Random Testing of a Higher-Order Blockchain Language (Experience Report)

TRAM HOANG, National University of Singapore, Singapore
ANTON TRUNOV, Zilliqa Research, Russia
LEONIDAS LAMPROPOULOS, University of Maryland, USA
ILYA SERGEY, National University of Singapore, Singapore

We describe our experience of using property-based testing—an approach for automatically generating random inputs to check executable program specifications—in a development of a higher-order smart contract language that powers a state-of-the-art blockchain with thousands of active daily users.

We outline the process of integrating QuickChick—a framework for property-based testing built on top of the Coq proof assistant—into a real-world language implementation in OCaml. We discuss the challenges we have encountered when generating well-typed programs for a realistic higher-order smart contract language, which mixes purely functional and imperative computations and features runtime resource accounting. We describe the set of the language implementation properties that we tested, as well as the semantic harness required to enable their validation. The properties range from the standard type safety to the soundness of a control- and type-flow analysis used by the optimizing compiler. Finally, we present the list of bugs discovered and rediscovered with the help of QuickChick and discuss their severity and possible ramifications.

CCS Concepts: • **Software and its engineering → Functional languages**.

Additional Key Words and Phrases: random testing, property-based testing, definitional interpreters, higher-order control-flow analysis, smart contracts, QuickChick, Scilla

ACM Reference Format:
Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. Random Testing of a Higher-Order Blockchain Language (Experience Report). *Proc. ACM Program. Lang.* 6, ICFP, Article 122 (August 2022), 16 pages. https://doi.org/10.1145/3547653

1 INTRODUCTION

What is worse than losing millions of dollars to a buggy program that manages decentralized financial transactions? The answer is: losing millions of dollars to a buggy program that has been marked as *bug-free* by a buggy type checker.

Smart contract languages define semantics of transactions executed by distributed blockchain consensus protocols. Programs written in such languages encode the custom logic of replicated computations that manipulate various digital assets in a decentralized fashion. Research in smart contract languages is on the rise, and many proposals with exciting features, allowing for safe and secure implementations of decentralized applications, have been recently put forward (Blackshear et al. 2019; Coblenz 2017; Das et al. 2021; IOHK Foundation 2019; Schrans 2018; Sergey et al. 2019; Tezos Foundation 2018). Striving to strike a good balance between expressivity and minimalism,

Authors' addresses: Tram Hoang, National University of Singapore, Singapore, tram.hoang@u.yale-nus.edu.sg; Anton Trunov, Zilliqa Research, Russia, anton@zilliqa.com; Leonidas Lampropoulos, University of Maryland, USA, leonidas@umd.edu; Ilya Sergey, National University of Singapore, Singapore, ilya@nus.edu.sg.

2475-1421/2022/8-ART122
https://doi.org/10.1145/3547653

many of those languages are based on a combination of the polymorphic lambda-calculus with linear or session types to statically enforce properties, such as preservation of assets (Blackshear et al. 2019) or adherence to a statically-checked communication protocol (Das et al. 2021).

The static guarantees provided by such languages in practice are, however, only as good as their implementations, by means of a definitional interpreter, virtual machine, or via compilation to some well-adopted back-end languages, such as EVM (Wood 2014) or wasm (Haas et al. 2017). That is, a single bug in the implementation of a contract type checker, interpreter, or compiler can defy the whole purpose of a strong type system by making it possible to violate its runtime guarantees—the fact that can (and will) be exploited by the adversarial system participants for monetary profits.

One well-studied approach to ensure the correctness of various language implementation components is to mechanize the entire development in a proof assistant, allowing to state and prove theorems about aspects such as type- and analysis soundness. While this approach has been successfully exercised for realistic languages, such as C (Leroy 2006) and ML (Kumar et al. 2014), it incurs a prohibitively high first-time implementation effort and maintenance overhead, as evolution of the codebase also requires evolving proofs, which are often hand-crafted. A more lightweight approach to ensure correctness is via lightweight property-based testing (Claessen and Hughes 2000) of language artifacts by using large numbers of randomly generated well-formed programs. While unable prove the absence of errors, this approach has been shown extremely efficient for discovering multiple bugs in industry-strength compilers of both imperative (Yang et al. 2011) and functional languages (Midtgaard et al. 2017; Pałka et al. 2011). Furthermore, incremental in nature, randomized testing allows for a gradual increase of the set of properties being checked, following the "pay-as-you-go" principle and requiring little to no change in the tests to account for a constant evolution of a tested codebase.

In this paper, we describe our experience of using QuickChick (Lampropoulos and Pierce 2018), a state-of-the-art framework, implemented on top of Coq proof assistant, for randomized property-based testing of Coq and OCaml programs, inspired by QuickCheck (Claessen and Hughes 2000), to find bugs in the implementation of Scilla smart contract language (Sergey et al. 2019).

Why Scilla? Scilla combines a pure functional calculus based on System F with state-managing imperative computations and message-passing semantics for communication between contracts. The combination of these aspects makes it challenging to define random generators that provide good distribution of well-typed programs, while avoiding degenerate cases, such as those with uninhabited polymorphic types. Most of these aspects are not specific to Scilla and make an appearance in other popular smart contract languages, such as Plutus (IOHK Foundation 2019) and Michelson (Tezos Foundation 2018). However, to the best of our knowledge, there are no random program generators for property-based testing of any of those language's implementations. Scilla's semantics is defined by means of a monadic definitional interpreter written in OCaml, which streamlines testing basic properties, such as the soundness of its type checker, and also, with some harness, testing advanced ones, such as soundness of various static analysers. Finally, Scilla has been deployed on top of Zilliqa—a real-world blockchain protocol; therefore, any bugs found in its implementation would be likely taken seriously and fixed by the developers.

Why QuickChick? QuickChick is a property-based testing framework for the Coq proof assistant. Its guiding design principle is to fully support power users writing and fine-tuning hand-crafted generators for random data, while adding type-based and specification-based automation on top of that (Lampropoulos 2018). This made it perfect for an extended case study like the one presented in this paper, as the automatically obtained generators could serve as a starting point for the more elaborated program generation strategies. In addition, while originally designed to test Coq code, QuickChick operates via extraction to OCaml, allowing for access to side effects (such as printing counterexamples and pseudo-random number generation). By adapting this extraction process, we were able to use QuickChick to test the extended OCaml codebase of Scilla. In contrast with other existing framework for property-based in OCaml (qcheck,[1] crowbar,[2] and base_quickcheck[3]) that struggle with generating good distributions for hierarchical data (*e.g.*, program terms), QuickChick combines coverage-guided fuzzing of properties (Lampropoulos et al. 2019), targeted property-based testing (Löscher and Sagonas 2017), and combinatorial testing (Goldstein et al. 2021), all of which have been shown to allow for defining fine-tuned program generators, while significantly cutting the space of tests required to discover bugs in language implementations.

Contributions and outline. In summary, this paper makes the following contributions that will be of interest to the practitioners who work on implementations of interpreters and compilers for higher-order languages, general-purpose ones and for blockchain programming in particular:

- We present a novel approach to efficiently generate well-typed programs in System F. We also outline how to generate imperative state-manipulating code and full-blown Scilla smart contracts, as well as inputs to simulate the contracts' executions on a blockchain (Sec. 3).
- We implement these generation strategies in QuickChick and use them to test the correctness of Scilla evaluator and type checker (Sec. 4).
- We demonstrate how a monadic implementation of Scilla interpreter (Sec. 2) allows for testing correctness of abstract interpretation-based higher-order control- and type-flow analyses by relating their results to the collecting semantics of a concrete program execution (Sec. 5).
- We present the bugs discovered and rediscovered in the Scilla type checker, interpreter, and type-flow analysis using QuickChick, and discuss their possible impact in the context of blockchain-based programming (Sec. 6).

2 A BRIEF OVERVIEW OF SCILLA

Scilla is an ML-style functional language combining pure functional and structured imperative programming, targeted at implementing smart contracts in the form of state-transition systems that communicate via message-passing (Sergey et al. 2019). Fig. 1 shows the Scilla encoding (with some details omitted) of the most popular smart contract used to define custom currencies (so-called *fungible tokens*), following the ERC-20 standard of the Ethereum community (Ethereum Foundation 2018). Once can think of this definition as a "template" for contracts implementing this logic, somewhat similar to Java classes. The implementation starts by declaring the Scilla version (line 1), followed by the library of pure functions (lines 3–5) that can be utilised by this contract as well as by other contracts that import the library. The contract definition (lines 7–29) features a number of immutable contract parameters (line 8), mutable fields (lines 10–12), and six transitions (lines 14–29) which determine how external entities may interact with the contract. A transition is invoked by an external entity passing a message to the contract. Each transition contains a number of effectful statements that result in a change to the contract state, the emission of externally observable events, and the addition of messages to the contract's outbox via the **`send`** statement. For instance, the transition `BalanceOf` responds with the token balance of the `tokenOwner` account by scheduling the corresponding message for delivery upon the end of the transition.

Individual transitions can only implement simple control flow logic, featuring straight-line code, branching by means of pattern matching on a value, and **`for`**-loops. This is a design choice aimed to deprecate contract reentrancy "in the middle of" a transition, which is considered a harmful practice in smart contract programming (Cecchetti et al. 2021; Gün Sirer 2016). Transitions may

[1] https://github.com/c-cube/qcheck
[2] https://github.com/stedolan/crowbar
[3] https://github.com/janestreet/base_quickcheck

```
1   scilla_version 0
2   library FungibleToken
3   let min_int : Uint128 ⟶ Uint128 ⟶ Uint128 = (* ... *)
4   let le_int  : Uint128 ⟶ Uint128 ⟶ Bool    = (* ... *)
5   let one_msg : Msg ⟶ List Msg = (* Return singleton List with a message *)
6
7   contract FungibleToken
8   (owner : ByStr20, total_tokens : Uint128, decimals : Uint32, name : String, symbol : String)
9
10  field balances : Map ByStr20 Uint128 =
11    let m = Emp ByStr20 Uint128 in builtin put m owner total_tokens
12  field allowed : Map ByStr20 (Map ByStr20 Uint128) = Emp ByStr20 (Map ByStr20 Uint128)
13
14  transition BalanceOf (tokenOwner : ByStr20)
15    bal ⟵ balances[tokenOwner];
16    match bal with
17    | Some v ⇒
18      msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : v};
19      msgs = one_msg msg; send msgs
20    | None ⇒
21      msg = {_tag : "BalanceOfResponse"; _recipient : _sender; address : tokenOwner; balance : zero};
22      msgs = one_msg msg; send msgs
23    end
24  end
25  transition TotalSupply () (* code omitted *) end
26  transition Transfer (to : ByStr20, tokens : Uint128) (* code omitted *) end
27  transition TransferFrom (from : ByStr20, to : ByStr20, tokens : Uint128) (* code omitted *) end
28  transition Approve (spender : ByStr20, tokens : Uint128) (* code omitted *) end
29  transition Allowance (tokenOwner : ByStr20, spender : ByStr20) (* code omitted *) end
```

Fig. 1. The FungibleToken smart contract implementation in SCILLA.

locally define and apply *pure* functions, as well as apply functions from the contract's own library or from imported libraries defined by other, previously deployed contracts. In the example above the `BalanceOf` transition uses the function `one_msg` defined in the contract's own library.

The pure fragment of SCILLA corresponds to an explicitly typed implementation of the System F calculus (Girard 1972; Reynolds 1974), extended with built-in primitive types and operations, user-defined (non-inductive) algebraic data types, as well as a number of embedded inductive polymorphic data types equipped with pre-defined recursion principles. That is, SCILLA programmers only get access to a number of polymorphic structural recursion combinators (*aka* folds) implementing traversals of collections in a strictly terminating manner.

For example, the following code fragment demonstrates the SCILLA encoding of the standard polymorphic implementation of list reversal by using one of the language's embedded recursion primitives, namely, `list_foldl`. The second line explicitly instantiates the polymorphic type scheme of the generic `list_foldl` with the type of the list element `'E` and the accumulated result (`List 'E`); the rest of the code applies the specialised `fold` to the iterated function and the initial value.

```
1   let list_reverse : forall 'E. List 'E ⟶ List 'E = tfun 'E ⇒
2     let foldl = @list_foldl 'E (List 'E) in
3     let iter = fun (z : List 'E) ⇒ fun (h : 'E) ⇒ Cons {'E} h z in
4     let init = Nil {'E} in foldl iter init
```

In practice, most non-trivial SCILLA code is written in its pure functional fragment, leaving only the most primitive state-manipulating logic (*e.g.*, reading and assigning to contract fields or map values—*cf.* line 15 of Fig. 1) to the imperative logic of transitions.

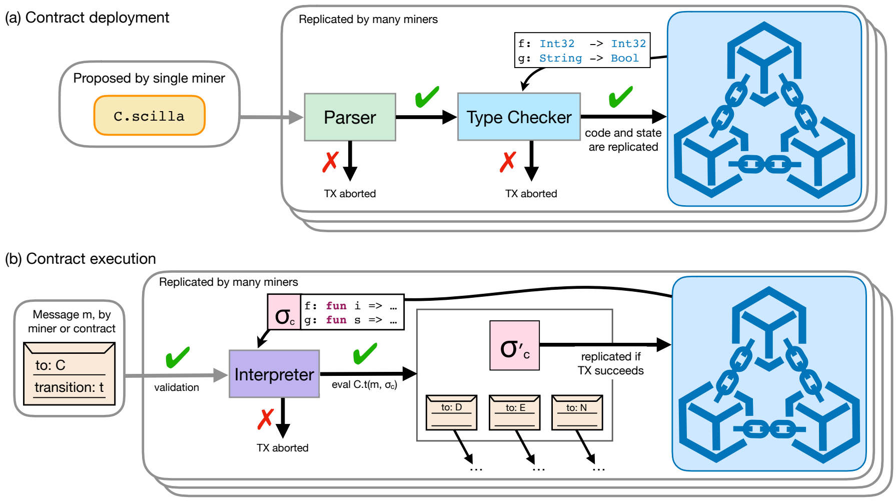

Fig. 2. Contract deployment (top) and contract execution (bottom) with the blockchain-replicated state.

2.1 Deploying and Executing Scilla Contracts

The top part (a) of Fig. 2 demonstrates the deployment pipeline. Once a contract, *e.g.*, `C.scilla` is proposed by a protocol participant (*aka miner*), it is parsed and type checked by a number of active miners participating in the protocol. Since the transition might use functions that are contained in other, previously deployed contracts (*e.g.*, `f` and `g` of types `Int32` → `Int32` and `String` → `Bool`, shown in the figure), type checking requires fetching their signatures from a local copy of the replicated state, available to each miner. Once the checks succeed, the contract's code and initial state are replicated and the transaction is considered committed. Once the described deployment procedure has been executed, a contract's code *cannot* be modified, and each network node will, from there on, have its own local copy of the contract.

The bottom part (b) of Fig. 2 shows a part of a transaction involving the invocation of a transition of an already deployed contract `C`. The transaction is initiated by a *message* `m` sent by a user to another account. Assuming this account belongs to a contract `C`, the sender also provides a name `t` of the transition to be invoked, as well as all necessary parameters as per the contract's signature. Each of the involved miners will then (*i*) validate the incoming message, (*ii*) fetch the most up-to-date state $\sigma_{\mathtt{C}}$ of `C` as well as external functions necessary for the execution of `t`, and (*iii*) invoke the Scilla interpreter. If successful, the execution will result in an updated state $\sigma'_{\mathtt{C}}$, as well as possibly a number of observable events and messages to be sent to other contracts and users. The order in which messages are processed is predetermined by the protocol semantics. Finally, if all the subsequently invoked transition executions succeed within the span of this transaction, the resulting states are stored locally, and thereby replicated by the miners across the blockchain.

2.2 Monadic Interpreter

The semantics of Scilla is implemented via a definitional interpreter written in a monadic style. Fig. 3 shows a characteristic example of evaluating some expressions from the pure fragment of Scilla: literals, variables, **`let`**-bindings, and cost-annotated `GasExpr (g, e')`, which will be explained soon. The definition also features semantics for the typed fixpoint combinator, although the combinator is only accessible to language users through the predefined well-founded *fold* functions.[4] The definition of the monad and its notation are imported from the module `EvalMonad.Let_syntax` in line 2. Monadic binding is encoded via the `let%bind` notation; the meaning of the `return` is standard. Using a monadic data type for representing intermediate execution states, the evaluator keeps track of the following program execution effects:

- whether the execution is successful so far or has failed, and
- the remaining execution budget (so-called "gas") tracked in the style of State monad.

The actual monad instance used by the evaluator is implemented by the following data type `t`:

```
type ('a,'b) result = Ok of 'a | Error of 'b
type nonrec ('a, 'b, 'c) t = (('a, 'b) result -> 'c) -> 'c
```

which is nothing but a slightly specialized continuation-passing style (CPS) monad.[5] A pleasant consequence of using the CPS monad is that it does not have to mention, *e.g.*, the gas-tracking state component explicitly, as it can be "added" by specializing the *return type* `'c`—a well-known trick for representing layered monads in CPS (Filinski 1994). Foreshadowing the discussion in Sec. 5, we have to note that the design choice of using CPS is what made it possible to harness the interpreter in a modular way for testing static analyses of Scilla programs.

Taking a closer look at Fig. 3, the reader might have noticed that the interpreter code is not perfectly monadic: at some point it has to "cut" the passed monadic datatype, revealing the internals of its implementation. Specifically, the interpreter must check, at each evaluation step, whether the user's allotted amount of gas covers the cost of the future execution, decreasing the remaining gas in the state component or raising an *out-of-gas* error. To enable this accounting in a "big step"-style evaluator, Scilla programs are pre-processed, so that all "expensive" expressions are annotated with their *gas costs* by wrapping them into `GasExpr`. When processing an expression of this kind, the interpreter calls `checkwrap` in line 13 of Fig. 3, which checks whether the user has depleted their gas resource, while handling the continuation `k` explicitly. Finally, lines 7 and 9 of Fig. 3 call the function `wrap_eval`, which virtualizes the recursive calls to the evaluator, supplementing them with additional runtime data (*e.g.*, `(e, E lval)`), which is ignored during the "normal" evaluation, but will come useful for testing purposes, as we will show in Sec. 5.

```
 1  let rec exp_eval (e, loc) env = match e with
 2    let open EvalMonad.Let_syntax in
 3    | Literal l -> return (l, env)
 4    | Var i ->
 5        let%bind v = Env.lookup env i in return (v, env)
 6    | Let (i, _, lhs, rhs) ->
 7        let%bind lval, _ = wrap_eval lhs env (e, U) in
 8        let env' = Env.bind env (get_id i) lval in
 9        wrap_eval rhs env' (e, E lval)
10    | GasExpr (g, e') ->
11        let thunk () = exp_eval e' env in
12        let%bind cost = eval_gas_charge env g in
13        checkwrap thunk (Uint64.of_int cost)
14          ("Insufficient gas")
15    | Fixpoint (g, _, body) -> (* Other cases *)
16
17    (* Gas accounting *)
18    let checkwrap op_thunk cost emsg k remaining_gas =
19      if remaining_gas > cost then
20        op_thunk () k (remaining_gas - cost)
21      else k (Error emsg) remaining_gas
```

Fig. 3. Fragment of interpreter for Scilla expressions.

3 GENERATION OF SCILLA PROGRAMS IN QUICKCHICK

The main building block of our system is the generation of well-typed lambda terms for the core calculus of Scilla. Generation of well-typed lambda terms is a rich research area, with Pałka et al. (2011) paving the way by finding bugs in Glasgow Haskell Compiler's strictness analyser by generating well-typed terms in simply-typed lambda calculus. In this work, Pałka et al. invert the

[4]Remember this fact, as it will make an appearance in Sec. 6.1, where we will discuss found bugs.
[5]The most general CPS monad type `('a -> 'c) -> 'c` is not used for performance reasons.

traditional reading of typing rules using them as generation rules. Given a context Γ and a type τ, we can generate a well-typed expression e by selecting at random a typing rule that can be used to conclude that an expression has the type τ, and then building up the expression in a way that satisfies the premises of the rule. For example, consider the standard typing rule for products:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}\ \textsc{TProd}$$

In order to generate an expression with a product type $\tau_1 \times \tau_2$, we need to construct a pair of two expressions e_1 and e_2. Looking at the premises of the rule, these expressions need to be recursively generated so that they each have types τ_1 and τ_2 respectively.

Unlike the work of Pałka et al., the core of Scilla is based on System F, not the simply typed lambda calculus. The additional layer of abstraction provides an interesting challenge, particularly when trying to generate type applications. Consider then, again standard, typing rules for type abstraction and application, where the context Γ keeps track of the types of variables in scope, and the context Δ keeps track of type variables that have been introduced through type abstraction:

$$\frac{\Gamma; \Delta, \alpha \vdash e : \tau}{\Gamma; \Delta \vdash \Lambda\alpha.e : \forall\alpha.\tau}\ \textsc{TTyAbs} \qquad \frac{\Gamma; \Delta \vdash e : \forall\alpha.\tau}{\Gamma; \Delta \vdash e\ \tau' : \tau[\tau'/\alpha]}\ \textsc{TTyApp}$$

Viewing the type abstraction rule as a generation rule is straightforward: in order to generate an expression of type $\forall\alpha.\tau$, we simply need to generate a type abstraction $\Lambda\alpha.e$, while recursively generating e in an expanded context that also contains the type variable α. The type application rule proves much more challenging: in order to generate an expression of some type σ, we would need to generate two types, τ and τ', such that σ is the result of the substitution $\tau[\tau'/\alpha]$, for some type variable α. Our solution to this problem is a generation process we called *unsubstitution.*

Unsubstitution. The key intuition behind unsubstitution is that we traverse our target type σ, looking for a closed *syntactic* sub-type τ' that appears somewhere in the tree structure of σ. Then, we traverse σ again and replace (with some probability) any occurrences of τ' we find with a fresh type variable α. The resulting type τ will satisfy the desired substitution equation $\tau[\tau'/\alpha] = \sigma$ by construction. While at a high level unsubstitution appears straightforward, the devil lies in the details, and the details, in this case, are in implementing and fine-tuning the distribution of generated (τ, τ') pairs to avoid generating degenerate cases of type substitution too often.

The trickiest part of unsubstitution, by far, is generating a closed sub-type τ' of a given type σ. Implementing such a generator recursively requires to not only select a random sub-type, but also to keep track of information that ensures closedness and allows for fine-tuning the distribution. To that end, the return type of this recursive process is a triple: the generator itself, the number of sub-types that this generator can produce (to fine tune the distribution), and a set of type variables (if any) that need to be abstracted over before the type becomes closed. This is shown in Figure 4.

Generating a closed sub-type of a type variable α is impossible; therefore *gClosed* does not return a generator ($\perp$) but returns the set $\{\alpha\}$ to signify that a type abstraction over α is needed before the potential sub-type becomes closed. For the rest of the base cases (such as various fixed-width integers, byte-strings, *etc*), we can just return the singleton generator that always produces this type (written as $\{1 \mapsto \tau\}$), and the empty set signifying that no type abstractions are needed.

In the type abstraction case $\forall\alpha.\tau$, we first recursively call *gClosed* on τ to obtain a triple (g, n, s). Then, we need to decide if $\forall\alpha.\tau$ is a closed subtype—the case when the only free variable that appears in τ is α, in which case s would be the singleton set $\{\alpha\}$. In this case, we construct a generator that with $\frac{1}{1+n}$ of the time produces $\forall\alpha.\tau$, and with $\frac{n}{1+n}$ of the time defers to the generator for closed subtypes of τ (which can produce n subtypes, as signified by its result). This way, we

```
$gClosed\ \alpha = (\bot, 0, \{\alpha\})$                                   Type variable case
$gClosed\ \tau = (\{1 \mapsto \tau\}, 1, \emptyset)$,                        Other base types
$gClosed\ (\forall\alpha.\tau) =$                                            Type abstraction case
    let $(g, n, s) = gClosed\ \tau$ in
    if $s = \{\alpha\}$ then $(\{1 \mapsto \forall\alpha.\tau\} \bigcup \{n \mapsto g\}, 1 + n, \emptyset)$
    else $(\{n \mapsto g\}, n, s/\alpha)$
$gClosed\ (op\ \overline{\tau_i}) =$                                         Other recursive type constructions
    let $(g_i, n_i, s_i) = gClosed\ \tau_i$ in
    if $\forall i, s_i = \emptyset$ then $(\{1 \mapsto op\ \overline{\tau_i}\} \bigcup_i \{n_i \mapsto g_i\}, 1 + \sum n_i, \emptyset)$
    else $(\bigcup_i \{n_i \mapsto g_i\}, \sum n_i, \bigcup s_i)$
```

Fig. 4. Generation of closed sub-types

maintain the invariant that *generation is uniform in the number of closed sub-types possible.* On the other hand, if there are still type variables that need to be abstracted, we simply propagate the inner result, making sure to remove α from the set of abstractions remaining to achieve closedness. Finally, in the rest of the recursive cases (such as arrow types, sums, and products), which we group together in Figure 4 as $op\ \overline{\tau_i}$, we operate similarly, calling *gClosed* for all types τ_i, and then combining the results including $op\ \overline{\tau_i}$ in the result only if all τ_i are closed (and therefore their corresponding s_i are the empty set).

Armed with *gClosed*, to complete unsubstitution we only need to traverse the type once more, looking for occurrences of τ' to abstract over. Since multiple such occurrences can exist, we only replace one with α probabilistically. This also ensures that sometimes the resulting substitution is a trivial one as with a small probability α will not exist in the resulting type.

Statements, transitions, and contracts. Using the generation of System F terms as a building block, the next step is to generate programs for the imperative fragment of the contract language that involves generating statements, events, and transitions. We implemented a goal-oriented approach to ensure that each transition finishes by either (1) raising an event, (2) sending a message, or (3) calls a helper function leading to one of the previous two outcomes. Both events and messages are quite similar - we simply need to generate a record that has some predefined fields and (potentially) some additional ones. To populate these records we follow an approach inspired from generation-by-execution (Hritcu et al. 2016): we keep track of an abstraction of the state of the program at any given point and generate statements one-by-one, ensuring that each one makes sense according to this abstraction. We also keep track of an environment containing all valid identifiers and use this environment to seed the functional fragment generator.

Inputs to contracts. After generating a contract, calling it with different inputs is a straightforward task. We can simply pick one of the transitions of the contract and invoke it with appropriately typed parameters using, once again, the generator of System F terms as the basic building block. The one caveat is that each contract in Scilla also comes with a number of *invariants* that are expected to hold on to its parameters (*e.g.*, a transition takes a list of unique keys). For our experiments, these invariants were also generated randomly. Most of them had to do with application-specific boolean constraints on a contract's parameters and fields. For those, the generator only produced small boolean expressions, which were straightforward to satisfy most of the time. This is why the generate-and-filter approach for data satisfying those invariants was effective. More intricate invariants, such as, for instance, ensuring that certain byte-strings have a fixed length, would indeed be almost impossible to generate-and-filter for; luckily we could trivially enforce them via ordinary Scilla types (*e.g.*, `ByStr20`). In the future, should these invariants prove too unwieldy to satisfy in this manner, we might need to turn to smarter methods of generation that take the structure of these invariants into account. Finally, to amortize the (non-trivial) cost of generating a contract, we re-use it to generate multiple inputs to invoke it with.[6]

4 TESTING BASIC LANGUAGE PROPERTIES

With the generation infrastructure described in the previous section in place, we proceeded to test various basic properties. Whenever developing generators as large as the ones required for this project (~1500 lines), it is imperative to establish confidence in the correctness of generation itself before testing language properties.

Property 1 (*Correctness of Program Generation*): Given a generator, all terms produced by the generator are well-typed in an empty environment.

In principle, every single term produced by a generator should be well-typed if we follow the algorithm outlined in the previous section. Any violations of this property then, would most likely imply an error in the implementation of the generator, which allows it to produce ill-typed terms. As an aside, the dual mistake, where a generator never produces some particular element of its intended co-domain is much more insidious and can't be ruled out through testing: the only known way would be to prove the generator's completeness (Lampropoulos et al. 2018), but that would be an extreme undertaking compared to its expected benefit. Sometimes, however, a violation of the property above can reveal an error in type checking, as the generator might produce a term that is wrongly rejected by the type system. Such errors do not lead to safety concerns while making the type system more restrictive than necessary. We didn't discover such errors in our case studies.

Property 2 (*Type Soundness*): For all well-typed terms e, e should evaluate to a value v of the same type without error, given sufficient gas.

This property is the Scilla version of the famous mantra: "well-typed programs don't go wrong". Given enough fuel to perform the necessary reductions, term evaluation should always succeed in producing a value of the same type. This property revealed the majority of the bugs described in Sec. 6. The gas accounting aspect has turned out to be quite important for discovering subtle bugs resource accounting implementation of Scilla. In our tests, we used a conservatively heuristic, allocating roughly an amount of gas quadratic in (the size of the program) × (the product of integers it manipulates with) × (the number of nested `folds`), assuming no reasonable computation should exceed this amount. With this strategy, an out-of-gas error helped to discover a gas accounting bug #9 (*cf.* Tab. 1), which we describe in more detail in Sec. 6.1.

5 TESTING STATIC ANALYSES WITH A STATE-COLLECTING INTERPRETER

In the previous section, we have outlined the properties of a language implementation, whose checking does not require any extra harness besides the definition of the property itself, as the corresponding artifacts: program generator, interpreter, and type checker are immediately available.

Checking the correctness of more intricate components, such as control-flow analysis, is a bit less straightforward and requires an additional harness for the definitional interpreter. The reason for that is that in higher-order languages the result of control- and data-flow analyses is typically defined as an over-approximation of the program's outcome in operational *state-collecting* semantics (Shivers 1991). Therefore, in order to test an analysis for soundness, the collecting

[6]As an aside note, generating state constraints randomly is a challenging research problem in its own right! A random collection of constraints is, with very high probability, either trivially satisfiable or trivially unsatisfiable. Producing such collections that are both satisfiable and hard to randomly satisfy (and then using them as transition invariants to generate inputs for) is an intriguing avenue for future work—but well beyond the scope of this paper.

semantics, which keeps intermediate execution states, needs to be implemented, ideally, with the minimal changes introduced to the main codebase.

5.1 Embedding Collecting Semantics to Scilla Evaluator

Luckily for us, the implementation of Scilla interpreter in a monadic form allows for introducing such changes in a modular way by piggy-backing on the original interpreter's structure (Sec. 2.2). In essence, the main additions that need to be done are (a) to refine the result type of the used CPS monad so, in addition to keeping track of gas, it would also collect the intermediate execution states, and (b) change the logic of `wrap_eval` accordingly. Instrumenting the evaluator with a state-collecting machinery follows the implementation of gas accounting in the original interpreter.

As previously mentioned, implementation of gas accounting is done by elaborating the continuation's return type `'c` into `uint64 → 'd`, where 'd is another abstract type, and `uint64` is an OCaml 64-bit integer representing the remaining amount of gas for the ongoing computation. By elaborating the type signature of the evaluator further to include another state component `CollectedStates` as an argument (*i.e.*, transforming `uint64 → 'd` to `uint64 → CollectedStates → 'd`), we can add the state-collecting component. The actual state accounting is handled by the `wrap_eval` function, which now records the current expression being evaluated at the corresponding evaluation stage, as well as some additional information passed along with it.

A note on related work. The idea of property-based testing of static analyses as a particular component of a compiler pipeline is, of course, not new. Prior work addressed validation of static analyses for soundness and precision by performing differential testing of an abstract and concrete execution (Klinger et al. 2019) and by using SMT solvers to obtain the most precise results, which are compared to the actual analysis outcomes (Taneja et al. 2020). Several works have even employed QuickCheck-inspired methodology to validate monotonicity of abstract state transformers as well as algebraic properties of the abstract domain on randomly generated abstract state components (Madsen and Lhoták 2018; Midtgaard and Møller 2015). That said, we are not aware of any work that uses randomly-generated programs to test soundness of Shivers-style static analyses for higher-order languages against state-collecting semantics.

The idea of implementing a collecting semantics, as a particular instance of a monad used to define a small-step of definitional evaluator has been explored before (Darais et al. 2017; Sergey et al. 2013), albeit not in the context of an OCaml implementation, and only for toy functional languages. Those approaches also showed how to define a static analysis semantics by picking a suitable monad instance, assuming an *interpreter-based* style of the analysis definition à la Might (2010). That analysis formulation, however, does not immediately support a more traditional way of defining a higher-order analysis (used, in particular, in the Scilla compiler) as a solution to a system of *data-flow constraints* (Nielson et al. 1999, Chapter 3).

The novelty of our work is, therefore, in adopting the idea of a monadic collecting semantics for *testing* soundness of a constraint-based analysis for a higher-order language.

5.2 Testing Runtime Type Conformance and Type-Flow Analysis

We use the state-collecting evaluator to test two properties: runtime type conformance and the type-flow analysis implemented as a pass of the Scilla to LLVM compiler (Nagaraj et al. 2020).[7]

5.2.1 Testing Type Conformance. The type checker correctness outlined in Sec. 4 ensures that the result of a functional expression in Scilla is consistent with its inferred type. In the context of imperative executions, it is also important to check that intermediate variables and function

[7]The compiler is not fully finished and has not yet been made an official part of the Zilliqa blockchain client.

parameters only get assigned values consistent according to their declared types—a property whose violation might result in runtime errors produced by a type-driven compiler. The collecting evaluator makes defining this property straightforward; the only noteworthy aspect to mention is that the conformance is checked with respect to *runtime types*, *i.e.*, at the moment when the type variables are already instantiated with ground types. While relatively simple, this check allowed us to discover a previously unknown bug in Scilla type checker, which we describe in Sec. 6.1.

5.2.2 Testing the Type-Flow Analysis. The compiler from Scilla to LLVM takes the full advantage of the closed-world setup, in which smart contracts are deployed, meaning that the sources of all used libraries are known at the compile time (*cf.* Sec. 2.1). This allows for an aggressive whole-program optimizations, such as, *e.g.*, full monomorphization of polymorphic definitions in the style of the MLton compiler of Standard ML (Weeks 2006). Unlike Standard ML and like System F, Scilla features rank-k polymorphism for arbitrary k, thus monomorphization in it requires a carefully designed *type-flow analysis* that would allow to conservatively determine, which type variables are instantiated with what *ground types* (*i.e.*, types without variables) at runtime, in order to generate specialized implementations for type-polymorphic functions.

An outcome of the type-flow analysis is a map from all unique type variables in a Scilla program to the sets of ground types they can be instantiated with. Equipped with the collecting interpreter, we can test the correctness of the analysis by checking whether those sets contain all type instantiations witnessed at run-time. For example, given a type function that takes a type variable `'X`, the analysis may predict that `'X` might be instantiated with types `Uint32` or `Bool`. Evaluating the function and *dynamically collecting* what type expressions are passed as arguments to the function confirms whether `'X` is indeed instantiated with `Uint32` and `Bool` types.

6 DISCOVERED BUGS

Tab. 1 catalogues the bugs in various components of Scilla implementation that have been discovered and rediscovered with the help of the framework. Some of the listed issues are marked as *known*: they have been previously discovered and fixed; we have reproduced them using QuickChick-powered automated program generation on the corresponding implementation snapshots. All of the newly found issues have been disclosed to the developers, who have fixed them and updated the version of Scilla implementation shipped with the Zilliqa blockchain client by the time of submission of this manuscript. Below, we elaborate on some of the most interesting issues discovered in each component of the language implementation. The test generator and the instructions for reproducing the bugs can be found in the accompanying artifact (Hoang et al. 2022).

6.1 Bugs in the Type Checker and the Evaluator

The definitional evaluator and the type checker outlined in Sec. 2 are the most critical pieces of the language implementation, as they are directly run by the miners who validate newly-deployed contracts and execute contract-involving transactions. Any bugs in these components are immediately marked as show-stoppers, as they can lead to potential exploits of the system.

The newly-discovered issues #3 and #4 both violate a particular aspect of Scilla type safety: non-serializable functional values (*e.g.*, closures) may not be hashed by the contracts, as their internal representation (and, hence, the result of hashing) can be a subject of future changes. We have checked the contracts deployed on the Zilliqa mainnet so far, and, luckily none of them have relied on this behaviour that is now disallowed. The type checker bug #5 is of an unusual nature and has been discovered thanks to the collecting semantics described in Sec. 5.2.1. The issue arose from the fact that values of fixed-length sub-types of `ByteString` (*e.g.*, `ByStr32`) were implicitly

Tab. 1. Discovered and rediscovered bugs in Scilla implementation, grouped by category

ID	Short bug description	Status
	Type checking and type inference	
#1	Closure values could be used as map keys	known
#2	Type variables were not properly shadowed; the bug allowed for encoding non well-formed recursion	known
#3	Type checker allowed for hashing closure values	new
#4	Type checker allowed for hashing polymorphically-typed values	new
#5	Sub-types of address type `ByteString` were implicitly up-cast to type `ByteString`	new
	Definitional interpreter	
#6	Conversion between `bech32` and `ByStr20` datatypes threw an exception	new
#7	Cryptographic built-in operations `ecdsa_verify` and `ecdsa_recover_pk` were throwing exceptions	new
#8	Cryptographic built-in `ecdsa_recover_pk` could abort Scilla interpreter with an OS-level exception	new
#9	The interpreter inadequately charged gas for the `power` arithmetic operation	new
	Type-flow analysis	
#10	Type-flow analysis does not terminate on programs that make use of impredicative polymorphism	known

up-cast to `ByteString`. While this bug did not compromise the safety of contract execution, it might have potentially affected the correctness of the compiler.

The issues #7–#8 in the evaluator have to do with built-in conversions of primitive data types (*e.g.*, byte-strings) as well as handling cryptographic signatures. According to the design of Scilla, those operations return a result of the type `Option`, that is, a failed value conversion or validation of a cryptographic signature results in returning `None`, which can be handled accordingly by the client code. In contrast, the evaluator threw exceptions in all those cases, resulting in the rollback of the corresponding transaction code and allowing for potential denial-of-service attacks.

The issue #9 is specific to smart contracts. For computing an exponential of a natural number, the implementation of the interpreter charged the cost logarithmic in the size of the exponent, while the implementation was linear. The bug was discovered due to the test case exceeding the allocated gas cost upon its execution. As the fix, the implementation has been changed to the efficient one, adequate to the gas cost to avoid possible gas-related exploits (Perez and Livshits 2020).

Finally, of the previously known issues #2 is particularly severe, as it undermined the runtime guarantees of Scilla, allowing to encode non-well-founded recursion by implementing fixed-point combinator. This bug has been discovered a members of the Scilla development team using a carefully designed example featuring rank-2 polymorphism. QuickChick was capable to generate a much more compact yet illuminating test case featuring arbitrary type instantiations, which were the root cause of the bug, resulting in run-time errors for seemingly well-typed programs. An example of a randomly generated Scilla function reproducing the bug is given below:

```
let a =
  let b =
    let c =
      let d = tfun 'V ⇒ fun (v1 : 'V) ⇒ tfun 'V ⇒ v1
      in @d (ByStr32)
    in
    let e = 0x0000000000000000000000000000000000000000000000000000000000000002
    in c e
  in @b (forall 'V. Nat)
in @a Nat
```

In the example above, the type of `d` is `forall 'V. 'V → forall 'W. 'V`. That is, the type `'V` of the result value `v1` might be different from the type `'W` of its closest bound type variable. However, since both bonds type variables in the definition share the same name `'V`, the bug occurs, as the rest of the program demonstrates. The definition of `c` instantiates the first type variable with `ByStr32`, and the definition of `c` subsequently applies the function to the correct argument byte-string `e`. The definition of `a` instantiates the *second* type variable in the definition of `d` with `forall 'V. Nat`. Finally, `a` itself is instantiated with `Nat`, which should be detected as a type error, as the type of `a` is `ByStr32`, not `forall 'V. Nat`! However, due to the type checker not performing alpha-renaming of nested bound type variables upon substitution, this error was not caught until runtime.

6.2 Type-Flow Analysis: Soundness and Precision

As mentioned in Sec. 5.2.2, we have used the state-collecting version of Scilla interpreter to validate the soundness of the type-flow analysis by checking that the set of ground types used to instantiate each type variable at runtime is a *subset* of the types predicted by the analyser.

The problem of designing a sound type-flow analysis for languages with higher-ranked polymorphism has been studied by Fluet (2012) in a more general setting that assumed polymorphic recursion, making the space of run-time types potentially infinite, and requiring an intricate abstract domain to soundly approximate them. Even though Scilla does not have polymorphic recursion, designing a simple yet finite abstract domain for its type-flow analysis appears to be challenging due to the possibility to, *e.g.*, dynamically convert "flat" integers into polymorphically-typed Church numerals. To keep things simple, the type-flow analysis currently implemented in Scilla compiler follows the constraint-based approach based on the standard definition of context-insensitive higher-order control-flow analysis adapted for System F. The analysis is somewhat conservative in that it rejects a small class of programs that introduce "expansive" polymorphic types that can grow arbitrarily large at runtime. This restriction has been introduced to address a subtle (but, alas, already known) bug #10 in the analysis design, which we have reproduced using our framework.

The bug was caused by programs that aggressively use System F's impredicativity, such as, *e.g.*, exponentiation of typed Church numerals—a program, in which a higher-kinded type is used to instantiate another higher-kinded type. The initial version of the analysis would not terminate when solving constraints introduced by such programs, and fixing this issue properly would require introducing a more complex abstract domain akin to the one by Fluet. It has been observed that (a) programs of this kind are extremely rare in practice and (b) an attempt to monomorphize them would result in an exponential explosion of the generated code in size. Luckily, the fix was relatively easy, and boiled down to detecting "expansive" cycles in the generated analysis constraints (which can be done in linear time in the size of the source program) and rejecting such programs altogether.

Other than that, the implementation of the type-flow analysis turned out to be surprisingly robust, and we did not manage to discover any new bugs. Somewhat curiously, our test revealed that on randomly-generated programs the analysis is amazingly precise: for almost all randomly generated programs it has inferred the *exact set* of ground types for the corresponding type variables. This precision can be explained by the fact that in many test programs generated by QuickChick a type abstraction has been syntactically instantiated *exactly once*, although often not immediately, *e.g.*, after being passed as a function parameter. In a small fraction of generated programs, several type instantiations took places in disjoint branches of a **match**-expression; this, obviously, has lead to a proper over-approximation with respect to a concrete collecting execution, which only visited one branch. Our testing framework discovered no bugs in such over-approximations.

7 DISCUSSION AND FUTURE WORK

Distribution of Efforts. We believe, our application of QuickChick is the largest to date, in terms of the size of the covered OCaml codebase and of the generator/shrinker development. That said, our choice to use QuickChick to test OCaml code was a double-edged sword. On the one hand, we were able to leverage its automation facilities to get a head start with generation, printing, and shrinking. On the other hand, the interoperability between OCaml and Coq proved surprisingly brittle, with a lot of manual effort put into maintaining this connection. For instance, we could not automatically port OCaml definitions of Scilla syntax to Coq, so we had to do so manually. We attempted to apply Coq-of-OCaml (Claret 2021) for automating this task, but discovered that it does not work so well with OCaml functors, which are used extensively in Scilla implementation for representing syntax with various annotations for different checking and interpretation phases.

Future Directions. While integrating QuickChick-powered property-based testing has been a considerable effort, now that this infrastructure is in place, we will use it for testing more interesting language properties, focusing specifically on multiple static analyses featured in the Scilla tool suite, such as *gas usage* and *cash-flow* analyses (Sergey et al. 2019, §§ 5.1 and 5.2). With a suitable definition of the abstraction function, we should be able to provide a similar validation procedure for the CoSplit analysis that derives contract signatures for sharding of contract-manipulating transactions executed on Zilliqa blockchain (Pîrlea et al. 2021). Finally, we are hopeful that our integration of QuickChick into a fairly sizeable and actively maintained OCaml development will provide a rich case study for future approaches contributing to the state of the art in property-based testing, fuzzing, and random generation of higher-order and effectful programs.

ACKNOWLEDGMENTS

We thank Vaivaswatha Nagaraj for his help with testing the Scilla to LLVM compiler. We are grateful to Andreea Costea, Kiran Gopinathan, and George Pîrlea for their feedback on drafts of this paper. We also thank the ICFP'22 anonymous PC and AEC reviewers for their constructive and insightful comments on the paper and the artifact.

The work by Sergey was supported in part by Singapore MoE Tier 1 Grant No. IG18-SG102. The work by Lampropoulos was supported in part by NSF award #2107206 "Efficient and Trustworthy Proof Engineering". Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

REFERENCES

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language With Programmable Resources. Available at https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf.

Ethan Cecchetti, Siqiu Yao, Haobin Ni, and Andrew C. Myers. 2021. Compositional Security for Reentrant Applications. In *42nd IEEE Symposium on Security and Privacy*. IEEE, 1249–1267. https://doi.org/10.1109/SP40001.2021.00084

Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*. ACM, 268–279. https://doi.org/10.1145/351240.351266

Guillaume Claret. 2021. Coq-of-OCaml, A Compiler from OCaml to Coq. Available at https://formal.land/docs/coq-of-ocaml/introduction.

Michael Coblenz. 2017. Obsidian: A Safer Blockchain Programming Language. In *ICSE (Companion)*. IEEE Press, 97–99. https://doi.org/10.1109/ICSE-C.2017.150

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *PACMPL* 1, ICFP (2017), 12:1–12:25. https://doi.org/10.1145/3110256

Ankush Das, Stephanie Balzer, Jan Hoffmann, Frank Pfenning, and Ishani Santurkar. 2021. Resource-Aware Session Types for Digital Contracts. In *CSF*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00004

Ethereum Foundation. 2018. ERC20 Token Standard. https://en.bitcoinwiki.org/wiki/ERC20 Online.

Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM Press, 446–457. https://doi.org/10.1145/174675.178047

Matthew Fluet. 2012. A Type- and Control-Flow Analysis for System F. In *IFL (LNCS, Vol. 8241)*. Springer, 122–139. https://doi.org/10.1007/978-3-642-41582-1_8

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse d'État. Université de Paris VII, Paris, France.

Harrison Goldstein, John Hughes, Leonidas Lampropoulos, and Benjamin C. Pierce. 2021. Do Judge a Test by its Cover - Combining Combinatorial and Property-Based Testing. In *ESOP (LNCS, Vol. 12648)*. Springer, 264–291. https://doi.org/10.1007/978-3-030-72019-3_10

Emin Gün Sirer. 2016. Reentrancy Woes in Smart Contracts. http://hackingdistributed.com/2016/07/13/reentrancy-woes/

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 185–200. https://doi.org/10.1145/3062341.3062363

Tram Hoang, Anton Trunov, Leonidas Lampropoulos, and Ilya Sergey. 2022. *Random Testing of a Higher-Order Blockchain Language (ICFP 2022 Artifact): Code and Commentary.* https://doi.org/10.5281/zenodo.6610599

Catalin Hritcu, Leonidas Lampropoulos, Antal Spector-Zabusky, Arthur Azevedo de Amorim, Maxime Dénès, John Hughes, Benjamin C. Pierce, and Dimitrios Vytiniotis. 2016. Testing Noninterference, Quickly. *J. Funct. Program.* 26 (2016), e4. https://doi.org/10.1017/S0956796816000058

IOHK Foundation. 2019. Plutus: A Functional Contract Platform. https://testnet.iohkdev.io/en/plutus/ Online; accessed 23 February 2022.

Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *ISSTA*. ACM, 239–250. https://doi.org/10.1145/3293882.3330553

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192. https://doi.org/10.1145/2535838.2535841

Leonidas Lampropoulos. 2018. *Random Testing for Language Design.* Ph. D. Dissertation. University of Pennsylvania.

Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* 3, OOPSLA, 181:1–181:29. https://doi.org/10.1145/3360607

Leonidas Lampropoulos, Zoe Paraskevopoulou, and Benjamin C. Pierce. 2018. Generating good generators for inductive relations. *PACMPL* 2, POPL (2018), 45:1–45:30. https://doi.org/10.1145/3158133

Leonidas Lampropoulos and Benjamin C. Pierce. 2018. QuickChick: Property-Based Testing In Coq *(Software Foundations series, Volume 4)*. Electronic textbook. https://softwarefoundations.cis.upenn.edu/

Xavier Leroy. 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*. ACM, 42–54. https://doi.org/10.1145/1111037.1111042

Andreas Löscher and Konstantinos Sagonas. 2017. Targeted Property-Based Testing. In *ISSTA*. ACM, 46–56. https://doi.org/10.1145/3092703.3092711

Magnus Madsen and Ondrej Lhoták. 2018. Safe and sound program analysis with Flix. In *ISSTA*. ACM, 38–48. https://doi.org/10.1145/3213846.3213847

Jan Midtgaard, Mathias Nygaard Justesen, Patrick Kasting, Flemming Nielson, and Hanne Riis Nielson. 2017. Effect-driven QuickChecking of compilers. *Proc. ACM Program. Lang.* 1, ICFP (2017), 15:1–15:23. https://doi.org/10.1145/3110259

Jan Midtgaard and Anders Møller. 2015. QuickChecking Static Analysis Properties. In *ICST*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICST.2015.7102603

Matthew Might. 2010. Abstract Interpreters for Free. In *SAS (LNCS, Vol. 6337)*. Springer, 407–421. https://doi.org/10.1007/978-3-642-15769-1_25

Vaivaswatha Nagaraj, Jacob Johannsen, Anton Trunov, George Pîrlea, Amrit Kumar, and Ilya Sergey. 2020. Compiling a Higher-Order Smart Contract Language to LLVM. *CoRR* abs/2008.05555 (2020). https://arxiv.org/abs/2008.05555

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer. https://doi.org/10.1007/978-3-662-03811-6

Michał H. Pałka, Koen Claessen, Alejandro Russo, and John Hughes. 2011. Testing an optimising compiler by generating random lambda terms. In *AST*. ACM, 91–97. https://doi.org/10.1145/1982595.1982615

Daniel Perez and Benjamin Livshits. 2020. Broken Metre: Attacking Resource Metering in EVM. In *NDSS*. The Internet Society.

George Pîrlea, Amrit Kumar, and Ilya Sergey. 2021. Practical Smart Contract Sharding with Ownership and Commutativity Analysis. In *PLDI*. ACM, 1327–1341. https://doi.org/10.1145/3453483.3454112

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium (LNCS, Vol. 19)*. Springer, 408–423. https://doi.org/10.1007/3-540-06859-7_148

Franklin Schrans. 2018. *Writing Safe Smart Contracts in Flint.* Master's thesis. Imperial College London.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *PLDI*. ACM, 399–410. https://doi.org/10.1145/2491956.2491979

Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer smart contract programming with Scilla. *PACMPL* 3, OOPSLA (2019), 185:1–185:30. https://doi.org/10.1145/3360611

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* Ph. D. Dissertation. School of Computer Science, Carnegie Mellon University.

Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing static analyses for precision and soundness. In *CGO*. ACM, 81–93. https://doi.org/10.1145/3368826.3377927

Tezos Foundation. 2018. Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/whitedoc/michelson.html Online; accessed 23 February 2022.

Stephen Weeks. 2006. Whole-Program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML*. ACM. https://doi.org/10.1145/1159876.1159877

Gavin Wood. 2014. Ethereum: A Secure Decentralized Generalised Transaction Ledger.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*. ACM, 283–294. https://doi.org/10.1145/1993498.1993532