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:
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 (SPLASH Companion '22), December 5–10, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3563768.3565550

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].

Inferred Grammar	Inferred Inputs
$s \rightarrow int \mid int$ `,` s	✗ (empty)
$int \rightarrow space^*\ (\texttt{+} \mid \texttt{-})^?\ digit\ (\texttt{\_}^?\ digit)^*\ space^*$	✓ `1,2,3`
$digit \rightarrow$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`	✓ `10_000,4`
$space \rightarrow$ `␣` | `\t` | `\n` | `\v` | `\f` | `\r`	✓ `+01_2,␣␣3␣`

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

Figure 1. An ad hoc parser and its inferred grammar.

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 security* 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).

SPLASH Companion '22, December 5–10, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9901-2/22/12.
https://doi.org/10.1145/3563768.3565550

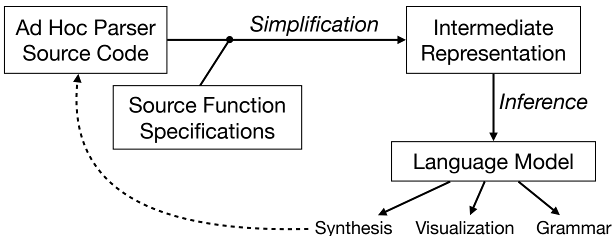

Figure 2. End-to-end grammar inference.

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,[1] an intermediate representation of ad hoc parser code. It is a small λ-calculus 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].

[1]Named after the ancient Indian grammarian Pāṇini [10], as well as the delicious Italian sandwiches.

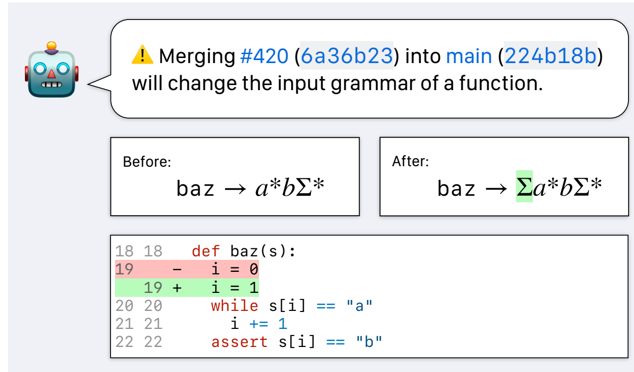

Figure 3. Grammar-aware semantic change tracking: a code review bot informs the programmer that a recent commit has introduced a change in input grammar.

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

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.

VCs might initially contain κ *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 κ variables need to be replaced by concrete refinement predicates. It is generally desirable to find the strongest satisfying assignments for all κ variables given the overall constraints.

$\boldsymbol{\lambda} s.$	$\forall s.\ \kappa_0(s) \Rightarrow$	
let x = charAt s 0 **in**	$0 < \lvert s\rvert \wedge \forall x.\ x = s[0] \Rightarrow$	$s[0] = x$
let p_1 = match x "a" **in**	$\forall p_1.\ p_1 \Leftrightarrow x = \texttt{"a"} \Rightarrow$	$(p_1 \wedge s[0] = \texttt{"a"}) \vee (\neg p_1 \wedge s[0] \neq \texttt{"a"})$
if p_1 **then**	$(p_1 \Rightarrow$	$s[0] = \texttt{"a"}$
let n = length s **in**	$\forall n.\ n \geq 0 \wedge n = \lvert s\rvert \Rightarrow$	$s[0] = \texttt{"a"} \wedge \lvert s\rvert = n$
let p_2 = equals n 1 **in**	$\forall p_2.\ p_2 \Leftrightarrow n = 1 \Rightarrow$	$(p_2 \wedge s = \texttt{"a"}) \vee \ldots$
assert p_2	$p_2)$	$s = \texttt{"a"}$
else	$\wedge\ (\neg p_1 \Rightarrow$	$s[0] \neq \texttt{"a"}$
let y = charAt s 1 **in**	$1 < \lvert s\rvert \wedge \forall y.\ y = s[1] \Rightarrow$	$s[0] \neq \texttt{"a"} \wedge s[1] = y$
let p_3 = match y "b" **in**	$\forall p_3.\ p_3 \Leftrightarrow y = \texttt{"b"} \Rightarrow$	$s[0] \neq \texttt{"a"} \wedge ((p_3 \wedge s[1] = \texttt{"b"}) \vee \ldots)$
assert p_3	$p_3)$	$s[0] \neq \texttt{"a"} \wedge s[1] = \texttt{"b"}$
		$(s = \texttt{"a"}) \vee (s[0] \neq \texttt{"a"} \wedge s[1] = \texttt{"b"})$

Figure 4. A Panini program (left), its verification condition (middle), and a derivation of κ_0 (right).

3.2 Grammar Inference

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

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

Figure 4 shows the equivalent Panini program, alongside the VC for the top-level function type—notice how it closely mirrors the program's structure. On the right, we show how to derive a precise assignment for κ_0, the unknown refinement over the input string s. Our key insight is that humans tend to write small parsers in a top-down, recursive descent, $LL(1)$ style. We can exploit this common structure and walk the VC's top-level consequent to build κ_0 piece by piece, using domain knowledge of string operations to minimize predicates until we satisfy the VC.

We begin with the constraint $0 < |s| \wedge \forall x.\ x = s[0] \Rightarrow \ldots$, which 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 that it definitely has at least this one. So we can make a preliminary assignment $\kappa_0 \cong s[0] = x$.

Next, the constraint $\forall p_1.\ p_1 \Leftrightarrow x = \texttt{"a"} \Rightarrow \ldots$ makes 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

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

As we continue on to subsequent constraints, we may be able to further refine and expand each of these branches, or to eliminate some of them altogether if they can never be satisfiable.

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

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

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.

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] 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

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

[7] 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.

[8] 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

[9] 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

[10] 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.

[11] 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

[12] 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

[13] 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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

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

[20] 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

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

[22] 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

[23] 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

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

[25] 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

[26] 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

[27] 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

[28] 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

[29] 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

[30] 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

[31] 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

[32] 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.

[33] 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

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

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

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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]

[43] 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

[44] 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

[45] 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

[46] 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

[47] 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.

[48] 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

[49] 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

[50] 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

[51] 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

[52] 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

[53] 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

[54] 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

[55] 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/

[56] 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

[57] 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

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

[59] 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

[60] 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

[61] 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

[62] 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/