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 perfom *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:

```
xs = 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]

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

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].

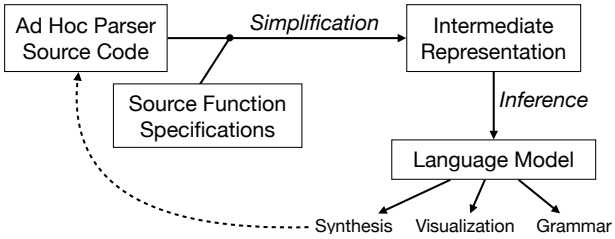

Figure 1. Sketch of a grammar inference system [21]

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 progamming 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, $\{\nu : \mathsf{int} \mid \nu \geq 0\}$ is the type of natural numbers and $(s : \mathsf{string}) \to \{\nu : \mathsf{int} \mid \nu \geq 0 \wedge \nu = |s|\}$

[1] This grammar assumes the semantics of Python 3.9.

[2] After the ancient Indian grammarian Pāṇini [4], as well as the delicious Italian sandwiches.

$$
\begin{aligned}
&\mathit{assert} : \{b : \mathsf{bool} \mid b\} \to \mathsf{unit} \\
&\mathit{equals} : (a : \mathsf{int}) \to (b : \mathsf{int}) \to \{c : \mathsf{bool} \mid c \Leftrightarrow a = b\} \\
&\mathit{length} : (s : \mathsf{string}) \to \{n : \mathsf{int} \mid n \geq 0 \wedge n = |s|\} \\
&\mathit{charAt} : (s : \mathsf{string}) \to \{i : \mathsf{int} \mid i \geq 0 \wedge i < |s|\} \to \\
&\qquad \{t : \mathsf{string} \mid t = s[i]\} \\
&\mathit{match} : (s : \mathsf{string}) \to (t : \mathsf{string}) \to \\
&\qquad \{b : \mathsf{bool} \mid b \Leftrightarrow s = t\}
\end{aligned}
$$

Figure 2. Panini specifications of standard functions

is a dependent function type whose outputs can refer to input types. Subtyping, e.g., $\{\nu : \mathsf{int} \mid \nu \geq 0\} \leqslant \mathsf{int}$, 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. VCs 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 κ *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 κ 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 Sprite 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 κ variable representing the refinement of the input string argument.

Example 3.1. The Python expression

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

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

$$
\begin{aligned}
&\boldsymbol{\lambda} s. \\
&\quad \textbf{let } x = \mathit{charAt}\ s\ 0 \textbf{ in} \\
&\quad \textbf{let } p = \mathit{match}\ x\ \texttt{"a"} \textbf{ in} \\
&\quad \mathit{assert}\ p
\end{aligned}
$$

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

$$\{s : \mathsf{string} \mid \kappa_0(s)\} \to \mathsf{unit}$$

under the verification condition:

$$
\begin{aligned}
&\forall s.\ \kappa_0(s) \Rightarrow \\
&\quad 0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \forall p.\ p \Leftrightarrow x = \texttt{"a"} \Rightarrow p
\end{aligned}
$$

Now we have to find an appropriate assignment for κ_0. It is clear that choosing $\kappa_0(s) \doteq \mathsf{true}$, i.e., allowing *any* string for s, is not a valid solution because it does not satisfy the constraint. On the other hand, choosing $\kappa_0(s) \doteq \mathsf{false}$ 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

$$\kappa_0(s) \doteq s = \texttt{"a"},$$

i.e., only allowing exactly the string "a" as a value for s. While this satisifes 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 fullfil these criteria.

The correct assignment for κ_0 is

$$\kappa_0(s) \doteq s[0] = \texttt{"a"},$$

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 $s \to \mathsf{a}\Sigma^*$, where Σ is any letter from the alphabet. Note that the solution is a minimized version of the consequent in the VC.

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

```
if s[0] == "a":
  assert len(s) == 1
else:
  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 κ_0 by walking the VC's top-level consequent, using our domain knowledge of parsing and string operations to minimize predicates.

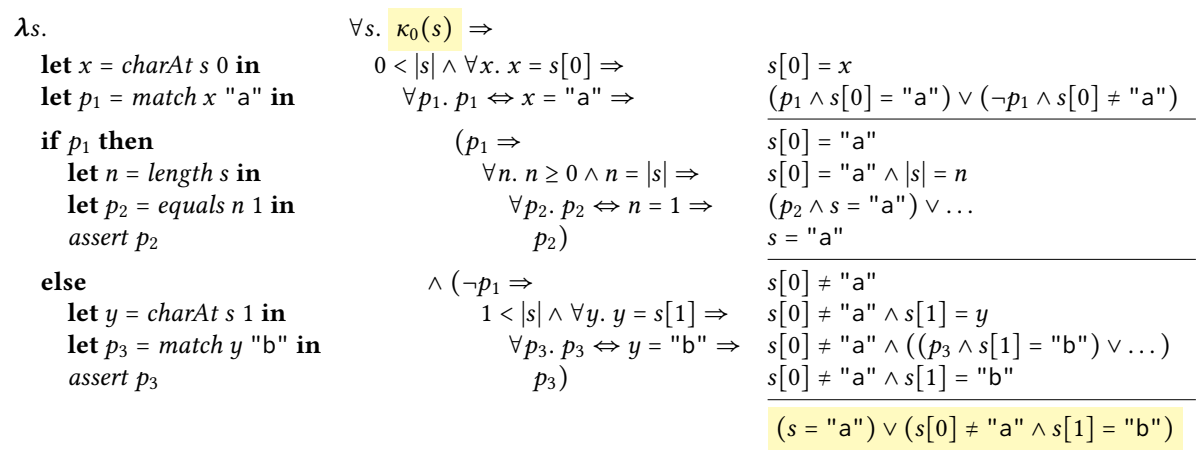

Figure 3. The Panini code for Example 3.2 (left), its verification condition (middle), and a derivation of κ_0 (right)

For example, the constraint $0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \ldots$ 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 $\kappa_0 \cong s[0] = x$. We then continue descending into quantifiers and implications, resolving names and simplifying equations, constructing κ_0 piece by piece. Constraints of the form

$$\forall p_1.\ p_1 \Leftrightarrow x = \texttt{"a"} \Rightarrow \ldots$$

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

$$\kappa_0 \cong (p_1 \wedge s[0] = \texttt{"a"}) \vee (\neg p_1 \wedge s[0] \neq \texttt{"a"}).$$

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

$$\kappa_0(s) \doteq (s = \texttt{"a"}) \vee (s[0] \neq \texttt{"a"} \wedge\ s[1] = \texttt{"b"}),$$

which can be equivalently written in grammar form as

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

4 Challenges and Future Work

- **Computing precise solutions for input strings.** Our goal is a constraint solving algorithm that can derive provably exact descriptions of input strings, up to some (to be determined) language complexity. Experience has shown that the structure of ad hoc parsers closely mirrors the structure of the languages they parse and that humans tend to write small parsers in a top-down, recursive-descent, $LL(1)$ style. We aim to efficiently find and minimize string predicates by exploiting this common structure and other recurring ad hoc parser patterns.
- **Constructing grammars from predicates.** 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.
- **Extracting relevant parts of the initial source code.** During transformation from general-purpose source language to Panini program, we want to extract only the information flows that are related to parsing the input string. Initial explorations have shown a form of program slicing [23] to be applicable here.
- **Ensuring source function specifications are correct.** We need precise specifications of all string functions and other externally defined operations used in Panini programs (cf. Figure 2). While these need to be provided only once per source language/library version, this kind of specification engineering can be cumbersome, and care must be taken to ensure that the axioms accurately reflect reality. We are exploring ways to mechanize this process.
- **Preserving precise source location information.** 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.
- **How expressive does Panini need to be?** Our language implementation is currently lacking features like type polymorphism and recursive data types. To 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.

References

[1] 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

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

[3] 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

[4] 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.

[5] 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

[6] 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

[7] 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

[8] 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

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

[10] 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

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

[12] 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

[13] 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

[14] 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.

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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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