CS 6110 Final Report, University of Utah

Verifying Contracts for Dynamic Typing Language Statically

Guannan Wei, Jian Lan

May 11, 2016

1 Introduction

This project presents a method to verify contracts of functions in dynamic typing programming languages via abstract interpretation.

Motivation

Many popular languages such as Python, Perl, Ruby, JavaScript, and Scheme are dynamic typing, which means the type of the value of an identifier may change in runtime, and the type checking are performed in runtime rather than compile-time. This feature helps programmers to build prototype rapidly because programmers don't need to firstly write type annotations and make the annotations be accepted by type checker.

But dynamic typing also makes program much easier to have bugs since programmer can not catch errors until it is raised in runtime. For instance, as the following example code shows, the function `add1` executes a simple arithmetic operation that adds `1` to `x`. But if we pass a string argument `"3"` to `add1`, the type error will be raised *only* when program runs to that call site.

```
(define (add1 x) (+ x 1))    ; Defining a function
(add1 3)                     ; OK
(add1 "3")                   ; Error
```

Modern languages such as Racket, a dialect of Scheme, provides a way to ease this problem: programmers can write contracts for functions, then the language's contract system will check if the arguments and returned values satisfy these contracts or not in runtime. The contracts are just composed of ordinary functions in the language that return a boolean value, and the way of checking contracts is to straightforwardly interpret these function with runtime value, if it returns `true` then it satisfies the contract, otherwise the contract system will properly blame somewhere violates the contract.

Contracts provide a way to describe refined behavior of functions, and programmers can utilize contract system and write test cases to syntactically cover the program to find bugs early, but the drawback is it may involve additional performance cost in runtime because contract checking happens on each function application.

The motivation of our project is we don't want to lose the advantages of dynamic typing, but we also want to write correct programs and catch errors as early as possible, for example, before running the program. So we present a method to check contracts statically in dynamic typing languages.

Approach

Our approach is based on abstracting abstract machine[5], which is a technique that using abstract machine such as CESK machine[1] to do abstract interpretation by transforming concrete semantics to abstract semantics.
The key idea of our approach is regarding predicate (i.e. the contract) as abstract value. A predicate is an approximation of a value, it describes the bound of the value. We build an non-deterministic abstract interpreter that do computations on these abstract values, the term *non-deterministic* means it will explore every possible behaviors of program's runtime. This approach based on abstract interpretation is sound, which means if the input of abstract interpretation is a sound approximation of concrete value, it's guaranteed that the result of abstract interpretation is still a sound approximation of returned concrete value. So that we could leverage abstract interpretation to check the behaviors of a function that depicted by contracts.
Section 2 describes the basic setting of the language, introduces the concrete semantics and abstract semantics for A-Normal Form lambda calculus on CESK machine. Section 3 shows the syntax of contracts in our language, how to computes abstract values over abstract values, and how to verify contracts with runtime information obtained from the abstract interpreter.

At last, I would like to explain the reason why we change the project name. At the very beginning, we propose our project as *Extending Scheme with Liquid Type*, where Liquid Type is the abbreviation of *Logically Qualified Data Types*. Liquid Type is a type system based on Hindley-Milner type with predicate abstraction and could infer the refinement part of a type[3]. It's a powerful type system that can help programmers to write pre-condition and post-condition of a function, and catch errors before running the program. But later, as we go further in this project, we realized that in such context of dynamic typing functional programming languages, *contract* is a more precise and conventional term to describe the pre-conditions and post-conditions. And what we are doing is actually verifying these contracts statically, i.e. without really running the program. Yet, the techniques behind the title does not change.

2 Abstracting Abstract Machine

In this section, we discuss the syntax and semantics of our core language. To formally describe the concrete and abstract semantics, we use CESK machine as execution model, and present the concrete CESK machine and abstract CESK machine, respectively. Finally, we show a special continuation of abstract CESK machine to trace the argument and returned value of function application, this information will be used in contract checking.

2.1 The Language

Our small-step abstract interpreter use a small language which is a variant of call-by-value *A-Normal Form* lambda calculus[2]. Traditionally A-Normal Form used is an intermediate representation of functional program as an alternative choice of Continuation-Passing Style (CPS) in compiler, it has several settings to simplify compiler as well as interpreter for our project, and more importantly, it's much more human-readable compared with CPS form. Programs in direct style can be easily transformed into A-Normal Form[2].
The Figure 2.1 shows the syntax of our tiny language.
Not surprisingly, we use `lambda` to form a function, and <*label*> is optional and mainly used for debugging. And the language has two primitive data types: integer and boolean, as well as primitive operations on these data type are provided. ANF has three kind of expressions:

- Atomic expressions (<*aexp*>) include primitive values, functions, and operations on primitive values. Evaluating these expressions are all trivial, that is immediately and always terminate, and no side effect.

⟨*var*⟩ ::= ⟨*symbol*⟩

⟨*label*⟩ ::= ⟨*symbol*⟩

⟨*integer*⟩ ::= ··· | `-1` | `0` | `1` | ···

⟨*boolean*⟩ ::= `true` | `false`

⟨*lambda*⟩ ::= (`lambda` [⟨*label*⟩] (⟨*var*⟩) ⟨*exp*⟩)

⟨*aexp*⟩ ::= ⟨*lambda*⟩
| ⟨*var*⟩
| ⟨*integer*⟩
| ⟨*boolean*⟩
| (⟨*prim*⟩ ⟨*aexp*⟩*)

⟨*cexp*⟩ ::= (⟨*aexp*⟩ ⟨*aexp*⟩)
| (`if` ⟨*aexp*⟩ ⟨*exp*⟩ ⟨*exp*⟩)
| (`letrec` ((⟨*var*⟩ ⟨*aexp*⟩)) ⟨*exp*⟩)

⟨*exp*⟩ ::= ⟨*aexp*⟩
| ⟨*cexp*⟩
| (`let` ((⟨*var*⟩ ⟨*exp*⟩)) ⟨*exp*⟩)

⟨*prim*⟩ ::= `+` | `-` | `*` | `=` | `and` | `or` | `not`

Figure 2.1: Syntax of ANF

- Complex expressions (<*cexp*>) are function applications, `if` expression and `letrec` expression. A notable thing is the operand and operator of an application must be an atomic expression, and as well as the condition of `if` expression and the right hand side of `lecrec` binding. Complex expression may not terminate, for example, an infinite recursive application.
- Expressions (<*exp*>) are either atomic expression, complex expression or a `let` expression. Unlike `letrec`, the right hand side of `let` binding can be a <exp>, so all non-trivial expressions must be bound by a `let` expression.

2.2 CESK Machine

This section we will introduce the Control, Environment, Store and Continuation (CESK) Machine[1] and its concrete semantics for our core language.
CESK machine as an abstract machine, is a formally defined tool for describing operational semantics of programming language. As we can see from its name, CESK machine has four components that compose a *state* of its execution: **Control** is the current program expression, **Environment** is a map associates variable name with address, **Store** is a simulation of memory that maps from address to value, **Continuation** represents the program stack. Figure 2.2 shows the concrete CESK state space.

The CESK machine is a state transition based machine. The execution of program is a procedure that explores the state graph, if all the state we have already seen, then the execution terminates. Note: since for now it is a concrete machine, so the state space is infinite.
Firstly we define an inject function to form the initial state by inserting the initial expression into a state with an empty environment, an empty store and a `Halt` continuation.

$$
\begin{aligned}
\varsigma \in \Sigma &= Exp \times Env \times Store \times Cont \\
\rho \in Env &= Var \to Addr \\
\sigma \in Store &= Addr \to D \\
\kappa \in Cont &= \texttt{Halt} \mid \texttt{LetK}(v, e, \rho, \kappa) \\
d \in D &= Clo + Integer + Boolean \\
clo \in Clo &= Lam \times Env \\
a \in Addr &\text{ is an infinite set of addresses}
\end{aligned}
$$

Figure 2.2: Concrete CESK state space

$$
\begin{aligned}
\mathcal{I} &: Exp \to \Sigma \\
\mathcal{I}(exp) &= \langle exp, [], [], \texttt{Halt} \rangle
\end{aligned}
$$

Next, we define the state transition relations $(\longmapsto_{CESK}) \subseteq \Sigma \times \Sigma$ that maps a state to another state.

Atomic Expression
If the current expression is an atomic expression, which means the expression is either primitive values, functions, or operation on primitive values, and we can evaluate it directly.

$$
\langle aexp, \rho, \sigma, \kappa \rangle \longmapsto_{CESK} \langle \mathcal{A}(aexp, \rho, \sigma), \rho, \sigma, \kappa \rangle
$$

where $\mathcal{A}$ is an auxiliary function to evaluate atomic expression and defined as follow:

$$
\begin{aligned}
\mathcal{A} : AExp \times Env \times Store &\to D \\
\mathcal{A}(v, \rho, \sigma) &= \sigma(\rho(v)) \\
\mathcal{A}(z, \rho, \sigma) &= z \\
\mathcal{A}(\texttt{true}, \rho, \sigma) &= \texttt{true} \\
\mathcal{A}(\texttt{false}, \rho, \sigma) &= \texttt{false} \\
\mathcal{A}(lam, \rho, \sigma) &= clo(lam, \rho) \\
\mathcal{A}((prim, aexp_0, \cdots aexp_i), \rho, \sigma) &= \mathcal{O}(prim)(\mathcal{A}(aexp_0, \rho, \sigma), \cdots \mathcal{A}(aexp_i, \rho, \sigma))
\end{aligned}
$$

where the $\mathcal{O} : Prim \to (Value^* \to Value)$ operator is a curry function that maps a name of primitive operation to its actual operation.

Function Application
For function application, since the A-Normal Form already required the operator and operand must be atomic expression, so that we can reuse the $\mathcal{A}$ to evaluate them, update the environment and store and then go into the function body.

$$
\langle (aexp_0\ aexp_1), \rho, \sigma, \kappa \rangle \longmapsto_{CESK} \langle body, \rho'', \sigma', \kappa \rangle
$$

where

$$
\begin{aligned}
clo((\lambda\ (v)\ body), \rho') &= \mathcal{A}(aexp_0, \rho, \sigma) \\
\rho'' &= \rho'[v \to a] \\
\sigma' &= \sigma[a \to \mathcal{A}(aexp_1, \rho, \sigma)]
\end{aligned}
$$

`let` Expression
`let` expression creates a binding in the body, and the right hand side and body are both expression. So we evaluate the right hand side firstly, and in the meantime create a new `LetK` continuation that save the current continuation for resuming the program in future.

$$\langle(\texttt{let}\ (lhs\ rhs)\ body), \rho, \sigma, \kappa\rangle \longmapsto_{CESK} \langle rhs, \rho, \sigma, \kappa'\rangle$$

where

$$\kappa' = \texttt{LetK}(lhs, body, \rho, \kappa)$$

If the current expression in the state is already a value, and the continuation is `LetK`, which means the expression can not be reduced further and we need to resume the program from continuation.

$$\langle v, \rho, \sigma, \texttt{LetK}(var, body, \rho', \kappa)\rangle \longmapsto_{CESK} \langle body, \rho'', \sigma', \kappa\rangle$$

where

$$\begin{aligned}
\rho'' &= \rho'[var \to a] \\
\sigma' &= \sigma[a \to v] \\
&a \text{ is a fresh address}
\end{aligned}$$

`letrec` Expression
`letrec` expression can be used for creating recursive binding, the right hand side of the binding can use name of itself. We can achieve this by firstly update the environment and then evaluate the right hand side of binding by $\mathcal{A}$, then step into the body of expression with new environment.

$$\langle(\texttt{letrec}\ (lhs\ rhs)\ body), \rho, \sigma, \kappa\rangle \longmapsto_{CESK} \langle body, \rho', \sigma', \kappa\rangle$$

where

$$\begin{aligned}
\rho' &= \rho[lhs \to a] \\
v &= \mathcal{A}(rhs, \rho', \sigma) \\
\sigma' &= \sigma[a \to v] \\
&a \text{ is a fresh address}
\end{aligned}$$

`if` Expression
The evaluation of `if` expression is straightforward: the condition of expression is an atomic expression, so we can directly use $\mathcal{A}$ to evaluate it, and then step into different branch according to its value.

$$\langle(\texttt{if}\ \text{cond thn els}), \rho, \sigma, \kappa\rangle \longmapsto_{CESK} \begin{cases} \langle thn, \rho, \sigma, \kappa\rangle & \mathcal{A}(cond, \rho, \sigma) \neq \texttt{false} \\ \langle els, \rho, \sigma, \kappa\rangle & \text{otherwise} \end{cases}$$

2.3 Abstracting CESK Machine

In this section, we will derive a non-deterministic abstracting CESK machine for performing k-CFA-like analysis[5]. There are several changes compared with concrete CESK machine:

- the continuation is store-allocated for eliminating recursive sturcture in state space
- the addresses are abstract, and it is an finite set; we will have two kinds of address `BindAddr` and `ContAddr` representing binding address and continuation address respectively
- all values are abstract value, namely a type tag with a set of predicates, which will be discussed in Section 3

- store is a mapping from address to a set of abstract values $\mathcal{P}$, the set represents all possible values for this address
- the state includes a *time* component, which encodes the execution contexts, practically it is a list of call string history.

Figure 2.3 shows the state space of abstracting CESK machine. Notation: we use a hat on term to denote it's an abstract component, for example, $\widehat{Env}$ is abstract environment.

$$
\begin{aligned}
\varsigma \in \widehat{\Sigma} &= Exp \times \widehat{Env} \times \widehat{Store} \times \widehat{Cont} \times Time \\
\rho \in \widehat{Env} &= Var \to \widehat{Addr} \\
\sigma \in \widehat{Store} &= \widehat{Addr} \to \mathcal{P}(\hat{D}) \\
\kappa \in Cont &= \texttt{Halt} \mid \texttt{LetK}(v, e, \rho, \kappa) \\
a \in \widehat{Addr} &= BindAddr + ContAddr \\
BindAddr &= Var \times Time \\
ContAddr &= Exp \times Time \\
d \in \hat{D} &= \widehat{Clo} + \widehat{Integer} + \widehat{Boolean} \\
clo \in \widehat{Clo} &= Lam \times \widehat{Env}
\end{aligned}
$$

Figure 2.3: Abstract CESK state space

Since in abstract CESK machine the store maps an address to a set of abstract value, so the $\mathcal{A}$ will also returns a set of abstract value $\mathcal{P}(\hat{D})$.

$$\mathcal{A} : AExp \times Env \times Store \to \mathcal{P}(\hat{D})$$

The detail of computation on abstract values will be discussed in Section 3.
We also use an injection function to construct initial abstract state:

$$\mathcal{I} : Exp \to \widehat{\Sigma}$$
$$\mathcal{I}(exp) = \langle exp, [], [], \texttt{Halt}, [] \rangle$$

Unlike the concrete CESK machine, the transition function in abstract CESK machine is from a state to a set of states(non-deterministic):

$$(\longmapsto_{\widehat{CESK}}) \subseteq \widehat{\Sigma} \times \widehat{\Sigma}^*$$

Like the concrete CESK machine, we have 6 transition rules, and figure 2.4 shows them, where $t' = tick(\sigma)$. *tick* is a function we used for performing k-CFA-like analysis and it records the last k call site: $tick\langle e, \_, \_, \_, t\rangle = \lfloor e : t \rfloor_k$

2.4 Tracing Types of Application

Now we have an abstract machine that could run the program abstractedly, next we need to trace each function call and record the argument and returned value which will be used when checking contract. Here we use a way as shown in Figure 2.4 that insert a new continuation $\texttt{AppK}(label, v, \kappa)$ when applying a function. `AppK` has three components: *label* is the function's name being called, v is the argument value, and κ is the current continuation. When we encounter a state which expression is a value, and the continuation is `AppK`, it means that expression is returned value of this function call, we should store the argument and returned value into an external hash table and resume program to κ.

$$
\begin{aligned}
\langle aexp, \rho, \sigma, \kappa, t\rangle &\longmapsto_{\widehat{CESK}} \langle v, \rho, \sigma, \kappa, t'\rangle \text{ where } v \in \mathcal{A}(aexp, \rho, \sigma) \\
\langle (aexp_0\ aexp_1), \rho, \sigma, \kappa, t\rangle &\longmapsto_{\widehat{CESK}} \langle body, \rho'', \sigma', \kappa', t'\rangle \\
\text{where } clo((\lambda\ label\ (var)\ body), \rho') &\in \mathcal{A}(aexp_0, \rho, \sigma) \\
\rho'' &= \rho'[var \to a] \\
\sigma' &= \sigma[a \to argv] \\
argv &= \mathcal{A}(aexp_1, \rho, \sigma) \\
\kappa' &= \texttt{AppK}(label, argv, \kappa) \\
\langle v, \rho, \sigma, \texttt{AppK}(label, argv, \kappa), t\rangle &\longmapsto_{\widehat{CESK}} \langle v, \rho, \sigma, \kappa, t'\rangle \\
\langle (\texttt{let}\ (lhs\ rhs)\ body), \rho, \sigma, \kappa, t\rangle &\longmapsto_{\widehat{CESK}} \langle rhs, \rho, \sigma', \kappa', t'\rangle \\
\text{where } \kappa' &= \texttt{LetK}(lhs, body, \rho, a) \\
a &= ContAddr((\texttt{let}\ (lhs\ rhs)\ body), t) \\
\sigma' &= \sigma[a \to \kappa] \\
\langle v, \rho, \sigma, \texttt{LetK}(var, body, \rho', a), t\rangle &\longmapsto_{\widehat{CESK}} \langle body, \rho'', \sigma', \kappa, t'\rangle \\
\text{where } \rho'' &= \rho'[var \to a'] \\
\sigma' &= \sigma[a' \to v] \\
a' &= BindAddr(v, t) \\
k &\in \sigma(a) \\
\langle (\texttt{letrec}\ (lhs\ rhs)\ body), \rho, \sigma, \kappa, t\rangle &\longmapsto_{\widehat{CESK}} \langle body, \rho', \sigma', \kappa, t'\rangle \\
\text{where } \rho' &= \rho[v \to a] \\
\sigma' &= \sigma[a \to \mathcal{A}(rhs, \rho', \sigma) \\
a &= BindAddr((\texttt{letrec}\ (lhs\ rhs)\ body), t) \\
\langle (\texttt{if}\ \text{cond thn els}), \rho, \sigma, \kappa, t\rangle &\longmapsto_{\widehat{CESK}} \begin{cases} \langle thn, \rho, \sigma, \kappa, t'\rangle \text{if } \texttt{true} \in v \\ + \\ \langle els, \rho, \sigma, \kappa, t'\rangle \text{if } \texttt{false} \in v \end{cases} \\
\text{where } v &= \mathcal{A}(cond, \rho, \sigma)
\end{aligned}
$$

Figure 2.4: Transition rules of abstract CESK machine

3 Verifying Contracts

As we mentioned before, the key idea is to take contracts as abstract values. In this part, we will describe the calculation on these contracts and how to check these contracts. Contracts depict the boundaries of a value, and in our project, we define contract as *a type tag with a set of predicates on value*.

3.1 The Syntax of Contract

The Figure 3.1 shows the syntax of our tiny sub-language of contract in BNF. Though we have this formal syntax definition of contract, we still would like to claify some points:

- `(`→ <contract> <contract> `)` is a contract for function. The first contract is pre-condition, and the second one is post-condition for function.
- `(Int` <predicate> `)` and `(Bool` <predicate> `)` are contracts on primitive data type.

⟨*define-function-contract*⟩ ::= `(:` ⟨*name*⟩ ⟨*contract*⟩`)`

⟨*contract*⟩ ::= `(`→ ⟨*contract*⟩ ⟨*contract*⟩`)`
| `(Int` ⟨*predicate*⟩* `)` | `(Bool` ⟨*predicate*⟩* `)`

⟨*predicate*⟩ ::= `#t` | `(`⟨*pred-op*⟩ ⟨*operand*⟩ ⟨*operand*⟩ `)`

⟨*pred-op*⟩ ::= `<` | `>` | ≤ | ≥ | `=` | `!=` | `and` | `or` | `not`

⟨*operand*⟩ ::= ⟨*predicate*⟩ | ⟨*literal*⟩ | ⟨*self*⟩ | ⟨*var*⟩

⟨*literal*⟩ ::= ⟨*integer*⟩ | ⟨*boolean*⟩

⟨*integer*⟩ ::= ⋯ | `-1` | `0` | `1` | ⋯

⟨*boolean*⟩ ::= `true` | `false`

⟨*self*⟩ ::= `_`

⟨*var*⟩ ::= ⟨*symbol*⟩

⟨*name*⟩ ::= ⟨*symbol*⟩

Figure 3.1: Syntax of Contract

- The wildcard `_` represents the identifier itself. Sometimes, we don't care the variable name, for instance, contract `(Int (< _ 3))` describes a variable which is an integer and less than 3.
- If the predicate part of a contract is `#t`, it means this value can be any concrete value of this given type.
- Programmers can provide multiple contracts to describe intersection type for one function.

3.2 Contract Calculation

In this part, we will introduce how to do computation on abstract values. Abstract value and contract have the same structure, they both have a type tag and a set of predicates. But for clarity, we use term *abstract value* to denote program's runtime information, which obtained from our abstract interpretation; and use term *contract* to represent user-defined contraints on program.
The core idea of doing calculation over contracts is to find a sound approximation that contains all possible result that comes from concrete calculation. We notice that the essence of a predicate is a union of sets that descirbe the domain of value, so the computation is actually find the upper bound and lower bound of result according to the operation.
For example, assume there are two contracts `(Int (or (and (> _ 3) (< _ 5)) (≥ _ 10)))` and `(Int (>_4))`, and we want to an addition on these contract, the result is $(7, +inf)$.
In our current implementation, we implements `+`, `-`, `*` operators over abstract integers, and `and` and `or` operators over abstract booleans.

3.3 Checking Contracts

Since a contract is type tag with a set of predicates, and we can obtain program's runtime behavior (i.e. the abstract value) from the abstract interpreter, so the contract checking are actually check is the runtime behavior a subset of user-defined contract. We say a concrete value satisfies a contract iff the type of them are same and all predicate on that value is true; and an abstract value satisfies

a contract iff type of them are are same and the predicates of abstract value is a subset of contract's predicates.

To checking the program's behaviors satisfies contract or not, we have two parts as following.

- Checking whether the user provided contracts and the actual behaviors of corresponding function are compatible, if not, the function should be rejected. E.g.: assume a given contract is `(: id (→ Int Bool))` and a given function is `(lambda id (x) x)`, this function will be rejected.
- Checking whether every argument and returned value of function call satisfy its corresponding contracts. For an example as follows, an identity function `id`, from this definition, we can say that `id` is polymorphism with input and output have the same type (and value).

```
(let ((id (lambda id (x) x)))
  (let ((one (id 1)))
    (let ((fls (id false)))
      one)))
```

There are two function applications: the first one is `(id 1)`, and the second one is `(id false)`. Now consider two cases with different contracts.

1) The given contract is `(: id (→ Int Int))`. Since the above code has an application that applies `false` on function `id`, while the contract only allows argument with type `Int`, the program will be rejected.

2) The given contracts are `(: id (→ Int Int))` and `(: id (→ Bool Bool))`. The program will be accepted since no function call violate any contracts, in other word, the contracts provides a sound description that contains all possible runtime behaviors.

4 Implementation

To verify our approach, we implemented a prototype verifier in Racket, which can be checked out from `https://github.com/Kraks/Verify-Contracts-for-Dynamic-Lang`

5 Related Work

Tobin-Hochstadt and Van Horn propose an approach to reasoning higher-order program by extending symbolic execution to use behavioral contracts as symbolic values[4], and then could verify contracts statically and properly blame value which violate the contract. Their work is very similar with our method, but since in the early of project we are focusing on refinement until we realized our work is actually verifying contracts, so our work in the project are mostly independent.

6 Conclusion and Future Work

Our project utilize abstract interpretation and build an analyzer to analyze the runtime function call behaviors of dynamic typed programs, then could verify the program with user-defined contracts by checking the runtime abstract value and contract is compatible or not. As a tool it cloud help programmer catch bugs before running the program when using dynamic typing programming language.

The prototype verifier can correctly checking contract with primitive data type such as `Int` or `Bool`, but still we may improve the it in several aspects:

- Supporting verification for function type. To achieve this, we can check is a function type a subtype of another function type by contravariance.
- Supporting dependent contract which means the post-condition can be dependent on some value from pre-condition. In order to do that, we cloud use a `lambda` which takes pre-condition as argument to wrap post-condition, and when checking post-condition, we apply the pre-condition into it.
- Extending the language and its contract system to support more data types such as `list`, and support more primitive function such as `call/cc`.

Reference

[1] Matthias Felleisen. *Control Operators, the SECD-machine, and the A-Calculus*. 1987.

[2] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[3] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM.

[4] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 537–554, New York, NY, USA, 2012. ACM.

[5] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 51–62, New York, NY, USA, 2010. ACM.