Toward Grammar Inference via Refinement Types

Extended Abstract

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

Jürgen Cito
TU Wien
Vienna, Austria
juergen.cito@tuwien.ac.at

1 Motivation

Ad hoc parsers are pieces of code that use common string functions like split, trim, or slice to effectively perform parsing—that is, "the process of structuring a linear representation in accordance with a given grammar" [13]—without employing any formal parsing techniques or frameworks; the "given grammar" remains entirely implicit.

As an example, here is a Python expression that turns a string of comma-separated numbers into a list of integers:

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

This is a typical ad hoc parser. It could be part of code handling command-line arguments, reading an ad hoc file format, or processing some other kind of user input. Often, this kind of code is deeply entangled with application logic, a phenomenon known as shotgun parsing [17].

The following is the above parser's input grammar, a finite but complete formal description of all values the input string can have without the program going wrong in some way:¹

```
s → int | int , s
int → space* (+ | -)* digit ( _ | digit)* space*
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
space → _ | \t | \n | \v | \f | \r
```

Parts of this grammar might be surprising and not at all obvious from looking at the code alone. And while it certainly takes some expertise to interpret a grammar correctly, it provides benefits beyond mere documentation, e.g., allowing one to automatically generate test inputs [1, 14], or reason about language-theoretic security properties [20].

There is an analogy here with types. Just like types, grammars are a form of specification. 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.

Types have one significant advantage over grammars, however: most type systems offer a form of type inference, allowing programmers to generally omit type annotations because they can be automatically recovered from the surrounding context. Is it possible to infer grammars just like we can infer types?

We believe that it is, and we have previously sketched an end-to-end grammar inference system (Figure 1) that would enable programmers to get input grammars from unannotated ad hoc parser source code "for free." We envision that this kind of static grammar inference enables a whole range of new possibilities, such as interactive documentation, bidirectional parser synthesis, grammar-based code search, and semantic change tracking [21].

In this extended abstract, we report our progress toward grammar inference via refinement types and highlight the challenges and work still ahead of us.

2 The Panini Language

Our approach for automatic grammar inference is centered around an intermediate representation, tentatively titled Panini,² a domain-specific language for parsing. It is a small λ-calculus in Administrative Normal Form (ANF) [12], 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 [5] and then into a Panini program via an SSA-to-ANF transformation [7]. 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 2).

Panini has a refinement type system in the Liquid Types tradition [19, 22]. Base types, like int or string, are decorated with predicates in an SMT-decidable logic, namely quantifier-free linear arithmetic with uninterpreted functions (QF_UFLIA) [2] extended with a theory of operations over strings [3]. For example, \{ v : int \mid v \geq 0 \} is the type of natural numbers and \{ s : string \} \rightarrow \{ v : int \mid v \geq 0 \land v = |s| \}

¹This grammar assumes the semantics of Python 3.9.

²After the ancient Indian grammarian Pāṇini [4], as well as the delicious Italian sandwiches.
assert : \{ b : \text{bool} | b \} \rightarrow \text{unit}  \\
equals: (a : \text{int}) \rightarrow (b : \text{int}) \rightarrow \{ c : \text{bool} | c \leftrightarrow a = b \}  \\
length: (s : \text{string}) \rightarrow \{ n : \text{int} | n \geq 0 \land n = |s| \}  \\
charAt: (s : \text{string}) \rightarrow \{ i : \text{int} | i \geq 0 \land i < |s| \} \rightarrow \{ t : \text{string} | t = s[i] \}  \\
machine: (s : \text{string}) \rightarrow \{ t : \text{string} \rightarrow \{ b : \text{bool} | b \leftrightarrow t = s \} \}

Figure 2. PANINI specifications of standard functions

is a dependent function type whose outputs can refer to input types. Subtyping, e.g., \{ v : \text{int} | v \geq 0 \} \subseteq \text{int}, generates entailment constraints, with function types decomposing into contra-variant inputs and co-variant outputs. These and other constraints produced in the course of type checking/synthesis are known as verification conditions (VCs). Their validity implies that the types are correct, i.e., that the program meets its specification. VC can be discharged by any off-the-shelf SMT solver; we currently use Z3 [10].

The generation and discharge of VCs can be quite involved, especially because the constraints might contain \kappa variables denoting unknown refinements. These arise naturally as part of type checking, e.g., to allow information to flow between intermediate terms, or if the user explicitly adds a refinement hole to a type signature. Various approaches exist to find satisfying assignments for \kappa variables, and it is generally (and in our case particularly) desirable to find the strongest or most precise solution given the overall constraints.

The current implementation of our system is based in large parts on the SmtRtE tutorial language by Jhala and Vazou [15], but incorporating ideas from various other systems [8, 11, 18]. Notably, we use the Fusion algorithm by Cosman and Jhala [8] 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.

As we continue working towards our goal of automatic grammar inference, we expect to make major modifications to our current type inference algorithm (by which we mean the combination of typing rules, predicate simplification procedures, and constraint solver) and are hopefully able to push the state-of-the-art forward.

3 Grammar Inference

The problem of inferring input grammars is equivalent to the problem of inferring precise preconditions for parsing functions. In the context of refinement types, this means finding the most precise solution for the \kappa variable representing the refinement of the input string argument.

Example 3.1. The Python expression

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

can be transformed to the following PANINI equivalent, assuming \(s\) to be the string whose grammar we wish to infer:

```java
λs.
  let x = charAt s 0 in
  let p = match x "a" in
  assert p
```

Given the specifications for `charAt`, `match`, and `assert` from Figure 2, we can infer the whole expression to have the type

\[
\{ s : \text{string} | \kappa_0(s) \} \rightarrow \text{unit}
\]

under the verification condition:

\[
\forall s. \kappa_0(s) \Rightarrow \\
0 < |s| \land \forall x. x = s[0] \Rightarrow \forall p. p \leftrightarrow x = "a" \Rightarrow p
\]

Now we have to find an appropriate assignment for \(\kappa_0\). It is clear that choosing \(\kappa_0(s) \doteq \text{true}\), i.e., allowing any string for \(s\), is not a valid solution because it does not satisfy the constraint. On the other hand, choosing \(\kappa_0(s) \doteq \text{false}\) trivially validates the constraint, but it implies that the function could never be called, as no string satisfies the predicate false. One possible assignment would be

\[
\kappa_0(s) \doteq s = "a",
\]

i.e., only allowing exactly the string "a" as a value for \(s\). While this satisfies the constraint 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.

The correct assignment for \(\kappa_0\) is

\[
\kappa_0(s) \doteq s[0] = "a",
\]

which ensures that the first character of the string is "a" and leaves the rest of the string unconstrained. Translated into a grammar, this could be written as \(s \rightarrow a\Sigma^*\), where \(\Sigma\) is any letter from the alphabet. Note that the solution is a minimized version of the consequent in the VC.

Example 3.2. Let’s look at another simple parser:

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

Figure 3 shows the equivalent PANINI program, alongside the VC for the top-level function type. It also shows how we can derive a precise assignment for \(\kappa_0\) by walking the VC’s top-level consequent, using our domain knowledge of parsing and string operations to minimize predicates.
Toward Grammar Inference via Refinement Types (Extended Abstract)

4 Challenges and Future Work

The challenges and future work are

1. Constructing grammars from predicates.
2. Extracting relevant parts of the initial source code.
3. Ensuring source function specifications are correct.
4. Preserving precise source location information.
5. How expressive does PANINI need to be?

For downstream use, we need to transform the logical predicates describing string preconditions into a representation that allows us to derive grammars in familiar forms (e.g., ABNF [9] or railroad diagrams [6]) and to compare grammars with each other [16]. We are currently exploring a graph representation with bounded edge constraints.

For practical applications, it will be necessary to accurately trace grammar productions back to their origins in the initial source code. As there are quite a number of steps between source and final grammar, including SMT solving, this is far from trivial. We need to ensure that identifier provenance is preserved throughout the whole process.

For what extent these and other language features are actually required mainly depends on the desired scope of our inference, i.e., what constructs are actually needed to represent real-world ad hoc parsers. We are conducting a study to determine this empirically.

For example, the constraint 0 < |s| \(\forall x. x = s[0] \Rightarrow \ldots \)
tells us that s is a string of at least one character and that we can identify this character by the variable x. The string might have more characters, but we know definitely that it has at least this one. So we can make a preliminary assignment \(\kappa_0 \equiv s[0] = x\). We then continue descending into quantifiers and implications, resolving names and simplifying equations, constructing \(\kappa_0\) piece by piece. Constraints of the form

\(\forall p_1. p_1 \iff x = \text{“a”} \Rightarrow \ldots\)

make us branch into two possible worlds: one where the predicate is true and one where its opposite is true. Accordingly, we update our preliminary assignment to

\(\kappa_0 \equiv (p_1 \land s[0] = \text{“a”}) \lor (\neg p_1 \land s[0] \neq \text{“a”})\).

Subsequent constraints may now allow us to further refine and expand each of these branches, or to eliminate some of them altogether if they can never be satisfiable.

Finally, we arrive at the correct assignment

\(\kappa_0(s) \equiv (s = \text{“a”}) \lor (s[0] \neq \text{“a”} \land s[1] = \text{“b”})\),

which can be equivalently written in grammar form as

\(s \rightarrow a \mid (\Sigma \backslash a)b\Sigma^*\).

4 Computing Future Work

- Constructing grammars from predicates.
- Extracting relevant parts of the initial source code.
- Preserving precise source location information.
- How expressive does PANINI need to be?