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.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.
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:

\[ xs \equiv \text{reverse } xs \]
Symbolic execution explores all execution paths through the \texttt{reverse} function that satisfy the constraint, and instantiates \texttt{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}

\textbf{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{Friedman:1991:FBP:110832.110836, Byrd:2002:SBP:777453.777491, Scheme:1990, Racket:2018}, a language for relational programming and constraint logic programming, which has been implemented in a wide range of different languages; notably Scheme \cite{Scheme:1990, Racket:2018}, but also, e.g., OCaml \cite{OCaml:2018}. 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{Friedman:2012:TFQ:2421342.2421364}. 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{Rosette:2010, Rosette:2012} is a solver-aided language that extends Racket \cite{Racket:2018} 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., \cite{Devadas:1990:ATF:100469.100474, Devadas:1991:ATF:113273.113277, Devadas:1993:ATF:113278.113284, Devadas:1994:ATF:745643.745644, Devadas:1996:ATF:113285.113289, Devadas:1997:ATF:113290.113294}. Many of these frameworks are so-called \textit{concolic} frameworks that work by instrumenting a concrete language runtime to track \textit{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.

\textbf{Contributions}

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

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

\section{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 \textit{Haskell} to implement a definitional interpreter for a functional language with pattern matching.

\subsection{Syntax}

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

\subsection{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 \texttt{EffVal} type class which in turn depends on a number of type classes that constrain the polymorphic notion of effects (defined by a monad \texttt{m}) and values (defined by a value type \texttt{val}) of the interpreter. The \texttt{EffVal} type class is thus a polymorphic embedding \cite{Haskell:2018} of a language that allows us to define a \textit{family} of interpreters for the same language,
MonadEnv is a specialized version of the classical reader
class
lowing 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:

\[
\text{newtype IfThenElse } m \ a = \text{ITE} (m \ a, m \ a)
\]

\[
\text{instance Monad } m \Rightarrow \\
  \text{MonadBranch Bool rval IfThenElse } m \ \text{where}
\]

\[
\begin{align*}
  \text{branch True } & \ \text{ITE} (t, \_) = t \\
  \text{branch False } & \ \text{ITE} (\_, f) = f
\end{align*}
\]

For our interpreter, which branches on values and returns values of the same type, we rely on the following more restrictive version of MonadBranch:\footnote{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.}

\[
\text{class Monad } m \Rightarrow \text{MonadMatch } \text{val fork m where}
\]

\[
\text{match :: val → fork m val → m val}
\]

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:

\[
\text{newtype Cases } m \ a = \text{Cases } [(\text{Patt, m a})]
\]

Values

The following type classes define the constructors for term values \text{con}_v and function closures \text{clos}_v, as well as operation \text{app} for applying a function to an argument and operation \text{eq} for checking equality between two term values.

\[
\begin{align*}
\text{class TermVal } \text{val where}
  \text{con}_v & \ :: \text{String} → [\text{val}] → \text{val} \\
\text{class FunVal } \text{val where}
  \text{clos}_v & \ :: \text{String} → \text{Expr} → \text{Env} \text{val} → \text{val} \\
\text{class FunApp } \text{val where}
  \text{app} & \ :: \text{val} → \text{val} → \text{val} \\
\text{class TermEq } \text{val where}
  \text{eq} & \ :: \text{val} → \text{val} → \text{val}
\end{align*}
\]

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: \text{mmap} maps a monadic function over a list; \text{mapSnd} maps a function over the second element of a tuple; and \text{resolve} resolves a name in an association list, or fails. The implementation of Letrec uses Haskell’s support for

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

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

\[
data ValExpr = VCon String [ValExpr] \\
  | \text{VLam } String Expr
\]

\[
data Patt = PVar String \\
  | \text{PCon } String
\]
The type class instances for this notion of value and monad allow the use of monads: (lazy) recursive definitions to define a recursive environment

\[
\text{ConcreteMonad}
\]

instance MonadMatch ConcreteValue Cases

ConcreteMonad where

match v (Cases ((p, m) : bs)) = case vmatch (v, p) of

\[\text{Just } v \rightarrow \text{local } (\lambda \text{nv} \rightarrow v + \text{nv}) m\]

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

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

vmatch :: (ConcreteValue, Patt) \rightarrow

Maybe (Env ConcreteValue)

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

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

\[
\text{runSteps} :: \text{Expr} \rightarrow \text{Env ConcreteValue} \rightarrow \text{Either String ConcreteValue} \\
\text{runSteps } e \text{ nv } = \text{runExcept (runReaderT (interp e) nv)}
\]

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 Bannen [39], the following data type defines a family of free monads:

\[
data \text{Free } c a = \text{Stop } a \\
\mid \forall b. \text{Step } (c b) (b \rightarrow \text{Free } c a)
\]
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:

```haskell
instance Monad (Free c) where
  return = Stop
  Step 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:

```haskell
data Cmd val = :: s → +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 :: val → val → Cmd val val
  Eq :: 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:

```haskell
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, 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:

```haskell
type Thread = Free (Cmd ConcreteValue)
step :: Thread, ConcreteValue →
  ConcreteMonad (Thread, 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 (λnv0 → nv + nv0) 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):

```haskell
drive :: Thread, 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:

```haskell
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:

```haskell
data SymbolicValue = ConV String [SymbolicValue]
  | ClosV String
  | Conv 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:

```haskell
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:

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

Here, Unifier witnesses how symbolic variables must be inst-
stantiated in order to complete a single transition step, representing a particular execution path of the program being symbolically executed. UnifierN 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 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 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 expressions 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 interesting 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)

step, (Step (Match _ (Cases [])) _) = mzero

step, (Step (Match v (Cases ((p, m) : bs))) k) = (do
    (nv, u) ← vmatch (v, p)
    'mplus' step, (Step (Match v (Cases bs)) k))

'catchError' (λ_ → step, (Step (Match v (Cases bs)) k))

step, (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 step, 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 function above for expressions of the form Eq v1 v2 checks whether v1 and v2 are unifiable. If they are unifiable with the empty unifier, there is only one possible execution 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 consider: one where v1 and v2 are equal, and one where they are not. The step 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)] →
    SymbolicMonad (Config, SymbolicValue,
        [Config, (Thread, SymbolicValue)]

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 negative 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 configuration is iterated by a single transition step, and drive is called recursively on the resulting list of configurations.

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...
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 implementation 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):

```plaintext
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 containing all input atoms in sequence.

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

```plaintext
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)))))
        `CEq` (atom "a" `cons` (atom "b" `cons` (atom "c" `cons` (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:

```plaintext
append02 :: Constraint
append02 =
    grab 6 (exists "x" (exists "y"
        ((append @@ (var "x") @@ (var "y")))
        `CEq` (atom "a" `cons` (atom "b" `cons` (atom "c" `cons` (atom "d" `cons` (atom "e" `cons` nil)))))
    )
```

```plaintext
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 4. A constraint solver for symbolic execution constraints
```
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 $e :: Expr$ 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.

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


