LLSC: A Parallel Symbolic Execution Compiler for LLVM IR

Guannan Wei
Purdue University
West Lafayette, IN, USA
guannanwei@purdue.edu

Shangyin Tan
Purdue University
West Lafayette, IN, USA
tan279@purdue.edu

Oliver Bračevac
Purdue University
West Lafayette, IN, USA
bracevac@purdue.edu

Tiark Rompf
Purdue University
West Lafayette, IN, USA
tiark@purdue.edu

ABSTRACT

We present LLSC, a prototype compiler for nondeterministic parallel symbolic execution of the LLVM intermediate representation (IR). Given an LLVM IR program, LLSC generates code preserving the symbolic execution semantics and orchestrating solver invocations. The generated code runs efficiently, since the code has eliminated the interpretation overhead and explores multiple paths in parallel. To the best of our knowledge, LLSC is the first compiler for fork-based symbolic execution semantics that can generate parallel execution code.

In this demonstration paper, we present the current development and preliminary evaluation of LLSC. The principle behind LLSC is to automatically specialize a symbolic interpreter via the 1st Futamura projection, a fundamental connection between interpreters and compilers. The symbolic interpreter is written in an expressive high-level language equipped with a multi-stage programming facility. We demonstrate the run time performance through a set of benchmark programs, showing that LLSC outperforms interpretation-based symbolic execution engines in significant ways.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; **Software testing and debugging**; **Automated static analysis**.

KEYWORDS

symbolic execution, compilation, program testing, staging

ACM Reference Format:
Guannan Wei, Shangyin Tan, Oliver Bračevac, and Tiark Rompf. 2021. LLSC: A Parallel Symbolic Execution Compiler for LLVM IR. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473108

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473108

1 INTRODUCTION

Symbolic execution is a popular software testing technique with origins going back to the 1970s [4, 11]. The underlying idea is to execute a program in a fashion that explores multiple execution paths, where some inputs are left symbolic (rather than concrete), and to collect constraints (path conditions) at branches describing how to trigger a specific execution path. At a branch, the symbolic execution engine nondeterministically explores multiple paths, synthesizing different path conditions as logical formulae. This process permits the automatic generation of concrete inputs that trigger a certain execution path. Modern engines employ SMT (Satisfiability Modulo Theories) solvers [3] for this purpose.

While useful for software testing and analysis, symbolic execution suffers from the path explosion problem, arguably its major drawback: There is an exponential or even infinite number of paths in large programs, and exploring all of them is intractable.

Taming the path explosion problem in symbolic execution is therefore critical to make it practical, which has resulted in different specialized flavors of symbolic execution [2]. For example, instead of exploring all paths, the execution engine may prune or merge paths according to some heuristics, or use concrete inputs to guide the exploration.

We present the LLSC symbolic compiler, which accelerates path exploration for symbolic execution in two ways: (1) lowering the cost per path execution, and (2) exploring multiple paths in parallel. These two improvements greatly amplify the path exploration *throughput*, i.e., the number of paths explored per unit of time. LLSC implements multipath nondeterministic symbolic execution, but its approach is not tied to any particular exploration strategy and accommodates more specialized variants.

In contrast to the typical way of implementing symbolic execution by interpretation, LLSC is a *compiler* that transforms input LLVM IR programs to C++ code, following the multipath symbolic execution semantics. The generated C++ code together with the runtime code are further compiled by an off-the-shelf optimizing C++ compiler, yielding the final executable. Running the executable performs the actual symbolic execution, invokes SMT solvers, and generates test cases. The generated code does not exhibit any interpretation overhead, such as traversing the AST/IR or dynamic dispatching based on AST/IR, which commonly occurs in interpretation-based engines. Eliminating the interpretation overhead drastically lowers the cost per-path execution. LLSC also generates code that explores paths in parallel on multicore CPUs.

The design and implementation of LLSC follows the 1st Futamura projection [7, 8], which allows us to readily obtain a symbolic execution compiler by specializing (staging) a symbolic interpreter in a high-level programming language. The symbolic interpreter is written in Scala [13] using the Lightweight Modular Staging (LMS) [18] framework. The LMS framework provides basic staging facilities to automatically convert an annotated symbolic interpreter into a symbolic compiler. The staged interpreter is written with functional programming design patterns [10, 23] (e.g., using monads to express effects). The generated code makes use of modern persistent data structures [17], leading to efficient and safe code that permits concurrent and parallel execution without race conditions. This combination leads to a clean, concise, and yet performant implementation of symbolic execution.

The development of LLSC is based on the approach of building symbolic execution engines presented in our prior work [24]. Compared with this prior work, LLSC supports compiling more instructions (46 at the moment) and a number of external functions, integrates an improved runtime with a parallel path exploration strategy, and provides an improved user interface. We evaluate the performance of LLSC on a synthetic benchmark and a set of realistic programs. We also compare the performance with KLEE [6], a state-of-the-art symbolic execution tool on LLVM. The evaluation shows that LLSC performs on average 10x faster than KLEE.

To the best of our knowledge, LLSC is the first prototype compiler for nondeterministic and parallel multipath symbolic execution. The source code and a demonstration video of LLSC are available at `https://continuation.passing.style/llsc`.

Organization. Section 2 reviews necessary background on Futamura projections and multi-stage programming. Section 3 presents the design and implementation of LLSC. Section 4 describes the end-user workflow for LLSC. Section 5 presents the empirical evaluation and compares the result with KLEE. Section 6 concludes with related work, limitations, and future work.

2 BACKGROUND

In this section, we briefly review the 1st Futamura projection [7, 8] and multi-stage programming [21], two of LLSC's key ingredients. Due to space limitations, we refer the reader to the survey paper by Baldoni et al. [2] for basic concepts of symbolic execution.

Futamura Projections. Interpreters and compilers are intimately connected. The 1st Futamura projection permits deriving a compiler from an interpreter in an entirely mechanical way. Consider an interpreter $interp(p, arg)$ taking a program p and an argument arg to that program, producing the output by interpretation. The 1st Futamura projection states that if we a have program specializer mix, then applying it to $interp$ with program p yields a target program: $mix(interp, p) = target$. Running the target program produces the same result as interpretation and as the original program: $[\![target]\!](arg) = interp(p, arg) = [\![p]\!](arg)$, where the semantic bracket $[\![\_]\!]$ maps programs to their meanings.

Multi-Stage Programming. One way to realize the 1st Futamura projection is to write the interpreter with annotations that can guide the automatic specialization. This programming paradigm is called multi-stage programming (MSP) or generative programming. The classic example of MSP is the `power` function to compute x^n:

```
def power(x: Int, n: Int): Int =
  if (n == 0) 1 else x * power(x, n-1)
```

If `n` is a statically-known constant c, one would expect a faster implementation that directly multiples `x` for c times with itself without paying the overhead of recursively calling `power`. To achieve that, we can rewrite `power` with the next-stage annotation `Rep` (using the notation from LMS [18]) denoting that `x` and the result of the function will be known in a later stage:

```
def power(x: Rep[Int], n: Int): Rep[Int] =
  if (n == 0) 1 else x * power(x, n-1)
```

Given a constant number for `n` (e.g., `n = 3`), the underlying MSP system generates a specialized power function that runs faster:

```
def power3(x: Int): Int = x * x * x
```

Compilation by Specialization. The power function has the structure of an interpreter: it is inductively defined over the natural number `n`. Likewise, an interpreter is inductively defined over the program AST/IR. One can consider the argument `n` of `power` as the known program to the interpreter, and the argument `x` as the unknown input of the target program to the interpreter. The process of specializing an interpreter with stage annotations is essentially compilation, following the slogan "*a staged interpreter is a compiler*".

3 DESIGN AND IMPLEMENTATION

The design of the staged symbolic interpreter in LLSC follows the semantics-first approach [24] we previously developed. We start from a big-step, operational, concrete semantics of LLVM IR and then refactor it into a symbolic execution semantics, i.e., by adding symbolic values, collecting path conditions, and exploring multiple paths nondeterministically. Then, we slightly adapt the symbolic interpreter to a *staged symbolic interpreter* (SSI), which is the core construction of LLSC.

The SSI is written in an expressive high-level language (Scala) using the LMS framework [18]. We use a parser generated by ANTLR [14] to parse programs in LLVM's assembly format into Scala objects that can be inspected by the SSI.

Running the SSI will generate code in C++. We use C++ functions to represent blocks and functions in the source IR program. The generated code is structurally isomorphic to the source IR, in the sense that for each branching/jump/call in the source IR, there is a function call of the corresponding target function in the target code. The C++ functions representing source blocks take a "state" and returns a list of "state" and potential returned "values", where the list represents possibly multiple results. The state contains information about the execution of a path at some program location, including the heap, the stack, and the path conditions. The generated C++ functions for source LLVM functions take a list of arguments in addition to the state.

Besides the generated code, there are a number of the runtime components written in C/C++: 1) definitions of the symbolic states and values, 2) a scheduler for parallel exploration, 3) summaries of external functions such as library functions and system calls, and 2) the third-party SMT solver library. These runtime components will be compiled and linked with the application-specific generated

```
1  case CallInst(ty, f, args) ⇒
2    for {
3      fv ⟵ eval(f) // evaluate function
4      vs ⟵ mapM(extValues(args))(eval) // evaluate arguments
5      _  ⟵ pushFrame
6      s  ⟵ getState
7      v  ⟵ reflect(fv(s, vs)) // apply
8      _  ⟵ popFrame(s.stackSize)
9    } yield v
```

Figure 1: Code snippet for `CallInst` in the SSI.

code. Currently LLSC uses the Simple Theorem Solver [9] in the backend to discharge constraints.

3.1 Staged Symbolic Interpreter

The core of LLSC is a staged symbolic interpreter that handles LLVM instructions and data types (integers, floating numbers, structs, arrays, etc.). For operations on unknown values, SSI constructs symbolic expressions and then continues execution. When encountering a branch instruction (e.g. `br` and `switch`), SSI adds the path conditions and explores both branches, respectively. The code generated by interpreter specialization precisely reflects what the interpreter does but without the interpretation overhead.

The implementation of the SSI is concise and close to an informal LLVM semantics. For example, Figure 1 shows the code that handles the `call` instruction. The for-comprehension syntax `for {...} yield v` contains a series of bindings in between the brackets. The for-comprehension evaluates the right-hand side expression of each binding sequentially, binds the result of the expression to the identifier at left, so that the subsequent evaluation can use previously evaluated expressions. To be concrete, to handle a `call` instruction, we need to (1) evaluate the function sub-term `f` first (line 4), (2) evaluate all arguments from left to right (line 5), (3) push a new stack frame for the call (line 6), (4) "jump" to and apply the new function with state `s` and arguments `vs`, yielding returned value `v`, and (5) clean up the frame by popping. The value of the whole for-comprehension is a computation producing `v`.

At line 4, the `eval` function maps an LLVM function value to a Scala function value that we can directly apply in Scala, i.e., `fv(s, vs)`. The application of function `fv` is fed with the whole state `s` before making the call. Applying this Scala function effectively generates a function call in C++, the target of which call is the C++ function representing the LLVM callee. The `reflect` function is the mechanism to convert the representation of generated code back to a result that can be used in the meta-program in Scala.

Remarkably, the staged symbolic interpreter in Figure 1 resembles an ordinary interpreter. It thus serves as an executable semantic specification and a compiler at the same time. The stage annotations are attached at the type level (i.e., `Rep` shown in the example of Section 2), minimizing the syntactic noise.

3.2 Generated Code and Runtime

The generated code and runtime components are written in C++ and make use of Immer's [17] persistent data structures to represent execution states, such as heaps and stacks. The choice of using immutable and persistent data structures greatly simplifies the generated code and the implementation of the runtime. Our

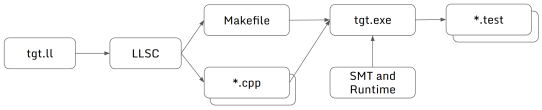

Figure 2: The workflow of LLSC.

experience shows that immutable data structures perform well, in contrast to common preconceptions about them.

To give a taste of the generated code, the following snippet shows a fragment of a block-function that corresponds to loading and assigning local variables in LLVM:

```
flex_vector<pair<SS, PtrVal>> fun_Block1(SS x1) {
  PtrVal x2 = x1.at(x1.env_lookup("a"));
  SS x3 = x1.assign("b", x2);
  PtrVal x4 = x3.at(x3.env_lookup("c")); ... }
```

The current way that LLSC implements parallel symbolic execution is to use the `std::async` and `std::future` functions (from the C++ standard library) in the generated code. We leave a more fine-tuned parallel execution scheduler for future work. The runtime also starts an additional thread to monitor the block coverage and path coverage. The maximum number of concurrent threads is configurable by the user.

3.3 Intrinsics and External Functions

The LLVM IR language models a number of features as intrinsics, such as variable arguments, `memcpy`, `memset`, and etc. Those intrinsics are opaque, since they have no definition. However, they usually have well understood or intuitive semantics. To symbolically execute programs with intrinsics, it is possible to manually provide executable summaries manipulating the states for intrinsics. In LLSC, an executable summary can be modularly implemented in two equivalent ways: 1) they can be written as meta-functions in Scala, which will be generated to C++ code at staging-time, or 2) directly written with the C++ runtime and linked together at the (C++) compile-time. In either way, the summary function manipulates or transforms LLSC's symbolic state representation.

The SSI models the LLVM IR language, but not the external environment. Another opaque part to applications is the C standard library and system calls. For library functions, we reuse the KLEE modification of μClibc [1], a minimal C library providing the same functionality as glibc. The involved library functions will be compiled by LLSC too, and be linked with the compiled application.

4 TOOL USAGE

Figure 2 shows the workflow of LLSC, which has a command line interface. By invoking with a file path of an LLVM module and an entrance function name:

```
llsc <.ll-filepath> <task-name> <entrance-fun> [n-sym]
```

LLSC generates a set of `.cpp` files and a `Makefile` for `<task-name>`. It is optional to specify the number of symbolic arguments to the entrance function. The makefile specifies the instructions to compile the `.cpp` files and to link them with the runtime and SMT solver library. After `make`-ing the executable file, the user can then run it with the number of threads:

```
./<task-name> [n-threads]
```

When the executable file starts running, the user will continuously observe the current timestamp and coverage, e.g.:

Table 1: Running times of 1 million path explorations.

	KLEE	LLSC-1	LLSC-4	LLSC-8	LLSC-16
Running time	329.2 s	48.99 s	38.20 s	32.87 s	26.97 s

```
[5 s] #blocks: 41/41; #paths: 198487; #threads: 4
```

While the task is running, test cases generated by the SMT solver for complete paths are written into files `tests/n.test`, where n is the id number of the path.

5 EVALUATION

In this section, we demonstrate the performance of LLSC and compare it with KLEE, a state-of-the-art symbolic execution tool. We conduct the experiments in two categories. The first one uses a synthetic benchmark that aims to measure the performance of exhaustive path exploration. The other one uses realistic programs of small sizes, and evaluates end-to-end executions. We conduct the experiments on a machine running Ubuntu 20.10 with AMD 3700X and 32 GB main memory. Other tools and their versions involved are KLEE 2.1, g++ 10.2.0, and STP 2.3.3.

Benchmarking Synthetic Programs. We use the contrived benchmark program `mp1m` from [24] that is small in terms of LOC but contains an exponential number of paths to measure the performance of exhaustive exploration. We run LLSC with different numbers of threads and compare with the symbolic interpreter KLEE. All timings reported in this experiment do not include the time spent by the SMT solver[1].

The benchmark `mp1m` has 285 LLVM instructions and 1048576 paths in total. LLSC compiles `mp1m` to a C++ program of 706 LOC within 452.08ms. Table 1 shows the running times of LLSC and KLEE, where n of LLSC-n indicates the number of parallel threads. As we can see, even without parallelism, LLSC is ~7x times faster than KLEE on this benchmark. Running with multiple threads further shortens the time, although the improvement is not proportional to n due to the overhead of context switching. When $n = 16$, we observe a ~12x speedup of path exploration.

Benchmarking Realistic Programs. We also conduct experiments on small programs that performs realistic tasks and compare with KLEE. Table 2 shows a summary of those programs and the benchmark results, including the numbers of instructions and numbers of valid paths of the program, as well as the staging times of LLSC and the actual symbolic execution times (including SMT solving) for LLSC and KLEE. Those programs have relatively small numbers of paths, and running them with multiple threads would not bring significant performance improvements, therefore the reported running times in the table stem from a single-threaded execution.

It is worth noting that KLEE by default uses a counterexample cache to query the solver less frequently than LLSC, and to make fair experiments we disable this single optimization and report times in T_{KLEE}. Table 2 shows that LLSC successfully explores all paths and generates all test cases, with running times significantly shorter than KLEE. We also report the running times of KLEE with all default optimizations in $T^{\text{Opt}}_{\text{KLEE}}$. In this case, LLSC still outperforms KLEE on most of the benchmarks where the outlier is kmp-match, which uses the solver intensively.

[1]For KLEE, we calculate the time by excluding the solver time reported by `klee-stats`.

Table 2: Evaluation result of benchmark programs.

Benchmark	#inst	#paths	T_{staging}	T_{LLSC}	T_{KLEE}	$T^{\text{Opt}}_{\text{KLEE}}$
maze	156	309	351.32ms	106ms	5.30s	270ms
k-of-st-arrays	198	252	312.11ms	556ms	9.12s	2.75s
kmp-match	254	1287	399.33ms	991ms	28.98s	202ms
binary-search	253	92	379.73ms	531ms	2.25s	1.36s
bubble-sort	121	24	204.17ms	184ms	1.26s	493ms
quick-sort	120	120	215.73ms	629ms	3.98s	1.59s
merge-sort	314	720	422.29ms	3.48s	22.82s	9.42s
knapsack	148	1666	265.95ms	45.35s	163.47s	76.73s

6 DISCUSSION

Related Work. KLEE [6] is one of the mature symbolic execution engines targeting LLVM IR. Symbolic execution engines are commonly implemented as interpreters (e.g. [6, 16, 22]). Compilation-based approaches only gained attention very recently [15, 24]. The development of LLSC follows the semantics-first approach using staging introduced in [24]. Futamura projections [7, 8] are the conceptual idea we use to build symbolic execution compilers. LLSC uses the LMS framework [18] in Scala to realize program specialization. A number of widely-used languages also have built-in support for staging [12, 19, 20]. Instrumentation is another popular approach and can be done by using specialized instrumentation tools [25] or general compilation frameworks [15]. Our approach can be viewed as generating instrumented code by specializing a symbolic interpreter. LLSC explores multiple paths in parallel by using threads in a single process. For a similar goal, Cloud9 [5] performs parallel and distributed symbolic execution, where each node in the cluster runs a separate instance of KLEE.

Limitation and Future Work. The symbolic execution data types supported by LLSC is currently limited to bitvectors. With an appropriate SMT solver backend, it can be extended to reason about richer types, such as floating-point numbers and symbolic arrays. Other optimizations appeared in mature tools, for example, query caching, can be integrated too. We are currently making effort to build better symbolic support for library functions and external environments, which will allow LLSC to execute more real-world programs. LLSC's code generation scheme is relatively simple by using persistent data structure and high-level concurrency primitives. It is possible to gain further performance improvement by using destructive data structures and finer-grained parallelism primitives. Although these would require a more complicated and delicate code generator to generate safe code.

Conclusion. We present LLSC, a symbolic execution compiler that generates parallel and nondeterministic symbolic execution code for LLVM IR. Our methodology combines an old idea, the Futamura projections, with modern programming abstractions and compiler technology, leading to a sweet spot between performance, safety, and developer effort. We believe that both the demonstrated tool and our disciplined approach are a significant step forward to building performant program analysis tools.

ACKNOWLEDGMENTS

This work was supported in part by NSF awards 1553471, 1564207, 1918483, 1910216, DOE award DE-SC0018050, as well as gifts from Google, Facebook, and VMware.

REFERENCES

[1] 2021. klee-uclibc. https://github.com/klee/klee-uclibc. Accessed: 2021-05-06.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.* 51, 3 (2018), 50:1–50:39. https://doi.org/10.1145/3182657

[3] Clark Barrett and Cesare Tinelli. 2018. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 305–343.

[4] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, California). ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/800027.808445

[5] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. 2011. Parallel symbolic execution for automated real-world software testing. In *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*, Christoph M. Kirsch and Gernot Heiser (Eds.). ACM, 183–198. https://doi.org/10.1145/1966445.1966463

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. 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* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 209–224.

[7] Yoshihiko Futamura. 1971. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls* 25 (1971), 45–50.

[8] Yoshihiko Futamura. 1999. Partial Evaluation of Computation Process–An Approach to a Compiler-Compiler. *Higher-Order and Symbolic Computation* 12, 4 (01 Dec 1999), 381–391. https://doi.org/10.1023/A:1010095604496

[9] Vijay Ganesh and David L. Dill. 2007. A Decision Procedure for Bit-Vectors and Arrays. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 519–531.

[10] John Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98

[11] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[12] Oleg Kiselyov. 2014. The Design and Implementation of BER MetaOCaml. In *Functional and Logic Programming*, Michael Codish and Eijiro Sumii (Eds.). Springer International Publishing, Cham, 86–102.

[13] Martin Odersky, Lex Spoon, and Bill Venners. 2008. *Programming in scala*. Artima Inc.

[14] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[15] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[16] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

[17] Juan Pedro Bolívar Puente. 2017. Persistence for the Masses: RRB-Vectors in a Systems Language. *Proc. ACM Program. Lang.* 1, ICFP, Article 16 (Aug. 2017), 28 pages. https://doi.org/10.1145/3110260

[18] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010*, Eelco Visser and Jaakko Järvi (Eds.). ACM, 127–136. https://doi.org/10.1145/1868294.1868314

[19] Tim Sheard and Simon Peyton Jones. 2002. Template Meta-Programming for Haskell. *SIGPLAN Not.* 37, 12 (Dec. 2002), 60–75. https://doi.org/10.1145/636517.636528

[20] Nicolas Stucki, Aggelos Biboudis, and Martin Odersky. 2018. A practical unification of multi-stage programming and macros. In *Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2018, Boston, MA, USA, November 5-6, 2018*, Eric Van Wyk and Tiark Rompf (Eds.). ACM, 14–27. https://doi.org/10.1145/3278122.3278139

[21] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242. https://doi.org/10.1016/S0304-3975(00)00053-0

[22] Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. ACM, New York, NY, USA, 530–541. https://doi.org/10.1145/2594291.2594340

[23] Philip Wadler. 1992. The Essence of Functional Programming. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico, USA) *(POPL '92)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/143165.143169

[24] Guannan Wei, Oliver Bračevac, Shangyin Tan, and Tiark Rompf. 2020. Compiling Symbolic Execution with Staging and Algebraic Effects. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 164 (Nov. 2020), 33 pages. https://doi.org/10.1145/3428232

[25] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. 2018. QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, William Enck and Adrienne Porter Felt (Eds.). USENIX Association, 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun