From definitional interpreter to symbolic executor

Mensing, Adrian D.; Van Antwerpen, Hendrik; Poulsen, Casper Bach; Visser, Eelco

DOI
10.1145/3358502.3361269

Publication date
2019

Document Version
Final published version

Published in
META 2019 - Proceedings of the 4th ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection, co-located with SPLASH 2019

Citation (APA)

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.
From Definitional Interpreter to Symbolic Executor

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

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

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

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.\footnote{https://github.com/MetaBorgCube/From-Definitional-Interpreter-To-Symbolic-Executor}

## 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 \cite{Friedman1991, Byrd1994, Mensing2007, Mensing2008}, a language for relational programming and constraint logic programming, which has been implemented in a wide range of different languages; notably Scheme \cite{Harper1993, Mensing2007}, but also, e.g., OCaml \cite{Byrd2013}. 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 \cite{Byrd2013}. 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 \cite{Mensing2016, Mensing2018} is a solver-aided language that extends Racket \cite{Poulsen2014} 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 domain-specific languages at large, by automatically deriving symbolic executors from definitional interpreters.

With Pattern Matching

### 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.

## 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.

### 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 [e₁, ..., eₙ]` represents an `n`-ary term whose head symbol is `f`, and whose sub-term values are the results of evaluating each expression `e₁, ..., eₙ`. Case `e [(p₁, e₁), ..., (pₙ, eₙ)]` is a pattern match expression which first evaluates `e` to a value and then attempts to match the resulting value against the patterns `p₁, ..., pₙ`, where patterns are given by the type `Patt`. `Letrec` expressions are restricted to bind value expressions, given by the type `ValExpr`.

### 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 \cite{Schrijvers2010} of a language that allows us to define a family of interpreters for the same language,
There are two reasons why we use a specialized version. The MonadEnv is a specialized version of the classical reader Monad m class operation for branching: m following type class constrains a monad reason, we treat conditional branching as an effect. The following Haskell type class constrains a monad [18, 24, 32]:

```
newtype 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:

```
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 return type a:

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

Values The following type classes define the constructors for term values con, and function closures clos, 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 :: String -> [val] -> val

class FunVal val where
  clos :: String -> Expr -> Env val -> val

class FunApp val 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

2The 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.

```
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
```
(lazy) recursive definitions to define a recursive environment

$$nv$$ that ValExprs 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]

**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**

$$\text{match } v (\text{Cases } ((p, m) : bs)) = \text{case } vmatch (v, p) \text{ of}$$

Just $$nv \rightarrow \text{local } (\lambda (nv_0 \rightarrow nv + nv_0)) \text{ (interp e)}$$

Nothing $$\rightarrow \text{match } v \text{ (Cases bs)}$$

match _ (Cases []) = throwError "Match failure"

**vmatch :: (ConcreteValue, Patt) -> Maybe (Env ConcreteValue)**

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

$$\text{runSteps :: Expr \rightarrow Env ConcreteValue \rightarrow}$$

Either String ConcreteValue

$$\text{runSteps } e \text{ } nv = \text{runExcept } (\text{runReaderT } (\text{interp } e) \text{ } nv)$$

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 Bannen [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 \( (b \to \text{Free } c \ a) \) of Step. The Free data type is a monad:

\[
\text{instance Monad (Free } c \text{) where }
\begin{align*}
\text{return} &= \text{Stop} \\
\text{Step } a &= k = k a \\
\text{Step } c \ f &= k = \text{Step } c (\lambda x \to f \ x) = k
\end{align*}
\]

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:

\[
\text{data Cmd } \text{val} \star \text{=} \ast \to \ast \text{where }
\begin{align*}
\text{Match } \text{:: } \text{val} & \to \text{Cases } \text{(Free } \text{Cmd val}) \text{ val} \to \\
\text{Cmd val val} \\
\text{Local } \text{:: } (\text{Env val} \to \text{Env val}) \to \text{Free } (\text{Cmd val}) \text{ val} \to \\
\text{Cmd val val} \\
\text{Ask } \text{:: } \text{Cmd val } \text{(Env val)} \\
\text{App } \text{:: } \text{val} \to \text{val} \to \text{Cmd val val} \\
\text{Eq } \text{:: } \text{val} \to \text{val} \to \text{Cmd val val} \\
\text{Fail } \text{:: } \text{String } \to \text{Cmd val val}
\end{align*}
\]

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

\[
\text{comp } \text{:: (TermVal val, FunVal val) } \Rightarrow \text{Cmd val val}
\]

The command trees that \( \text{comp} \) 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 \( \text{Thread} \), 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 \( \text{step} \) function:

\[
\text{type } \text{Thread} = \text{Free } (\text{Cmd ConcreteValue})
\]

\[
\text{step } \text{:: } \text{Thread } \to \text{ConcreteValue } \Rightarrow \\
\text{ConcreteMonad } \text{(Thread, ConcreteValue)}
\]

\[
\begin{align*}
\text{step } (\text{Stop } x) &= \text{return } (\text{Stop } x) \\
\text{step } (\text{Step } (\text{Match } \_ (\text{Cases } )) \_ ) &= \\
\text{throwError } "\text{Pattern match failure}" \\
\text{step } (\text{Step } (\text{Match } v (\text{Cases } ((p, m) : bs))) k) &= \\
\text{case } \text{vmatch } (v, p) \Rightarrow \\
\text{Just } nv =
\end{align*}
\]

\[
\text{return } (\text{Step } (\text{Local } (\lambda n v) \to n v + n v) m) k)
\]

\[
\text{Nothing } \Rightarrow \text{step } (\text{Step } (\text{Match } v (\text{Cases } bs)) k)
\]

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

\[
\text{drive } \text{:: } \text{Thread } \to \text{ConcreteValue } \Rightarrow \\
\text{ConcreteMonad } \text{ConcreteValue}
\]

\[
\begin{align*}
\text{drive } (\text{Stop } x) &= \text{return } x \\
\text{drive } c &= \text{do } r \leftarrow \text{step } c; \text{drive } r
\end{align*}
\]

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

\[
\text{runSteps } \text{:: } \text{Expr } \to \text{Env ConcreteValue } \Rightarrow \\
\text{Either } \text{String ConcreteValue}
\]

\[
\text{runSteps } e = \text{runExcept } (\text{runReaderT } (\text{drive } (\text{comp } e)) e)
\]

### 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 \( \text{ConcreteValue} \) data type from § 2.3 with a symbolic variable constructor, \( \text{SymV} \):

\[
\text{data } \text{SymbolicValue } = \text{Conv'} \text{ String } [\text{SymbolicValue}] \\
| \text{Closv'} \text{ String } \text{Expr} \\
| \text{SymV } \text{String}
\]

#### 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 \( \text{Int} \) field for keeping track of a fresh supply of symbolic variable names:

\[
\text{type } \text{SymbolicMonad } = \\
\text{ReaderT } (\text{Env SymbolicValue}) \\
(\text{StateT } \text{Int} (\text{Except String}))
\]

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 \( \text{set} \) 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:

\[
\text{type } \text{Unifier } = \mathbb{[} \text{String, SymbolicValue}] \\
\text{type } \text{Unifier}_N = \mathbb{[} \text{(SymbolicValue, SymbolicValue)}]
\]
type SymbolicSetMonad =
  StateT (Unifier, UnifierN) (ListT SymbolicMonad)

Here, Unifier witnesses how symbol variables must be instan-
tiated in order to complete a single transition step, repre-
senting a particular execution path of the program being sym-
boically executed. UnifierN represents a set of negative uni-
fication constraints. We motivate the use and need for the-
se shortly. The ListT monad transformer generalizes the
return type of a monadic computation $m$ to return a list of
as; 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 sat-
sifies 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. effects are interpreted. Thus
all cases of the transition function step, (below) are identical
to the small-step transition function from §3, except for the
cases for the Match and Eq. Furthermore, the definitional
interpreter from Fig. 2 is unchanged. We summarize the in-
teresting cases for the step function, which takes a symbolic
interpretation thread, Thread, as input, and returns a set of
threads (note the use of SymbolicSetMonad):

type Thread = Free (Cmd SymbolicValue)

step :: Thread, SymbolicValue →
  SymbolicSetMonad (Thread, SymbolicValue)

step₁ = (Step (Match _ (Cases []))) _ = mzero

step₂ = (Step (Match v (Cases ((p, m) : bs)))) k = (do
  (nv, u) ← vmatch (v, p)
  (applySubst u (Step (Local (λnv0 → nv + n0) m) k))
  'mplus' step₁ (Step (Match v (Cases bs) k))
  'catchError' (λ_ → step₁ (Step (Match v (Cases bs) k)))

step₁ = (Step (Eq. v₁ v₂) k) =
  case unify v₁ v₂ of
    Just [] → return (k (ConV "true" []))
    Just u → do
      (applySubst u (k (ConV "true" []))) 'mplus'
      (constrainUnify 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 function
(elided for brevity). If the value contains symbolic variables,
the vmatch 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 step₁ 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 step₁ function above for expressions of the
form Eq. v₁ v₂ checks whether v₁ and v₂ are unifiable. If
they are not, their unifier, there is only one possible execu-
tion path to consider, namely the execution path where v₁
and v₂ are equal. Otherwise, if v₁ and v₂ have a non-empty
unifier, there are two possible execution paths to con-
sider: one where v₁ and v₂ are equal, and one where they
are not. The step₁ 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 v₁ and v₂, such that
v₁ and v₂ cannot be unified at any point in the future during
symbolic execution.

Driver Loop The driver loop for symbolic execution is gen-
eralized to operate on sets of possible execution paths, where
each execution path is given by a configuration Config:

type Config, a = (a, Env SymbolicValue, UnifierN)

drive :: [Config] (Thread, SymbolicValue) →
  SymbolicSetMonad (Config, SymbolicValue, [Config])

drive [] = throwError "No solution found"

drive, ts =
  case isDone ts of
    (Just c, ts') → return (c, ts')
    _ → do
      ts' ← iterate ts
      drive, ts'

A configuration comprises a value, an environment which
may contain terms with symbolic variables, and a list of neg-
ative unification constraints (UnifierN). The drive 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 con-
genration is iterated by a single transition step, and drive 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 sym-
boitic executor from the concrete definitional interpreter from
§3. Invoking this symbolic executor with input programs
we are symbolically executing (Fig. 1), and what we can do with our derived symbolic executor and small
variables that make the result of symbolic execution of the
::
append0
Here, append is a recursive function defined in the language
we are symbolically executing (Fig. 1), and @@ is syntactic
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 cx is a “top-level” constraint for picking n
solutions to a constraint cx that contains existentially
quantified symbolic variables. CEx x cx introduces an existentially
quantified symbolic variable, by populating the environment
of a symbolic interpreter with a symbolic variable value
binding SymV x for x, where x is a fresh symbolic variable
name. CEq e1 e2 is a constraint that e1 and e2 evaluate to
the same value, and CNEq e1 e2 is a constraint that e1 and e2
evaluate to different values.

Our approach to constraint solving is given by the solve
function in Fig. 4 which, in turn, calls the search function
whose type signature is shown in the figure, but whose imple-
mentation we omit for brevity. search e ts c eq 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
data Constraint = CTake Int ExConstraint
data ExConstraint = CEx String ExConstraint
  | CEq Expr Expr
  | CNEq Expr Expr

Figure 3. Syntax for a tiny constraint language

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

Figure 3. Syntax for a tiny constraint language

solve :: Constraint → SymbolicMonad [Env SymbolicValue]
solve (CTake n cx) = solvec cx n
solvec :: ExConstraint → Int →
  SymbolicMonad [Env SymbolicValue]
solvec (CEx x cx) n = do
  nx ← fresh'
  Reader.local (λnv → (x, SymV nx) : nv) (solvec cx n)
solvec (CEq e1 e2) n = do
  nv ← ask
  search e1 [(interp e2, nv, [])] unify n
solvec (CNEq e1 e2) n = do
  nv ← ask
  search e1 [(interp e2, nv, [])]
  (λv1 v2 → case unify v1 v2 of
    Just _ → Nothing
    Nothing → Just [])

type SymbolicEq =
  SymbolicValue → SymbolicValue → Maybe Unifier
search :: Expr →
  [Config, (Thread, SymbolicValue)] →
  SymbolicEq →
  Int →
  SymbolicMonad [Env SymbolicValue]

Figure 4. A constraint solver for symbolic execution con-
straints

sugar for ‘App’. Solving the append0 constraint yields the
instantiation of q to the list containing all input atoms in
sequence.

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")))

17
We have shown how to derive a symbolic executor from a work, Rozplokhas et al. [36] provide a certified definition of we conjecture a correctness proposition for our symbolic execution within the framework of abstract interpretation. This formalization is only available in French [11].

But it is an open question how the small-step interpretation strategy based on free monads that we adopted in § 3 will eventually yield, if we were to iterate that configuration.

2. Any symbolic execution path, given by calling runSteps, under nv, 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.

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.

---

‘CEq (atom "a" ‘cons’ (atom "b" ‘cons’ (atom "c" ‘cons’ (atom "d" ‘cons’ (atom "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 be a function that uses the drive function to drive an expression to a final value and pool of alternative execution paths that may yet yield a final result:

\[
\text{runSteps} : \text{Expr} \rightarrow \text{Env} \rightarrow \text{SymbolicValue} \rightarrow \text{Either} \ \text{String} \ \text{(SymbolicValue, [Config, (Thread, SymbolicValue)])}
\]

We conjecture that, for any pair of concrete environment nv and symbolic environment nv', 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 returns; or yields a value that one of the configurations in runSteps, will eventually yield, if we were to iterate that configuration.

2. Any symbolic execution path, given by calling runSteps, under nv, 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'.

We believe that abstract interpretation [13] is a suitable framework for formalizing the correspondence between concrete and symbolic execution. 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.

---

3Indeed, 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].
References


