From Definitional Interpreter to Symbolic Executor

Adrian D. Mensing
Delft University of Technology
Netherlands
a.d.mensing-1@student.tudelft.nl

Hendrik van Antwerpen
Delft University of Technology
Netherlands
h.vanantwerpen@tudelft.nl

Casper Bach Poulsen
Delft University of Technology
Netherlands
c.b.poulsen@tudelft.nl

Eelco Visser
Delft University of Technology
Netherlands
e.visser@tudelft.nl

Abstract

Symbolic execution is a technique for automatic software validation and verification. New symbolic executors regularly appear for both existing and new languages and such symbolic executors are generally manually (re)implemented each time we want to support a new language. We propose to automatically generate symbolic executors from language definitions, and present a technique for mechanically (but as yet, manually) deriving a symbolic executor from a definitional interpreter. The idea is that language designers define their language as a monadic definitional interpreter, where the monad of the interpreter defines the meaning of branch points. Developing a symbolic executor for a language is a matter of changing the monadic interpretation of branch points. In this paper, we illustrate the technique on a language with recursive functions and pattern matching, and use the derived symbolic executor to automatically generate test cases for definitional interpreters implemented in our defined language.

CCS Concepts • **Theory of computation** → **Program schemes**; • **Software and its engineering** → *Formal methods*; Automatic programming.

Keywords Symbolic Execution, Monads, Haskell, Definitional Interpreter

ACM Reference Format:
Adrian D. Mensing, Hendrik van Antwerpen, Casper Bach Poulsen, and Eelco Visser. 2019. From Definitional Interpreter to Symbolic Executor. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection (META '19), October 20, 2019, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3358502.3361269

1 Introduction

Symbolic execution [27] is a meta-programming technique that is at the core of techniques for boosting developer productivity, such as *automated testing* [3, 9, 17, 19, 38] and *program synthesis* [14, 20, 35]. A symbolic executor allows exploration of possible execution paths by running a program with symbolic variables in place of concrete values. By strategically instantiating symbolic variables, a symbolic executor can be used to systematically analyze which parts of a program are reachable, with which inputs.

Constructing symbolic executors is non-trivial, and enabling support for symbolic execution for general-purpose languages, such as C [4, 19, 38], C++ [31], Java [1, 37], PHP [2], or Rust [33], is the topic of entire publications at major software engineering conferences. We propose that techniques for symbolic execution are reusable between languages, and investigate the foundations of how to define and implement symbolic executors, by deriving them from *definitional interpreters*. Our long-term goal is to integrate these techniques into language workbenches, such as Spoofax [25], Rascal [29], or Racket [15], to enable the automatic generation of programmer productivity boosting tools, such as automated testing frameworks and program synthesizers.

In this paper we explore how to mechanically derive symbolic executors that explore possible execution paths through programs by instantiating and specializing symbolic variables, following a breadth-first search strategy. Our exploration revolves around a dynamically-typed language with recursive functions and pattern matching. Using Haskell as our meta-language, and working with its integrated support for generic and monadic programming, we implement a definitional interpreter for this language. This definitional interpreter is parameterized with an interface which we instantiate in two different ways to obtain first a concrete interpreter, and then a symbolic executor. The "derivation" thus amounts to instantiating the interface operations in a manner that yields a symbolic executor.

The symbolic executor we derive allows us to explore the solution space for constraints such as the following constraint that a list *xs* must be a palindrome:

$$xs \equiv reverse\ xs$$

Symbolic execution explores all execution paths through the *reverse* function that satisfy the constraint, and instantiates *xs* accordingly, thereby generating palindromes. This paper is a literate Haskell file, and we invite interested readers to download the Haskell version of the paper to experiment with, and extend, the framework we present.[1]

Related Previous Lines of Work The techniques that we develop in this paper are closely related to the techniques used for *relational programming*, pioneered by Friedman and Byrd in miniKanren [5, 8, 16, 22], a language for relational programming and constraint logic programming, which has been implemented in a wide range of different languages; notably Scheme [7, 16], but also, e.g., OCaml [30]. The miniKanren language and many of its implementations have been developed and researched for more than a decade, with new developments and improvements appearing each year, such as new and better heuristics for guiding the exploration of execution paths [34]. The motivation for this paper is to bring similar benefits as found in miniKanren to programming languages at large, by automatically deriving symbolic executors from definitional interpreters.

Rosette [40, 41] is a solver-aided language that extends Racket [15] to provide framework for implementing solver-aided domain-specific languages, by means of a symbolic virtual machine and symbolic compiler. This VM brings the benefits of symbolic execution and model checking to languages implemented in Rosette via general-purpose symbolic abstractions that support sophisticated symbolic reasoning, beyond the relatively simple constraints found in (most variants of) miniKanren. A main goal of Rosette is to implement solver-aided languages, but the symbolic abstractions and techniques that Rosette implements could also be used to address the problem that is the motivation for this paper, namely the problem of automatically deriving symbolic executors from "traditional" definitional interpreters.

There has been much work on symbolic execution in the literature on software engineering; e.g., [1, 2, 4, 19, 31, 33, 37, 38]. Many of these frameworks are so-called *concolic* frameworks that work by instrumenting a concrete language runtime to track *symbolic path constraints*. After each concrete execution, these path constraints are collected and solved in order to cover a different path through the program in a subsequent run of the program. Concolic testing is typically implemented by generating test inputs randomly, rather than systematically solving path constraints. In this paper, we explore a symbolic execution strategy which interleavingly explores multiple execution paths concurrently, rather than a concolic testing approach which would require a relatively sophisticated constraint solver in order to explore execution paths in an equally systematic manner.

[1] https://github.com/MetaBorgCube/From-Definitional-Interpreter-To-Symbolic-Executor

Contributions

- Techniques (in § 3) for deriving symbolic executors from definitional interpreters, by using *free monads* to compile programs into *command trees*, and interpreting these using a small-step execution strategy.
- A symbolic executor (in § 4) for a language with algebraic datatypes that illustrates these techniques.
- A simple example application (in § 6): automated test generation for definitional interpreters.

The rest of this paper is structured as follows. In § 2 we introduce a definitional interpreter for a language with recursion and pattern matching. In § 3 we present a definitional interpretation of the effects, by means of a free monad, using a small-step semantics execution strategy. In § 4 we generalize the definitional interpretation of effects from § 3, to obtain a symbolic executor, whose correctness we discuss in § 5. Finally, in § 6 we discuss a case study application of the symbolic executor: generating tests for definitional interpreters, and § 7 concludes.

2 Definitional Interpreter for a Language With Pattern Matching

Definitional interpreters define the meaning of a (new) object language by implementing an interpreter for it in an existing, well-understood, language. We use *Haskell* to implement a definitional interpreter for a functional language with pattern matching.

2.1 Syntax

The abstract syntax of the language we consider is summarized in Fig. 1. The expression constructors for *Var*, *Lam*, and *App* are standard expressions for variables, unary functions, and function application. An expression constructor expression *Con* f $[e_1, ..., e_n]$ represents an n-ary term whose head symbol is f, and whose sub-term values are the results of evaluating each expression $e_1 \ldots e_n$. *Case* e $[(p_1, e_1), ..., (p_n, e_n)]$ is a pattern match expression which first evaluates e to a value and then attempts to match the resulting value against the patterns $p_1 \ldots p_n$, where patterns are given by the type *Patt*. *Letrec* expressions are restricted to bind value expressions, given by the type *ValExpr*.

2.2 Prelude to a Definitional Interpreter: Effects and Values

The definitional interpreter for the language we consider in this paper is given in Fig. 2. The interpreter depends on the *EffVal* type class which in turn depends on a number of type classes that constrain the polymorphic notion of effects (defined by a monad m) and values (defined by a value type *val*) of the interpreter. The *EffVal* type class is thus a polymorphic embedding [23] of a language that allows us to define a *family* of interpreters for the same language,

```
data Expr = Con String [Expr]
          | Case Expr [(Patt, Expr)]
          | Var String
          | Lam String Expr
          | App Expr Expr
          | Let [(String, Expr)] Expr
          | Letrec [(String, ValExpr)] Expr
          | EEq Expr Expr

data ValExpr = VCon String [ValExpr]
             | VLam String Expr

data Patt = PVar String
          | PCon String [Patt]
```

Figure 1. Syntax for a language with pattern matching, functions, let, and letrec

akin to the finally tagless approach of Carette et al. [10]. We summarize the type classes that *EffVal* comprises.

Effects The language that we define has two classes of effects: lexically-scoped functions and pattern matching. The following Haskell type class constrains a monad *m* to provide two operations for accessing environments (*ask*), and altering which local environment is passed down to recursive calls of the interpreter (*local*):

```
type Env val = [(String, val)]
class Monad m ⇒ MonadEnv val m where
  ask   :: m (Env val)
  local :: (Env val → Env val) → m val → m val
```

MonadEnv is a specialized version of the classical reader monad [18, 24, 32]:

```
class Monad m ⇒ ClassicalMonadReader r m where
  ask_c   :: m r
  local_c :: (r → r) → m a → m a
```

There are two reasons why we use a specialized version. The reason we specialize the type of environments, as opposed to an arbitrary type *r*, is to help Haskell's type class instance resolution engine (using GHC v8.6.4). The reason we insist that the return type is *val* for the computation that *local* takes as argument, is a desire to know that this particular computation is value-producing, for reasons we explain § 3.

The purpose of symbolic execution is to decide which inputs cause which parts of a program to execute. For this reason, we treat conditional branching as an effect. The following type class constrains a monad *m* to provide a generic operation for branching:

```
class Monad m ⇒ MonadBranch cval rval fork m where
  branch :: cval → fork m rval → m rval
```

This type class is parameterized by: (1) a value type *cval* that branch selection is conditional upon; (2) a value type *rval* for the return type of computations in branches; and (3) a *fork* type, an abstract notion of branches comprising computations described by *m* and *val*. To illustrate, consider the following instance of *MonadBranch* which represents a classical if-then-else expression:

```
newtype IfThenElse m a = ITE (m a, m a)
instance Monad m ⇒
      MonadBranch Bool rval IfThenElse m where
  branch True  (ITE (t, _)) = t
  branch False (ITE (_, f)) = f
```

For our interpreter, which branches on values and returns values of the same type, we rely on the following more restrictive version of *MonadBranch*:[2]

```
class Monad m ⇒ MonadMatch val fork m where
  match :: val → fork m val → m val
```

And our interpreter uses the following notion of *fork* over a list of pairs consisting of a pattern and a (monadic) computation where each computation has the same return type *a*:

```
newtype Cases m a = Cases [(Patt, m a)]
```

Values The following type classes define the constructors for term values con_v and function closures $clos_v$, as well as operation *app* for applying a function to an argument and operation *eq* for checking equality between two term values.

```
class TermVal val where
  con_v :: String → [val] → val
class FunVal val where
  clos_v :: String → Expr → Env val → val
class FunApp val m where
  app  :: val → val → m val
class TermEq val m where
  eq   :: val → val → m val
```

2.3 A Definitional Interpreter for a Language with Pattern Matching

The interpreter in Fig. 2 relies on the effect and value type classes summarized in the previous section. Additionally, the interpreter makes use of a few auxiliary functions whose definitions we elide: *mmap* maps a monadic function over a list; *mapSnd* maps a function over the second element of a tuple; and *resolve* resolves a name in an association list, or fails. The implementation of *Letrec* uses Haskell's support for

[2]The main motivation for using the more specific notion of *MonadMatch* here is to help Haskell's type class resolution engine (using GHC v8.6.4). Morally, *MonadBranch* should do.

```
interp :: EffVal m val ⇒ Expr → m val
interp (Con c es) = do
  vs ← mmap interp es
  return (con_v c vs)
interp (Case e bs) =
  let vbs = map (mapSnd interp) bs in do
  v ← interp e
  match v (Cases vbs)
interp (Var x) = do
  nv ← ask
  return (resolve x nv)
interp (Lam x e) = do
  nv ← ask
  return (clos_v x e nv)
interp (App e1 e2) = do
  f ← interp e1
  a ← interp e2
  app f a
```

```
interp (Let xes e) = do
  nv ← mmap interpSnd xes
  local (λnv0 → nv ++ nv0) (interp e)
  where interpSnd (x, e) = do
    v ← interp e; return (x, v)
interp (Letrec xves e) = do
  nv ← ask
  let nv_b = map (mapSnd (interpval nv_r)) xves
      nv_r = nv_b ++ nv in
    local (λ_ → nv_r) (interp e)
interp (EEq e1 e2) = do
  v1 ← interp e1
  v2 ← interp e2
  eq v1 v2
interpval :: (TermVal val, FunVal val) ⇒
           Env val → ValExpr → val
interpval nv (VLam x e) = clos_v x e nv
interpval nv (VCon x es) =
  con_v x (map (interpval nv) es)
```

Figure 2. A definitional interpreter for a language with pattern matching

(lazy) recursive definitions to define a recursive environment nv_r that *ValExpr*s are evaluated under.

To run our definitional interpreter we must provide concrete instances of the abstract type classes from § 2.2. We use the following notion of value and monad:

```
data ConcreteValue = ConV String [ConcreteValue]
                   | ClosV String Expr (Env ConcreteValue)
type ConcreteMonad =
  ReaderT (Env ConcreteValue) (Except String)
```

Here *ReaderT* is a monad transformer [32] for the classical reader monad, and *Except* is the exception monad. So *ConcreteMonad* is isomorphic to:

```
type ConcreteMonad' a =
  Env ConcreteValue → Either String a
```

The type class instances for this notion of value and monad are defined in the obvious way. *MonadMatch* attempts to pattern match a value against a list of cases by attempting each from left-to-right until a match succeeds:

```
instance MonadMatch ConcreteValue Cases
                    ConcreteMonad where
  match v (Cases ((p, m) : bs)) = case vmatch (v, p) of
    Just nv → local (λnv0 → nv ++ nv0) m
    Nothing → match v (Cases bs)
  match _ (Cases [ ]) = throwError "Match failure"
vmatch :: (ConcreteValue, Patt) →
          Maybe (Env ConcreteValue)
```

Using these type class instances, our definitional interpreter can be run as follows:

```
runSteps :: Expr → Env ConcreteValue →
            Either String ConcreteValue
runSteps e nv = runExcept (runReaderT (interp e) nv)
```

3 Towards a Symbolic Executor

The definitional interpreter presented in § 2.3 uses standard monads and monad transformers to implement the definitional interpreter given in Fig. 2. But it gives meta-programmers little control over how interpretation proceeds. Our goal is to implement a symbolic executor for running a program in a way that interleavingly explores all possible execution paths. To this end, we want a symbolic executor that can operate on a pool of concurrently running threads where each thread represents a possible path through the program. We will approach this challenge by adopting a small-step execution strategy for each thread. In this section we provide alternative type class instances that give meta-programmers more fine-grained control over how interpretation proceeds. Concretely, we adopt a small-step execution strategy for effect interpretation, by using *free monads.*

Following Kiselyov and Ishii [28] and Swierstra and Baanen [39], the following data type defines a family of free monads:

```
data Free c a = Stop a
              | ∀b.Step (c b) (b → Free c a)
```

Following Hancock and Setzer [21], we call values of this data type *command trees*: each *Step* represents an application of a command *c b*, corresponding to a monadic operation, which yields a value of type *b* when interpreted. This value is passed to the continuation (*b* ⟶ *Free c a*) of *Step*. The *Free* data type is a monad:

```
instance Monad (Free c) where
  return       = Stop
  Stop a   >>= k = k a
  Step c f >>= k = Step c (λx → f x >>= k)
```

By defining a suitable notion of command, we can define a free monad instance which satisfies the type class constraints for our definitional interpreter from Fig. 2. The following data type defines such a notion of command:

```
data Cmd val :: * → * where
  Match :: val → Cases (Free (Cmd val)) val →
           Cmd val val
  Local :: (Env val → Env val) → Free (Cmd val) val →
           Cmd val val
  Ask   :: Cmd val (Env val)
  App_c :: val → val → Cmd val val
  Eq_c  :: val → val → Cmd val val
  Fail  :: String → Cmd val a
```

By instantiating each of the type classes we obtain a *compiler* from expressions into command trees:

```
comp :: (TermVal val, FunVal val) ⇒
        Expr → Free (Cmd val) val
comp = interp
```

The command trees that *comp* yields are the sequences (or rather trees) of effectful operations that define the meaning of object language expressions. But the meaning of command trees is left open to interpretation. We define the meaning of command trees by means of a small-step transition function and a driver loop for the transition function. This small-step transition function operates on a single command tree (whose type we abbreviate $Thread_c$, since the command tree represents a thread of interpretation), and yields a single command tree as result (or raises an exception). For brevity, we show just a few cases of the *step* function:

```
type Thread_c = Free (Cmd ConcreteValue)
step :: Thread_c ConcreteValue →
        ConcreteMonad (Thread_c ConcreteValue)
step (Stop x)                       = return (Stop x)
step (Step (Match _ (Cases [ ])) _) =
  throwError "Pattern match failure"
step (Step (Match v (Cases ((p, m) : bs))) k) =
  case vmatch (v, p) of
    Just nv →
      return (Step (Local (λnv_0 → nv ++ nv_0) m) k)
    Nothing → step (Step (Match v (Cases bs)) k)
```

The driver loop for the step function is straightforwardly defined to continue interpretation until the current thread of interpretation terminates successfully (or fails):

```
drive :: Thread_c ConcreteValue →
         ConcreteMonad ConcreteValue
drive (Stop x) = return x
drive c        = do r ← step c; drive r
```

Thus an alternative definitional interpreter for the language in Fig. 2 is given by the following function:

```
runSteps :: Expr → Env ConcreteValue →
            Either String ConcreteValue
runSteps e nv = runExcept (runReaderT (drive (comp e)) nv)
```

4 From Definitional Interpreter to Symbolic Executor

In this section we derive a symbolic executor from the definitional interpreter in § 3, by: (1) generalizing the notion of value from previous sections to also incorporate symbolic variables; and (2) generalizing the semantics (monad and small-step transition function) to support instantiation of symbolic variables and fork new threads of interpretation.

Symbolic Values The updated notion of value is an extension of the notion of *ConcreteValue* data type from § 2.3 with a symbolic variable constructor, *SymV*:

```
data SymbolicValue = ConV′  String [SymbolicValue]
                   | ClosV′ String Expr (Env SymbolicValue)
                   | SymV   String
```

Monad The monad for evaluating a step of symbolic execution has an environment and may raise an exception, just like the monad in § 3 for evaluating a step of concrete execution. Additionally, the monad has a stateful *Int* field for keeping track of a fresh supply of symbolic variable names:

```
type SymbolicMonad =
  ReaderT (Env SymbolicValue)
          (StateT Int (Except String))
```

Since symbolic execution should explore all possible execution paths through a program, we generalize the small-step transition relation from § 3 by letting the transition relation take a single thread of interpretation as input, but return a *set* of possible continuation threads. Each step may result in unifying a symbolic variable in order to explore a possible execution path. Our generalized notion of monad is thus given by the following types:

```
type Unifier   = [(String, SymbolicValue)]
type Unifier_N = [(SymbolicValue, SymbolicValue)]
type SymbolicSetMonad =
  StateT (Unifier, Unifier_N) (ListT SymbolicMonad)
```

Here, *Unifier* witnesses how symbolic variables must be instantiated in order to complete a single transition step, representing a particular execution path of the program being symbolically executed. *Unifier*$_N$ represents a set of *negative unification constraints*. We motivate the use and need for these shortly. The *ListT* monad transformer generalizes the return type of a monadic computation $m\ a$ to return a list of *a*s; i.e., $m\ [a]$. Note that, although we call *ListT* a monad transformer, it is well-known that *ListT* in Haskell is not guaranteed to yield a monad that satisfies the monad laws. For the purpose of this paper, it is not essential whether the particular definition of *SymbolicSetMonad* above actually satisfies the monad laws.

Small-Step Transition Function Our symbolic executor is derived from the concrete semantics of effects in § 3 by altering how we *Match* and *Eq*$_c$ effects are interpreted. Thus all cases of the transition function *step*$_s$ (below) are identical to the small-step transition function from § 3, except for the cases for the *Match* and *Eq*$_c$. Furthermore, the definitional interpreter from Fig. 2 is unchanged. We summarize the interesting cases for the *step*$_s$ function, which takes a symbolic interpretation thread, *Thread*$_s$, as input, and returns a *set* of threads (note the use of *SymbolicSetMonad*):

```
type Thread_s = Free (Cmd SymbolicValue)
step_s :: Thread_s SymbolicValue →
      SymbolicSetMonad (Thread_s SymbolicValue)
step_s (Step (Match _ (Cases [ ])) _) = mzero
step_s (Step (Match v (Cases ((p, m) : bs))) k) = (do
    (nv, u) ← vmatch_s (v, p)
    (applySubst u (Step (Local (λnv0 → nv ++ nv0) m) k))
      `mplus` step_s (Step (Match v (Cases bs)) k))
  `catchError` (λ_ → step_s (Step (Match v (Cases bs)) k))
step_s (Step (Eq_c v1 v2) k) =
  case unify v1 v2 of
    Just [ ]  → return (k (ConV′ "true" [ ]))
    Just u    → do
      (applySubst u (k (ConV′ "true" [ ]))) `mplus`
        (constrainUnif_N u (k (ConV′ "false" [ ])))
    Nothing →
      return (k (ConV′ "false" [ ]))
```

As in § 3, there are two cases for *Match*: one for the case where we have exhausted the list of patterns to match a value against, and one for the case where there are more cases to consider. In case we have exhausted the list of patterns to match a value against, we now use *mzero* to return an empty set of result threads. Otherwise, we match a value against a pattern, using the side-effectful *vmatch*$_s$ function (elided for brevity). If the value contains symbolic variables, the *vmatch*$_s$ function computes a unifier to be be applied to the symbolic variables in order to make the pattern match succeed. The transition function returns the thread resulting from applying that unifier to the matched branch, unioned with (via the '*mplus*' operation of the *SymbolicSetMonad*) any other threads contained in branches with patterns that may succeed to match (via the recursive call to *step*$_s$ in the second *Match* case above). This way, the transition function computes the set of all possible execution paths for a given expression.

The case of the *step*$_s$ function above for expressions of the form $Eq_c\ v_1\ v_2$ checks whether v_1 and v_2 are unifiable. If they are unifiable with the empty unifier, there is only one possible execution path to consider, namely the execution path where v_1 and v_2 are equal. Otherwise, if v_1 and v_2 have a non-empty unifier, there are two possible execution paths to consider: one where v_1 and v_2 are equal, and one where they are not. The *step*$_s$ function returns the union (again, using '*mplus*') of two threads representing each of these execution paths. For safety, we register a *negative unification constraint* for the execution path that disequates v_1 and v_2, such that v_1 and v_2 cannot be unified at any point in the future during symbolic execution.

Driver Loop The driver loop for symbolic execution is generalized to operate on *sets* of possible execution paths, where each execution path is given by a configuration *Configs*$_s$:

```
type Configs_s a = (a, Env SymbolicValue, Unifier_N)
drive_s :: [Configs_s (Thread_s SymbolicValue)] →
      SymbolicMonad (Configs_s SymbolicValue,
                     [Configs_s (Thread_s SymbolicValue)])
drive_s [ ] = throwError "No solution found"
drive_s ts =
  case isDone ts of
    (Just c, ts′) → return (c, ts′)
    _             → do
      ts′ ← iterate ts
      drive_s ts′
```

A configuration comprises a value, an environment which may contain terms with symbolic variables, and a list of negative unification constraints (*Unifier*$_N$). The *drive*$_s$ function takes a list of configurations as input, uses *isDone* to check if one of the input configurations is a value, and returns a pair of that configuration and the remaining configurations. If none of the input configurations are values, each input configuration is *iterate*d by a single transition step, and *drive*$_s$ is called recursively on the resulting list of configurations.

A Constraint Language for Symbolic Execution We have shown how to alter the interpretation of the effects in the definitional interpreter presented in Fig. 2, to derive a symbolic executor from the concrete definitional interpreter from § 3. Invoking this symbolic executor with input programs

```
data Constraint   = CTake Int ExConstraint
data ExConstraint = CEx String ExConstraint
                  | CEq Expr Expr
                  | CNEq Expr Expr
```

Figure 3. Syntax for a tiny constraint language

that contain symbolic variables gives rise to a breadth-first search over possible instantiations of symbolic variables, to synthesize concrete terms. We provide programmers with control over which parts of a program (s)he wishes to synthesize by defining a small constraint language on top of the definitional interpreter from Fig. 2.

The syntax for this constraint language is summarized in Fig. 3. *CTake n* c_x is a "top-level" constraint for picking *n* solutions to a constraint c_x that contains existentially quantified symbolic variables. *CEx x* c_x introduces an existentially quantified symbolic variable, by populating the environment of a symbolic interpreter with a symbolic variable value binding *SymV* x_f for *x*, where x_f is a fresh symbolic variable name. *CEq* e_1 e_2 is a constraint that e_1 and e_2 evaluate to the same value, and *CNEq* e_1 e_2 is a constraint that e_1 and e_2 evaluate to different values.

Our approach to constraint solving is given by the *solve* function in Fig. 4 which, in turn, calls the $search_s$ function whose type signature is shown in the figure, but whose implementation we omit for brevity. $search_s$ *e ts ceq n* implements a naive constraint solving strategy which uses a symbolic executor to search for *n* different instantiations of symbolic variables that make the result of symbolic execution of the input expression *e* equal to the result of symbolic execution of a configuration in *ts*, modulo a custom notion of *SymbolicEquality*.

Example: Synthesizing Append Expressions To illustrate what we can do with our derived symbolic executor and small constraint language, let us consider list concatenation as an example, inspired by the relational programming techniques and examples given by Byrd et al. [6]. The *append0* program below grabs a single solution to the constraint which equates "q" and the result of concatenating (*append*) a list consisting of three atoms (*a*, *b*, *c*) with a list of two atoms (*d*, *e*):

```
append0 :: Constraint
append0 =
  grab 1 (exists "q"
    ((append @@ (atom "a" `cons` (atom "b"
                    `cons` (atom "c" `cons` nil)))
             @@ (atom "d" `cons` (atom "e" `cons` nil)))
      `CEq` (var "q")))
```

Here, *append* is a recursive function defined in the language we are symbolically executing (Fig. 1), and @@ is syntactic sugar for '*App*'. Solving the *append0* constraint yields the instantiation of *q* to the list contaning all input atoms in sequence.

```
solve :: Constraint → SymbolicMonad [Env SymbolicValue]
solve (CTake n c_x) = solve_x c_x n
solve_x ::  ExConstraint → Int →
            SymbolicMonad [Env SymbolicValue]
solve_x (CEx x c_x) n = do
  n_x ← fresh′
  Reader.local (λnv → (x, SymV n_x) : nv) (solve_x c_x n)
solve_x (CEq e_1 e_2) n = do
  nv ← ask
  search_s e_1 [(interp e_2, nv, [ ])] unify n
solve_x (CNEq e_1 e_2) n = do
  nv ← ask
  search_s e_1 [(interp e_2, nv, [ ])]
        (λv_1 v_2 → case unify v_1 v_2 of
                      Just _   → Nothing
                      Nothing → Just [ ])
        n
type SymbolicEq =
  SymbolicValue → SymbolicValue → Maybe Unifier
search_s :: Expr →
        [Configs_s (Thread_s SymbolicValue)] →
        SymbolicEq →
        Int →
        SymbolicMonad [Env SymbolicValue]
```

Figure 4. A constraint solver for symbolic execution constraints

We can also use symbolic execution to synthesize inputs to functions:

```
append01 :: Constraint
append01 =
  grab 1 (exists "q"
    ((append @@ (var "q")
             @@ (atom "d" `cons` (atom "e" `cons` nil)))
      `CEq` (atom "a" `cons` (atom "b" `cons` (atom "c"
        `cons` (atom "d" `cons` (atom "e" `cons` nil)))))))
```

Solving the *append01* constraint yields the instantiation of *q* to the list containing the atoms *a*, *b*, *c*.

We can even use symbolic execution to synthesize multiple inputs:

```
append02 :: Constraint
append02 =
  grab 6 (exists "x" (exists "y"
    ((append @@ (var "x") @@ (var "y"))
```

$$`CEq` \, (atom \, \texttt{"a"} \, `cons` \, (atom \, \texttt{"b"} \, `cons` \, (atom \, \texttt{"c"} \, `cons` \, (atom \, \texttt{"d"} \, `cons` \, (atom \, \texttt{"e"} \, `cons` \, nil)))))))$$

Solving the *append02* constraint yields the 6 different possible instantiations of x and y that satisfy the constraint.

5 Correctness

We have shown how to derive a symbolic executor from a concrete semantics. The derivation was driven by an intuitive understanding of what needs to happen in a symbolic executor (instantiating and refining symbolic variables, forking new threads of interpretation) in order to ensure that the symbolic executor explores *all possible execution paths*, but *only* possible execution paths (i.e., no execution paths that do not correspond to an actual execution path). In this section we conjecture a correctness proposition for our symbolic evaluator, and discuss directions for making this correctness proposition more formal.

Let $runSteps_s$ be a function that uses the $drive_s$ function to drive an expression to a final value and pool of alternative execution paths that may yet yield a final result:

$$runSteps_s :: Expr \to Env \; SymbolicValue \to Either \; String \; (SymbolicValue, [Config_s \; (Thread_s \; SymbolicValue)])$$

We conjecture that, for any pair of concrete environment nv and symbolic environment nv_s that are equal up-to-unification:

1. Any concrete execution path, given by calling *runSteps* from § 3 under nv with any $e{::}Expr$ either yields a value that is equal up-to-unification to the *SymbolicValue* that $runSteps_s$ returns; or yields a value that one of the configurations in $runSteps_s$ will eventually yield, if we were to iterate that configuration.
2. Any symbolic execution path, given by calling $runSteps_s$ under nv_s with any $e :: Expr$ yields a symbolic value and set of configurations that exhaustively describe any concrete execution path resulting from evaluating e under any nv' that is equal up-to-unification to nv_s.

We believe that *abstract interpretation* [13] is a suitable framework for formalizing the correspondence between concrete and symbolic execution.[3] The methodology due to Keidel et al. [26] for defining static analyzers with compositional soundness proofs is attractive to consider for this purpose. But it is an open question how the small-step interpretation strategy based on free monads that we adopted in § 3 and § 4 to realize our symbolic executor fits into the framework and methodology of Keidel et al. [26]. In very recent work, Rozplokhas et al. [36] provide a certified definition of miniKanren. In future work, we will investigate how to port their verification technique to the development in this paper.

[3]Indeed, it seems Cousot [12] has considered how to formalize symbolic execution within the framework of abstract interpretation. This formalization is only available in French [11].

6 Case Study: Automatic Test Generation for Definitional Interpreters

In order to test the symbolic executor we have developed, we defined various interpreters for the simply-typed lambda calculus, and attempt to synthesize program terms that yield different results for correct and wrong interpreters. Specifically, we have implemented a canonical, environment-based interpreter, and variations on this interpreter with scoping mistakes. Symbolic execution is able to automatically synthesize test programs that will detect these mistakes, by looking for programs whose results differ between the correct interpreter and the wrongly-scoped interpreter. For brevity, we omit discussion of these test cases. The Haskell version of this paper contains the test cases that we invite interested readers to consult. Using GHCi (v8.6.4), symbolic execution takes <1s to synthesize each test program.

Byrd et al. [6] also compare interpreters with lexical and dynamic scope in their functional pearl on using miniKanren to solve programming problems. Their implementation is engineered to use miniKanren's relational programming constructs to allow them to yield example terms more efficiently than naively written interpreters. Our case study does not come near the efficiency of the interpreters with lexical and dynamic scope of Byrd et al. [6], which synthesize 100 example programs in <2s. But we did not attempt to optimize the interpreter implementations either, neither at the meta-language nor the object-language level, to make it easier for the symbolic execution strategy to find solutions.

7 Conclusion

In this paper we studied how to derive a symbolic executor from concrete definitional interpreters, and presented techniques for structuring definitional interpreters to ease this derivation: free monads for compiling a definitional interpreter into a command tree with a small-step execution strategy, suitable for forking threads of interpretation and doing breadth-first search over how to instantiate symbolic variables in ways that correspond to execution paths through a program, subject to constraints. We introduced a small constraint language on top of our symbolic executor, and used this language to derive test cases for definitional interpreters for the simply-typed lambda calculus.

In future work, we intend to explore how to make the derivation techniques presented in this paper formally correct, how to automate them, and how to make them efficiently executable, akin to, e.g., miniKanren [5, 16].

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions for improvements and future research directions, and Sven Keidel for his timely and helpful comments.

References

[1] Saswat Anand, Corina S. Pasareanu, and Willem Visser. 2007. JPF-SE: A Symbolic Execution Extension to Java PathFinder. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings (Lecture Notes in Computer Science)*, Orna Grumberg and Michael Huth (Eds.), Vol. 4424. Springer, 134–138. https://doi.org/10.1007/978-3-540-71209-1_12

[2] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit M. Paradkar, and Michael D. Ernst. 2010. Finding Bugs in Web Applications Using Dynamic Test Generation and Explicit-State Model Checking. *IEEE Trans. Software Eng.* 36, 4 (2010), 474–494. https://doi.org/10.1109/TSE.2010.31

[3] Stefan Bucur, Johannes Kinder, and George Candea. 2014. Prototyping symbolic execution engines for interpreted languages. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 239–254. https://doi.org/10.1145/2541940.2541977

[4] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. IEEE Computer Society, 443–446. https://doi.org/10.1109/ASE.2008.69

[5] William E. Byrd. 2010. *Relational Programming in miniKanren: Techniques, Applications, and Implementations*. Ph.D. Dissertation. Indiana University.

[6] William E. Byrd, Michael Ballantyne, Gregory Rosenblatt, and Matthew Might. 2017. A unified approach to solving seven programming problems (functional pearl). *PACMPL* 1, ICFP (2017), 8:1–8:26. https://doi.org/10.1145/3110252

[7] William E. Byrd and Daniel P. Friedman. 2006. From Variadic Functions to Variadic Relations: A miniKanren Perspective. In *Proceedings of the 2006 Scheme and Functional Programming Workshop*. http://scheme2006.cs.uchicago.edu/12-byrd.pdf

[8] William E. Byrd, Eric Holk, and Daniel P. Friedman. 2012. miniKanren, live and untagged: quine generation via relational interpreters (programming pearl). In *Proceedings of the 2012 Annual Workshop on Scheme and Functional Programming, Scheme 2012, Copenhagen, Denmark, September 9-15, 2012*, Olivier Danvy (Ed.). ACM, 8–29. https://doi.org/10.1145/2661103.2661105

[9] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[10] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2009. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.* 19, 5 (2009), 509–543. https://doi.org/10.1017/S0956796809007205

[11] Patrick Cousot. [n.d.]. Symbolic Execution is a case of Abstract Interpretation? Theoretical Computer Science Stack Exchange. https://cstheory.stackexchange.com/q/42290

[12] Patrick Cousot. 1978. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. https://tel.archives-ouvertes.fr/tel-00288657

[13] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, Robert M. Graham, Michael A. Harrison, and Ravi Sethi (Eds.). ACM, 238–252. https://doi.org/10.1145/512950.512973

[14] Shingo Eguchi, Naoki Kobayashi, and Takeshi Tsukada. 2018. Automated Synthesis of Functional Programs with Auxiliary Functions. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science)*, Sukyoung Ryu (Ed.), Vol. 11275. Springer, 223–241. https://doi.org/10.1007/978-3-030-02768-1_13

[15] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA (LIPIcs)*, Thomas Ball, Rastislav Bodík, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.), Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 113–128. https://doi.org/10.4230/LIPIcs.SNAPL.2015.113

[16] Daniel P. Friedman, William E. Byrd, and Oleg Kiselyov. 2005. *The reasoned schemer*. MIT Press.

[17] Aggelos Giantsios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2017. Concolic testing for functional languages. *Sci. Comput. Program.* 147 (2017), 109–134. https://doi.org/10.1016/j.scico.2017.04.008

[18] Andy Gill. 2019. The Monad Transformer Library. http://hackage.haskell.org/package/mtl-2.2.2/

[19] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, Vivek Sarkar and Mary W. Hall (Eds.). ACM, 213–223. https://doi.org/10.1145/1065010.1065036

[20] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. Program Synthesis. *Foundations and Trends in Programming Languages* 4, 1-2 (2017), 1–119. https://doi.org/10.1561/2500000010

[21] Peter Hancock and Anton Setzer. 2000. Interactive Programs in Dependent Type Theory. In *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings (Lecture Notes in Computer Science)*, Peter Clote and Helmut Schwichtenberg (Eds.), Vol. 1862. Springer, 317–331. https://doi.org/10.1007/3-540-44622-2_21

[22] Jason Hemann and Daniel P. Friedman. 2013. μKanren: A Minimal Functional Core for Relational Programming. In *Proceedings of the 2013 Workshop on Scheme and Functional Programming (Scheme'13)*.

[23] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. 2008. Polymorphic Embedding of DSLs. In *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*, Yannis Smaragdakis and Jeremy G. Siek (Eds.). ACM, 137–148. https://doi.org/10.1145/1449913.1449935

[24] Mark P. Jones. 1995. Functional Programming with Overloading and Higher-Order Polymorphism. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text (Lecture Notes in Computer Science)*, Johan Jeuring and Erik Meijer (Eds.), Vol. 925. Springer, 97–136. https://doi.org/10.1007/3-540-59451-5_4

[25] Lennart C. L. Kats and Eelco Visser. 2010. The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, 444–463. https://doi.org/10.1145/1869459.1869497

[26] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *PACMPL* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

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

[28] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. https://doi.org/10.1145/2804302.2804319

[29] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[30] Dmitry Kosarev and Dmitry Boulytchev. 2016. Typed Embedding of a Relational Language in OCaml. In *Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara, Japan, September 22-23, 2016. (EPTCS)*, Kenichi Asai and Mark R. Shinwell (Eds.), Vol. 285. 1–22. https://doi.org/10.4204/EPTCS.285.1

[31] Guodong Li, Indradeep Ghosh, and Sreeranga P. Rajan. 2011. KLOVER: A Symbolic Execution and Automatic Test Generation Tool for C++ Programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings (Lecture Notes in Computer Science)*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.), Vol. 6806. Springer, 609–615. https://doi.org/10.1007/978-3-642-22110-1_49

[32] Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 333–343. https://doi.org/10.1145/199448.199528

[33] Marcus Lindner, Jorge Aparicius, and Per Lindgren. 2018. No Panic! Verification of Rust Programs by Symbolic Execution. In *16th IEEE International Conference on Industrial Informatics, INDIN 2018, Porto, Portugal, July 18-20, 2018*. IEEE, 108–114. https://doi.org/10.1109/INDIN.2018.8471992

[34] Kuang-Chen Lu, Weixi Ma, and Daniel P. Friedman. 2019. Towards a miniKanren with fair search strategies. In *Proceedings of the Workshop on miniKanren 2019, Berlin, Germany*. http://minikanren.org/workshop/2019/minikanren19-final1.pdf

[35] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, David Grove and Steve Blackburn (Eds.). ACM, 619–630. https://doi.org/10.1145/2737924.2738007

[36] Dmitry Rozplokhas, Andrey Vyatkin, and Dmitri Boulytchev. 2019. Certified Semantics for miniKanren. In *Proceedings of the Workshop on miniKanren 2019, Berlin, Germany*. http://minikanren.org/workshop/2019/minikanren19-final5.pdf

[37] Koushik Sen and Gul Agha. 2006. CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings (Lecture Notes in Computer Science)*, Thomas Ball and Robert B. Jones (Eds.), Vol. 4144. Springer, 419–423. https://doi.org/10.1007/11817963_38

[38] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Michel Wermelinger and Harald C. Gall (Eds.). ACM, 263–272. https://doi.org/10.1145/1081706.1081750

[39] Wouter Swierstra and Tim Baanen. 2019. A Predicate Transformer Semantics for Effects. *PACMPL* ICFP (2019). https://doi.org/10.1145/3236767

[40] Emina Torlak and Rastislav Bodík. 2013. Growing solver-aided languages with rosette. In *ACM Symposium on New Ideas in Programming and Reflections on Software, Onward! 2013, part of SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, Antony L. Hosking, Patrick Th. Eugster, and Robert Hirschfeld (Eds.). ACM, 135–152. https://doi.org/10.1145/2509578.2509586

[41] Emina Torlak and Rastislav Bodík. 2014. A lightweight symbolic virtual machine for solver-aided host languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 530–541. https://doi.org/10.1145/2594291.2594340