employing any formal parsing techniques, such as combinator frameworks [38] or parser generators [35, 49]; the “given grammar” remains entirely implicit.

The Python expression in Figure 1 is a typical example of an ad hoc parser. It turns a string of comma-separated numbers into a list of integers. 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. Commonly, this kind of parsing code is deeply entangled with application logic, a phenomenon known as shotgun parsing [45].

Figure 1 also demonstrates our vision of grammar inference. In the same way that type inference allows programmers to generally omit type annotations because they can be automatically recovered from the surrounding context, grammar inference lets programmers recover the implicit input grammars of their ad hoc parsers. A parser without an explicit 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.

The grammar in Figure 1 immediately reveals a great deal about a deceptively simple looking expression, e.g., that the empty string is not a valid input (it will in fact crash the program) or that single _ characters can be used for grouping digits. Grammars are finite but complete formal descriptions of all values an input string can have without the program going wrong. They help assure us that our input languages have favorable properties and our parsers do not contain otherwise hidden features or bugs. The language-theoretic community regards grammars as vital in assuring the correctness and safety of input handling routines [54, 55].

We propose an end-to-end grammar inference system [56] (Figure 2) that would allow programmers to get input grammars from unannotated ad hoc parser source code “for free.” This enables a range of exciting new possibilities:

- **Interactive Documentation** that is closely linked to the underlying code and always up-to-date [39] (Figure 1).
- **Bi-directional Parser Synthesis**, combining grammar inference with parser generation to enable grammar-based program transformations [17], program sketching [40, 50, 58], and live bi-directional programming [15, 43].
- **Grammar Mining & Learning**, which allows us to detect parser code clones [36, 61], enhance semantic code search [26, 44, 51], and add grammar-awareness to semantic change tracking [30, 52] (Figure 3).
2 Problem

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 [20, 21, 31].

Obtaining input grammars of programs has been heavily pursued by the fuzzing community [42, 62] for use in grammar-based fuzzing [4, 32]. Black-box approaches try to infer a language model by poking the program with seed inputs and monitoring its runtime behavior [8, 27]. 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 use techniques like taint tracking to monitor data flow between variables [33] or observing character accesses of input strings [28]. 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. They also do not provide any guarantees about the accuracy of the inferred grammars.

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 [1]. However, collecting string constraints again usually requires (dynamic) symbolic execution [37], and practical SCS applications are generally concerned with the inverse of our problem: modeling the possible strings a function can return or express [13], instead of the strings a function can accept.

We view grammar inference as a special case of precondition inference. But despite a wide variety of approaches for computing preconditions [5, 18, 23, 48, 57], we are not aware of any that focus specifically on string operations, or that would allow us to reconstruct an input grammar.

3 Approach

3.1 The Panini Language

Our approach is centered around Panini, an intermediate representation of ad hoc parser code. It is a small λ-calculus

\[(\lambda \rightarrow SCS), a declarative paradigm\]

\[\text{Ad Hoc Parser}\]

\[\text{Source Code}\]

\[\text{Intermediate}\]

\[\text{Representation}\]

\[\text{Language Model}\]

\[\text{Grammar Visualization}\]

\[\text{Simplification}\]

\[\text{Inference}\]

\[\text{Synthesis}\]

\[\text{Source Function Specifications}\]

\[\text{Figure 2. End-to-end grammar inference.}\]

\[\text{⚠ Merging #420 (6a36b23) into main (224b18b) will change the input grammar of a function.}\]

\[\text{Before:}\]

\[\text{baz → a*bΣ*}\]

\[\text{After:}\]

\[\text{baz → Σa*bΣ*}\]

\[\text{Figure 3. Grammar-aware semantic change tracking: a code review bot informs the programmer that a recent commit has introduced a change in input grammar.}\]

\[\text{In A-normal form (ANF) [25] that is solely intended for type synthesis, Panini programs are neither meant to be executed nor written by hand. Ad hoc parser source code, written in a general-purpose programming language like Python, is first transformed into static single assignment (SSA) form [11] and then into a Panini program via an SSA-to-ANF transformation [14].}\]

\[\text{Panini has a refinement type system in the Liquid Types tradition [53, 59]. Base types like int or string are decorated with predicates in a logic decidable using satisfiability modulo theories (SMT) [7], specifically quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) [6] extended with a theory of operations over strings [9]. For example, }\{(v : int \mid v ≥ 0)\} \text{ is the type of natural numbers and }\{(s : string) → (v : int \mid v ≥ 0 ∧ v = |s|)\} \text{ is a dependent function type whose outputs can refer to input types. Type synthesis generates verification conditions (VCs) [47], which are constraints in the refinement logic whose validity implies that the synthesized types are a correct specification of the program. VCs can be discharged by most off-the-shelf SMT solvers; we currently use Z3 [22].}\]

\[\text{We based Panini on the Sprite tutorial language by Jhala and Vazou [34], and incorporated ideas from various other systems [16, 24, 46]. Notably, we use the Fusion algorithm by Cosman and Jhala [16] to enable inference of the most precise local refinement type for all program statements, without requiring any prior type annotations except for library functions. Another advantage of the Fusion approach is the preservation of scoping structure, yielding VCs that more closely match the original program structurally.}\]

\[\text{VCs might initially contain } k \text{ variables denoting unknown refinements. These arise naturally as part of type synthesis, e.g., to allow information to flow between intermediate terms, but can also be added explicitly as refinement holes. Before discharging a VC, all of its } k \text{ variables need to be replaced by concrete refinement predicates. It is generally desirable to find the strongest satisfying assignments for all } k \text{ variables given the overall constraints.}\]
\( \lambda s. \quad \forall s, \ k_0(s) \Rightarrow \)

\[
\begin{align*}
\text{let } x &= \text{charAt} s \ 0 \ \text{in} \quad 0 < |s| \land \forall x. \ x = s[0] \Rightarrow \\
\text{let } p_1 &= \text{match} x \ "a" \ \text{in} \quad \forall p_1. \ p_1 \iff x = "a" \Rightarrow \\
\text{if } p_1 \ \text{then} \quad (p_1 \Rightarrow \\
\text{let } n &= \text{length} s \ \text{in} \quad \forall n. \ n \geq 0 \land n = |s| \Rightarrow \\
\text{let } p_2 &= \text{equals} n \ 1 \ \text{in} \quad \forall p_2. \ p_2 \iff n = 1 \Rightarrow \\
\text{assert } p_2 \\
\text{else} \quad \land (\neg p_1 \Rightarrow \\
\text{let } y &= \text{charAt} s \ 1 \ \text{in} \quad 1 < |s| \land \forall y. \ y = s[1] \Rightarrow \\
\text{let } p_3 &= \text{match} y \ "b" \ \text{in} \quad \forall p_3. \ p_3 \iff y = "b" \Rightarrow \\
\text{assert } p_3 \\
\text{end} &
\end{align*}
\]

\[ s[0] = x \]
\[ (p_1 \land s[0] = "a") \lor (\neg p_1 \land s[0] \neq "a") \]

\[ (s = "a"") \lor (s[0] \neq "a" \land s[1] = "b") \]

\[ s = "a" \lor (s[0] \neq "a" \land s[1] = "b") \]

Figure 4. A Panini program (left), its verification condition (middle), and a derivation of \( k_0 \) (right).

### 3.2 Grammar Inference

To infer a parser’s input grammar, we need to find the most precise solution for the \( \kappa \) variable representing the refinement of the parser’s input string argument. Consider the following simple parser:

\[
\begin{align*}
\text{if } s[0] &= "a" : \\
\text{assert } \text{len}(s) &= 1 \\
\text{else:} \\
\text{assert } s[1] &= "b"
\end{align*}
\]

After we have descended into all quantifiers and implications, resolved all names, and simplified all equations, we arrive at the final assignment

\[ k_0(s) = (s = "a") \lor (s[0] \neq "a" \land s[1] = "b") \]

which can be equivalently written in grammar form as

\[ s \rightarrow a | (\Sigma \backslash a) b \Sigma^* \]

### 4 Methodology

Our 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. We intend to prove the soundness of our approach and to demarcate its limits. More practically, we will provide an implementation of the Panini language.

Our ultimate goal is an end-to-end grammar inference system (Figure 2). This necessitates solving a number of additional technical problems surrounding the inference step: extracting the relevant parts of the initial source code, e.g., using a form of program slicing [60]; ensuring source function specifications are correct (ideally in a mechanized way); preserving precise source location information to allow traceability of grammar productions; and transforming refinement predicates into representations that facilitate grammar comparisons [41] and can be shown in familiar form, e.g., ABNF [19] or railroad diagrams [12] (we found a graph representation with bounded edge constraints to be promising).

We intend to ensure the effectiveness of our system by evaluating it on a corpus of curated ad hoc parser samples from the real world. Additionally, we plan on building prototypes of at least some of our proposed applications (§1) to demonstrate practical viability. We also intend on conducting a large-scale mining study of inferred grammars, and are currently conducting a user study on grammar comprehension to determine the benefits and drawbacks of different textual and visual grammar representations.