Higher-Order Demand-Driven Symbolic Evaluation

ZACHARY PALMER, Swarthmore College, USA
THEODORE PARK, Swarthmore and Hopkins, USA
SCOTT SMITH, The Johns Hopkins University, USA
SHIWEI WENG, The Johns Hopkins University, USA

Symbolic backwards execution (SBE) is a useful variation on standard forward symbolic evaluation; it allows a symbolic evaluation to start anywhere in the program and proceed by executing in reverse to the program start. SBE brings goal-directed reasoning to symbolic evaluation and has proven effective in e.g. automated test generation for imperative languages.

In this paper we define DDSE, a novel SBE which operates on a functional as opposed to imperative language; furthermore, it is defined as a natural extension of a backwards-executing interpreter. We establish the soundness of DDSE and define a test generation algorithm for this toy language. We report on an initial reference implementation to confirm the correctness of the principles.

CCS Concepts: • Software and its engineering → Software testing and debugging: Formal software verification; • Theory of computation → Programming logic; Logic and verification; Verification by model checking.

Additional Key Words and Phrases: Symbolic Execution; Test Generation; Demand-Driven Execution

ACM Reference Format:
Zachary Palmer, Theodore Park, Scott Smith, and Shiwei Weng. 2020. Higher-Order Demand-Driven Symbolic Evaluation. Proc. ACM Program. Lang. 4, ICFP, Article 102 (August 2020), 28 pages. https://doi.org/10.1145/3408984

1 INTRODUCTION

Symbolic execution, the evaluation of a program over symbolic ranges of values instead of over concrete values, has proven to be a useful technique with real-world applications from lightweight program verification to automated test generation; see [Baldoni et al. 2018] for a recent survey of the area. Path explosion is a major shortcoming with symbolic execution: a vast number of the explored paths never get near the target program point in forward runs. A backward-running approach can avoid searching many of those paths.

This paper focuses on symbolic backwards execution (SBE) [Baldoni et al. 2018, §2.3], a variation on symbolic evaluation where evaluation can start at any point in the program and proceed in reverse to the program start. This reverse propagation is similar in spirit to how Dijkstra weakest-preconditions (wps) are propagated, and how classic backward program analyses propagate constraints in reverse. The advantage of SBE is the same as any goal-directed reasoning: by focusing on the goal from the start, there are fewer spurious paths taken.

Authors’ addresses: Zachary Palmer, Swarthmore College, 500 College Ave., Swarthmore, PA, 19081, USA, zachary.palmer@swarthmore.edu; Theodore Park, Swarthmore and Hopkins, USA, tedpark7@gmail.com; Scott Smith, The Johns Hopkins University, 3400 N. Charles St., Baltimore, MD, 21218, USA, scott@cs.jhu.edu; Shiwei Weng, The Johns Hopkins University, 3400 N. Charles St., Baltimore, MD, 21218, USA, wengshiwei@jhu.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2020 Copyright held by the owner/author(s).
2475-1421/2020/8-ART102
https://doi.org/10.1145/3408984
SBEs have been developed for imperative languages; examples include [Chandra et al. 2009; Charreteur and Gotlieb 2010; Dinges and Agha 2014; Ma et al. 2011]. These reverse techniques are useful for goal-directed reasoning about paths leading to a particular program point: if a condition at a program point can be propagated back to the program start, this will deduce its validity. The aforementioned systems are capable of automatically generating tests exercising a particular program point, using backward symbolic execution to accumulate constraints required to reach the target. To be clear: SBE does not single-handedly solve the problem of symbolic execution performance, but it is a fundamentally different approach that has advantages in some contexts.

These imperative language systems do not directly generalize to functional languages. Functional languages have a combination of non-local variables and a control flow that can itself depend on (function) data flow which makes this gap non-trivial. In this paper, we develop DDSE: a demand-driven symbolic evaluator for higher-order functional languages which also propagates constraints backwards. We show how, unlike existing SBEs, DDSE may be constructed as a direct generalization of a backward concrete evaluator; this follows how forward symbolic evaluators are constructed as generalizations of forward concrete evaluators and lends a regularity to the process. With this regularity it is also possible to formally prove DDSE is correct, something not previously proven for any SBE. In order to show applicability of DDSE, we develop a theory and implementation of test generation for a functional language. While the paper focuses on the test generation application to show that concrete results are possible, DDSE is also applicable to other goal-directed problems that SBEs can address.

There exist demand-driven program analyses in parallel with demand-driven symbolic evaluators, both for imperative languages [Horwitz et al. 1995] and more recently for functional languages [Facchinetti et al. 2019; Germane et al. 2019]; DDSE is built on the infrastructure of one particular higher-order demand-driven program analysis, DDPA [Facchinetti et al. 2019].

In Section 2 we give a high-level overview of the principles behind the approach. Section 3 defines a novel demand-driven operational semantics which serves as the basis of our symbolic demand-driven evaluator. Section 4 extends the demand-driven operational semantics to symbolic DDSE and shows how it can be used for test generation. We formally prove that the symbolic interpreter extends the concrete one, and that tests inferred will in fact exercise the indicated line of code they were supposed to. Section 5 describes the implementation of the test generation algorithm and its performance on small benchmarks. Section 6 gives related work, and we conclude in Section 7. Proofs are found in the Supplemental Material associated with the paper [Palmer et al. 2020b].

2 OVERVIEW

Goal-directed program reasoning has a long tradition in programming languages, dating back to Dijkstra weakest-precondition (wp) propagation. We review a very simple example in Figure 1 to recollect wp propagation.

Suppose we started at line 6 with true as our (vacuous) assertion. By wp propagation since we know we are coming only from the true branch of the conditional, before line 3 we must have precondition \( \{ x < 25 \} \), and continuing to propagate, we have \( \{ x > 0 \land x < 25 \} \) in line 2. So, it means that input must be in the range of 1...24 for the target line 6 to be reached. This example gives some idea of how existing first-order symbolic backward executors (SBEs) [Chandra et al. 2009; Dinges and Agha 2014] work: they start with a vacuous precondition and back-propagate to the start of the program.

The goal of this paper is to show how a demand-driven symbolic evaluator can be developed for higher-order functional languages. Weakest precondition logic was designed for first-order stateful programs, and we aim to design a similar reverse propagation for functional programs.
Recall that the general case of higher-order functions includes two key differences from first-order programs: functions are passed as data, thus causing data flow to influence control flow, and function bodies capture non-local variables in closures. The aforecited systems give partial consideration of higher-order functions: they accommodate virtual method calls by an iterative process for estimating the call graph. However, no soundness properties are claimed in those works. By starting with a higher-order functional basis, we can develop a direct and provably sound demand-driven symbolic evaluator. We will describe DDSE in stages here: first defining the demand-driven evaluator, then extending it to deal with input, and finally performing symbolic evaluation starting from an arbitrary program point.

### 2.1 Demand-Driven Functional Evaluators

A functional evaluator can be written to be more demand-driven than the standard closure-based, environment-based, or substitution-based evaluators: the evaluator only needs to retain the current stack of function calls invoked, and from this information it is possible to reconstruct any variable’s value. Consider for example the program\(^1\) in Figure 2.

For the \( f y \) call on line 5, a standard evaluator would pass in the actual value 0 by some means. In our truly demand-driven evaluator, however, the body \( x + 1 \) executes without any binding for \( x \), only knowing that the function was called from line 5. When \( x \)'s value is needed in the body to add 1 to it, we rely on the fact that the call site \( f y \) was recorded. (We use unique variables in our A-normalized programs, so variable definitions serve to uniquely identify program points.) We know that \( x \)'s value will take on the value of \( y \) at that call site, which in turn can be seen to be 0. Although there is a call \( f 1 \) on line 6, we know that our parameter \( x \) does not have the argument value 1 here because the call site \( f y \) (and not the call site \( f 1 \)) was recorded. When the evaluator executes the \( f 1 \) call, it will again compute \( x + 1 \) but this time under the call site stack \( f 1 \), and there \( x \) will have value 1.

We now trace this more precisely. We will formally define a variable lookup assertion \( L([x], n, C) \equiv v \) to mean that \( v \) is the result of a lookup of variable \( x \), starting the (reverse) search from program line \( n \), and assuming the current call site stack context is \( C \). We will pun this relation as a function since lookup is deterministic, writing it as a function \( L([x], n, C) \) returning \( v \) equivalently.

The \([x]\) is just a singleton list; it can in general be a non-singleton for looking up non-local variables, a topic we address shortly in Section 2.2. The call stack \( C \) is not the forward execution stack, since we are \( wp \)-style walking the program in reverse; it merely denotes the calls entered and not yet exited in this reverse-order sequence. We now illustrate lookup in detail by showing how the value of \( \text{ret} \) in the above program is looked up from the program end and empty call stack (since we are starting outside any function calls), \( L([\text{ret}], 7, []) \).

---

\(^1\)This section uses an informal OCaml-like syntax. In our formal presentation below, we A-normalize our programs to clarify operator ordering; we give the formal grammar for our A-normalized form (ANF) in the following section.
(1) \( L([\text{ret}], 7, []) \equiv L([\text{fy}], 6, []) + L([\text{f1}], 6, []) \): Line 7 defines \( \text{ret} \) in terms of two other variables, so we now have two lookup sub-goals to find \( \text{ret} \)'s value: looking up \( \text{fy} \) and \( \text{f1} \) starting from the previous line. We will trace only \( \text{fy} \) in this example since \( \text{f1} \) is similar.

(2) \( L([\text{fy}], 6, []) \equiv L([\text{fy}], 5, []) \): We may skip over \( \text{f1} \)'s definition as we do not find \( \text{fy} \) on that line, making it irrelevant to \( \text{fy} \)'s lookup. (An astute reader may notice that skipping is not sound if the call does not terminate, so skipping is not always correct; we address this case at the end of this subsection.) In line 5, we find the definition of \( \text{fy} \) is not yet a value but is a function call: \( \text{fy} \)'s value is in fact the result of the application \( \text{f y} \). To obtain value, we first need to look up the definition of \( \text{f} \).

(3) \( L([\text{f}], 2, []) \equiv \text{fun } x \rightarrow \ldots \). We have looked up the definition of \( \text{f} \), so now we need to search for the result value of the function body, the contents of \( \text{fret} \). We perform the lookup \( L([\text{fret}], 4, [\text{fy}]) \), pushing the call site \( \text{fy} \) onto the call stack since the search has entered that function body.\(^2\)

(4) \( L([\text{fret}], 4, [\text{fy}]) \equiv L([x], 3, [\text{fy}]) + 1 \): When we perform the lookup, we see that \( \text{fret} \) is defined in the current line as the expression \( x + 1 \), so we next lookup \( x \) from the previous line.

(5) \( L([x], 3, [\text{fy}]) \equiv L([y], 2, [\text{fy}]) \): \( x \) is immediately seen as a parameter to the \( \text{f} \), so we want to look up the value of this parameter at the original call site. We know to examine call site \( \text{fy} \) since it was recorded on the call stack. So we induce a lookup of the argument \( y \) from the main program line right before \( \text{fy} \), which is line 2.

(6) \( L([y], 2, []) \equiv L([y], 1, []) \equiv \emptyset \): The lookup fails to find \( y \) on line 2, so we skip to line 1, where \( y \) is observed to be \( \emptyset \).

(7) Now that we have found our values, we can pop off lookup obligations. We have seen that \( L([x], 3, [\text{fy}]) \equiv L([y], 2, []) \equiv \emptyset \), so \( L([\text{fret}], 3, [\text{fy}]) \equiv \emptyset + 1 = 1 \), so \( L([\text{fy}], 6, []) \equiv 1 \).

(8) A similar lookup of \( \text{f1} \), yields \( L([\text{f1}], 6, []) \equiv 2 \), so \( L([\text{ret}], 7, []) \equiv 1 + 2 = 3 \).

Notice that in this process we did not use structures common in operational semantics of higher-order functions: there were no environments or closures and no term substitution was performed. That is because all variables are looked up on demand by a backtrace to their origin.

The trace above glosses over one detail needed for test generation: lookup is a data flow operation but test generation is a search for a control flow path from a program point to the start of the program. So, we in fact need to look up all variables encountered in the reverse search and cannot skip over any statement as it may have an input or non-termination side effect. Additionally, to make sure we reach the start of the program we initiate lookup with the very first variable in the program from our target point; the lookups traced above will be side effect lookups of that first variable lookup. In this process we will map the full control of the program up to the line we initially targeted as a consequence of tracing from that line back to the origin. Note that it will still be the case that only a limited portion of the program needs to be exercised: we only examine earlier control flows and not all branches need be taken.

\(^2\)Note that this logic is for call-by-name function call; for the call-by-value implemented here we need to also verify that the argument is not divergent by looking it up.

---

Fig. 3. Non-local variable example

```
let g =
  (fun x ->
   let gret =
     (fun y ->
      let gyret = x + y in gyret) in gret) in
let g5 = g 5 in
let g51 = g5 1 in
let g6 = g 6 in
let g62 = g6 2 in
let ret = g51 + g62 in ret
```
2.2 Non-Local Variables

Non-local variables are variables *used in a function but whose definitions lie outside of that function*; they must be placed in closures in standard functional language evaluators. To support non-local variable lookup in a reverse evaluator, we cannot rely on closures as they are a forward-passed structure. So, we instead use a stack to record the chain of call frames we need to walk back through to find where a non-local variable is local. This process is related to access links in compiler implementations of non-local variable lookup. This is why the variable looked up in the previous examples were singleton lists \([x]\) – in the general case it can be multiple function definitions followed by the variable, \(X = [f_1, \ldots, f_n, x]\) and lookup is generally \(\mathbb{L}(X, n, C) = v\). The lookup stack gets longer than two elements when the non-locals are themselves functions – the stack in effect is increasing as we climb the type hierarchy of non-local atomic values, functions, functionals, functions on functionals, etc.

We now perform an example lookup of a non-local variable to clarify this process. Consider the example of a Curried addition function in Figure 3; the use of \(x\) in line 5 is a non-local variable. Suppose we want to look up the value of \(g51\) for use in line 10.

1. \(\mathbb{L}([g51], 10, []) \equiv \mathbb{L}([g51], 7, [])\): We skip lines 8 and 9 to reach \(g51\)'s definition in line 7, which is the function application \(g5\) 1.
2. \(\mathbb{L}([g5], 6, []) \equiv (\text{fun } y \rightarrow \ldots)\): To look up \(g5\), we find that is defined as the result of function application \(g\) 5. So, we need to find the result returned by this call and so enter \(g\) and search for the value of its result variable \(g\text{ret}\). We find that \(g\text{ret}\) is immediately defined as the function \(\text{fun } y \rightarrow \ldots\) so we finally have found the value of \(g5\) and can resume finding the result of the call \(g5\) 1. Inspecting the source of \(\text{fun } y \rightarrow \ldots\), the return variable is \(g\text{ret}\) and so to find the result of the call we need to find the result of that variable from within this call, i.e. with \(g51\) on the call stack.
3. \(\mathbb{L}([g\text{ret}], 5, [g51]) \equiv \mathbb{L}([x], 4, [g51]) + \mathbb{L}([y], 4, [g51])\): Continuing, \(g\text{ret}\) is defined in line 5 as expression \(x + y\), so we need to look up \(x\) (and \(y\)) from the previous line in order to get \(g\text{ret}\)'s value. In this sub-lookup \(x\) is not defined in the current context: it is a non-local. So, we will have to work harder to find its value. First, we exit the \(g51\) call since the definition is not local, and redirect our search. The key idea is we can find the definition of \(x\) if we look up where function \(g5\) if defined: *that* must be a point in the program where \(x\) is also defined since it must be lexically in scope of that function definition (think about it: once we are at the function definition, \(x\) must have a value under static scoping convention). This is a subtle observation and is at the root of how we can avoid computing closures or similar structures.
4. \(\mathbb{L}([g5, x], 6, []) \equiv \mathbb{L}([x], 3, [g5])\): When performing this lookup, we are at line 6 at the top level of the program looking for the definition point of \(g5\); the lookup stack \([g5, x]\) here indicates that once we have found the definition point of \(g5\) in line 3, we will need to pop \(g5\) off of the non-locals stack, giving us a goal of looking up \(x\) in the context \(g5\). This implements the intuition for non-local lookup just described. The \([g5]\) call stack reflects that we had to enter the \(g5\) call site to find the \(\text{fun } y \rightarrow \ldots\) definition.
5. \(\mathbb{L}([x], 3, [g5]) \equiv 5\): We now are at a program point where \(x\) is defined (as the function parameter in this case) and can simply perform a local parameter lookup, specifically for the parameter at the \(g5\) call site which in turn we find is 5. This completes the non-local portion of the lookup, the rest is straightforward.

The above examples give a basic idea of the process; for even deeper lexically nested variables there will be further chaining back through a series of function definitions. For recursive functions, definable here via self-passing, the context stack may grow unboundedly but there is no need for any special handling in the lookup definition. The demand-driven evaluator described here is novel
but is based on ideas in [Facchinetti et al. 2019] (the ωDDPAc evaluator there), where it was used for the purpose of proving a program analysis sound.

2.3 Demand-Driven Symbolic Execution: DDSE

The primary contribution of this paper is to make a demand-driven symbolic evaluator, DDSE, based on the demand-driven evaluator of the previous subsection, and to show how DDSE may then be used to infer tests to reach an arbitrary line of code. There are several extensions to the above evaluator that are needed in order for it to evaluate symbolically and to infer tests. First, rather than lookup returning values, constraints on values are accumulated, in the form of logical formulae. This yields a symbolic evaluator. Then, the symbolic evaluator is modified to include input and to allow (reverse) execution to commence from any point in the program including inside a function or conditional; this latter modification may be used to generate a test reaching that program point. We will now work through these extensions.

2.3.1 Constraint-Based execution. First we show how the demand-driven evaluator can be extended to a symbolic demand-driven evaluator. The basic idea is simple: to accumulate all constraints on variable values in a global formula Φ which must remain satisfiable, and to define lookup to return a variable over which constraints are constructed. If a variable is directly receiving input, it obviously can’t have a concrete value. But additionally, any variable depending on input cannot have a concrete value; so, for uniformity, the symbolic evaluator always produces (constrained) variables. Still, there are subtleties on how to name variables given there may be many activations of the same variable at runtime. Fortunately, the pair of variable name and current call site stack serves as a unique reference into the runtime heap, assuming that the initial program had any duplicate variable definitions renamed (i.e. the program was alphatized). This property is formally established in the Supplementary Appendix as Lemma C.8. We use notation \( x^S \) for the pair of variable \( x \) annotated with context stack \( C \) to uniquely identify runtime heap locations. Note that every such pair denotes where the variable is defined; at a program point where we have only the use of a variable we must look it up to find its defining variable pair, as equations on variables must be on their definitions (equivalently, their heap locations) and not their uses. Every variable use is invariably a chain of variable aliases back to a definition of the variable; such chains may go through function calls. The symbolic lookup relation is of the form \( L(f, x, n, C) \equiv C^x_n \), the \( S \) standing for “symbolic”. Analogously with concrete lookup, we will equivalently write this in function form as \( L^S(f, x, n, C) \) returning \( C^x_n \). We will additionally produce a global set of constraints \( \Phi \) over all lookups which is an implicit result.

Let us re-evaluate the Figure 2 example symbolically to illustrate the differences, specifically by looking up the variable \( \text{ret} \) from line 7.

1. \( L^S(\text{ret}, 7, [0]) \equiv [\text{ret}] \) and \( [\text{ret}] = L^S(\text{fy}, 6, [0]) + L^S(f1, 6, [0]) \): We proceed with looking up \( \text{fy} \) and \( f1 \) to find their variable definitions, which we need in order to complete the second equation constraining \( [\text{ret}] \). We trace only \( L^S(\text{fy}, 6, [0]) \) since looking up \( f1 \) is similar.

2. \( L^S(\text{fy}, 6, [0]) \equiv L^S(f\text{ret}, 4, [\text{fy}]) \): Similar to how we looked up \( \text{fy} \) in the demand evaluator, we skip \( f1 \)’s definition and find that we need to look up \( f \\
's \) result value \( \text{ret} \).

3. \( L^S(f\text{ret}, 4, [\text{fy}]) \equiv [\text{fy}]f\text{ret} \) and \( [\text{fy}]f\text{ret} = L^S(\text{fy}, 3, [\text{fy}]) + 1 \): We have reached a point where a concrete value is constructed, so the latter equation (after the remaining lookup has completed) will be added to \( \Phi \). Note that here, \( [\text{fy}]f\text{ret} \) is the defining variable: the stack annotation \( [\text{fy}] \) disambiguates to mean the \( \text{fy} \) call site allocation of \( f\text{ret} \), as opposed to the \( f1 \) allocation - we have succeeded in avoiding variable name clashes.
(4) $L^s([x], 3, [fy]) \equiv L^s([y], 2, []) \equiv \|y \equiv 0$: After tracing back through the lookups, we find that we have reached a value definition with $\|y \equiv 0$, which we can add to $\Phi$ without any further lookup. This also entails that both $L^s([y], 2, [])$ and $L^s([x], 3, [fy])$ will return $\|y$.

(5) $\{fy\} fret = \|y + 1$: Given the above results, we can construct this equation and add it to $\Phi$. We can then return $\{fy\} fret$ as the defining variable for $fy$.

(6) $\{if\} ret = \{fy\} fret + \{f1\} fret$: Here, we complete the lookup not only for $fy$, but also for $f1$, which completes the equation in step 1 and allows us to add it to $\Phi$.

The final constraint set $\Phi$ for this lookup (including constraints added when looking up $f1$) is:

$$\Phi = \{\|\text{ret} = \{fy\} \text{fret} + \{f1\} \text{fret}, \{fy\} \text{fret} = \|y + 1, \{f1\} \text{fret} = 1 + 1, \|y = 0\}$$

By basic arithmetic we can conclude that $\|\text{ret} = 3$ is logically deducible from satisfiable $\Phi$. In the DDSE implementation we simply call out to a SMT solver to check for satisfiability of $\Phi$; the implementation will be discussed in Section 5 below.

2.3.2 Adding Input. While relatively easy in forward evaluators, adding input in this backward evaluation model is somewhat challenging. Forward evaluators simply process input in sequence as they execute the program. In the demand-driven process described above, however, values are looked up in the (reverse) order in which they are used rather than the order in which they are defined. For example, consider the program in Figure 4. Here, the input keyword reads an integer off standard input. When looking up $if\text{ret}$, we must first establish the value of the conditional $i2$, which is neither the first or last value in the input sequence. All we know about this value is that it is an input which was allocated to the heap on line 2.

To address this issue, we record inputs not as a stream but as a mapping from call-site annotated variables to values, in a similar manner to how we used annotated variables to uniquely identify heap locations in the symbolic evaluator above. If the input sequence of the original program were $[1, 2, 3]$, we would re-cast it as $i = \{i1 \mapsto 1, i2 \mapsto 2, i3 \mapsto 3\}$. Since all three variables are defined at top level, their call stack annotations are empty. We formally establish that the two notions of input are isomorphic.

On this simple example these inputs look similar to unconstrained variables, and could in fact be modeled as such. But for general test generation, inputs can repeated unboundedly (e.g. input a number $n$, and then input $n$ numbers to construct an integer list of length $n$). Importantly, we aim to show our ideas scale to arbitrary inputs, so we include them in the theory even though they introduce complications.

2.3.3 Test Generation. We may finally consider how to generate a test exercising any particular line of the program. Consider again the Figure 4 input example and suppose our goal is to find inputs which reach (i.e., cover) line 7. Lookup of $f\text{retp}$ from line 7 is non-trivial: this search begins inside the body of $f$, there are two call sites to $f$, and either could have been the calling point. So, a search is fundamentally required. A conservative approach would be that any call site could have called $f$, but a simple program analysis can build a conservative call graph to winnow out nearly all
call sites from contention. For this search process one additional global structure is present, along with formula $\Phi$: a path choice, $\Pi$, which records which call site was chosen in the current search attempt.

Suppose here we arbitrarily guessed $f \ i2$ in line 14 as the call site that got us to line 7; let us at a high level describe the lookup constraints $\Phi$ produced. In this case, we also know that with this choice the call stack when we started must be $[f i2]$ since the call site is at the top level. In general, we only have a partial notion of the full call stack if we start deeply in the program, but we can incrementally construct an isomorphic structure on-the-fly which we term a relative stack; see Section 4.1 for details.

We began our search in the then branch, so we know the conditional expression to be true; we will ultimately express this with a constraint of the shape $L^S([x], [f i2]) = 0$ where $L^S([x], [f i2])$ is the defining variable obtained by looking up $x$. This lookup in turn has the same result as looking up $f i2$’s call site argument, $i2$, reducing our work to determining $L^S([i2], 12, []).$ Since we came from the else branch, we will build a constraint of the shape $L^S([i2], 12, []) \neq 0$ to remember that this condition must have failed. Continuing with $L^S([i2], 12, []),$ we finally arrive at its definition in line 2. Since $i2$ is defined as an arbitrary input, it is only constrained to be an integer. Having completed lookup, we can fill in the unresolved lookups in the previous constraints: the else constraint is $|i2| \neq 0$ (because $L^S([i2], 12, []) = |i2|$) and the original then constraint is $|i2| = 0$ (since $L^S([x], [f i2]) = L^S([i2], 12, []) = |i2|$). These two constraints, $|i2| \neq 0$ and $|i2| = 0$, are immediately contradictory, meaning that this path will never happen and so the $f i2$ call site choice can be discarded.

So, rewinding back to the start, we will this time record in $\Pi$ that the $f i1$ call site was the caller of $f$. Skipping the details, it produces $\Phi = \{|i1| = 0\}$. Any input mapping conforming to these constraints, such as $\{|i1| \mapsto 0, |i2| \mapsto 0, |i3| \mapsto 0\}$, will exercise line 6, and so we have successfully deduced a test case.

Note that if all potential paths are unsatisfiable, we have a proof that there is no such test reaching the line, i.e. is it dead code. It also could be the case that the code is unreachable but there are infinitely many paths and so the search for a path will never terminate and the algorithm will be unable to prove the code is unreachable.

3 A REVERSE CONSTRUCTION INTERPRETER

In this section we construct a demand-driven interpreter for our language. In the following section we will then be able to symbol-ize this interpreter to make a sound demand-driven symbolic interpreter. Section 2.1 gave an informal treatment of how a demand-driven interpreter operates; here we make that informal intuition precise. This interpreter design was initially inspired by the $\omega$DDPAc interpreter of [Facchinetti et al. 2019]; unlike that interpreter it depends on and constructs no control flow graph in a forward direction, so it is “even more demand driven” than [Facchinetti et al. 2019].

The grammar of our language appears in Figure 5. This grammar is in $A$-normalized form (ANF) to clarify order of execution and to simplify the presentation of the theory below. Expressions $e$ in ANF are then just lists of clauses $c$. We now define notation and well-formedness conditions on this grammar.

**Definition 3.1** Notation and well-formedness.

1. We assume there is a fixed program $e_{glob}$ we are working over in this section.
2. We use notation $[x_1, \ldots, x_n]$ for lists and $||$ for list concatenation.
We require that $e_{\text{glob}}$ is closed and that each clause $c$ in $e_{\text{glob}}$ is uniquely identified by the variable it defines (i.e., our programs are alphatized).

For technical reasons, $e_{\text{glob}}$ cannot begin with a function definition clause. (W.o.l.o.g. the reader can assume there is a dummy $= \emptyset$ first clause.)

$C(x)$ is the unique $c$ where $c = (x=b)$ occurs in $e_{\text{glob}}$ (there is at most one such clause in $e_{\text{glob}}$ by alphatization).

Clauses $\text{fun } x \rightarrow$ may only be used as the initial clause in an $f$.

Clauses $x ! \beta$ may only appear at the beginning of a conditional expression’s branch and all conditional branches must start with such a clause. Variable $x$ must match the variable defined by the conditional; boolean value $\beta$ must match the condition (true or false) used to enter the branch.

Observe that functions $f$ are themselves just lists of clauses; for uniformity the entry of a function “$\text{fun } x \rightarrow$” is formally just a clause so functions are of the form $[\text{fun } x \rightarrow, c_1, \ldots, c_n]$. For readability we may use $\text{fun } x \rightarrow e$ as an abbreviation of $[\text{fun } x \rightarrow] || e$. This notion makes it easy to define a uniform notion of “predecessor line” used in the reverse program traversal of lookup. Here is the Figure 2 example in this formal grammar for illustration:

$$[o = 1, y = 0, f = [\text{fun } x \rightarrow, \text{fret} = x + o], fy = f y, f1 = f o, \text{ret} = fy + f1]$$

Similarly we add a clause $x ! \beta$ at the front of conditionals marking which branch was taken; this clause just serves to label the true vs false branches to know which branch we are at the start of. For example, the first conditional in Figure 4 would formally be the clauses

$$[bv = x = 0, \text{fret} = bv ? [\text{fret} ! \text{true}, \text{fretp} = x + 1] : [\text{fret} ! \text{false}, \text{fretm} = x - 1]]$$

Notice that in the informal examples we concluded each let with the variable returned by the final clause in the let, as this is the implicit semantics of ANF: the value of a list of ANF clauses is the last variable assigned in the list. Also, observe that constants are not inlined in ANF; they are predefined to reduce the number of cases in our definitions below, so we have added an initial clause to the program above defining 1.

Figure 5 also contains all of the constructs needed to define the reverse interpreter.

Note that there is no environment for looking up values; as was described in the overview, all variables are looked up by tracing back in the program to find their value.

The interpreter also needs inputs for test generation, as was motivated in Section 2.3.2. We model inputs as a mapping $i$ since the usual stream ordering is not compatible with a reverse-running interpreter; in Appendix C.1.3 of the Supplement we show how mapping and streaming versions are provably interconvertible, so we will use the mapping view to simplify our presentation here. The domain of $i$ is a pair $(C, x)$ which is more succinctly notated $C x$ - it is a variable plus a call stack to disambiguate which runtime version of the variable is being referenced$^3$.

\footnote{Lemma C.9 in the Appendix Supplement shows how this is a sufficient notion of freshening in a forward interpreter.}
3.1 Variable value lookup

Lookup was informally described in the Overview; here we make rigorous the intuitions described there. There must be some whole program expression \(e_{\text{glob}}\) being executed and fixed input mapping \(\iota\) defining the program inputs; we will often make these implicit parameters on relations as they are globally fixed. Lookup also proceeds with respect to a current context stack \(C\) which corresponds to the runtime call stack. The context stack is used to align calls and returns to rule out cases of looking up a variable based on a non-sensical call stack. Step (4) in the example of Section 2.1 for example shows how the context stack, there \([fy]\), is used for this purpose in lookup.

Lookup also proceeds with respect to a lookup stack \(X\). The topmost variable of this stack is the variable currently being looked up. The rest of the stack is used to remember non-local variable(s) we are in the process of looking up while searching for the lexically enclosing context where they were defined. Section 2.2 described how the non-locals stack may be used to search for non-local variable values.

Lookup of a variable value proceeds by “walking backwards” through the program. To accomplish this, we define a notion of a syntactic predecessor: all clauses which are not the start of an expression have a predecessor. We define a partial function \(\text{Pred}\) to formalize this concept and provide other definitions to assist in the formalization.

**Definition 3.2 Clause operations.**

1. \(\text{Pred}(c) = c’\) iff \([\ldots, c’, c, \ldots]\) occurs in \(e_{\text{glob}}\) (to find the syntactic predecessor of a clause)
2. \(\text{Pred}(x) = \text{Pred}(\text{Cl}(x))\)
3. \(\text{RetCl}([c_1, \ldots, c_n]) = c_n\) (to extract the return (last) clause of a function or conditional body)

Note that \(\text{Pred}(c)\) is partial and invertible: some clauses (like the first clause in the program) do not have predecessors and every clause can be the predecessor to at most one other clause.

Lookup finds the value of a variable starting from a given program point. In the context of a fixed program \(e_{\text{glob}}\) and input mapping \(\iota\), we write \(\mathcal{L}(X, c, C) \equiv v\) to denote that lookup using lookup stack \(X\) relative to program point \(c\) with context \(C\) returns \(v\) as a result. For instance, a lookup of variable \(x\) from program point \(c\) with empty context returning \(v\) would be written \(\mathcal{L}([x], c, []) \equiv v\). Note that this refers to looking for values of \(x\) starting at that program line – the definition could be in the line we start on. In Section 2 we informally used program line numbers in place of clauses \(c\); otherwise the lookups in that section directly correspond with the formalization here.

**Definition 3.3.** Given fixed program \(e_{\text{glob}}\) and input mapping \(\iota\), the relation \(\mathcal{L}(X, c, C) \equiv v\) holds iff there is a proof using the rules of Figure 6.

\(\text{First}(x, c, C)\) used in the Figure holds iff we can look up the very first variable in the program from the current clause/stack: \(x \neq \text{FirstV}(e_{\text{glob}})\) implies \(\exists v’. \mathcal{L}([\text{FirstV}(e_{\text{glob}})], \text{Pred}(c), C) \equiv v’ \land x \equiv \text{FirstV}(e_{\text{glob}})\) implies \(C = []\).

Since the call stack forces calls and returns to align it is not difficult to show the interpreter is deterministic.

**Lemma 3.4.** \(\mathcal{L}(X, c, C) \equiv v\) is deterministic: given fixed \(e_{\text{glob}}, \iota\) and \(X, c, C\) there is at most one \(v\) such that a proof can be constructed.

The proof appears in Appendix A of the Supplement.

Given the determinism of the lookup relation, we can overload lookup as a partial function: \(\mathcal{L}(X, c, C) = v\) if and only if relation \(\mathcal{L}(X, c, C) \equiv v\) holds.

The intuitions for Definition 3.3 were given in examples in Figures 2 and 3 in Section 2.1. There informally we used line numbers, and formally the clause in that line is used; additionally, for context stack elements in the overview we used the defining variable of the clause and here we
use the full clause. For example, the informal lookup \( \mathbb{L}([x], 3, [f y]) \) from the overview is formally \( \mathbb{L}([x], (\text{fun } x \to), [(f y = f y)]) \).

Before tracing through an overview example, here are a few high-level points about the rules. **FUNCTION EXIT** is the rule to "back into a function", transitioning from a call site to the last clause in the called function. And, the two **FUNCTION ENTER** rules are transition out of the function body from the front; one rule is for the case we were searching for the function parameter, and the other not - **PARAMETER** vs **NON LOCAL**. Note that the **FUNCTION ENTER NON-LOCAL** rule pushes \( x_f \) on to the lookup stack to implement the non-local lookup strategy: first find the function, then resume looking for the variable.

The **CONDITIONAL rules** are similar in several ways to the **FUNCTION rules**: **CONDITIONAL TOP** transitions out of the front of the conditional, and **CONDITIONAL BOTTOM** transitions into the last clause in one of the two branches. This latter rule shows the purpose of the \( x_1 ! \beta \) clause: it serves to mark which branch of the conditional we are about to exit out the front of. In both rules we

---

**Value Discovery**

| Rule | Description |
|------|-------------|
| \( \text{First}(x, C_l(x), C) \) | \( \mathbb{L}([x], (x = v), C) \equiv v \) |
| \( \text{Input} \) | \( \mathbb{L}([x], (x = \text{input}), C) \equiv v \) |

**Value Discard**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}(x, \text{Pred}(x), C) \equiv v \) | \( \mathbb{L}([x] \parallel X, (x = f), C) \equiv v \) |
| \( \text{Alias} \) | \( \mathbb{L}([x] \parallel X, (x = x'), C) \equiv v \) |

**Binop**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x'], \text{Pred}(x), C) \equiv v' \) | \( \mathbb{L}([x''], \text{Pred}(x), C) \equiv v'' \) |
| \( \mathbb{L}([x], (x = x' \odot x''), C) \equiv v' \odot v'' \) |

**Function Enter**

| Rule | Description |
|------|-------------|
| **Parameter** | \( c = (x_v = x_f \ x_v) \) |
| \( \mathbb{L}([x_v] \parallel X, \text{Pred}(c), C) \equiv v \) | \( \mathbb{L}([x_f], \text{Pred}(c), C) \equiv [\text{fun } x \to] \parallel e \) |
| \( \mathbb{L}([x] \parallel X, (\text{fun } x \to), [c] \parallel C) \equiv v \) |

**Function Enter Non-Local**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x_f, x] \parallel X, \text{Pred}(c), C) \equiv v \) | \( \mathbb{L}([x_f], \text{Pred}(c), C) \equiv [\text{fun } x'' \to] \parallel e \) |
| \( \mathbb{L}([x] \parallel X, (\text{fun } x'' \to), [c] \parallel C) \equiv v \) |

**Function Exit**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x'] \parallel X, (x' = b), [C_l(x)] \parallel C) \equiv v \) | \( \mathbb{L}([x_f], \text{Pred}(c), C) \equiv [\text{fun } x'' \to] \parallel e \) |
| \( \mathbb{L}([x] \parallel X, (x = x_f \ x_v), C) \equiv v \) |

**Skip**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x] \parallel X, \text{Pred}(x''), C) \equiv v \) | \( \exists v_0. \mathbb{L}([x''], C_l(x''), C) \equiv v_0 \) |
| \( \mathbb{L}([x] \parallel X, (x' = b), C) \equiv v \) |

**Conditional Top**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x_2], \text{Pred}(x_1), C) \equiv \beta \) | \( \mathbb{L}(X, \text{Pred}(x_1), C) \equiv v \) |
| \( \mathbb{L}(X, (x_1 \ ! \beta), C) \equiv v \) |

**Conditional Bottom**

| Rule | Description |
|------|-------------|
| \( \mathbb{L}([x''] \parallel X, (x' = b), C) \equiv v \) | \( \mathbb{L}([x''], C_l(x''), C) \equiv v \) |
| \( \mathbb{L}([x_2], \text{Pred}(x_1), C) \equiv v \) |
| \( \mathbb{L}([x_1] \parallel X, (x_1 = x_2 \ ? \ e_{\text{true}} : e_{\text{false}}), C) \equiv v \) |

Fig. 6. Value Lookup Rules
verify that we are in the correct branch; this is only needed in CONDITIONAL Top for the case that
the lookup started in the middle of a branch as otherwise the condition was already verified before
entering the branch.

The VALUE DISCARD rule is the case where the function \( f \) was found; the rule then pops off the
search for the function, resuming the search for the variable. Values are finally found and returned
in the VALUE DISCOVERY rule. The FIRST conditions in VALUE DISCOVERY and INPUT address the
corner case where a value is reached but it could be that the current code is dead, i.e. is it not
accessible from the start of the program due to for example an infinite loop before this point. To
verify this is not the case, we require there is a path back to the program start via a lookup of the
first variable.

Let us trace the same example lookup of the overview through the formal definition; we will
use the stringent ANF syntax for Figure 2 given at the start of this section as that is what the rules
work over.

\( L([fy], (f1 = f \ 1), []) \): Apply the SKIP rule as \( fy \neq f \). This induces a lookup of \( fy \) from the
clause \( \text{Pred}(f1 = f \ 1) = (fy = f \ y) \). The last precondition of SKIP is needed to ensure that
the computation we skipped over (f1’s in this case) is not diverging; that is, there exists some
value that could have been looked up for \( f1 \). In many practical cases non-termination checks
are not needed so this can be skipped. For brevity we will not prove that precondition now,
but the implementation currently always performs this check.

\( L([fy], (fy = f \ y), []) \): Apply the FUNCTION EXIT rule. This induces a lookup on \( \text{fret} \) which
pushes this call site onto the call stack. In addition, we need to satisfy these preconditions:
(a) Look up \( f \) to ensure that it is indeed a function. By VALUE DISCOVERY we see it holds:

\( L([f], (f = [\text{fun} \ x ->, \ldots]), []) = (f = [\text{fun} \ x ->, \text{fret} = x + o]) \).

(b) Look up \( f \)'s return variable \( \text{fret} \), the last variable defined in the body:

\( \text{RetCl}([\text{fun} \ x ->, \text{fret} = x + o]) = (\text{fret} = x + o) \).

\( L([\text{fret}], (\text{fret} = x + o), [(fy = f \ y)]) \): Apply the BINOP rule, looking up both terms in
the addition clause.

We will first look up \( x \):

\( L([x], (\text{fun} \ x ->), [(fy = f \ y)]) \): Apply the FUNCTION ENTER PARAMETER rule. We are looking
for the variable \( x \), which is the parameter passed into our function, so we pop out of the
function, pop the call site \( fy = f \ y \) from the stack, and search for the formal parameter \( y \)
from \( \text{Pred}(fy = f \ y) = (f = \ldots) \).

\( L([y], (f = \ldots), []) \): Apply the SKIP rule.

\( L([y], (y = \emptyset), []) \): Apply the VALUE DISCOVERY rule to obtain \( \emptyset \) as the result for \( y \) and thus
for \( x \).

Now we look up \( o \) from the BINOP:

\( L([o], (\text{fun} \ x ->), [(fy = f \ y)]) \): Apply the FUNCTION ENTER NON-LOCAL rule. In this stringent
ANF version of the example, \( o \) is a non-local variable in the function, giving us an
opportunity to exercise this rule. For the next lookup we will need to push \( f \) onto the lookup
stack to first find the definition point of \( o \), and pop the current call site from the call stack
since we are exiting the function body.

\( L([f, o], (f = \ldots), []) \): Apply the VALUE DISCARD rule, since we are sitting right on \( f \)'s
definition. We can now pop \( f \) off the lookup stack and continue lookup for \( o \) at the predecessor
clause.

\( L([o], (o = \emptyset), []) \): Apply the SKIP rule.

\( L([o], (o = 1), []) \): Apply the VALUE DISCOVERY rule, returning 1.
At this point both preconditions of Binop have completed and the result there is known to be \(0 + 1 = 1\), and chaining back that means the above \(f_{re t}\) and \(f_{y}\) lookups are also 1.

The input and conditional rules were not covered in the above example but they are more interesting for the symbolic case so will be described in the next section.

### 3.2 Equivalence of Demand and a Forward Interpreter

We need to show that the above lookup definition does not deviate from what a standard operational semantics would produce: if it is to be the basis for a demand symbolic evaluator, it must be sound and complete with respect to a standard evaluator. A full proof of this equivalence is found in Appendix C in the Supplement. It is the most conceptually deep Lemma in the paper as it aligns forward- and reverse-running interpreters; a large number of invariants need to be added to align the two.

Additionally, Appendix C.1.3 justifies the mapping view of input \(\iota\) used here by showing an isomorphism between the standard stream-based and mapping-based inputs, as well as constructions to build one form from the other.

### 4 A SYMBOLIC DEMAND-DRIVEN EVALUATOR

In this section we modify the demand-driven interpreter of the previous section to produce DDSE, the symbolic demand-driven evaluator that is the goal of this paper. While the core structure of lookup is mostly unchanged, two key modifications are required. First, we must make the interpreter symbolic, allowing arbitrary ranges of input values to be searched simultaneously. Second, we must replace the absolute stacks \(C\) in the lookup process with relative stacks \(\hat{C}\); this supports variable lookups that start in the middle of the program without knowing how we may have arrived at that point. If the reverse interpreter of the previous section were to begin lookup in a function body with an empty call stack, it would never be able to pop out of that call and no lookup proof can be constructed; relative stacks soundly support pops in such cases.

We will use a global cache of constraints \(\Phi\) to simultaneously constrain all run-time variables in the program. In order to disambiguate different runtime versions of the same variable due to recursion, we index each variable by its call stack: variables in \(\Phi\) are pairs of the form \(\hat{C}x\) (where the relative stacks \(\hat{C}\) are explained below). Lookup paths are realizable only if the constraints of \(\Phi\) can be met; that is, \(\Phi\) must always be satisfiable for some variable assignment. The implementation uses an SMT solver to verify this condition.

In a symbolic interpreter, there is often no single path of execution. First, since we can start lookup mid-program, we may start deep inside an (unknown) call stack and, as the correct caller is not known, we must search through all potential callers. We address this issue by adding a parameter \(\Pi\) to lookup which is an oracle to consult for which calling sites to choose in a particular lookup. Second, conditionals could have both true and false branches satisfiable if we had not (yet) accumulated any constraints in \(\Phi\) to indicate otherwise; as with the call site exploration, both branches must be tried one at a time. These two points of non-determinism capture the unknown control flows which may lead to the program point at which our lookup started; when they are fixed, the symbolic lookup process becomes deterministic.

#### 4.1 Relative Stacks

Since our symbolic interpreter does not know the program stack at the time that lookup starts, it uses a relative stack \(\hat{C}\) to characterize the stack state. At the beginning of symbolic lookup, the stack is completely unknown; if we start within a function body, for instance, we do not know from where that function was called. As we move backward through the program using decisions
in the aforementioned $\Pi$, we also \textit{retrospectively} learn what the stack was when lookup began. The relative stack $\hat{C}$ is thus a pair (co-stack, concrete stack) written $C?C$: the co-stack represents those frames which have been \textit{popped} since we started a lookup from inside a function body, and the concrete stack represents those frames which have been pushed since we began lookup. Once execution has reached the start of the program we can in retrospect transform all relative stacks $\hat{C}$ into absolute stacks $C$.

Before giving the formal definitions we provide some motivation. Consider the example of Figure 4 where we wanted to find inputs exercising line 7. When lookup begins the relative stack will be $C = [][]$: we have not yet exited or entered any calls. Tracing back as with the interpreter rules we will arrive at the start of $f$ looking for parameter $x$ but there are two call sites for the function: in line 13 and in line 14. So, $\Pi$ will contain one of those two as its arbitrary choice, say line 13. When the search for $x$ turns into a search for the parameter value $i1$ there, the relative stack will now be $[(f i1 = f i1)][]$ since that call site has been popped upon exiting $f$. We are now at the top of the program, and when we learn that fact we can in retrospect convert relative stack $[][]$ that we started lookup with to absolute stack $[(f i1 = f i1)]$, as the pops to get from line 7 in reverse to the start of the program must be the same as the pushes in a forward run to get from the front of the program to line 7.

The general case is more complex as we may have exited multiple functions creating a longer co-stack by the time we arrive at the program start, and we will need to reverse this top-level co-stack to arrive at the absolute stack. Additionally, in the reverse search we may have not just exited functions, we may have also entered new calls in the reverse walk; those are placed in the concrete stack and are treated as concrete push-pop operations.

4.2 Notation for Symbolic Lookup

The new notation needed for symbolic lookup is summarized in Figure 7. Along with the $\Phi$ grammar we define the relative stack grammar as described above. The formal definitions of operations on these stacks are as follows.

\begin{definition} Notation for pushing, popping, and concretizing relative stacks is as follows.
\begin{enumerate}
  \item $\text{Push}(c_1, \ldots, c_n)[c_1', \ldots, c_{n'}], c) = [c_1, \ldots, c_n][c, c_1', \ldots, c_{n'}],$
  \item $\text{Pop}(c_1, \ldots, c_n)[c], c) = [c, c_1, \ldots, c_n][].$
  \item $\text{Pop}(c_1, \ldots, c_n)[c_1', \ldots, c_{n'}], c) = [c_1, \ldots, c_n][c_2, \ldots, c_{n'}]$ for $c = c_1'$.
  \item $[c_1, \ldots, c_n][c_1', \ldots, c_{n'}]$ is empty iff $n' = 0$ (the stack is empty, the co-stack may not be).
  \item $\text{Concretize}(C[][]) = \text{Reverse}(C)$
\end{enumerate}
\end{definition}

The definitions are straightforward when the purpose is kept in mind: if there are callsites on the concrete stack and it is time to pop, pop the callsite on top of the concrete stack. Only if there are no concrete callsites does a pop add a frame to the co-stack. Pop is undefined if the concrete stack is non-empty but $c$ is not the top of the stack.

Function $\text{Concretize}(C[][])$ is used when the lookup search reaches the top level with stack $C[][$, this function extracts the actual stack that the program point the search started on. Since the concrete stack and co-stack grow oppositely, the co-stack at the top needs to be reversed to
obtain the stack at the start point. We use the atomic constraint stack = C of Figure 7 to record this top-level inferred stack. There will only be at most one such constraint present in Φ.

Above, we described how non-determinism in the lookup function needs to be addressed and how an additional lookup parameter Π is used to represent the call site choices made. Concretely, in the grammar Π ranges over mappings from relative stacks ̂C to function call sites c = (xf = xf x0).

Mapping key ̂C is the current relative stack and the mapped call site is the one which should be stepped (backwards) to by the lookup function under this call stack. The lookup function takes a Π oracle map as parameter; our implementation will need to search through the space of potential Π mappings. Since in large programs it is usually the case that the space of call sites paired with what function called there is sparse (most call sites usually call only a very limited number of functions in the whole program), the implementation in fact optimizes Π search by using an initial analysis pass to remove many provably-invalid cases.

Formulae Φ of Figure 7 are similarly oracular in the specification, and the implementation for them is similarly oracular in the specification, and the implementation for them is similarly oracular. We use the following notation for formulae and their properties:

Definition 4.2. We use the following notation for formulae and their properties:

1. Φ = ϕ1 ∧ · · · ∧ ϕn in some contexts is punned as the set of its atomic conjunctions, Φ = {ϕ1, . . . , ϕn}.
2. isSAT(Φ) holds if there is a satisfying assignment for Φ.
3. SATs(Φ) is the set of all satisfying assignments M that map (annotated) variables in Φ to values v.

Inputs need no special handling, they can simply be recorded by constraints on the input variable in Φ. So, the ι of the interpreter is subsumed by Φ here.

4.3 Variable Lookup Defined

We are now ready to define symbolic variable lookup.

Definition 4.3. For implicit fixed program εglob, Φ with isSAT(Φ) holding, and path mapping Π, DDSE variable lookup, L̂(X, Φ, Π, c, ̂C) ≡ ̂C x holds iff there is a proof using the rules of Figure 8. Since Φ and Π are fixed in most places we take them as implicit parameters in the Figure and elsewhere, writing the equivalent shorthand L̂(X, c, ̂C) ≡ ̂C x. In this Figure, we use a few additional notational abbreviations:

- FirstV([x1 = b1, . . . ]) = x1 extracts the first variable defined in the program εglob = [x1 = b1, . . . ].
- L̂s(X, c, ̂C) ≡ v abbreviates Φ̂C x0. L̂s(X, c, ̂C) ≡ ̂C ̂C x0 ∧ (̂C x0 = v) ∈ Φ
- L̂s(X, c, ̂C) ≡ _ abbreviates L̂s(X, c, ̂C) ≡ ̂C ̂C x0 for some ̂C x0.
- FirstS(x, c, ̂C) holds iff x ̂≠ FirstV(εglob) implies L̂s([FirstV(εglob)], Pred(c), ̂C) ≡ _

Understanding The Symbolic Lookup Rules. The lookup rules of the reverse interpreter in Figure 6 closely mirror the symbolic rules of Figure 8, and the reader should look through those rules and their detailed explanations before tackling the symbolic lookup rules. There are several key differences which we now outline.

Instead of returning a value v as the result we return the defining variable of the value, allowing us to return symbolic constraints in Φ instead of concrete values. For example, the Value Discovery rule directly returns the defining variable x of (x = v) paired with the stack, ̂C x. Note that some of the lookup assertions have non-variables on the right, but that is just a notational shorthand, described in Definition 4.3. This defining variable may then be used by other rules such as the Binop rule, which invokes lookup on both operator parameters and uses the defining variables to
Fig. 8. Symbolic Lookup Rules

build the equation constraining the binary operator behavior in Φ. By using stack-indexed variable definitions \( \hat{c}_x \) in the Φ constraints, we have a guarantee that there are no collisions of different activations of the same variable; they are relative stacks in the rules but upon completion of lookup can be converted to absolute stacks as we describe below.
The Function Enter . . . rules are extended to support the case that a search is started lexically within a function body. In this case Π is consulted for the choice to make and a call site frame is added to the co-stack by the Pop. The rules contain an additional precondition to look up the function definition variable \( x_f \) which is not present in the reverse interpreter; there the function was already looked up by Function Exit and so it is unnecessary to look up the function again.

The Conditional Bottom rule could match both branches in the implementation, and so it may be necessary to search through both possibilities. Here in the specification we assume the answer was already wired into Φ, similar to how Π has pre-wired the single possibility for the function call site choice. The requirement that Φ be satisfiable precludes both rules from firing.

All rules also use the relative stack push/pop operations of Definition 4.1 in place of the (absolute) stack operations of the interpreter. The Value Discovery rule when on the first clause of the program will require the constraint (\( \text{stack} = \text{Concretize}(\tilde{C}) \)) be in Φ to record the absolute stack for subsequent stack normalization.

The Input rule adds only the constraint \( \varsigma_{\text{true}} = (\tilde{C}x = \tilde{c}x) \) which implicitly constrains the input variable to be an integer. (Equality in the language is defined only on integers.) Otherwise there are no constraints added since the goal is to find inputs; as with Value Discovery the stack also needs to be recorded if this is the first line in the program.

Section 2.3 informally traces some examples through this definition; with the formal definition the results of that example can be confirmed. Note that the informal notation \( \tilde{L}^{x}(\{x\}, n, C) \equiv Cx' \) used there abbreviates \( \tilde{L}^{x}(\{x\}, c, C) \equiv Cx' \) for line \( n \) containing program clause \( c \). We also did not use relative stacks in the examples for simplicity, but we outlined the relative stack version for Figure 7 lookup in Section 4.1 above.

4.4 Defining Test Generation and Showing Computability

This section uses the symbolic lookup definition to formally define a function \( T \) which generates a test input exercising a particular clause in a program. We show \( T \) to be partially recursive and complete: if an input sequence exists which will test a particular line of code, \( T \) will find it.

We begin by observing the determinism of symbolic lookup, in analogy to the determinism of the reverse interpreter. To be clear, test generation is non-deterministic as the input choices in Φ and branch choices in Π are not fixed, but for fixed Φ/Π there can be only one result.

**Lemma 4.4.** \( \tilde{L}^{x}(\{x\}, \Phi, \Pi, c, []?[]) \equiv \tilde{C}x_0 \) is deterministic: given \( e_{\text{glob}}, x, c, \Pi \) and satisfiable \( \Phi \), there is at most one \( \tilde{C}x_0 \) such that a proof can be constructed.

This and all subsequent proofs in this section are found in Appendix B of the Supplement.

The goal of test generation is to find inputs exercising a particular clause in the program.

**Definition 4.5 Test Generation Predicate.** Given fixed expression \( e_{\text{glob}} \) and a clause \( c \) in \( e_{\text{glob}} \),

- \( T(e_{\text{glob}}, c, \Phi, \Pi) \) holds if \( \tilde{L}^{x}(\{\text{FirstV}(e_{\text{glob}})\}, \Phi, \Pi, c, []?[]) \equiv \_\_ \).
- \( T(e_{\text{glob}}, c) \) holds iff \( \tilde{T}(e_{\text{glob}}, c, \Phi, \Pi) \) for some \( \Phi, \Pi \).

The above definition looks up the first variable in the program, guaranteeing that we do not prematurely stop our reverse lookup somewhere in the middle, and in fact be in dead code or past a termination point.

The above definition does not produce program inputs \( t \) directly. However, an \( t \) may always be constructed from \( \Phi \); in particular, an SMT solver can produce such an input mapping. Formally, we say \( t \) satisfies the constraints \( \Phi \) if \( \text{isSAT}(\Phi \cup \{ \tilde{x} = v \mid \tilde{x} \mapsto v \in t \}) \), and all input variables (identified by constraints \( \varsigma_{\text{true}} = (\tilde{C}x = \tilde{c}x) \) in Φ) are mapped by \( t \). We first observe that, for any consistent Φ, such a mapping always exists.
Lemma 4.6. Given expression \( e_{\text{glob}} \) and clause \( c \), if \( \mathcal{T}(e_{\text{glob}}, c, \Phi, \Pi) \) for some \( \Phi, \Pi \) then there exists an \( i \) such that \( i \) satisfies \( \Phi \).

Searching for such a mapping is not decidable, but it is recursively enumerable:

Lemma 4.7. Given \( e_{\text{glob}} \) and \( c \), finding a \( \Phi, \Pi \) for which \( \mathcal{T}(e_{\text{glob}}, c, \Phi, \Pi) \) holds is recursively enumerable.

Lemma 4.7 demonstrates that, if \( \mathcal{T}(e_{\text{glob}}, c) \), then we can eventually find a suitable input sequence. But, the enumeration strategy used in the Lemma is not at all practical. Fortunately, in practice we can incrementally accumulate constraints during lookup and only need to perform a nondeterministic search for the cases where a function may have had multiple callers (we can use a program analysis to rule out nearly all cases), and where a conditional clause may be either true or false. Our implementation is described in the following section.

4.5 Correctness

Here we show that DDSE is fully and faithfully modeling the demand interpreter from the previous section (which was itself shown to be equivalent to a forward interpreter; see Section 3.2).

Before getting into the meat of the proof we prove a small auxiliary Lemma showing how a unique \( \text{stack} = C \) constraint must always show up in any successful symbolic lookup of the first program variable.

Lemma 4.8. If \( \mathcal{L}([\text{FirstV}(e_{\text{glob}})], \Phi, \Pi, c_0, \hat{C}_0) \equiv \hat{C}x \) then there is exactly one constraint of the form \( \text{stack} = C \) in \( \Phi \) for some \( C \).

4.5.1 Absolutizing the Relative Stack. In our first step toward showing correctness, we will replace the relative stacks used by symbolic lookup with absolute stacks to align them with the stacks in the demand interpreter. We formalize the notation \( |\hat{C}|_C = C_a \) to mean the normalization of a relative stack \( \hat{C} \) with respect to \( C \), the call stack at the program point where we start lookup (which we only learn in retrospect), has absolute equivalent \( C_a \).

Definition 4.9. Relative stacks, and variables and formulae so-labeled are absolutized as follows.

- \( |C_c ? C_s|_C \equiv C_s || C' \) where \( C = \text{Reverse}(C_c) || C' \) for some \( C' \); this operator is undefined if the equation fails for all \( C' \).
- \( |\hat{C}x|_C = |C|_C x \).
- \( |X|_C = \{|\hat{C}x|_C | \hat{C}x \in X\} \) and
- \( |\Phi|_C = \Phi[\{X|_C/X\}], \) for \( X \) being the set of all variables in \( \Phi \), and in addition replacing constraint \( \text{(stack} = C \text{)} \) in \( \Phi \) with (stack = []).

For example, \( [[a]?[b]]_{[a,c]} = [b, c] \): relative stack \( [a]?[b] \) is the state of exiting \( a \) and then entering \( b \), and once we learn in retrospect that we started lookup with call stack \( [a, c] \), we see that the state is the stack \( [b, c] \). Using this operation, we can replace all relative stacks with absolute stacks and replay the lookup isomorphically. We also establish the converse to help show completeness.

Lemma 4.10 Equivalence of relative and absolute stacks.

1. If \( \mathcal{L}^{\hat{C}}(X, \Phi, \Pi, c_0, \hat{C}_0) \equiv \hat{C}x \) and \( \text{(stack} = C \text{)} \in \Phi \) then \( |\hat{C}|_C \) and \( |\hat{C}_0|_C \) are defined and \( \mathcal{L}^{\hat{C}}(X, |\Phi|_C, \Pi, c_0, [??]|\hat{C}_0|_C) \equiv |\hat{C}|_C x \).
2. If \( \mathcal{L}^{\hat{C}}([\text{FirstV}(e_{\text{glob}})], \Phi, \Pi, c_0, [??]|C) \equiv |\hat{C}|_C x \) then \( \mathcal{L}^{\hat{C}}([\text{FirstV}(e_{\text{glob}})], \Phi', \Pi, c_0, [??]) \equiv \hat{C}x \) for some \( \hat{C} \) with \( C' = |\hat{C}|_C \), \( \text{(stack} = C \text{)} \in \Phi' \), and \( \Phi = |\Phi'|_C \).
5 IMPLEMENTATION

In this section we describe the reference implementation of DDSE. This implementation closely follows the specification, and is primarily designed to confirm correctness of the specification; while it includes some optimizations, many more are needed for good performance. First we describe the implementation and then we describe its performance on example programs. The language implemented is the grammar of Figure 5.

5.1 The OCaml Implementation of DDSE

Definition 4.3 gives the lookup function which symbolically executes a program in a demand-driven fashion. This definition is declarative, treating the logical formulae $\Phi$ and the search path $\Pi$ as oracular, to improve readability. For a feasible implementation, however, we must construct $\Phi$ and $\Pi$ as we search for control flows which satisfy the lookup rules. Designing this algorithm presented three key challenges — nondeterminism, nontermination, and caching — which we now discuss.

Our implementation searches for paths through a program to a desired destination by moving backward and applying the rules of Definition 4.3. This naturally gives rise to a nondeterministic
algorithm: if multiple rules or uses of a rule apply, each of those choices is attempted. As choices are made, incoherent universes (e.g. with $\Phi$ containing unsatisfiable formulae) are discarded. This application of nondeterminism is common in proof searching algorithms, transitive closures, and similar domains; however, nondeterminism is non-trivial to combine with other features of computation [Zwart and Marsden 2018].

One example of this poor interaction is with non-terminating computations. As stated in Section 4.4, lookup is recursively enumerable (Lemma 4.7) but not decidable. Thus, traditional implementations of nondeterminism (such as the concatMap approach used in Haskell’s list monad) do not enumerate correctly: even if one thread of nondeterministic computation fails to terminate, we want other threads to be productive. We address this by modeling computations as promises, routinely yielding control. This allows our implementation to explore the tree of nondeterministic computations in a breadth-first fashion via a simple continuation-based worklist algorithm. We use a simple priority function on the worklist: we prioritize the shortest relative stacks with the most unique frames (fewest recursions).

Finally, most lookup rules include multiple child lookups and the descendants of those children often overlap. In the Skip Rule, for instance, the lookup of $x''$ and $x$ may share sub-lookups. This recursion is exponential akin to naive Fibonacci algorithms and can be resolved the same way: via caching. Here, we must contend with caching nondeterministic and potentially nonterminating computations. Our implementation introduces a publish/subscribe messaging model: a cached computation publishes its results as the worklist algorithm produces them, while computations relying upon the cache consume value messages to produce a promise of future work. This publish/subscribe messaging model is global to the evaluation — that is, cached computations do not recursively maintain their own caches — ensuring that cached values are shared between all subordinate lookups regardless of where they appear in the computational tree. Simpler caching models, such as associating each nondeterministic computation with its own cache, proved in practice to be little different from no caching at all.

We developed the implementation with OCaml 4.09.0 using Z3 4.8.1 to check formulae. The symbolic interpreter is implemented in monadic style: one module defines a monad addressing the above challenges while another implements each rule of Definition 4.3 as straight-line imperative code. This design permits additional language features to be supported with minimal effort.

Source code and instructions on how to build the implementation and run the tests and benchmarks are included with the supplementary software artifact [Palmer et al. 2020a].

5.2 Methodology

We have performed a preliminary evaluation of our reference implementation of the test generator. Since there is no existing benchmark suite for general higher-order symbolic execution, we modified some of the model-finding benchmarks from SMBC\(^4\), as well as standard Scheme benchmarks from Larceny and P4F\(^5\), and additionally made some benchmarks of our own which allow for arbitrary-sized inputs. The SMBC benchmarks are for SMT model finding with total recursive functions and declared datatypes [Cruanes 2017]. Larceny is a standard set of Scheme benchmarks. The P4F benchmarks are for higher-order program analysis.

The SMBC benchmarks are SMT programs in SMT-LIB syntax [Barrett et al. 2017]. The original programs consist of function and datatype definitions, assertions and a goal. We can directly translate the functions to our syntax. We put the assertions and the goal in an expression if

\(^4\)From [https://cedeela.fr/~simon/files/cade_17.tar.gz](https://cedeela.fr/~simon/files/cade_17.tar.gz) and [https://github.com/c-cube/smbc](https://github.com/c-cube/smbc).

\(^5\)From [http://www.larcenists.org/benchmarks/AboutR7.html](http://www.larcenists.org/benchmarks/AboutR7.html) and [https://github.com/adamsmd/paper-push-down-for-free-prototype/tree/master/benchmarks](https://github.com/adamsmd/paper-push-down-for-free-prototype/tree/master/benchmarks).
(assertions and goal) then target else nothing at the end of our program and start test generation from target. We don’t directly support datatype declarations or solve for uninterpreted functions, so those features of their benchmarks we encode by writing helper functions which generate them. For instance, to generate an arbitrary list of integers, we use the code

```ocaml
let rec gen_list () = let t = input in if 0 == t then nil else t::(gen_list ());
```

to express an uninterpreted function on booleans, we can use

```ocaml
let bt = input in let bf = input in fun x -> if x then (bt == 0) else (bf == 0)
```

Using such encodings, we can convert the original SMT programs with unknown terms to closed programs with an input-dependent goal.

For an example of this adaptation process, Figure 9(a) shows our adaption of the facehugger.scm benchmark. For the Scheme benchmark, the (OCaml equivalent of the) last expression is \((\text{id } f) 3) + ((\text{id } g) 4)\) and the evaluated result is 30. To construct an interesting symbolic test generation benchmark, we change the argument 4 here to be an input \(y\) at the beginning of the program. We then add a final condition checking whether the sum equals 30 and target a test to reach the then-clause. Our test generator in fact automatically infers an input, 4, which matches the argument of the original text. So, we have transformed an existing Scheme benchmark into a benchmark for symbolic test generation. We apply a similar methodology to all the Scheme benchmarks adapted for our evaluation.

The formal ANF syntax of Figure 5 is difficult to code in, so we also implemented a translator which allows direct coding in an ML-like syntax; this syntax was used in the Section 2 examples. Recursive functions or loops appearing in the original benchmarks are encoded via self-application in the ANF. Lists are encoded as records, for example

\[
\{\text{last } = \text{false}, \text{elem } = 1, \text{next } = \{\text{last } = \text{false}, \text{elem } = 2, \text{next } = \{\text{last } = \text{true}}\}\}
\]

We use ML-like syntax for lists in the figures for clarity.

### 5.3 Evaluation

#### 5.3.1 Performance on Simple Benchmarks.

We now briefly describe the existing SMBC and Scheme benchmarks and how we modified them to make test generation benchmarks. The benchmarks are from the Scheme examples unless otherwise noted below.

- **fold** The original SMBC benchmark synthesizes an accumulator function when folding a list, that can distinguish between two lists of booleans with one different element. We generate this function from the input.

- **palindrome** The original SMBC benchmark looks for a list which is palindrome and satisfies the constraints on the length and the sum of elements. We add an unbound recursive function to generate one list from the input before checking for those constraints.

- **pigeon** This SMBC benchmark encodes the classic pigeon hole problem. We choose 4 holes for 5 pigeons and specify a program point to reach if no solution found.

- **sorted** This SMBC benchmark finds a list of natural numbers which is sorted and satisfies the constraints on the length and the sum of elements. We use the similar methods in palindrome.

- **blur** This benchmark combines non-local function definitions and recursion. We replace a constant call \((lp \text{ false } 2)\) with \((lp \text{ false } x)\) where \(x\) is an added input that the test generator must infer to get the correct answer.

- **eta** Tests spurious function calls that do not affect the lookup subject. We simply check whether the benchmark runs properly with symbolic execution.

- **facehugger** The modifications to this benchmark appear in Figure 9(a) and were described above.
let y = input in
let rec list_in acc =
  let x = input in
  if 0 == x then acc
  else list_in (x :: acc)
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in
let rec list_build acc =
  let x = input in
  if 0 == x then acc
  else list_build (x :: acc)
in
let rec f n =
  if n <= 1 then 1
  else n * (f (n-1))
in
let rec g n =
  if n <= 1 then 1
  else n * (g (n-1))
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in
let rec list_build acc =
  let x = input in
  if 0 == x then acc
  else list_build (x :: acc)
in
let rec f n =
  if n <= 1 then 1
  else n * (f (n-1))
in
let rec g n =
  if n <= 1 then 1
  else n * (g (n-1))
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in
let rec list_build acc =
  let x = input in
  if 0 == x then acc
  else list_build (x :: acc)
in
let rec f n =
  if n <= 1 then 1
  else n * (f (n-1))
in
let rec g n =
  if n <= 1 then 1
  else n * (g (n-1))
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in
let rec list_build acc =
  let x = input in
  if 0 == x then acc
  else list_build (x :: acc)
in
let rec f n =
  if n <= 1 then 1
  else n * (f (n-1))
in
let rec g n =
  if n <= 1 then 1
  else n * (g (n-1))
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in
let rec list_build acc =
  let x = input in
  if 0 == x then acc
  else list_build (x :: acc)
in
let rec f n =
  if n <= 1 then 1
  else n * (f (n-1))
in
let rec g n =
  if n <= 1 then 1
  else n * (g (n-1))
in
let rec list_map f l =
  let rec lp lst =
    match lst with
    | [] -> []
    | x::xs -> (f x)::(lp xs)
in
let rec sum_list l =
  match lst with
  | [] -> 0
  | x::xs -> x + (sum_list xs)
in

(a) Facehugger function
(b) List build, map and sum

Fig. 9. Benchmark Source Code Examples

flatten The original test flattens a fixed deeply nested list. We change it to test double nested lists, and replace the hard-coded list elements by input values and check whether those inputs appear in the right points after flattening.

k-cfa-2, k-cfa-3 These benchmarks test worst-cases for k-CFA. We simply check whether the symbolic evaluator can get the original result.

map This Scheme benchmark tests the classic list map function on several example lists. We change some of the list elements to be inputs and constrain the mapped list elements to be equal to the result list returned by the Scheme benchmark, forcing the test generator to generate those constant elements.

mj09 Tests the alignments of calls and returns. We simply check whether symbolic execution can get the original result.

sat-1 A brute-force SAT solver on a hard-coded formula with four boolean variables. It recursively tries both assignments for each variable to see if the formula is satisfiable. We again simply verify whether symbolic execution correctly runs the benchmark.

sat-1-direct This is our own variation on sat-1 where we take formula variables from input and demand that the test generator find values for the variables satisfying the formula.

All of these benchmarks run successfully in our test generator, and generate satisfying inputs (if any). For each benchmark, Table 1 lists the elapsed time, taken as the average of three runs. The
total steps is an internal count of the number of coroutine monad binds and should only be viewed relatively – the vast majority of these binds are trivially discharged, allowing the code to be highly compositional. All of the times for these short programs are reasonably fast, in spite of the lack of optimizations in our current implementation. sat-1-direct is much faster than sat-1 because the satisfiability search is entirely within Z3 whereas sat-1 is a naive SAT solver checking cases one by one and needing a different control flow path for each. The fold benchmark also has to do a control flow space search as we have to encode an uninterpreted boolean function by conditioning on integer inputs; the SMBC system has native support for uninterpreted functions.

In the above examples we ask the test generator to find only one input stream reaching the target, but it is also possible to query for multiple input streams which also must use a different control flow path to reach the target.

5.3.2 Synthesizing Unbounded Inputs. To test the ability to synthesize tests in the presence of somewhat more complex higher-order functions and recursion with an unbounded number of inputs, we craft some longer examples based on List.map, a common functional idiom; see the code in Figure 9(b).

This program first defines a function that inputs a list of unbounded length. The list build finishes when 0 is input at line 4. list_map and sum_list work as their names suggest. After these function definitions, the main workflow is to build a list lst, to make an adder function adder, to map over it to get a new list lst2, to sum the new list s, and finally check whether we can get the desired sum to reach the specified program point target.

The first input y parameterizes how much adder adds, the last 0 is required to exit list_build, and the remaining input is the list lst. We fed this program into the test generator requesting four unique control flows reaching target line 28. We get the following results requiring 7.1 seconds to infer all four input sequences:

| Input generated       | Steps of symbolic evaluation |
|-----------------------|------------------------------|
| [9, 1, 0]             | 5878                         |
| [1, -1, 9, 0]         | 10793                        |
| [-1, 1, -1, 13, 0]    | 17947                        |
| [1, -1, -1, -1, 9, 10]| 27618                        |

For each control flow path taken, we ask the underlying SMT solver for (only) one solution to the constraints. To get a different control flow path we need a longer list, so in each run the lists generated get longer. Notice that the steps of symbolic evaluation needed increase nearly-linearly in the length of the list needed: the underlying algorithm is linear, and few spurious paths are taken since we are demand-driven.

5.3.3 Strengths of DDSE. The problem with a pure forward symbolic execution is if the wrong control flow path is taken early on, it will then perform a lot of wasted work and in general it can be just “shooting in the dark” to find the proper control flow. It is particularly bad for the common case of a branch inside of recursion as there will be an unbounded number of bad control flow choices to make. In such cases, demand-driven test generation can be much more efficient.

Table 1. DDSE performance on SMBC and Scheme benchmarks modified for test generation

| Benchmark | Time  | Steps  |
|-----------|-------|--------|
| fold      | 20.51s| 77587  |
| palindrome| 4.01s | 20585  |
| pigeon    | 0.55s | 8055   |
| sorted    | 2.46s | 18377  |
| blur      | 0.24s | 2880   |
| eta       | 0.04s | 268    |
| facehugger| 0.98s | 12226  |
| flatten   | 0.85s | 8157   |
| k-cfa-2   | 0.09s | 1537   |
| k-cfa-3   | 0.25s | 4308   |
| map       | 1.19s | 10631  |
| mj09      | 0.07s | 767    |
| sat-1     | 4.42s | 20213  |
| sat-1 direct | 0.07s | 861    |
For example, consider the example of Figure 10. For this program, pure forward symbolic execution doesn’t have any guidance on how many times the count loop should run in the two calls to count, and so \(ca\) and \(cb\) do not initially get constrained. However, the later code requires \(ca\) to be twice the value of \(cb\) and a forward symbolic interpreter must perform a search of all possible iterations for both loops until it luckily picks one that is twice the other, making many fruitless attempts before reaching this goal. DDSE on the other hand starts from the target point, which immediately gives it a constraint as to what previous control flow paths will work, and significantly shrinks the search space for control flows – DDSE successfully infers input stream \([1,1,1,1,1,1,0,1,1,1,0]\) with 69823 steps required. This code is a simplified example of common real-world cases which contain an assertion deep inside a program but which may have a massive number of forward paths to explore to reach it.

5.3.4 Weaknesses of DDSE. While our implementation performs caching of the lookup function, it is of the completely obvious sort: if the exact same arguments are seen again the cached result is used. But there are many lookups which are nearly identical but have slightly different arguments and those will all require separate lookups even in cases where they are provably the same. One obvious example is imagine the first line declares a constant \(k = 0\) and \(k\) is used throughout the program. Each lookup of \(k\) in a different call stack context will be a different lookup and so no caching will be possible in our primitive caching scheme. A more advanced caching methodology would be able to cache over schemas of call stacks. Additionally, since the source code is fixed, lookup can be partially evaluated for a given function body producing a table summary; this is called function summarization in the first-order symbolic analysis literature [Baldoni et al. 2018]. We aim to make a more real-world implementation addressing these and other performance issues.

While the goal-directed proof search strategy used here is generally considered superior to forward search due to the much greater number of spurious paths in a forward search, the problem we are trying to solve is undecidable and pure demand-driven searches can also get overwhelmed with cases. It can be useful to combine forward- and reverse-symbolic methods, a topic there has been some progress on in the first-order space and which we discuss next in the related work.

Overall, the test generator is performing correctly on all the benchmarks, showing that our symbolic evaluator is promising as a tool for goal-directed symbolic execution of higher-order programs.

6 RELATED WORK

Symbolic execution has been an active area of research for almost 50 years; we refer readers to a recent complete survey for broader background [Baldoni et al. 2018].

Our work lies under the umbrella of symbolic backwards execution (SBE) [Blackshear et al. 2013; Bourdoncle 1993; Chandra et al. 2009; Charreteur and Gotlieb 2010; Cousot et al. 2011; Dinges and Agha 2014; Manevich et al. 2004]. Our general philosophy is similar in principle to these works. In detail, however, there are many differences as we are addressing higher-order functional languages and these papers address imperative languages. We also have only a small toy language and implementation, but with rigorous semantics, an implementation that very closely follows the semantics, and correctness proofs of the semantics. (Note that there have been some correctness proofs for forward symbolic executions of C programs [Aissat et al. 2016], but not for higher-order
demand symbolic execution). CCBSE [Ma et al. 2011] is a forward evaluator which steps back incrementally from the target to try to "hit" it, giving it some character of the SBE school. Their Mix-CCBSE system combines forward symbolic execution with this partial-reverse strategy; it improves performance by combining the advantages. [Dinges and Agha 2014] combines SBE with concrete forward execution to narrow the search space. The idea of combining forward and backward would also likely benefit DDSE: a forward phase would eliminate a class of search paths by propagating some constraints forward which would preclude those paths from ever being considered in the backward phase.

Many of the issues and challenges of these systems we also share. The key problem in symbolic execution is the well-known path explosion problem [Anand et al. 2013, 2008]: the search space grows far too rapidly and the algorithm founders. We currently use a simple cache of the lookup function \( L \) to avoid repeated lookup, but it would not be hard to extend this to caching of whole families of lookups: in many cases parameters are fully or partly irrelevant. Some caching of function summaries is performed by Snugglebug [Chandra et al. 2009]. Snugglebug speeds up SMT queries by solving most of them internally rather than calling out to a solver; this is because the logic is nearly always very simple and an industrial SMT solver is overkill. Our DDSE artifact also performs simple on-the-fly SAT checks to eagerly catch obvious inconsistencies. All symbolic interpreters suffer when the logical assertions are beyond the capabilities of the solver; we share that weakness. These existing systems have many complex phases and heuristics; one advantage of DDSE is how the formal specification can fit on a page (Definition 4.3), and it is a direct generalization of a non-symbolic interpreter (Definition 3.3) to symbolic data.

The demand-driven interpreter which we symbolize here was inspired by the \( \omega \)DDPAc interpreter [Facchinetti et al. 2019; Palmer and Smith 2016], which was developed solely to show soundness of a demand-driven program analysis. The interpreter of this paper is significantly simplified from \( \omega \)DDPAc in that it does not require (forward) construction of a control flow graph, and for this reason it is purely demand-driven unlike \( \omega \)DDPAc. But the main contribution here beyond [Facchinetti et al. 2019] is to show how demand interpreters also can be turned into demand symbolic evaluators, producing the first such known system for a higher-order functional language. Beyond the very different natures of program analyses and symbolic evaluators, several particular features of DDSE were not found in previous work, including the modeling of input equivalently as a mapping; the use of stack-indexed variables in formulae to disambiguate across different activations; and, the novel relative stack construction for building a call stack starting from the middle of a program and working backward.

While we focus on functional code here, there is in principle no problem with extending these results to include side effects beyond input and non-termination. For mutation for example, a demand evaluator in this style finds the most recent assignment to the cell, verifying no aliases of it were skipped over [Facchinetti et al. 2019].

Automated test generation is a well-studied research topic with many complementary approaches; see [Anand et al. 2013] for a survey. Simple automated test generators such as QuickCheck [Claessen and Hughes 2000] are very useful but test coverage will often be incomplete: some lines of code will still have no test exercising them. Some variations allow the distribution of data to be altered [Lampropoulos et al. 2017] to improve coverage, but code structure is not taken into account; this ameliorates the incompleteness problem but does not solve it. In general, there is an infinite search space of possible inputs and, in practice, test generation algorithms will be incapable of reaching some program points. This is a consequence of path explosion and is a major problem in automated test generation. As mentioned above, SBE [Chandra et al. 2009; Dinges and Agha 2014; Ma et al. 2011] aims squarely at this issue, taking a goal-directed approach to deal with path explosion: paths
that would never lead to the goal line are not even initiated. DDSE aims to extend the SBE approach to functional languages.

Forward symbolic evaluators have been developed for extended functional languages, e.g. Rosette [Torlak and Bodik 2013] for Racket and Kaplan [Köksal et al. 2012] for Scala. Higher-order contract verification [Tobin-Hochstadt and Van Horn 2012] is a generalized form of higher-order forward symbolic execution. Contract verification is a step closer to program logic; one particular reason why this current study excites us is for the potential applications as a more general logic.

If functions are required to be total and data types are all declared, it is possible to take a more structural view to automated counterexample search in the presence of higher-order functions; one state-of-the-art forward symbolic evaluator in this area is [Cruanes 2017]. Their notion of conflict-driven clause learning (CDCL) allows earlier identification of useless search paths. Our approach is more general in that we have no requirement for data type declarations or for functions to be always terminating and we support unbounded inputs. These differences make our approach more directly applicable to mainstream programming languages; conversely, CDCL can take advantage of these restrictions to make the search more efficient. We showed in Section 5.3 how their benchmark examples are also successfully solved by our system.

In the first-order program space, incorrectness logic [O’Hearn 2019] shows how symbolic execution can be extended to a refutation logic, and by analogy our symbolic evaluator is “just an induction rule” away from being a higher-order refutation logic. Note that incorrectness logic supports both forward- and reverse reasoning, a generalization of forward-reverse symbolic execution [Dinges and Agha 2014; Ma et al. 2011], but it has no automated proof search methodology and so does not define a symbolic execution.

Dijkstra monads [Swamy et al. 2013] are in a related but different space to our work; they show how in functional programs with monadic effects there is a natural porting of \( wp \) logic on effect-ful code to the monadic presentation of that effect-ful code. They can then use \( wp \) propagation to generate verification conditions for semi-automatic verification of program properties. Our work is not focused not only on side effects ordering, but on also the order of operation inherent even in pure functional code.

7 CONCLUSIONS

Here we developed the theory and reference implementation of DDSE, a symbolic backwards executor (SBE) for higher-order functional programs. Unlike existing SBE’s, DDSE works on higher-order functional languages and is characterized as a direct symbolic generalization of a (non-symbolic, backward) interpreter. This places demand symbolic interpreters closer to forward symbolic interpreters, which are also direct generalizations of forward non-symbolic interpreters. We described initial results from a reference implementation.

This paper represents the initial effort in this direction; handling more language features and a more optimized implementation are key extensions needed. There are also several general fronts on which this approach could lead to new applications. Currently the test generation approach only generates tests for whole programs. By using type and data structure information it should also be able to generate tests for code fragments, to e.g. be used to generate unit tests. The underlying logic of DDSE lookup embodies a novel approach to reasoning about programs and may be useful as a program logic: its goal-directed nature naturally aligns with theorem provers.

ACKNOWLEDGEMENTS

We thank POPL and ICFP reviewers for comments which led to a significantly improved paper. We acknowledge Kelvin Qian and Leandro Facchinetti for helpful comments and suggestions.
REFERENCES

Romain Aissat, Frédéric Voisin, and Burkhart Wolff. 2016. Infeasible Paths Elimination by Symbolic Execution Techniques. In Interactive Theorem Proving, Jasmin Christian Blanchette and Stephan Merz (Eds.). Springer International Publishing, Cham, 36–51.

Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, J. Jenny Li, and Hong Zhu. 2013. An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software 86, 8 (2013), 1978 – 2001. https://doi.org/10.1016/j.jss.2013.02.061

Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. 2008. Demand-Driven Compositional Symbolic Execution. In Tools and Algorithms for the Construction and Analysis of Systems, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 367–381.

Roberto Baldoni, Emilio Coppa, Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi. 2018. A Survey of Symbolic Execution Techniques. ACM Comput. Surv. 51, 3, Article 50 (2018).

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. The SMT-LIB Standard: Version 2.6. Technical Report. Department of Computer Science, The University of Iowa. Available at www.smt-lib.org.

Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013). Association for Computing Machinery, New York, NY, USA, 275–286. https://doi.org/10.1145/2491956.2462186

François Bourdoncle. 1993. Abstract Debugging of Higher-order Imperative Languages. In Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI ’93). ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/150590.150595

Satish Chandra, Stephen J. Fink, and Manu Sridharan. 2009. Snugglebug: A Powerful Approach to Weakest Preconditions. In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’09). ACM, New York, NY, USA, 363–374. https://doi.org/10.1145/1542476.1542517

Florence Charrette and Arnaud Gotlieb. 2010. Constraint-Based Test Input Generation for Java Bytecode. In 2010 IEEE 21st International Symposium on Software Reliability Engineering.

Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In Proceedings of the Fifth ACM SIGPLAN Conference on Functional Programming (ICFP ’00). ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

Patrick Cousot, Radhia Cousot, and Francesco Logozzo. 2011. Precondition Inference from Intermittent Assertions and Application to Contracts on Collections. In Verification, Model Checking, and Abstract Interpretation, Ranjit Jhala and David Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 150–168.

Simon Cruanes. 2017. Satisfiability Modulo Bounded Checking. In Automated Deduction – CADE 26, Leonardo de Moura (Ed.). Springer International Publishing, Cham, 114–129.

Peter Dinges and Gul Agha. 2014. Targeted Test Input Generation Using Symbolic-concrete Backward Execution. In Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE ’14). ACM, New York, NY, USA, 31–36. https://doi.org/10.1145/2642937.2642951

Leandro Facchinetti, Zachary Palmer, and Scott Smith. 2019. Higher-Order Demand-Driven Program Analysis. TOPLAS 41 (July 2019). Issue 3. https://doi.org/10.1145/3310340

Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. 2019. Demand Control-Flow Analysis. In VMCAI (Lecture Notes in Computer Science), Constantin Enea and Ruzica Piskac (Eds.), Vol. 11388. Springer, 226–246. http://dblp.uni-trier.de/db/conf/vmcai/vmcai2019.html#GermaneM0M19

Susan Horwitz, Thomas Reps, and Mooyi Sagiv. 1995. Demand Interprocedural Dataflow Analysis. In Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering (SIGSOFT ’95). ACM, New York, NY, USA, 104–115. https://doi.org/10.1145/222124.222146

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints As Control. In Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL ’12). ACM, New York, NY, USA, 151–164. https://doi.org/10.1145/2103656.2103675

Leonidas Lampropoulos, Diane Gallois-Wong, Cătălin Hriţcu, John Hughes, Benjamin C. Pierce, and Li-yao Xia. 2017. Beginner’s Luck: A Language for Property-based Generators. In Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017). ACM, New York, NY, USA, 114–129. https://doi.org/10.1145/3009837.3009868

Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. 2011. Directed Symbolic Execution. In Proceedings of the 18th International Conference on Static Analysis (SAS’11). Springer-Verlag, Berlin, Heidelberg, 95–111. http://dl.acm.org/citation.cfm?id=2041552.2041563

Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. PSE: Explaining Program Failures via Postmortem Static Analysis. SIGSOFT Softw. Eng. Notes 29, 6 (Oct. 2004), 63–72. https://doi.org/10.1145/1041685.1029907

Proc. ACM Program. Lang., Vol. 4, No. ICFP, Article 102. Publication date: August 2020.
Peter W. O’Hearn. 2019. Incorrectness Logic. Proc. ACM Program. Lang. 4, POPL, Article Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078

Zachary Palmer, Theodore Park, Scott Smith, and Shiwei Weng. 2020a. Higher-Order Demand-Driven Symbolic Evaluation: Software Artifact. Zenodo. https://doi.org/10.5281/zenodo.3923023

Zachary Palmer, Theodore Park, Scott Smith, and Shiwei Weng. 2020b. Higher-Order Demand-Driven Symbolic Evaluation: Supplementary Appendices. ACM Digital Library. https://doi.org/10.1145/3408984

Zachary Palmer and Scott F. Smith. 2016. Higher-Order Demand-Driven Program Analysis. In 30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs)), Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.19

Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. 2013. Verifying Higher-Order Programs with the Dijkstra Monad. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’13). Association for Computing Machinery, New York, NY, USA, 387–398. https://doi.org/10.1145/2491956.2491978

Sam Tobin-Hochstadt and David Van Horn. 2012. Higher-order Symbolic Execution via Contracts. In Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA ’12). ACM, New York, NY, USA, 537–554. https://doi.org/10.1145/2384616.2384655

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2013). ACM, New York, NY, USA, 135–152. https://doi.org/10.1145/2509578.2509586

Maaike Zwart and Dan Marsden. 2018. Don’t Try This at Home: No-Go Theorems for Distributive Laws. arXiv:math.CT/1811.06460 https://arxiv.org/abs/1811.06460