From definitional interpreter to symbolic executor

Mensing, Adrian D.; van Antwerpen, Hendrik; Poulsen, Casper; 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.
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

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]. 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.

**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 [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`.

## 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.

---

¹https://github.com/MetaBorgCube/From-Definitional-Interpreter-To-Symbolic-Executor
There are two reasons why we use a specialized version. The first is a specialized version of the classical reader

\[
\text{class MonadEnv val m where}
\begin{array}{l}
\text{ask :: m (Env val)}\\
\text{local :: (Env val \rightarrow Env val) \rightarrow m val \rightarrow m val}
\end{array}
\]

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 \(\text{MonadBranch}\) which represents a classical if-then-else expression:

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

\[
\text{instance Monad m \Rightarrow MonadBranch Bool rval IfThenElse m where}
\begin{array}{l}
\text{branch True (ITE (t, \_)) = t}\\
\text{branch False (ITE (\_, f)) = f}
\end{array}
\]

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

\[
\text{class Monad m \Rightarrow MonadMatch val fork m where}
\begin{array}{l}
\text{match :: val \rightarrow fork m val \rightarrow m val}
\end{array}
\]

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 \(a\):

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

Values

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

\[
\text{class TermVal val where}
\begin{array}{l}
\text{con :: String \rightarrow [ val ] \rightarrow val}\\
\text{clos :: String \rightarrow Expr \rightarrow Env val \rightarrow val}\\
\text{app :: val \rightarrow val \rightarrow m val}\\
\text{eq :: val \rightarrow val \rightarrow m val}
\end{array}
\]

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 \(\text{Letrec}\) uses Haskell’s support for
The type class instances for this notion of value and monad are defined in the obvious way:

\[
\text{ConcreteMonad} \quad \text{is isomorphic to:}\n\]

\[
\text{Except} \quad \text{ReaderT} (\text{Env ConcreteValue}) (\text{Except String})
\]

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

\[
\text{type ConcreteMonad} = \\
\quad \text{ReaderT (Env ConcreteValue) (Except String)}
\]

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:

\[
\text{instance MonadMatch ConcreteValue Cases}
\]

where

\[
\begin{align*}
\text{match } v \text{ (Cases ((p, m) : bs))} &= \text{ case vmatch } (v, p) \text{ of} \\
&\quad \text{Just } nv \rightarrow \text{ local } (\lambda n v_0 \rightarrow n v + n v_0) m \\
&\quad \text{Nothing } \rightarrow \text{ match } v \text{ (Cases bs)} \\
&\quad \text{match } _- \text{ (Cases [ ]) } = \text{ throwError "Match failure"}
\end{align*}
\]

\[
\text{vmatch} :: (\text{ConcreteValue}, \text{Patt}) \rightarrow \\
\quad \text{Maybe (Env ConcreteValue)}
\]

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

\[
\text{runSteps} :: \text{Expr} \rightarrow \text{Env ConcreteValue} \rightarrow \\
\quad \text{Either String ConcreteValue}
\]

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

### 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:

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

![Figure 2. A definitional interpreter for a language with pattern matching](image)
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 :: val → val → Cmd val val
Eq :: val → val → Cmd val val
Fail :: String → Cmd val z

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

cmp :: (TermVal val, FunVal val) ⇒
       Expr → Free (Cmd val)

The command trees that cmp 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:

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 (λnv → nv + nv) 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 (cond e)) nv)

4 From Definitional Interpreter to
Symbolic Executor

In this section we derive a symbolic executor from the defini-
tional 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 exten-
sion of the notion of ConcreteValue data type from § 2.3
with a symbolic variable constructor, SymV:

data SymbolicValue = ConV String [SymbolicValue]
       | ClosV String
       | (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 execu-
tion 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 UnifierN = [(SymbolicValue, SymbolicValue)]
where we have exhausted the list of patterns to match a value
As in § 3, there are two cases for \( \text{match} \) against a pattern, using the side-effectful

to consider. In case we have exhausted the list of patterns
against, and one for the case where there are more cases
possible execution path to consider, namely the execution
path where \( v_1 \) and \( v_2 \) are equal, and one where they
are not. The \( \text{step} \) function returns the union (again, using
\'mplus\') of two threads representing each of these execution
paths. For safety, we register a \text{negative unification constraint}
for the execution path that disequalizes \( v_1 \) and \( v_2 \), such that
\( v_1 \) and \( v_2 \) cannot be unified at any point in the future during
symbolic execution.

\textbf{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 \( \text{Config} \):

\textbf{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-

bolic executor from the concrete definitional interpreter from
§ 3. Invoking this symbolic executor with input programs

```haskell
type SymbolicSetMonad =
  StateT (Unifier, UnifierN) (ListT SymbolicMonad)

Here, \( \text{Unifier} \) witnesses how symbolic variables must be in-
stantiated in order to complete a single transition step, repre-
senting a particular execution path of the program being
symbolically executed. \( \text{UnifierN} \) represents a set of \text{negative}
unification constraints. We motivate the use and need for
these shortly. The \( \text{ListT} \) monad transformer generalizes the return type of a monadic computation \( m a \) to return a list of
\text{as}; i.e., \( m \ [ \ a \] \). Note that, although we call \( \text{ListT} \) a monad
transformer, it is well-known that \( \text{ListT} \) in Haskell is not
guaranteed to yield a monad that satisfies the monad laws.

\textbf{Small-Step Transition Function} Our symbolic executor
is derived from the concrete semantics of effects in § 3 by
altering how we \( \text{Match} \) and \( \text{Eq} \) effects are interpreted. Thus
all cases of the transition function \( \text{step}_1 \) (below) are identical
to the small-step transition function from § 3, except for the
cases for the \( \text{Match} \) and \( \text{Eq} \). Furthermore, the definitional
interpreter from Fig. 2 is unchanged. We summarize the in-
teresting cases for the \( \text{step} \) function, which takes a symbolic
interpretation thread, \( \text{Thread} \), as input, and returns a set of
threads (note the use of \( \text{SymbolicSetMonad} \)):

\begin{verbatim}
type Thread, = Free (Cmd SymbolicValue)

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

step_1 (Step Match _ (Cases [])) _ = mzero
step_2 (Step (Match v (Cases ((p, m) : bs))) k) = (do
  (nv, u) ← vmatch1 (v, p)
  mplus' step_3 (Step (Match v (Cases bs))) k)

where

vmatch1 :: SymbolicValue →
  Match v (Cases bs) → Match v (Cases bs)

\end{verbatim}

As in § 3, there are two cases for \( \text{Match} \): 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 \( \text{mzero} \) to return an
empty set of result threads. Otherwise, we match a value
against a pattern, using the side-effectful \( \text{vmatch} \) function
(elided for brevity). If the value contains symbolic variables,
the \( \text{vmatch} \) function computes a unifier to be be applied

```
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 $SymV\ x_f$ 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. $search\ s\ ts\ c\ eq\ n$ 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**

```haskell
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:

**append01 :: Constraint**

```haskell
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**

```haskell
append02 =
  grab 6 (exists "x" (exists "y")
    ((append @@ (var "x") @@ (var "y")))
```
We conjecture that, for any pair of concrete environment `nv`, we can derive a symbolic executor from a definitional interpreter. Rozplokhas et al. [36] provide a certified definition of abstract environment and methodology of Keidel et al. [26].

In very recent work, we conjecture a correctness proposition for our symbolic execution strategy based on free monads that we adopted in § 3 and § 4 to realize our symbolic executor fits into the framework of abstract interpretation. This formalization is engineered to use miniKanren's relational programming 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 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.

### 6 Case Study: Automatic Test Generation for Definitional Interpreters

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].

### 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


