SATyros: Step-by-Step SAT/SMT Library for a Pedagogical Purpose

Junyoung Jang

Abstract

Satisfiability Modulo Theory (SMT) solvers are one of the most widely adopted tools for constraint problems. Unfortunately, despite their importance, we have a limited number of beginner-friendly interactive materials that explain the principles of SMT solvers. One barrier against providing such materials is the burden of implementation of an interactive SMT solver. This burden inflates more since an instructor who teaches the internal details of SMT solvers often wants to demonstrate the difference between SMT implementation strategies; thus, they need to implement several interactive SMT solvers to produce a tutorial. Here, we present SATyros, a Haskell library to implement interactive SAT/SMT solvers easily. With its API design based on the free monad, SATyros allows instructors to implement interactive SMT solvers in an ideal code structure with a minimum amount of duplicated code across multiple solver implementations. We demonstrate the power of SATyros with two examples of SMT solver implementations and a web tutorial built on those two SMT solvers. The library code and web tutorial are available at `https://github.com/Ailrun/satyros` and `https://Ailrun.github.io/satyros`.

1 Introduction

A Satisfiability Modulo Theory (SMT) problem is the decision/search problem that, first, checks whether there exists an assignment that satisfies given mathematical constraints, and second, computes that satisfying assignment. SMT problems appear in diverse areas: software verification (SLAM [Ball *et al.*, 2004], Spec# [Barnett *et al.*, 2005], KLEE[Cadar *et al.*, 2008], F* [Swamy *et al.*, 2013]), test generation (SAGE [Godefroid *et al.*, 2008]), program synthesis (cvc4sy [Reynolds *et al.*, 2019]), and model checking (Pono [Mann *et al.*, 2021]). Due to this ubiquity, SMT solvers have attracted increasing attention from both industrial and academic researchers. Microsoft has been applied their Z3 SMT solver [de Moura and Bjørner, 2008] to a number of products [Ball *et al.*, 2004; Barnett *et al.*, 2005; Godefroid *et al.*, 2008; Swamy *et al.*, 2013]. There are also many academia-driven SMT solvers such as CVC4 [Barrett *et al.*, 2011], Yices2 [Dutertre, 2014], and MathSAT5 [Cimatti *et al.*, 2013], and possible extensions of those solvers have been actively studied [Reynolds and Blanchette, 2015; Tiwari *et al.*, 2015; Reynolds *et al.*, 2016; Cimatti *et al.*, 2018; Graham-Lengrand *et al.*, 2020].

Nevertheless, there are not so many beginner-friendly materials that explain the principles of SMT solvers. Most of the explanatory materials on SMT are journal or conference articles. Other more beginner-friendly materials either provide only non-interactive static explanations or focus more on the usage of real-world SMT solver implementations than the principles of SMT solvers.

One possible root of this lack of interactive materials is the difficulty of SMT implementations. The implementations of SMT solvers require non-trivial work, especially when we want to allow some user interactions during problem-solving. The situation can be even worse when a tutorial author wants to compare a few different implementation strategies; they need to implement several SMT solvers! This issue brings one idea: what if we can provide a library that makes implementations of interactive SMT solvers simpler based on highly-composable APIs so that a tutorial author, instructor, and other experts can easily write an interactive tutorial?

We introduce SATyros, a Haskell library for SAT/SMT solver implementations, which exactly serves that purpose. SATyros provides

- All essential parts for the implementation of an interactive SMT solver based on a free monad effect system
- A comprehensible set of effects, each of which represents a meaningful atomic step of an SMT execution
- Common components of an SMT solver to ease the implementation

The rest of this article is organized as follows. Section 2 discusses the related work. Section 3 explains background concepts used in the SATyros library. Section 4 introduces

SATYROS and discusses its APIs, including the solvers and effects provided by SATYROS. Section 5 shows two examples of SMT implementations using SATYROS, and we evaluate the effectiveness of SATYROS library. Lastly, Section 6 concludes the discussion and suggests some future extensions of SATYROS.

2 Related Work

CDCL Tutorial [Gritman *et al.*, 2017] provides a web-based interactive tutorial with a rich graphical interface. Their work covers the detail of the Conflict-Driven Clause Learning [Marques-Silva and Sakallah, 1999], one of the most crucial SAT optimizations. However, the tutorial does not cover anything beyond SAT, and the original code of the tutorial is not publicly available. Moreover, the post-compile JavaScript code of the tutorial, which is accessible through the developer consoles of browsers, shows that the naïve DPLL solver and CDCL solver of the tutorial repeats a large portion (around 30%, 100 LOC) of code, which can be even bigger for SMT solvers if the authors implemented them using the same code structure.

Z3 Playground [Morales, 2021] offers a beginner-friendly tutorial for Z3 [de Moura and Bjørner, 2008]. The tutorial explains how to use Z3 to solve some elementary constraint problems with a thorough explanation. However, it does not describe the principles of SMT solvers. In addition, the interactive features of the tutorial are quite limited as it allows users only to run SMT-LIB code and shows the result from Z3.

Z3 wiki [Z3, 2021] presents a wide range of information on Z3, including the link to Z3 Playground [Morales, 2021]. However, it does not introduce any other tutorial than Z3 Playground, and other information is Z3 specific or links to paper articles. CVC4 manual [CVC4, 2021] provides detailed documents on CVC4, but not on general principles of SMT solvers.

3 Background

Operations with *computational effects*, such as stateful operations or IO operations, are essential for any useful programs. For example, the targets of SATYROS, i.e., interactive solvers, are necessarily effectful; they need to stop during computation, respond to user's query for the current state, and undo a step of a computation.

In this section, we explain how to elegantly handle effectful computations with algebraic effects and free monad, as SATYROS use them to handle interactive effects.

Algebraic Effects Since the first semantic definition of computational effects[Moggi, 1991], researchers have been sought a way to construct, combine, reason, and handle effects safely. Among their attempts, the *algebraic effect* introduced by [Plotkin and Power, 2001; Plotkin and Pretnar, 2009], stands out for its extensibility and flexibility.

An algebraic effect system allows programmers to define, raise, and handle effects as in Listing 1. When a program raises an effect, the execution is paused and delegated to an effect handler. This execution is resumed only when the effect handler asks to resume. For example, in Line 24 of Listing 1, the handler does not resume `greetUser`. Thus, the `testApp` function will return without going into `greetUser` again.

This approach allows users to compose effectful functions without a huge boilerplate and to provide multiple handlers that handle only some effects that the users want to handle.

Free Monad One issue of algebraic effect systems is that there are not so many industrial-level programming languages that natively support the systems. In fact, algebraic effect systems that are currently available in industrial-level programming languages are all based on a single language feature called the *monad*[Moggi, 1991]. The monad is a mathematical concept that can explain computational effects, including algebraic effect systems[Plotkin and Pretnar, 2009]. There are many languages, such as Haskell, Scala, and PureScript, that support the monad natively.

However, defining a monad that follows its mathematical laws is often cumbersome and error-prone. The free monad, whose usage is suggested by [Swierstra, 2008], provides a clear distinction between the monad as a lawful mathematical object and as an effect interpreter. This allows users of the free monad to focus on effect definition and effect handlers than the monad laws.

4 SATYROS

In this section, we describe our new library, SATYROS. We first explain the solver components provided SATYROS. Then, we list the available effects for each solver and discuss their meanings.

4.1 Solver Components

The current version (`0.3.1.4`) of SATYROS supports two solver types: boolean satisfiability solver (SAT) and quantifier-free integer difference logic solver (QFIDL).

SAT solver components For a SAT solver implementation, SATYROS provides the following components of a DPLL-based SAT solver[Davis *et al.*, 1962; Marques-Silva and Sakallah, 1999]:

- **Random decision procedure** is a decision procedure to assign a value to an unassigned variable. The current version of SATYROS supports only a random decision procedure.
- **Boolean constraint propagation** is an algorithm to propagate boolean constraints and to detect conflict by checking unit clauses and falsified clauses according to the current assignment.
- **1-UIP conflict-driven clause construction** is one strategy to construct a clause that is derived from a conflict. This strategy searches the first unique implication point (UIP) for the current decision level and uses the UIP to construct a conflict-driven clause.
- **RelSAT conflict-driven clause construction** is another strategy to construct a conflict-driven clause. This strategy uses the current decision assignment (the last UIP) to construct a conflict-driven clause.
- **Chronological backtracking** is a backtracking strategy that does not require a conflict-driven clause. A SAT

Listing 1 Effect definition, raising, and handling in algebraic effect system (pseudo C++-like syntax)

```
// Definitions of two effects occurring in console:
effect WriteOutput(String) = void   // WriteOutput takes a string and yields nothing
effect ReadInput()         = String // ReadInput takes no argument and yields a string

void greetUser() {                              // A function that raises console effects
  raise WriteOutput("Name?");                   // Print a query
  String name = raise ReadInput();              // Get a user input
  raise WriteOutput("Hello " + name + "!"); // Print a greeting
}

// Handle effects in two different ways:
void realApp() {
  try { greetUser(); } handle {
    case WriteOutput(s): print(s); resume; // Print s to stdout and resume greetUser
    case ReadInput():
      String s = read();                       // Read s from stdin, and
      resume s;                                // resume greetUser execution with value s
  }
}
void testApp() {
  try { greetUser(); } handle {
    case WriteOutput(s):
      if (s == "Name?" || s == "Hello user!") resume; // If s is expected string, resume
      else print("Found a wrong ouptut!");            // Otherwise, print a test error
    case ReadInput(): resume "user";                  // Resume with a dummy input.
  }
}
```

solver will backtrack by one decision level with this strategy.

- **Non-chronological backtracking** is a backtracking strategy that depends on a conflict-driven clause. This strategy allows a SAT solver to backtrack by more than one level if the cause of a conflict does not belong to the previous decision level.

Note that no components mention other components in their descriptions. Backtracking components mention conflict-driven clauses, but they do not specify from where those clauses should come. They might be constructed by 1-UIP or RelSAT component but can be an input value from a user as well. These independent components allow tutorial authors to compose the components freely to serve their needs.

QFIDL solver components For a QFIDL solver implementation, which is often based on shortest path algorithms, SATYROS provides all components of the Bellman-Ford algorithm[Cormen *et al.*, 2001]. The Bellman-Ford algorithm is chosen because it comprises two simple stages with the straightforward purposes: shortest path computation and negative cycle checking. This clear stage distinction makes the algorithm easy to explain, which fits well to the pedagogical objective of SATYROS. The following describes each component for QFIDL:

- **Shortest path propagation** is a stage for the shortest path computation. This allows a QFIDL solver to decide an assignment for QFIDL variables if there exists valid one, and otherwise lets the negative cycle checking to find a contradiction in constraints.
- **Negative cycle checking** is a stage to check whether the target graph contains a negative cycle or not. This requires that shortest path propagation has been done before.

It should be mentioned that QFIDL solver components are more tightly bound to each other than SAT components. However, this is not an issue for a QFIDL solver, as it is based on a less flexible algorithm and does not have much freedom in its algorithm design from the beginning.

4.2 Effects from Solver Components

Each component introduced in the previous section raises a dedicated set of effects. These effects are meant to support common code structures for SAT/SMT solvers.

Effects from SAT solver components SAT solver components raise these effects

- The decision procedure component can raise a `DecisionResult` and `DecisionComplete` effect. `DecisionResult` occurs when the decision procedure chooses a new decision variable and the value assigned to it. `DecisionComplete` occurs when the decision procedure finds that there are no unassigned variables.
- The boolean constraint propagation (BCP) component can raise `BCPUnitClause`, `BCPConflict`, and

BCPComplete. BCPUnitClause is raised when BCP meets a unit clause. BCPConflict is for when BCP meets a falsified clause. Lastly, BCPComplete occurs when BCP finishes to check all clauses.

- The conflict-driven clause construction components can raise BCPConflictDrivenClause when the components finish the construction.
- The backtracking components can raise either of BacktraceExhaustion and BacktraceComplete. BacktraceExhaustion is for when there is no decision level left to backtrack. BacktraceComplete is raised when the components successfully backtrack to a level according to the strategies of the components.

As shown in the cases of the conflict-driven clause components and backtracking components, all components that provide the same functionality share the same set of effects. With these common effects, SATYROS users can easily replace one component with another to implement a different version of SMT/SAT solver.

Effects from QFIDL solver components The effects from QFIDL solver components follow

- The shortest path propagation component can raise PropagationCheck, PropagationFindShorter, and PropagationEnd. PropagationCheck is raised when the component checks an edge. Likewise, PropagationFindShorter is when the component finds that the distance to the end node of the edge should be updated. Once the component is done, it raises PropagationEnd.
- The negative cycle component raise NegativeCycleCheck, NegativeCycleFind, and NegativeCyclePass. NegativeCycleCheck occurs when the component checks an edge. If it finds a negative cycle, it raises NegativeCycleFind. Otherwise, it raises NegativeCyclePass.

Unlike SAT effects which allow users to apply different strategies for SAT implementation, most of the QFIDL effects are meant to provide better interactivity during the QFIDL solving. Two exceptions are NegativeCycleFind and NegativeCyclePass, which are necessary to integrate a QFIDL solver with a SAT solver.

5 Example and Evaluation

We describe two examples that demonstrate the power and simplicity of SATYROS library in this section. Both of them are SMT solvers that utilize QFIDL solvers and SAT solvers together.

5.1 Loosely Integrated DPLL(T)

The first SMT solver we explain is a loosely integrated DPLL(T). In a loosely integrated DPLL(T), we first translate any SMT constraints into SAT constraints and solve the SAT constraints to get a complete assignment to boolean variables. Once we get the complete assignment, we translate the assignment back to theory constraints and solve them using a theory solver (in this case, QFIDL solver). If the theory solver rejects, we add an extra constraint for the current assignment and repeat the process from SAT solving. Otherwise, we output an assignment to SMT variables from the theory solver.

We illustrate a SATYROS implementation of this approach in Listing 2. In the implementation, we handle only BCPComplete, DecisionComplete, and NegativeCyclePass because handlers for other effects are mere component calls and easily delegated to a common handler commonHandler (which uses the **RelSAT** and **non-chronological backtracking** components).

The code faithfully reflects the strategy of a loosely integrated DPLL(T). When BCPComplete occurs, we start the decision procedure. When we get a complete assignment (DecisionComplete), we initialize a QFIDL solver and run it (runQFIDL). If the theory solver accepts the assignment (NegativeCyclePass), we finish the execution.

5.2 Tightly Integrated DPLL(T)

The other SMT solver we describe here is a tightly integrated DPLL(T). We also start with the translation from any SMT constraints into SAT constraints in this strategy. However, we run a SAT solver only for a single decision level, i.e., for one decision and boolean propagations. After that, we run a QFIDL solver to validate the decision. If it is invalid, we add a constraint to block the decision and backtrack. Otherwise, we allow the SAT solver to run for one more decision level. We repeat this until we get a full assignment that the QFIDL solver accepts.

We show the example code for this strategy in Listing 3. As in the previous example, we handle only the crucial effects and delegate handling of other effects to commonHandler.

This code also faithfully reflects its strategy. When BCP is done (BCPComplete), we initialize a QFIDL solver and run it. If the theory solver accepts a partial assignment (NegativeCyclePass), we continue to the next decision procedure. When there are no more variables to assign (DecisionComplete), we finish the execution.

5.3 Evaluation

As shown in the previous examples, one can simply implement interactive SMT implementations based on different implementation strategies with SATYROS. SATYROS requires only the minimum amount of deviation across strategy and allows an abstraction over common code using a handler like commonHandler. Moreover, even with its simplicity, it provides full interactivity; a user can execute an SMT solver for one "step" for any meaning of "step" (which may subsume multiple effects), undo such a step, query the SMT state, and modify the state. A SMT web tutorial implemented with SATYROS (https://Ailrun.github.io/satyros) is implemented upon these interactive functionalities.

6 Conclusion

We present SATYROS, a Haskell library for an interactive SAT/SMT implementation. With its API design, SATYROS allows a simple yet fully-featured interactive SMT solver implementation. Some future plan for SATYROS is

Listing 2 SATyros-based implementation of loosely integrated DPLL(T)

```
looseDPLLT BCPComplete                       = decision $> False
looseDPLLT DecisionComplete                  = runQFIDL
looseDPLLT (InsideTheory NegativeCyclePass) = pure True
looseDPLLT eff                               = commonHandler eff
```

Listing 3 SATyros-based implementation of tightly integrated DPLL(T)

```
tightDPLLT BCPComplete                       = runQFIDL
tightDPLLT DecisionComplete                  = pure True
tightDPLLT (InsideTheory NegativeCyclePass) = decision $> False
tightDPLLT eff                               = commonHandler eff
```

- To support different decision procedures: random decision procedure is enough to demonstrate basic design choices but does not fit well when one wants to describe more complex design choices. By adding other decision procedures, SATyros can handle such a case too.
- To support other theory solvers than QFIDL solver: Although an algorithm for QFIDL solver is easy and thus helps beginners to understand the structural design of SMT solvers, its power is fairly limited and cannot expose the full power of SMT solvers. By providing, for example, linear integer arithmetic or bit-vector theory solver, it becomes more obvious for the learners how powerful SMT solvers are.
- To allow theory solvers to be combined: Since the current version of SATyros has components for only one theory solver (QFIDL), it does not provide a way to combine multiple theory solvers such as Nelson-Oppen[Nelson and Oppen, 1979].

All of these extensions have no theoretical barrier. Thus, we expect to enhance SATyros with these features near future.

References

[Ball *et al.*, 2004] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods (IFM) 2004*, pages 1–20, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[Barnett *et al.*, 2005] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# Programming System: An Overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS) 2004*, pages 49–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[Barrett *et al.*, 2011] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification (CAV) 2011*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Cadar *et al.*, 2008] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 209224, USA, 2008. USENIX Association.

[Cimatti *et al.*, 2013] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2013*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Cimatti *et al.*, 2018] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions. *ACM Trans. Comput. Logic*, 19(3), aug 2018.

[Cormen *et al.*, 2001] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2nd edition, 2001.

[CVC4, 2021] About cvc4. https://cvc4.github.io/, 2021. Accessed Dec 1, 2021.

[Davis *et al.*, 1962] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Commun. ACM*, 5(7):394397, jul 1962.

[de Moura and Bjørner, 2008] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2008*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Dutertre, 2014] Bruno Dutertre. Yicesă2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification (CAV) 2014*, pages 737–744, Cham, 2014. Springer International Publishing.

[Godefroid *et al.*, 2008] Patrice Godefroid, Michael Y Levin, and David Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of Network and Distributed Systems Security (NDSS) 2008*. the Internat Society, 2008.

[Graham-Lengrand *et al.*, 2020] Stéphane Graham-Lengrand, Dejan Jovanović, and Bruno Dutertre. Solving Bitvectors with MCSAT: Explanations from Bits and Pieces. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 103–121, Cham, 2020. Springer International Publishing.

[Gritman *et al.*, 2017] Add Gritman, Anthony Ha, Tony Quach, and Derek Wenger. Conflict driven clause learning. https://cse442-17f.github.io/Conflict-Driven-Clause-Learning/, 2017. Accessed Oct 23, 2021.

[Mann *et al.*, 2021] Makai Mann, Ahmed Irfan, Florian Lonsing, Yahan Yang, Hongce Zhang, Kristopher Brown, Aarti Gupta, and Clark Barrett. Pono: A Flexible and Extensible SMT-Based Model Checker. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification (CAV) 2021*, pages 461–474, Cham, 2021. Springer International Publishing.

[Marques-Silva and Sakallah, 1999] J.P. Marques-Silva and K.A. Sakallah. GRASP: a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.

[Moggi, 1991] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[Morales, 2021] Jose F. Morales. Z3 playground. https://jfmc.github.io/z3-play, 2021. Accessed Oct 23, 2021.

[Nelson and Oppen, 1979] Greg Nelson and Derek C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245257, oct 1979.

[Plotkin and Power, 2001] Gordon Plotkin and John Power. Adequacy for algebraic effects. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, pages 1–24, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Plotkin and Pretnar, 2009] Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 80–94, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[Reynolds and Blanchette, 2015] Andrew Reynolds and Jasmin Christian Blanchette. A Decision Procedure for (Co)datatypes in SMT Solvers. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 197–213, Cham, 2015. Springer International Publishing.

[Reynolds *et al.*, 2016] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A Decision Procedure for Separation Logic in SMT. In Cyrille Artho, Axel Legay, and Doron Peled, editors, *Automated Technology for Verification and Analysis*, pages 244–261, Cham, 2016. Springer International Publishing.

[Reynolds *et al.*, 2019] Andrew Reynolds, Haniel Barbosa, Andres Nötzli, Clark Barrett, and Cesare Tinelli. cvc4sy: Smart and Fast Term Enumeration for Syntax-Guided Synthesis. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification (CAV) 2019*, pages 74–83, Cham, 2019. Springer International Publishing.

[Swamy *et al.*, 2013] Nikhil Swamy, Juan Chen, and Ben Livshits. Verifying Higher-order Programs with the Dijkstra Monad. In *ACM Programming Language Design and Implementation (PLDI) 2013*. ACM, June 2013.

[Swierstra, 2008] Wouter Swierstra. Data Types á la Carte. *Journal of Functional Programming*, 18(4):423436, 2008.

[Tiwari *et al.*, 2015] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program Synthesis Using Dual Interpretation. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 482–497, Cham, 2015. Springer International Publishing.

[Z3, 2021] Z3prover/z3 wiki. https://github.com/Z3Prover/z3/wiki, 2021. Accessed Dec 1, 2021.