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, and furthermore it is defined as a direct abstraction 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.

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 and a backward approach will avoid searching a vast number of paths which would never get near the target.

This paper focuses on symbolic *backwards* execution (SBE) [Baldoni et al. 2018, §2.3], a variation on symbolic evaluation where evaluation can start anywhere in the middle of the program and proceed *in reverse* to the program start. This reverse propagation is similar in spirit to how Dijkstra weakest-preconditions (*wp*s) 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 to be taken.

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 aforecited systems are capable of automatically generating tests exercising a particular program point, using backward symbolic execution to accumulate constraints required to reach the target.

These imperative language systems unfortunately 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. So, in this paper we develop DDSE, a **d**emand-**d**riven **s**ymbolic **e**valuator 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 easy to formally prove DDSE is correct, something not

Authors' addresses: Zachary Palmer, Swarthmore College, 500 College Ave. Swarthmore, PA, 19081, USA; Theodore Park, Swarthmore and Hopkins, 500 College Ave. Swarthmore, PA, 19081, USA; Scott Smith, The Johns Hopkins University, 3400 N. Charles St. Baltimore, MD, 21218, USA; Shiwei Weng, The Johns Hopkins University, 3400 N. Charles St. Baltimore, MD, 21218, USA.

2019. XXXX-XXXX/2019/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

previously proven for an 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.

In Section 2 we give a high-level overview of the principles behind the approach. Section 3 defines a largely standard operational semantics for a simple functional language that also includes input; this interpreter is then extended with a non-standard notion of fresh variable generation to support demand-driven execution. Section 4 defines the (concrete) demand-driven interpreter and shows it equivalent to the standard one. Section 5 extends the demand-driven interpreter to the 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 6 describes the implementation of the test generation algorithm and its performance on small benchmarks. Section 7 gives related work, and we conclude in Section 8.

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 $\{\mathtt{x} < 25\}$, and continuing to propagate, we have $\{\mathtt{x} > 0 \wedge \mathtt{x} < 25\}$ in line 2. So, it means that input must be in the range of $1 \ldots 24$ 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.

```
1  let  x = input in
2  (* {x > 0 ∧ x < 25} *)
3  if x > 0 then
4    (* {x < 25} *)
5    if x < 25
6      then x+1   (* {true} *)
7      else x-1
8    else x-2
```

Fig. 1. Weakest precondition propagation

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 there does not exist any direct analogue for higher-order 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 some consideration to functions as data: they accommodate virtual method calls by an iterative process for estimating the call graph. However, no soundness properties are asserted in those works and it is not clear if the analysis is sound in the face of a mis-estimated call graph. 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: 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, 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 3. When `x`'s value is needed in the body to add `1` to it, we rely on the fact that the call site was recorded as being at the location[2] `fy`. 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`. We know that `x` does not have the value `1` from the other call site since `fy`, and not `f1`, was the call site recorded. When the evaluator executes the `f1` call, it will again compute `x + 1` but this time under the call site stack `f1`, and there `x` will have value `1`.

We now trace this more precisely. We define a variable lookup function $\mathbb{L}^{\omega}(x, n, C) = v$ to mean that v is the result of a lookup of variable x, starting the (reverse) search from just before line n, and assuming the current call site stack context is C. This call stack C is not the forward execution stack, since we are *wp*-style walking the program in reverse; it denotes the calls entered and not yet exited in this *reverse*-order sequence. This lookup function lacks some accounting parameters for clarity of discussion but is otherwise the heart of the evaluator we formally define in later sections. We now illustrate lookup in detail by showing how the value of `result` in the above program is looked up from the program end and empty call stack, $\mathbb{L}^{\omega}(\texttt{result}, 8, [])$.

```
1  let y = 0 in
2  let f = (fun x ->
3    let fresult = x + 1 in
4      fresult) in
5  let fy = f y in
6  let f1 = f 1 in
7  let result = fy + f1
8  in result
```

Fig. 2. Simple demand-driven evaluation example

(1) This lookup first inspects the previous line: line 7. This line defines `result` in terms of two other variables, so we now have two lookup sub-goals to find `result`'s value: looking up `fy` and `f1` will allow us to establish the result $\mathbb{L}^{\omega}(\texttt{fy}, 7, []) + \mathbb{L}^{\omega}(\texttt{f1}, 7, [])$. We will trace only `fy` in this example since `f1` is similar.
(2) Considering $\mathbb{L}^{\omega}(\texttt{fy}, 7, [])$ and proceeding backward, we may skip over `f1`'s definition as it is irrelevant. In line 5, we find the definition of `fy` which is not yet a value but is a function call. So, `fy`'s value is in fact the *result* of the function call by a similar reverse process: first look up the definition of function `f` ($\mathbb{L}^{\omega}(\texttt{f}, 7, []) =$ `fun x ->`...), and then in turn search for the *result value* of the function body, `fresult`, having pushed this call site on the call stack: $\mathbb{L}^{\omega}(\texttt{fresult}, 4, [\texttt{fy}])$.[3]
(3) Performing $\mathbb{L}^{\omega}(\texttt{fresult}, 4, [\texttt{fy}])$, `fresult` can be seen to be defined in the previous line, line 3, as expression `x + 1`. The value for `fresult` is therefore $\mathbb{L}^{\omega}(\texttt{x}, 4, [\texttt{fy}]) + 1$, recursively invoking lookup on `x`.
(4) Looking up `x`, it can immediately be seen as the parameter to the function, so we want to look up the *value* of this parameter at the original call site; we know that is `fy` as it was recorded on the call stack. So, $\mathbb{L}^{\omega}(\texttt{x}, 4, [\texttt{fy}])$ induces a lookup of the parameter value `y` from the call site `fy`: $\mathbb{L}^{\omega}(\texttt{y}, 5, [])$. This lookup steps back over line 2 and to line 1 where `y` is observed to be `0` and so $\mathbb{L}^{\omega}(\texttt{y}, 5, []) = 0$.
(5) Popping off lookup obligations, $\mathbb{L}^{\omega}(\texttt{x}, 2, [\texttt{fy}]) = \mathbb{L}^{\omega}(\texttt{y}, 5, []) = 0$, which means $\mathbb{L}^{\omega}(\texttt{fresult}, 2, [\texttt{fy}]) = 0 + 1 = 1$, which means $\mathbb{L}^{\omega}(\texttt{fy}, 7, []) = 1$.

[1]We A-normalize our programs to clarify operator ordering. We use an informal OCaml-like syntax in this section and give a formal ANF grammar in the following section.
[2]We use unique variables in our A-normalized programs, so variable definitions serve to uniquely identify program points.
[3]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.

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 an additional stack to record the chain of call frames we need to walk back through to find where a non-local variable is local. For readers familiar with access links in compiler implementations of non-local variable lookup, it is an analogous stucture. So, lookup is now more generally $\mathbb{L}^{\omega}(X, n, C) = v$, where $X = [x_1, \ldots, x_n]$ is a chain (list) of function definitions to be searched for. We have removed the variable looked up, x, from the argument list, instead interpreting list head x_1 as this variable; all lookups in the previous example were singletons, $\mathbb{L}^{\omega}([x], n, C) = v$.

2.1.1 Non-Local Variables. Consider the example of a Curried addition function in Figure 3; x in line 3 is a non-local variable. Suppose we want the value of g51 in line 9, $\mathbb{L}^{\omega}([\texttt{g51}], 9, [])$; this is the value of the call in line 6, which means we need to find the function and then call it (in reverse). First the function definition g5 must be found, via $\mathbb{L}^{\omega}([\texttt{g5}], 6, [])$; as in the previous example it is the result of the application in line 5 so we in turn look up g there and trace into it's gresult to find it is the fun y ... in line 2: $\mathbb{L}^{\omega}([\texttt{g5}], 6, []) =$ fun y ....

```
1   let g = (fun x ->
2     let gresult = (fun y ->
3       let gyresult = x + y in gyresult)
4     in gresult) in
5   let g5 = g 5 in
6   let g51 = g5 1 in
7   let g6 = g 6 in
8   let g62 = g6 2 in
9   let result = g51 + g62
10  in result
```

Fig. 3. Non-local variable example

Now that we have found the function body being executed in the line 6 call, we need to lookup the result variable of that function, gyresult: $\mathbb{L}^{\omega}([\texttt{gyresult}], 3, [\texttt{g51}])$ (by entering the function body we now are inside the call from site g51, thus g51 is on the call stack). Continuing, gyresult is defined in line 3 as expression x + y, so we need to look up x (and y) at this point in order to get gyresult's value: $\mathbb{L}^{\omega}([\texttt{x}], 3, [\texttt{g51}])$. In this sub-lookup, we are inside fun y ... and x is not defined in the current context: it is a non-local. So, since x is not in the current body, we must exit the g51 call and redirect our search: since x is non-local, we can find its definition point if we (again) look up the function definition g5, where x has to be defined as it must be lexically in scope of that function definition. Once we reach g5's definition, we will be able to resume our lookup of x. In other words, we need to perform $\mathbb{L}^{\omega}([\texttt{g5}, \texttt{x}], 6, [])$. When performing this lookup, we are at line 6 at the top level of the program looking for the definition point of g5. Once we have found the definition point of g5 in line 2, will we pop g5 off of the non-locals stack giving goal $\mathbb{L}^{\omega}([\texttt{x}], 2, [\texttt{g5}])$ (the [g5] call stack here reflects that we had to enter the g5 call site to find the fun y definition). $\mathbb{L}^{\omega}([\texttt{x}], 2, [\texttt{g5}])$ is now just a local parameter lookup: this is the parameter at the g5 call site which in turn can be traced to be 5.

The above examples describe the core ideas; for deeper lexical nesting the non-locals stack will just get deeper, similar to how access links extend. 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. Demand-driven functional evaluators of the above form were to our knowledge first developed in [Palmer and Smith 2016] (the ωDDPAc evaluator there), for the purpose of proving a program analysis sound.

2.2 Demand-Driven Symbolic Execution: DDSE

The novel contribution of this paper is to make a *symbolic demand-driven* 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.

2.2.1 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 can be constructed. But, there are subtleties on how to name variables given there may be many activations of the same variable at runtime. Fortunately, the runtime heap location is uniquely determined by the pair of variable name and current call site stack (assuming that the initial program had any duplicate variable definitions renamed, i.e. the program was alphatized). We use notation ${}^{C}x$ 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 (heap locations), not 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 function is of the form $\overset{S}{\mathbb{L}}(x, n, C) = {}^{C'}x'$, and will additionally produce a global set of constraints Φ over all lookups. (For simplicity here we revert to single-variable lookup notation x.)

Let us re-evaluate the Figure 2 example symbolically to illustrate the differences. Consider looking up the variable `result` from line 8 with an empty call stack, $\overset{S}{\mathbb{L}}(\texttt{result}, 8, [])$. Proceeding to line 7 we have arrived at a definition of `result` and are looking to add a constraint of the form ${}^{[]}\texttt{result} = \overset{S}{\mathbb{L}}(\texttt{fy}, 7, []) + \overset{S}{\mathbb{L}}(\texttt{f1}, 7, [])$ to global set Φ. The two lookups for `fy` and `f1` in the above equation need to be computed to find the defining variables to put in the equation, and they may entail other constraints. Let us trace only $\overset{S}{\mathbb{L}}(\texttt{fy}, 7, [])$ since `f1` is similar. This lookup is structurally identical to the demand evaluator above. So, as previously, lookup of `fy` entails (and has the result of) lookup of `f`'s result `fresult`: $\overset{S}{\mathbb{L}}(\texttt{fy}, 7, [])$ is equal to $\overset{S}{\mathbb{L}}(\texttt{fresult}, 4, [\texttt{fy}])$.

Now, $\overset{S}{\mathbb{L}}(\texttt{fresult}, 4, [\texttt{fy}])$ immediately finds the definition of `fresult`, `x + 1`. We have reached a point where a concrete value is constructed, so the equation ${}^{[\texttt{fy}]}\texttt{fresult} = \overset{S}{\mathbb{L}}(\texttt{x}, 3, [\texttt{fy}]) + 1$ must be generated. Here, ${}^{[\texttt{fy}]}\texttt{fresult}$ is the defining variable: the stack annotation `[fy]` disambiguates to mean the `fy` call site allocation of `fresult`. As with the call site above, we have induced a lookup of `x` required to complete the equation. So, looking up $\overset{S}{\mathbb{L}}(\texttt{x}, 3, [\texttt{fy}])$ as in the evaluator above, we see it entails and has the result of $\overset{S}{\mathbb{L}}(\texttt{y}, 5, [])$ which is defined on line 1, and yields the constraint ${}^{[]}\texttt{y} = 0$ since it is a value definition. Further, $\overset{S}{\mathbb{L}}(\texttt{y}, 5, [])$ returns ${}^{[]}\texttt{y}$ and so $\overset{S}{\mathbb{L}}(\texttt{x}, 3, [\texttt{fy}])$ also returns ${}^{[]}\texttt{y}$. Given these results, we construct the addition equation ${}^{[\texttt{fy}]}\texttt{fresult} = {}^{[]}\texttt{y} + 1$ and we return ${}^{[\texttt{fy}]}\texttt{fresult}$ as the defining variable for `fy` which finishes half of the original equation in step 1. A similar process repeats for `f1` and also completes the equation in step 1.

The constraint set Φ for this lookup is then fully:

$$\Phi = \{{}^{[]}\texttt{result} = {}^{[\texttt{fy}]}\texttt{fresult} + {}^{[\texttt{f1}]}\texttt{fresult}, {}^{[\texttt{fy}]}\texttt{fresult} = {}^{[]}\texttt{y} + 1, {}^{[\texttt{f1}]}\texttt{fresult} = 1 + 1, {}^{[]}\texttt{y} = 0\}$$

By basic arithmetic we can conclude that ${}^{[]}\texttt{result} = 3$ is logically deducible from satisfiable Φ. In the DDSE implementation we simply call out to a SMT solver to check for satisfiability of Φ; the implementation will be discussed in Section 6 below.

2.2.2 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 `ifresult`, 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 $\iota = \{{}^{[]}\mathtt{i1} \mapsto 1, {}^{[]}\mathtt{i2} \mapsto 2, {}^{[]}\mathtt{i3} \mapsto 3\}$. Since all three variables are defined at top level, their call stack annotations are empty.

Section 3 formalizes both the stream and mapping notion of input and proves their equivalence in a conventional (forward) evaluator; Section 4 then develops a demand-driven evaluator which uses the mapping model of input and which is also proved to be equivalent to the forward evaluator. The symbolic evaluator of Section 5 also uses the mapping notion of input.

```
 1  let i1 = input in
 2  let i2 = input in
 3  let i3 = input in
 4  let f = fun x ->
 5    let fresult =
 6      if x = 0
 7         then let fresultp = x + 1 in fresultp
 8         else let fresultm = x - 1 in fresultm
 9      in fresult
10  in
11  let ifresult =
12     if i2 = 0
13       then let fi1 = f i1 in fi1
14       else let fi2 = f i2 in fi2
15  in ifresult
```

Fig. 4. Input Example

2.2.3 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 `fresultp` 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 two additional global structures are present, along with formula Φ: a path choice, Π, which records which call site was chosen in the current search attempt; and a conservative call graph, G.

Suppose here we arbitrarily guessed `f i2` in line 14 as the call site that got us to line 7, recording this choice in Π; let us at a high level describe the lookup constraints Φ produced. In this case, we also know that with this choice the call stack when we started must be `[fi2]` 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 5.1 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 $\overset{S}{\mathbb{L}}(\mathtt{x}, 5, [\mathtt{fi2}]) = 0$ where $\overset{S}{\mathbb{L}}(\mathtt{x}, 5, [\mathtt{fi2}])$ is the defining variable obtained by looking up `x`. This lookup in turn has the same result as looking up `fi2`'s call site argument, `i2`, reducing our work in determining $\overset{S}{\mathbb{L}}(\mathtt{i2}, 14, [])$. We now exit the conditional on that line and, since we came from the `else` branch, we will eventually record a constraint of the shape $\overset{S}{\mathbb{L}}(\mathtt{i2}, 14, [])$`<>0` to remember that this condition must have failed.

Continuing with $\overset{S}{\mathbb{L}}(\texttt{i2}, 5, [])$, we finally arrive at its definition in line 2. Since `i2` is defined as an `input`, we do not constrain it. Having completed lookup, we can fill in the holes in the previous constraints: the `else` constraint is ${}^{[]}\texttt{i2} <> 0$ (because $\overset{S}{\mathbb{L}}(\texttt{i2}, 11, []) = {}^{[]}\texttt{i2}$) and the original `then` constraint is ${}^{[]}\texttt{i2} = 0$ (since $\overset{S}{\mathbb{L}}(\texttt{x}, 5, [\texttt{fi2}]) = \overset{S}{\mathbb{L}}(\texttt{i2}, 11, []) = {}^{[]}\texttt{i2}$). These two constraints, ${}^{[]}\texttt{i2<>0}$ and ${}^{[]}\texttt{i2} = \texttt{0}$, are immediately contradictory, meaning that this path will never happen and so the `fi2` call site choice is discarded.

So, rewinding back to the start, we will this time record in Π that the `fi1` call site was the caller of `f`. Skipping the details, it produces $\Phi = \{{}^{[]}\texttt{i2} = \texttt{0}\}$. Any input mapping conforming to these constraints, such as $\{{}^{[]}\texttt{i1} \mapsto \texttt{0}, {}^{[]}\texttt{i2} \mapsto \texttt{0}, {}^{[]}\texttt{i3} \mapsto \texttt{0}\}$, 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.

The discussion above glosses over an important detail: lookup is a *data flow* operation but we are searching for a *control flow* path. In fact, we needed to look up *all* variables encountered in the reverse search and cannot completely skip over any statement as it may have an input or non-termination side effect. In this process we will map the full control of the program as a consequence of tracing the origin of every value definition needed: *the complete data flow of a functional program fully defines the control flow.* The same lookup is triggered many times because of this and the implementation caches it to avoid blowup.

Section 5 formally presents DDSE symbolic evaluator, and our reference implementation is described in Section 6.

3 A STANDARD FORWARD OPERATIONAL SEMANTICS

We begin the formal development by defining a standard forward-proceeding operational semantics. The grammar of our language appears in Figure 5. This figure additionally includes a few elements needed for the operational semantics: environments E are mappings from variables to values and input sequences I are lists of values which will be provided as input during execution.

$$
\begin{array}{lcll}
e & ::= & [c, \ldots] & \textit{expressions} \\
c & ::= & {}^{F}x = b & \textit{clauses} \\
x & ::= & \textit{(identifiers)} & \textit{variables} \\
b & ::= & v \mid {}^{F}x \mid \texttt{input} \mid {}^{F}x\ {}^{F}x & \textit{bodies} \\
 & & \mid {}^{F}x \mathrel{?} e : e \mid {}^{F}x \odot {}^{F}x & \\
f & ::= & \texttt{fun}\ {}^{F}x\ \texttt{->}\ (\,e\,) & \textit{functions} \\
\odot & ::= & + \mid - \mid < \mid = \mid \texttt{and} \mid \texttt{or} \mid \texttt{xor} & \textit{binops} \\
v & ::= & f \mid \mathbb{Z} \mid \mathbb{B} & \textit{values} \\
F & ::= & \epsilon \mid \top \mid \ldots & \textit{freshening tags} \\
 & & & \\
I & ::= & [v, \ldots] & \textit{input list} \\
E & ::= & [{}^{F}x = v, \ldots] & \textit{environments}
\end{array}
$$

Fig. 5. Langauge Grammar

The grammar in Figure 5 is in a shallow A-normal form: all steps are variable assignments and each computation step's arguments are variables. We also require all variable bindings to be unique (i.e., alphatized). These two properties ensure that each program point is uniquely identified by the variable it defines; we will thus pun between variables and program points throughout the paper.

Additionally, in this development we need to be careful with how variables are freshened, and so we formally define variables as pairs of tags F and underlying variable x, which we will notate ${}^{F}x$. The F's can be any countable set which includes two distinct values: $\top$ (which we discuss below) and ϵ. The ϵ value indicates an unfreshened variable; all source programs contain only unfreshened variables. Variable ${}^{\epsilon}x$ will be freshened by replacing ϵ with a fresh F to give ${}^{F}x$. This explicit freshening notation may look heavyweight, but it is critical to distinguish different activations of

Alias
$$\frac{({}^{F_2}x_2 = v) \in E}{E \,||\, [{}^{F_1}x_1 = {}^{F_2}x_2] \,||\, e, I \longrightarrow^1 E \,||\, [{}^{F_1}x_1 = v] \,||\, e, I}$$

Input
$$\frac{I = [v] \,||\, I'}{E \,||\, [{}^{F_1}x_1 = \texttt{input}] \,||\, e, I \longrightarrow^1 E \,||\, [{}^{F_1}x_1 = v] \,||\, e, I'}$$

Binop
$$\frac{\{{}^{F_2}x_2 = v_2, {}^{F_3}x_3 = v_3\} \subseteq E \qquad v = v_2 \odot v_3}{E \,||\, [{}^{F_1}x_1 = {}^{F_2}x_2 \odot {}^{F_3}x_3] \,||\, e, I \longrightarrow^1 E \,||\, [{}^{F_1}x_1 = v] \,||\, e, I}$$

Call
$$\frac{e_0 = E \,||\, [{}^{F_1}x_1 = {}^{F_2}x_2\ {}^{F_3}x_3] \,||\, e \qquad ({}^{F_2}x_2 = \texttt{fun}\, x_4 \texttt{ -> } e') \in E \qquad ({}^{F_3}x_3 = v) \in E \qquad e'' = \textsc{Freshen}_{e_0}(e')}{e_0, I \longrightarrow^1 E \,||\, [{}^{F}x_4 = v] \,||\, e'' \,||\, [{}^{F_1}x_1 = \textsc{RetV}(e'')] \,||\, e, I}$$

Conditional True
$$\frac{{}^{F_1}x_1 = \texttt{true} \in E \qquad {}^{F}x' = \textsc{RetV}(e_1)}{E \,||\, [{}^{F}x = ({}^{F_1}x_1 \ ?\ e_1 \ :\ e_2)] \,||\, e, I \longrightarrow^1 E \,||\, e_1 \,||\, [{}^{F}x = {}^{F}x'] \,||\, e, I}$$

Conditional False
$$\frac{{}^{F_1}x_1 = \texttt{false} \in E \qquad {}^{F}x' = \textsc{RetV}(e_2)}{E \,||\, [{}^{F}x = ({}^{F_1}x_1 \ ?\ e_1 \ :\ e_2)] \,||\, e, I \longrightarrow^1 E \,||\, e_2 \,||\, [{}^{F}x = {}^{F}x'] \,||\, e, I}$$

Fig. 6. Standard Forward Operational Semantics

the same variable in the symbolic evaluator in Section 5. To prevent clutter, we occasionally use unmarked variables x to refer to their unfreshened counterparts ${}^{\epsilon}x$.

We define the small step operational semantics as a relation $e \longrightarrow^1 e$ using the following definitions:

Definition 3.1. Let $\textsc{RetV}(e) = x$ if $e = [c, \ldots, x = b]$. That is, x is the last variable of e.

Definition 3.2. $e \longrightarrow^1 e'$ holds iff a proof exists in the system of Figure 6. We write $e_0 \longrightarrow^* e_n$ to denote $e_0 \longrightarrow^1 \ldots \longrightarrow^1 e_n$.

The rules in Figure 6 are largely straightforward. The auxiliary function RetV returns the last variable defined in an expression, which is taken to be the expression's result. I is the input stream during evaluation[4].

The variable freshening function $\textsc{Freshen}_e({}^{\epsilon}x)$ used in the rules is any function returning ${}^{F'}x'$ for F' not occurring in e; one (inefficient) version of such a function would be for $F = e$, tagging the variable with the full computation state. We will never re-freshen variables: $\textsc{Freshen}_e({}^{F}x)$ for $F \neq \epsilon$ is identity. Below we will make a more nuanced version which is our primary purpose in focusing on freshening.

We extend this freshening function to clauses and expressions. Function freshening $\textsc{Freshen}_e(f)$ freshens only *non-local* variable uses in the function's body because the function is not being invoked at this point. All other clause bodies as well as clauses and expressions are freshened homomorphically; e.g. expressions are freshened as $\textsc{Freshen}_e([c_1, \ldots, c_n]) = [\textsc{Freshen}_e(c_1), \ldots, \textsc{Freshen}_e(c_n)]$. Note that all clause variable definitions are also freshened, $\textsc{Freshen}_e({}^{\epsilon}x = b) = (\textsc{Freshen}_e({}^{\epsilon}x) = \textsc{Freshen}_e(b))$.

[4]Note that we use a finite list I to represent the input stream for simplicity. A finite n-step run of a program will only consume at most the first n elements of an infinite stream; any non-terminating evaluation can be modeled as the limit of incrementally larger finite step executions.

Source programs contain only un-freshened variables ${}^{\epsilon}x$, but we maintain an invariant that variables in the top level of a program under evaluation are freshened. To set up this invariant initially, The evaluation of a source program e begins by computing $e' = \textsc{Freshen}_{\top}(e)$ which freshens all top-level variables by giving them the initial freshening tag $\top$; the resulting e' is then evaluated by Definition 3.2. Concretely, we will use stacks of call sites as freshening tags, and $\top$ will be the empty stack, as explained below.

3.1 Freshening with call site stacks

For the symbolic interpreter we need to have a more focused notion of freshening. In particular, we would like to use the stack of call sites as indices on variables and not the whole execution state as in our example freshening function above. But we need to show that this will satisfy the requirements of a freshening function: it will rename variables to names not currently in use anywhere in the global execution state.

We start by defining each element of F to be either the unique ϵ or a call stack C. Here, a call stack is a list of call sites:

$$C = [x_1 = x_1'\ x_1'', \ldots, x_n = x_n'\ x_n'']$$

with the head of the list meaning the top of stack; i.e. the most recent call here was at call site $x_1 = x_1'\ x_1''$. These call sites are the literal clauses in the original program: variables x_i are just the program (base) variables and are not freshened. F must include a distinguished element $\top$ and here we fix $\top = []$. (ϵ remains distinguished and not an element of the form C.) We can then equivalently define freshened variables as an x, C pair which we will notate ${}^{C}x$.

We then may define freshening as

$$\textsc{Freshen}^{\text{cs}}_{E\,||[{}^{C}x_1={}^{C'}x_2\ {}^{C''}x_3]\,||\,e}(x) = {}^{([x_1=x_2\ x_3]\,||\,C)}x$$

The context C on the defined variable invariably represents the call stack at the point when this particular call was executed, and so we want to simply add the current call site (the source name for it, not the freshened version) to C.

Lemma 3.3. *$\textsc{Freshen}^{cs}_{E\,||[{}^{C}x_1={}^{C'}x_2\ {}^{C''}x_3]\,||\,e}(x)$ is a freshening function: for any execution $e_0 \longrightarrow^* \ldots \longrightarrow^* E\,||[{}^{C}x_1 = {}^{C'}x_2\ {}^{C''}x_3]\,||\,e \longrightarrow^* \ldots e_n$, for any variable x, $\textsc{Freshen}^{cs}_{E\,||[{}^{C}x_1={}^{C'}x_2\ {}^{C''}x_3]\,||\,e}(x)$ does not occur in $(E\,||[{}^{C}x_1 = {}^{C'}x_2\ {}^{C''}x_3]\,||\,e)$.*

Proof. Proceed by induction on the length of the derivation to prove the lemma (referred herein as (1)) strengthened with the additional assertions that (2) each call site ${}^{C}x_1 = {}^{C'}x_2\ {}^{C''}x_3$ is executed (i.e. replaced) only once (since each time the same point in the original program is revisited, the variables will be fresh), and (3) that all variable definitions in each program state are unique.

So, assume previous steps have these three properties, and wolog show the step evaluating the call ${}^{C}x_1 = {}^{C'}x_2\ {}^{C''}x_3$ will preserve them. For (2), since by induction all variable definitions are unique, there is only one occurrence of the call ${}^{C}x_1 = {}^{C'}x_2\ {}^{C''}x_3$ in the program, and the call rule removes that clause and so it will no longer appear in the program. So, since the call is removed in the remainder of the derivation it cannot be invoked again. Now for the lemma property, (1). We know by induction that ${}^{C}x_1$ must be unique. By property (2) we know there must not have been another call of ${}^{C}x_1$ earlier in the derivation, so there will be no variables ${}^{[x_1=x_2\ x_3]]\,||\,C}x$ for any x occurring in the execution state. Thus, all variable definitions ${}^{[x_1=x_2\ x_3]]\,||\,C}x$ are fresh and (1) is established. For (3), all variable definitions by induction were unique and by showing (1) just now we established all new variable definitions are also unique with respect to existing variables, thus all variable definitions are unique, establishing (3). □

From now on we will assume that $\textsc{Freshen}^{\text{cs}}$ is the function used for variable freshening.

$$\begin{tikzcd}
\text{input stream interpreter} & \boxed{e, I \longrightarrow^* e, I} \arrow[r, Rightarrow] & \boxed{e, I, \iota \longrightarrow^* e, I, \iota} \arrow[d, Rightarrow] & \iota\text{-building interpreter} \\
I\text{-building interpreter} & \boxed{e, I \longrightarrow^*_\iota e, I} \arrow[u, Rightarrow] & \boxed{e \longrightarrow^*_\iota e} \arrow[l, Rightarrow] & \text{input map interpreter}
\end{tikzcd}$$

Fig. 7. Modeling Input as a Mapping

3.2 Modeling input equivalently as a mapping

Streaming is the most natural model for input in a forward-running operational semantics. As illustrated above, however, a demand-driven interpreter looks up inputs in the (reverse) order of how they are *used*, not how they are *defined*. We address this by modeling input for the demand-driven interpreter as a mapping ι from heap locations (uniquely identified by freshened variables) onto the values they will input. By Lemma 3.3, freshened variables are used at most once, so such a mapping is sufficient to disambiguate inputs at runtime and we can build an isomorphism between the stream and mapping views.

To do this, we will make an instrumented version of the substitution-based interpreter above. In this instrumented interpreter, each evaluation of a clause ${}^{C}x = \texttt{input}$ adds the mapping ${}^{C}x \mapsto v$ to ι. By executing the entire computation, we may incrementally build the full ι. Once ι is defined we can build another interpreter $\longrightarrow^*_\iota$ which only needs ι but is otherwise equivalent. We will later need the reverse direction as well, constructing I from ι and a dual process will work there. This approach is summarized in Figure 7.

Formally we will define ι by the following grammar rule.

$\iota \quad ::= \quad \{{}^{C}x \mapsto v, \ldots, {}^{C}x \mapsto v\}$ *inputs*

We now define $e, I, \iota \longrightarrow^1 e, I, \iota$. On the non-input rules the additional arguments I and ι are just related as identity, so we leave out those rules.

$$\textsc{Input}\quad \frac{I = [v] \,||\, I'}{E \,||\, [{}^{C}x_1 = \texttt{input}] \,||\, e, I, \iota \longrightarrow^1 E \,||\, [{}^{C}x_1 = v] \,||\, e, I', \iota \cup \{{}^{C}x_1 \mapsto v\}}$$

$e, I, \iota \longrightarrow^1 e', I', \iota'$ holds if a proof exists in the above system; $e_0, I_0, \{\} \longrightarrow^* e_n, I_n, \iota_n$ is the usual closure. Note that ι_0 here is the empty mapping $\{\}$; the mapping ι_n is synthesized by the computation.

With an inferred ι we may then define computation which takes as given a fixed input mapping. This is the system that can be related to the demand-driven and symbolic interpreters in the following sections.

So, we next define $\longrightarrow^*_\iota$, where ι is a (fixed) parameter on the relation. Again the rules are nearly the unchanged, so we only present the Input rule:

$$\textsc{Input}\quad \frac{({}^{C}x_1 \mapsto v) \in \iota}{E \,||\, [{}^{C}x_1 = \texttt{input}] \,||\, e \longrightarrow^1_\iota E \,||\, [{}^{C}x_1 = v] \,||\, e}$$

$e \longrightarrow^1_\iota e'$ holds if a proof exists in this system; $e_0 \longrightarrow^*_\iota e_n$ is the usual closure.

We can then establish soundness of the stream-to-mapping conversion.

Lemma 3.4. *If* $e_0, I_0, \{\} \longrightarrow^* e_n, I_n, \iota_n$ *then* $e_0 \longrightarrow^*_{\iota_n} e_n$.

For test generation we also need to be able to convert in the opposite direction: we are going to find test inputs using the mapping notion of input, and would like to map them back to a standard interpreter. For this we can create another system which infers I from ι. Note that if $\iota({}^{C}x)$ is not defined, that means the input was never used and so the demand-driven inference algorithm never needed the value; for these cases an arbitrary value can be chosen.

$$\textsc{Input}\quad \frac{({}^{C}x_1 \mapsto v) \in \iota \text{ or } {}^{C}x_1 \notin \textsc{Dom}(\iota)}{E \,||\, [{}^{C}x_1 = \texttt{input}] \,||\, e, I \longrightarrow^1_\iota E \,||\, [{}^{C}x_1 = v] \,||\, e, (I \,||\, [v])}$$

$e, I \longrightarrow^1_\iota e', I'$ holds if a proof exists in the above system; $e_0, I_0 \longrightarrow^*_\iota e_n, I_n$ is the usual closure.

Observe that we built the input mapping by adding the elements as they appear to the end of the input; in the stream-based system we read off the front of the final stream inferred. That is reflected in the following soundness property.

Lemma 3.5. *If* $e_0, [] \longrightarrow^*_\iota e_n, I_n$ *then* $e_0, I_n \longrightarrow^* e_n, []$.

Given this isomorphism between stream and mapping forms of input, we will hereafter use the mapping form; our test inference algorithm will produce a mapping ι and we can then use the above stream inference method to produce an input stream usable by a standard compiler or interpreter to exercise the test case.

4 A REVERSE CONSTRUCTION INTERPRETER

In this section we construct a demand-driven interpreter for our language following the formal definition of the ωDDPAc interpreter in [Facchinetti et al. 2019]. This section follows Section 5.4 of [Facchinetti et al. 2019], but here we also add in input statements needed for test generation, as well as conditional branching statements as well as atomic data and operations; we will refer to the resulting language as $\omega\iota$DDPAc with the ι standing for the input addition.

Figure 8 contains the language grammar as well as all of the constructs needed to define the interpreter.

Definition 4.1. We use the following notational sugar for control flow graph edges:

- $a_1 \ll a_2 \ll \ldots \ll a_n$ abbreviates $\{a_1 \ll a_2, \ldots, a_{n-1} \ll a_n\}$.
- $a' \ll \{a_1, \ldots, a_n\}$ (resp. $\{a_1, \ldots, a_n\} \ll a'$) denotes $\{a' \ll a_1, \ldots, a' \ll a_n\}$ (resp. $\{a_1 \ll a', \ldots, a_n \ll a'\}$). That is, $\ll$ implicitly generates a product on sets.
- $a' \ll [a_1, \ldots, a_n]$ (resp. $[a_1, \ldots, a_n] \ll a'$) denotes $\{a' \ll a_1 \ll \ldots \ll a_n\}$ (resp. $\{a_1 \ll \ldots \ll a_n \ll a'\}$). That is, $\ll$ preserves ordering for lists (specifically because expressions e are a form of list).
- We write $a \ll a'$ to mean $a \ll a' \in G$ for some graph G understood from context.

We use standard notation $[x_1, \ldots, x_n]$ etc for lists and $||$ for list append.

The definition of lookup 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.

Lookup also proceeds with respect to a *lookup stack* X. The topmost variable of this stack is the variable currently being looked up.

$$\begin{array}{rcll}
V & ::= & \{v, \ldots\} & \textit{value sets} \\
a & ::= & c \mid x \overset{\text{◖}c}{=} x \mid x \overset{\text{◗}c}{=} x & \textit{annotated clauses} \\
 & & \mid v\text{◖}c \mid \textsc{Start} \mid \textsc{End} & \\
g & ::= & a \ll a & \textit{control flow edges} \\
G & ::= & \{g, \ldots\} & \textit{control flow graphs} \\
C & ::= & [c, \ldots] & \textit{clause stacks} \\
\mathbf{C} & ::= & \{C, \ldots\} & \textit{clause stack sets} \\
X & ::= & [x, \ldots] & \textit{continuation stacks}
\end{array}$$

Fig. 8. Structures for graph-based interpreter $\omega\iota$DDPAc

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.

Lookup finds the value of a variable starting from a given graph node. Given a control flow graph G, we write $\mathbb{L}^{\omega}(G, X, a_0, C, \iota)$ to denote a lookup using stack X in G relative to graph node a_0 with context C and input mapping ι (recall that in Section 3.2 we re-mapped inputs from a stream to a mapping to better align with our demand interpreters). For instance, a lookup of variable x from program point a with empty context would be written $\mathbb{L}^{\omega}(G, [x], a, [], \iota)$. Note that this refers to looking for values of x *upon reaching* program point a but *before* that point is executed (much like the convention of interactive debuggers); we are looking for a definition of x in the *predecessors* of a but not within a itself.

It is possible (and likely) that the variable for which we are searching is found before lookup walks back to the start of the program. This is because we are attempting to establish a complete *control flow* using lookup, a *data flow* operation. If we permit lookup to produce a result without reaching the start of the program, it may produce results based upon e.g. dead code. To help address this problem, we require each discovered value to prove it arises from a valid control flow from the start of the program. To aid this, we define $\textsc{FirstV}([x_1 = b_1, \ldots]) = x_1$.

Definition 4.2. Given control flow graph G for program e, $\mathbb{L}^{\omega}(G, X, a_0, C, \iota)$ is the function returning the least set of values V satisfying the following conditions given some $a_1 \ll a_0$:

(1) $\boxed{\textsc{Value Discovery}}$
If $a_1 = (x = v)$ and $X = [x]$, then $v \in V$,
provided that if $x \neq \textsc{FirstV}(e)$, $\mathbb{L}^{\omega}(G, [\textsc{FirstV}(e)], a_1, C, \iota)$ is non-empty.

(2) $\boxed{\textsc{Input}}$
If $a_1 = (x = \texttt{input})$ and $X = [x]$, then $v \in V$ if $v \in \mathbb{Z}$, and $\iota({}^{C}x) = v$,
provided that if $x \neq \textsc{FirstV}(e)$, $\mathbb{L}^{\omega}(G, [\textsc{FirstV}(e)], a_1, C, \iota)$ is non-empty.

(3) $\boxed{\textsc{Value Discard}}$
If $a_1 = (x_1 = v)$ and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $\mathbb{L}^{\omega}(G, [x_2, \ldots, x_n], a_1, C, \iota) \subseteq V$.

(4) $\boxed{\textsc{Alias}}$
If $a_1 = (x = x')$ and $X = [x] \,||\, X'$ then $\mathbb{L}^{\omega}(G, [x'] \,||\, X', a_1, C, \iota) \subseteq V$.

(5) $\boxed{\textsc{Binop}}$
If $a_1 = (x = x' \odot x'')$, $X = [x] \,||\, X'$, $v' \in \mathbb{L}^{\omega}(G, [x'], a_1, C, \iota)$ and $v'' \in \mathbb{L}^{\omega}(G, [x''], a_1, C, \iota)$, then $v' \odot v'' \in V$, provided that (1) $\mathbb{L}^{\omega}(G, [\textsc{FirstV}(e)], a_1, C, \iota)$ is non-empty, and
(2) if $X' \neq []$, then $\mathbb{L}^{\omega}(G, X', a_1, C, \iota)$ is non-empty.

(6) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (x \overset{\text{◖}c}{=} x')$, $X = [x] \,||\, X'$, $C = [c] \,||\, C'$, $c = (x_r = x_f\ x_a)$, and $\texttt{fun}\ x \texttt{ -> (} e \texttt{)} \in \mathbb{L}^{\omega}(G, [x_f], a_1, C', \iota)$, then $\mathbb{L}^{\omega}(G, [x'] \,||\, X', a_1, C', \iota) \subseteq V$.

(7) $\boxed{\textsc{Function Enter Non-Local}}$
If $a_1 = (x'' \overset{\text{◖}c}{=} x')$, $X = [x] \,||\, X'$, $C = [c] \,||\, C'$, $x'' \neq x$, $c = (x_r = x_f\ x_a)$, and $\texttt{fun}\ x'' \texttt{ -> (} e \texttt{)} \in \mathbb{L}^{\omega}(G, [x_f], a_1, C', \iota)$, then $\mathbb{L}^{\omega}(G, [x_f, x] \,||\, X', a_1, C', \iota) \subseteq V$.

(8) $\boxed{\textsc{Function Exit}}$
If $a_1 = (x \overset{\text{◗}c}{=} x')$, $X = [x] \,||\, X'$, $c = (x_r = x_f\ x_a)$, $x' = \textsc{RetV}(e)$, and $\texttt{fun}\ x'' \texttt{ -> (} e \texttt{)} \in \mathbb{L}^{\omega}(G, [x_f], c, C, \iota)$, then $\mathbb{L}^{\omega}(G, [x'] \,||\, X', a_1, [c] \,||\, C, \iota) \subseteq V$.

(9) $\boxed{\textsc{Skip}}$
If $a_1 = (x'' = b)$, $X = [x] \,||\, X'$, $x'' \neq x$, and $\mathbb{L}^{\omega}(G, [x''], a_0, C, \iota)$ is non-empty, then $\mathbb{L}^{\omega}(G, X, a_1, C, \iota) \subseteq V$.

(10) $\boxed{\textsc{Conditional Top}}$
If $a_1 = v \triangleleft c$, $c = (x = x_1 \mathrel{?} e_1 : e_2)$, and $v \in \mathbb{L}^{\omega}(G, [x_1], c, C, \iota)$,
then $\mathbb{L}^{\omega}(G, X, a_1, \dot{C}, \iota) \subseteq V$.

(11) $\boxed{\textsc{Conditional Bottom - True}}$
If $a_1 = (x \overset{\triangleright c}{=} x')$, $c = (x = x_1 \mathrel{?} e_1 : e_2)$, $X = [x] \,||\, X'$, `true` $\in \mathbb{L}^{\omega}(G, [x_1] \,||\, X', a_1, C, \iota)$, and $x' = \textsc{RetV}(e_1)$, then $\mathbb{L}^{\omega}(G, [x'] \,||\, X', a_1, C, \iota)$.

(12) $\boxed{\textsc{Conditional Bottom - False}}$
If $a_1 = (x \overset{\triangleright c}{=} x')$, $c = (x = x_1 \mathrel{?} e_1 : e_2)$, $X = [x] \,||\, X'$, `false` $\in \mathbb{L}^{\omega}(G, [x_1] \,||\, X', a_1, C, \iota)$, and $x' = \textsc{RetV}(e_2)$, then $\mathbb{L}^{\omega}(G, [x'] \,||\, X', a_1, C, \iota)$.

This represents an extension of Definition 5.12 of [Facchinetti et al. 2019] to include conditionals and atomic data and operations. The intuitions for this definition were given in examples in Figures 2 and 3 in Section 2.1: the lookup operations described there are informal versions of the operation in Definition 4.2.

Some lookup arguments were implicitly left out in the informal presentation. In the Figure 2 description, for example, the informal lookup $\mathbb{L}^{\omega}($`result`$, 8, [])$ is formally $\mathbb{L}^{\omega}(G, [$`result`$], a_9, [], \iota)$. The informal OCaml-ANF used in the overview does not exactly correspond to our formal grammar which consists of clauses only; here, a_9 is an extra line such as `end = result` appended to that figure to allow a search for the final value. G and ι were elided in the informal discussion as they are constant throughout lookup. Further, the lookup stack in that example was always a singleton, so lookup was presented using a single variable rather than a stack. The trace of the Figure 3 example shows the case where the lookup stack is not a singleton.

4.1 CFG construction

See Figure 9 for the single-step relation which incrementally builds a minimal CFG: it only wires in edges that are in fact needed in a lookup. The relation is parameterized by a fixed input mapping ι.

Definition 4.3. Fixing e and ι, $G \longrightarrow^1_\iota G'$ holds if a proof exists in the system of Figure 9. We write $G_0 \longrightarrow^*_\iota G_n$ iff $G_0 \longrightarrow^1_\iota \ldots \longrightarrow^1_\iota G_n$, and $G \downarrow_\iota G'$ iff $G \longrightarrow^*_\iota G'$ and no $G'' \neq G'$ exists such that $G' \longrightarrow^1_\iota G''$.

Definition 4.4. The initial embedding of an expression into a graph, $\textsc{Embed}([c_1, \ldots, c_n])$, is the graph $G = \textsc{Start} \lll c_1 \lll \ldots \lll c_n \lll \textsc{End}$. We write $e \downarrow_\iota G$, the CFG constructed for e over input mapping ι, to mean $\textsc{Embed}(e) \downarrow G$.

This CFG construction process is an extension of Definition 5.14 in [Facchinetti et al. 2019] to include conditionals.

4.2 A CFG for Symbolic Evaluation

Since different inputs may lead to different CFGs (some code/wirings will be dead with some inputs and not others), producing a CFG for each different set of inputs is undecidable as it would require running the program on all possible inputs. Fortunately, we don't need the completely accurate CFG for our symbolic evaluation algorithm to be useful: the rules verify that a wiring is correct in context by first looking up the function. So, the CFG only needs to be a conservative approximation of the actual CFG for those inputs – it may have spurious function call wirings which will not happen in any program run, and they will not change the final result of any lookup. So, it is trivial to construct a maximal CFG which wires all function bodies to all call sites and the lookup process would still produce the same results, albeit more slowly due to all the traversing of invalid call

Application

$$\frac{c = (x_1 = x_2\ x_3) \qquad C \in \textsc{Active}(c, G) \qquad (\texttt{fun}\ x_0\ \texttt{->}\ (\ e\ )) \in \mathbb{L}^{\omega}(G, [x_2], c, C, \iota) \qquad v \in \mathbb{L}^{\omega}(G, [x_3], c, C, \iota)}{G \longrightarrow^1_\iota G \cup \big(\textsc{Preds}(c) \lll (x_0 \overset{\llparenthesis c}{=} x_3) \lll e \lll (x_1 \overset{\rrparenthesis c}{=} \textsc{RetV}(e)) \lll \textsc{Succs}(c)\big)}$$

Conditional True

$$\frac{c = x_1 = (x_2\ ?\ e_1\ :\ e_2) \qquad C \in \textsc{Active}(c, G) \qquad \texttt{true} \in \mathbb{L}^{\omega}(G, [x_2], c, C, \iota)}{G \longrightarrow^1_\iota G \cup \big(\textsc{Preds}(c') \lll \texttt{true}\,\llparenthesis c \lll e_1 \lll (x_1 \overset{\rrparenthesis c}{=} \textsc{RetV}(e_1)) \lll \textsc{Succs}(c)\big)}$$

Conditional False

$$\frac{c = x_1 = (x_2\ ?\ e_1\ :\ e_2) \qquad C \in \textsc{Active}(c, G) \qquad \texttt{false} \in \mathbb{L}^{\omega}(G, [x_2], c, C, \iota)}{G \longrightarrow^1_\iota G \cup \big(\textsc{Preds}(c') \lll \texttt{false}\,\llparenthesis c \lll e_2 \lll (x_1 \overset{\rrparenthesis c}{=} \textsc{RetV}(e_2)) \lll \textsc{Succs}(c)\big)}$$

where in the above

$\textsc{Preds}(a) = \{a' \mid a' \lll a\}$, $\textsc{Succs}(a) = \{a' \mid a \lll a'\}$, and
$\textsc{Active}(G, a)$ is the least set $\boldsymbol{C}$ conforming to the following conditions:
1. If $c \lll a$ then $\textsc{Active}(G, c) \subseteq \boldsymbol{C}$.
2. If $a' \lll a$ for $a' = (x \overset{\llparenthesis c}{=} x')$ and $C \in \textsc{Active}(G, a')$, then $(C \,||[c]) \in \boldsymbol{C}$.
3. If $\textsc{Start} \lll a$ then $[] \in \boldsymbol{C}$.

Fig. 9. The Single-Step CFG Construction Relation $\longrightarrow^1_\iota$

paths. Formally, we require the CFG G to at least have all the edges needed for any possible inputs. We term any such G to be *complete*.

There are two obvious complete CFGs which an implementation can use. The first is to use the maximal CFG, but this choice incurs a significant performance penalty. A more efficient approach is to run some simple program analysis which conservatively approximates the CFG; such a CFG will be complete since the analysis is conservative. Any program analysis that constructs a conservative call graph suffices in this case.

We define the "best" (minimal complete) CFG for symbolic evaluation as union of CFGs over all possible inputs mappings, and a "good" (complete) CFG is any superset.

Definition 4.5. Given program e the *minimal complete CFG* is $\bigcup_\iota\{G \mid e \downarrow_\iota G\}$. And, a *complete* CFG for program e is any superset of the above minimal complete CFG.

In order to correctly perform lookups, any complete CFG will suffice.

Lemma 4.6. *Given two complete CFGs G and G' for program e, for any input ι and lookup parameters $x/c/C$, $\mathbb{L}^{\omega}(G, [x], c, C, \iota) = \mathbb{L}^{\omega}(G', [x], c, [], \iota)$.*

4.3 Equivalence of Demand and Forward Interpreters

The standard forward operational semantics of Section 3 is equivalent to the demand interpreter of this section, a fact we now establish. We show a bisimulation relation between a chain of interpreters to prove the result. We will fix the interpreters to all take the ι view of input, as a mapping, so the forward interpreter we will compare with is $\longrightarrow^1_\iota$.

For notational convenience, we write $\downarrow_\iota$ to indicate a sequence of $\longrightarrow^1_\iota$ (in all interpreters) which cannot make further progress. We then phrase equivalence as follows:

Lemma 4.7 (Equivalence of Demand and Forward Interpreters). *For any e, $e \downarrow_\iota e'$ if and only if* $\{\textsc{Start} \ll e \ll \textsc{End}\} \downarrow_\iota G$ *such that* $\textsc{Active}(G, \textsc{End}) \neq \emptyset$.

We now sketch this argument working from the analogous proof in [Facchinetti et al. 2019], Theorem 5.19; the proof in that paper is over 10 pages and requires development of several intermediate interpreters to ease the proof burden. Since there are relatively minor differences between the above assertion and the analogous Theorem 5.19 of [Facchinetti et al. 2019], we are not going to reproduce it here. Instead, we will briefly outline the differences and how they can be accounted for. First, the language of [Facchinetti et al. 2019] lacks input, conditionals, and integers; conditionals and integers are found in an earlier version [Palmer and Smith 2016]. Input only amounts to reading values from the fixed input map ι; the early interpreters in the chain of [Facchinetti et al. 2019] all need to be revised use the $\textsc{Freshen}^{cs}$ method for freshening variables to align with ι's domain. In the final $\omega\iota$DDPAc interpreter, the call stack C must be paired with the variable x to give the appropriate ${}^{C}x$ in the domain of ι. Lastly, the interpreter chain in [Facchinetti et al. 2019] starts with a closure-based interpreter and Lemma is needed to show the forward interpreter we start with here is equivalent to the initial closure-based interpreter of [Facchinetti et al. 2019]; fortunately, this equivalence is well-known.

5 A SYMBOLIC DEMAND-DRIVEN EVALUATOR

Now we may finally modify the $\omega\iota$DDPAc lookup operation to produce DDSE, the symbolic demand-driven evaluator that is the goal of this paper. 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 concrete stacks C in the lookup process with *relative* stacks $\dot{C}$; this supports variable lookups that start in the middle of the program without knowing how we may have arrived at that point.

For the symbolic interpreter, we use a global cache of constraints expressed as a (single) logical formula Φ which can 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 Φ are of the form ${}^{\dot{C}}x$. Lookup paths are realizable only if the constraints of Φ can be met; that is, Φ must always be satisfiable for some variable assignment. The implementation uses an SMT solver to verify this condition.

In a symbolic interpreter, there is no single path of execution. First, since we can start loopup mid-program, we may start very deep inside an (unknown) call stack and, as the correct caller is not known, we must search through all callers. We address this issue by adding a parameter Π to lookup which remembers the call sites chosen by a particular lookup path. Second, conditionals could have both true and false branches satisfiable if we had not (yet) accumulated any constraints in Φ 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.

Since our symbolic interpreter does not know the program stack at the time that lookup started, it uses a *relative* stack $\dot{C}$ in lieu of the concrete stack from Figure 8. At the beginning of symbolic lookup, the stack is *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 recording decisions in the aforementioned Π, we also *retrospectively* learn what the stack was when lookup began. The relative stack $\dot{C}$ consists of two parts: the normal stack, which represents those frames which have been pushed since we began lookup; and the *co-stack*, which represents those frames which have been popped since we began lookup.

In the $\omega\iota$DDPAc interpreter we had the luxury of incrementally building a CFG customized for the particular concrete input values. In this system there are many input values, and the CFGs could be radically different based on the input values. So, for this symbolic interpreter we assume a CFG G has previously been constructed which must include any edge possible in any input: it must be a complete CFG as defined in Definition 4.5. As with the interpreter, having spurious edges in the CFG will not introduce incompleteness into the algorithm: the lookup function verifies the correctness of each CFG call edge it uses by looking up the called function to see if it in can arrive at the current call site under the current context.

5.1 Stacks which start mid-program

Here we give the details on how a relative stack is constructed. Consider a lookup for a point in the program which could, at runtime, be reached with the stack $[c_3c_2c_1]$ (where c_3 is the most recently called site). We will show how this call stack can be constructed without the use of an oracle.

Lookup begins with a relative stack $\dot{C} = []?[]$; here, the empty stack on the left is the co-stack while the empty stack on the right is the normal stack. Moving backward through the program, we exit the call site c_3; we then proceed with a new relative stack, $\dot{C} = [c_3]?[]$. This relative stack indicates that, if we considered an execution of the program leading to the lookup site, the runtime stack of our current location could be derived from the runtime stack of the place at which lookup began by popping c_3. Likewise, we would likely next encounter the relative stack $\dot{C} = [c_2c_3]$, indicating that c_2 has also been removed since lookup started (and was removed most recently).

Throughout a particular lookup, a relative stack is isomorphic to a concrete runtime stack. At the end of the lookup algorithm, when the start of the program has been reached, we can reconstruct the concrete stack which existed at the start of lookup using the relative stack we had when lookup reached the top of the program. For example, given that the full stack is $[c_3c_2c_1]$, the relative stack $[c_3]?[]$ is (in retrospect) the concrete stack $[c_2c_1]$ and $[c_2c_3]?[]$ is $[c_1]$: the concrete stack is derived by reversing the co-stack (as those pops were discovered by walking *backward* through the program) and then removing them from the head of the known, final concrete stack $[c_3c_2c_1]$ to give the corresponding stack at that point during lookup.

The other part of the relative stack is used to address functions we have entered (in reverse) during lookup. Even though we do not yet know the outer frames, we can track these inner frames and ensure that they are correctly popped by our lookup path. Continuing the above example, $\dot{C} = [c_2c_3]?[c_5c_4]$ means that, while walking backward, we popped out of c_3 and c_2 but then entered two (known) call sites: first c_4 and then c_5.

5.2 Formal preliminaries to lookup

First the C grammar and stack operations are replaced with relative stacks $\dot{C} = [c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}]$. The operations on these stacks are defined as follows.

(1) $\textsc{Push}([c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}], c) = [c_1, \ldots c_n]?[c, c'_1, \ldots c'_{n'}]$,
(2) $\textsc{Pop}([c_1, \ldots c_n]?[], c) = [c, c_1, \ldots c_n]?[]$,
(3) $\textsc{Pop}([c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}], c) = [c_1, \ldots c_n]?[c'_2, \ldots, c'_{n'}]$ for $c = c'_1$,
(4) $[c_1, \ldots c_n]?[c'_1, \ldots c'_{n'}]$ is *empty* iff $n' = 0$ (the stack is empty, the co-stack may not be).

Above, we described how non-determinism in the lookup function is problematic and, as we indicated, we will include an additional parameter Π which represents the choices made. Concretely, we let Π range over mappings from relative stacks $\dot{C}$ to function entry wiring edges $x \overset{(\!|c}{=} x'$ (where c is a call site). Key $\dot{C}$ is the current relative stack and the mapped wiring edge is the one which should be chosen by the lookup function under this call stack. The lookup function takes a full Π as parameter; our implementation will need to search through the space of Π mappings. Formula Φ

is similarly oracular in the specification, and the implementation builds Φ mostly-monotonically. Φ also contains conditional branch choices which are not monotonic if both cases are satisfiable, and a search over the choice is made in the implementation.

The following is a summary of the grammar changes discussed above.

$$
\begin{array}{lcll}
{}^{\dot{C}}x & & & \textit{annotated variables} \\
\mathcal{X} & & & \textit{annotated variable sets} \\
\dot{C} & ::= & C?C & \textit{relative stacks} \\
\varsigma & ::= & {}^{\dot{C}}x \mid \varsigma_{\mathtt{true}} & \textit{formulae symbols} \\
\phi & ::= & {}^{\dot{C}}x = {}^{\dot{C}}x \odot {}^{\dot{C}}x \mid {}^{\dot{C}}x = {}^{\dot{C}}x \mid {}^{\dot{C}}x = v \mid \mathtt{stack} = C & \textit{formulae atoms} \\
\Phi & ::= & \phi \wedge \ldots \wedge \phi & \textit{formulae} \\
\Pi & ::= & \{\dot{C} \mapsto x \overset{(\!|c}{=} x, \ldots\} & \textit{search paths}
\end{array}
$$

5.3 Lookup, finally

In the definition of lookup, we use the following notational abbreviations and auxiliary definitions:

- We may pun $\Phi = \phi_1 \wedge \ldots \phi_n$ equivalently as the set of its atomic conjunctions, $\Phi = \{\phi_1, \ldots, \phi_n\}$.
- We define isSAT(Φ) to hold if there is a satisfying assignment for Φ.
- We define SATs(Φ) to be the set of all satisfying assignments M mapping (annotated) variables in Φ to values v.
- We substitute annotated variable sets for annotated variables to abbreviate set comprehensions. For instance, $(\mathcal{X} = {}^{\dot{C}}x)$ abbreviates the set $\{{}^{\dot{C}}x' = {}^{\dot{C}}x \mid {}^{\dot{C}}x' \in \mathcal{X}\}$.
- We overload $\overset{\mathrm{S}}{\mathbb{L}}(G, X, a_0, \dot{C}, \Phi, \Pi)$ as a predicate, to mean $\overset{\mathrm{S}}{\mathbb{L}}(G, X, a_0, \dot{C}, \Phi, \Pi) \neq \emptyset$.
- Function Stackize($C?[]$) = Reverse(C): 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 normal stack and co-stack grow oppositely, the co-stack at the top needs to be reversed to obtain the stack at the start point. This function is undefined if the stack portion of the pair is non-empty, a condition which never arises at the top level of the program (where the first variable is declared). We add constraint form $\mathtt{stack} = C$ for this; these constraints fix what the initial stack must have been at program start, converting the final relative stack into the (now-known) concrete stack, C.

Definition 5.1. Given control flow graph G, Φ with isSAT(Φ) holding, and path mapping Π, DDSE variable lookup, $\overset{\mathrm{S}}{\mathbb{L}}(G, X, a_0, \dot{C}, \Phi, \Pi)$, is the function returning the least set of annotated variables $\mathcal{X}$ satisfying the following conditions given some $a_1 \ll a_0$:

(1) $\boxed{\text{Value Discovery}}$
If $a_1 = (x = v)$, $X = [x]$, then ${}^{\dot{C}}x \in \mathcal{X}$ and $({}^{\dot{C}}x = v) \in \Phi$,
provided (1) if x = FirstV(e) then $(\mathtt{stack} = \text{Stackize}(\dot{C})) \in \Phi$, and (2) if $x \neq$ FirstV(e), then $\overset{\mathrm{S}}{\mathbb{L}}(G, [\text{FirstV}(e)], a_1, \dot{C}, \Phi, \Pi)$.

(2) $\boxed{\text{Input}}$
If $a_1 = (x = \mathtt{input})$ and $X = [x]$, then ${}^{\dot{C}}x \in \mathcal{X}$ and $\varsigma_{\mathtt{true}} = ({}^{\dot{C}}x = {}^{\dot{C}}x) \in \Phi$,
provided (1) if x = FirstV(e) then $(\mathtt{stack} = \text{Stackize}(\dot{C})) \in \Phi$, and (2) if $x \neq$ FirstV(e), then $\overset{\mathrm{S}}{\mathbb{L}}(G, [\text{FirstV}(e)], a_1, \dot{C}, \Phi, \Pi)$.

(3) $\boxed{\text{Value Discard}}$
If $a_1 = (x_1 = v)$ and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $\overset{\mathrm{S}}{\mathbb{L}}(G, [x_2, \ldots, x_n], a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

(4) $\boxed{\text{Alias}}$
If $a_1 = (x = x')$ and $X = [x] \,||\, X'$ then $\overset{\mathrm{S}}{\mathbb{L}}(G, [x'] \,||\, X', a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

(5) $\boxed{\textsc{Binop}}$
If $a_1 = (x = x' \odot x'')$, $X = [x] \,||\, X'$, $\mathcal{X}' = \mathbb{L}^{S}(G, [x'], a_1, \dot{C}, \Phi, \Pi)$, and $\mathcal{X}'' = \mathbb{L}^{S}(G, [x''], a_1, \dot{C}, \Phi, \Pi)$, then $\mathbb{L}^{S}(G, [\textsc{FirstV}(e)], a_1, \dot{C}, \Phi, \Pi)$, provided that (1) ${}^{\dot{C}}x = \mathcal{X}' \odot \mathcal{X}'' \in \Phi$ for some ${}^{\dot{C}}x \in \mathcal{X}$, and (2) if $X' \neq []$, then $\mathbb{L}^{S}(G, X', a_1, \dot{C}, \Phi, \Pi)$.

(6) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (x \overset{\llparenthesis c}{=} x')$, $X = [x] \,||\, X'$, $\dot{C}' = \textsc{Pop}(\dot{C}, c)$, $c = (x_r = x_f\ x_a)$, $\mathcal{X}_f = \mathbb{L}^{S}(G, [x_f], c, \dot{C}', \Phi, \Pi)$, $\Pi(\dot{C}) = (x \overset{\llparenthesis c}{=} x')$, and $(\mathcal{X}_f = \texttt{fun}\ x \texttt{ -> (} e \texttt{)}) \in \Phi$,
then $\mathbb{L}^{S}(G, [x'] \,||\, X', a_1, \dot{C}', \Phi, \Pi) \subseteq \mathcal{X}$.

(7) $\boxed{\textsc{Function Enter Non-Local}}$
If $a_1 = (x'' \overset{\llparenthesis c}{=} x')$, $X = [x] \,||\, X'$, $\dot{C}' = \textsc{Pop}(\dot{C}, c)$, $x'' \neq x$, $c = (x_r = x_f\ x_a)$, $\mathcal{X}_f = \mathbb{L}^{S}(G, [x_f], c, \dot{C}', \Phi, \Pi)$, $\Pi(\dot{C}) = (x'' \overset{\llparenthesis c}{=} x')$, and $(\mathcal{X}_f = \texttt{fun}\ x \texttt{ -> (} e \texttt{)}) \in \Phi$,
then $\mathbb{L}^{S}(G, [x_f, x] \,||\, X', a_1, \dot{C}', \Phi, \Pi) \subseteq \mathcal{X}$.

(8) $\boxed{\textsc{Function Exit}}$
If $a_1 = (x \overset{\rrparenthesis c}{=} x')$, $X = [x] \,||\, X'$, $c = (x_r = x_f\ x_a)$, $\mathcal{X}_f = \mathbb{L}^{S}(G, [x_f], c, \dot{C}, \Phi, \Pi)$, $x' = \textsc{RetV}(e)$, and $(\mathcal{X}_f = \texttt{fun}\ x \texttt{ -> (} e \texttt{)}) \in \Phi$,
then $\mathbb{L}^{S}(G, [x'] \,||\, X', a_1, \textsc{Push}(\dot{C}, c), \Phi, \Pi) \subseteq \mathcal{X}$.

(9) $\boxed{\textsc{Skip}}$
If $a_1 = (x'' = b)$, $X = [x] \,||\, X'$, $x'' \neq x$, and $\mathbb{L}^{S}(G, [x''], a_0, \dot{C}, \Phi, \Pi)$,
then $\mathbb{L}^{S}(G, X, a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

(10) $\boxed{\textsc{Conditional Top}}$
If $a_1 = v \llparenthesis c$, $c = (x = x_1\ ?\ e_1 : e_2)$, $\mathcal{X}_1 = \mathbb{L}^{S}(G, [x_1], c, \dot{C}, \Phi, \Pi)$, and $(\mathcal{X}_1 = v) \in \Phi$,
then $\mathbb{L}^{S}(G, X, a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

(11) $\boxed{\textsc{Conditional Bottom - True}}$
If $a_1 = (x \overset{\rrparenthesis c}{=} x')$, $c = (x = x_1\ ?\ e_1 : e_2)$, $X = [x] \,||\, X'$, $\textsc{RetV}(e_1) = x'$, $\mathcal{X}_1 = \mathbb{L}^{S}(G, [x_1], c, \dot{C}, \Phi, \Pi)$, and $(\mathcal{X}_1 = \texttt{true}) \in \Phi$,
then $\mathbb{L}^{S}(G, [x'] \,||\, X', a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

(12) $\boxed{\textsc{Conditional Bottom - False}}$
If $a_1 = (x \overset{\rrparenthesis c}{=} x')$, $c = (x = x_1\ ?\ e_1 : e_2)$, $X = [x] \,||\, X'$, $\textsc{RetV}(e_2) = x'$, $\mathcal{X}_1 = \mathbb{L}^{S}(G, [x_1], c, \dot{C}, \Phi, \Pi)$, and $(\mathcal{X}_1 = \texttt{false}) \in \Phi$,
then $\mathbb{L}^{S}(G, [x'] \,||\, X', a_1, \dot{C}, \Phi, \Pi) \subseteq \mathcal{X}$.

Understanding The Clauses of Symbolic Lookup. The clauses of Definition 5.1 closely mirror those of the concrete $\omega\iota$DDPAc lookup function in Definition 4.2. However, instead of returning a value v as the result we return the defining variable of the value, allowing us to return *symbolic* constraints (which will be in Φ) instead of concrete values. For example, the $\boxed{\textsc{Value Discovery}}$ rule directly returns the defining variable ${}^{C}x$. This defining variable may then be used by other rules such as the $\boxed{\textsc{Binop}}$ rule, which invokes lookup on both operator parameters and uses the defining variables to build the equation constraining the binary operator behavior in Φ. By using stack-indexed variable definitions ${}^{C}x$ in the Φ constraints, we can be guaranteed that there are no collisions of different activations of the same variable. Lemma 3.3 establishes this.

The $\boxed{\textsc{Function Enter} \ldots}$ 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 frame is added to the co-stack by the $\textsc{Pop}$.

The $\boxed{\textsc{Conditional Bottom} \ldots}$ rules may both match 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.

Section 2.2 informally traces some examples through this definition; with the formal definition the results of that example can be confirmed.

5.4 Defining test generation and showing computability

This section uses the above symbolic lookup function to formally define a function $\mathbb{T}$ which generates a test exercising a particular line of code. We show $\mathbb{T}$ to be partially recursive and complete: if an input sequence exists which will test a particular line of code, $\mathbb{T}$ will find it.

We begin by observing a bound on the sets produced by lookup:

Lemma 5.2. $\overset{S}{\mathbb{L}}(G, [x], a, []?[], \Phi, \Pi)$ *is deterministic: the result is either a singleton or empty set.*

Proof. By induction on the clauses of Definition 5.1. □

Lookup yields a singleton set if the line of code is reachable and an empty set if it is not. A line may be unreachable because (1) it is dead code, i.e. no input values will exercise that line; (2) the program diverged before reaching the relevant line; or (3) there was a run-time type error. We are primarily concerned with singleton sets, as a singleton set corresponds to a viable input sequence.

The goal of test generation is to find inputs exercising a particular line (clause) in the program. We will formally define it as looking up the first variable in the program from that clause.

Definition 5.3. We write $\overset{S}{\mathbb{L}}(G, a, \dot{C}, \Phi, \Pi)$ to mean $\overset{S}{\mathbb{L}}(G, [\textsc{FirstV}(e)], a, \dot{C}, \Phi, \Pi)$. We analogously define $\overset{\omega}{\mathbb{L}}(G, a, C, \iota)$ for the interpreter.

We now define the test generation function itself:

Definition 5.4. Given expression e, a complete CFG G for e, and a clause a in e, we say $\mathbb{T}(e, a, G, \Phi, \Pi)$ holds if $\overset{S}{\mathbb{L}}(G, a, []?[], \Phi, \Pi)$ is non-empty. We say e, G, a is *coverable* iff $\mathbb{T}(e, a, G, \Phi, \Pi)$ holds for some Φ, Π.

Often the e and G will be fixed in context, and so we may then assert a program point a is *coverable* (or not) as shorthand.

The above definition is a predicate and does not produce program inputs ι directly. However, an ι may always be constructed from Φ; in particular, the SAT solver can produce such an input mapping. Formally, we say ι satisfies the constraints Φ if $\textsc{isSAT}(\Phi \cup \{{}^{\dot{C}}x = v \mid {}^{\dot{C}}x \mapsto v \in \iota\})$; that is, Φ is not inconsistent with the mapping ι. We first observe that, for any consistent Φ, such a mapping always exists:

Lemma 5.5. *Given expression e and complete CFG G for e and clause a, if* $\mathbb{T}(e, a, G, \Phi, \Pi)$ *for some* Φ, Π *then there exists an ι such that ι satisfies* Φ.

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

Lemma 5.6. *Given e, a and complete G, Finding a* Φ, Π *for which* $\mathbb{T}(e, a, G, \Phi, \Pi)$ *holds is recursively enumerable.*

Proof. The space of (Φ, Π) pairs is trivially recursively enumerable since each set is r.e. and pairing preserves r.e. Lookup is recursively enumerable as it is a non-deterministic recursive function with boundedly finite branching, so dovetailing lookups over the Φ, Π enumeration will enumerate the valid Φ, Π. □

Lemma 5.6 demonstrates that, if a a in some e, G is coverable, then $\mathbb{T}$ will eventually find a suitable input sequence. While we assume the control flow graph G to have been computed, any complete G (including the trivially computable case in which all calls are wired to all call sites) suffices. While the enumeration strategy presented here is perhaps underwhelming – a naive implementation would run out of time or memory and the worst-case complexity is the same as brute-force dovetailing through all inputs on a standard interpreter – Lemma 5.6 leads us to the observation that *we can find suitable test inputs by working backward from the destination.* In practice, this allows us to constrain our search by observing operations on data (rather than using heuristics to predict which control flow paths will ultimately lead us to our goal) and should lead to more efficient implementations. We discuss in Section 6.1 the approach we use to search more efficiently for input coverings.

5.5 Correctness

Here we show that DDSE is fully and faithfully modeling the demand interpreter $\omega\iota$DDPAc from the previous section (which was itself shown to be equivalent to a standard substitution-style interpreter in Lemma 4.7).

5.5.1 Concretizing the relative stack. In our first step toward showing correctness, we will replace the relative stacks used by symbolic lookup with concrete stacks to align them with the stacks in $\omega\iota$DDPAc. We formalize the notation $|\dot{C}|@C = C'$ to mean the normalization of a relative stack $\dot{C}$ to its concrete equivalent C' given the stack at the starting point in retrospect was learned to be C.

Definition 5.7. $|C_c?C_s|@C = C_s \,||\, C'$ where $C = \textsc{Reverse}(C_c) \,||\, C'$; this operator is undefined if the latter equation fails for all C'. Let $|{}^{\dot{C}}x|@C = {}^{|\dot{C}|@C}x$, let $|\mathcal{X}|@C = \{|{}^{\dot{C}}x|@C \mid {}^{\dot{C}}x \in \mathcal{X}\}$, and let $|\Phi|@C = \Phi[(|\mathcal{X}|@C)/\mathcal{X}]$, for $\mathcal{X}$ being the set of all variables in Φ.

For example, $|[\mathsf{a}]?[\mathsf{b}]|@[\mathsf{a}, \mathsf{c}] = [\mathsf{b}, \mathsf{c}]$. We demonstrate that, using this operation, we can concretize the arguments of a lookup such that all formulae and variables use concrete stacks rather than relative stacks.

Lemma 5.8. *If* ${}^{\dot{C}}x \in \overset{S}{\mathbb{L}}(G, X, a_0, \dot{C}_0, \Phi, \Pi)$ *and* $(\mathtt{stack} = C) \in \Phi$ *then* $|\dot{C}|@C$ *is defined and* ${}^{|\dot{C}|@C}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, |\Phi|@C, \Pi)$.

5.5.2 Eliminating the search path. Now that we have a concrete stack, we no longer need the argument Π: the call site invoking a function is always on the call stack. Formally, we define $\Pi^{\max}$ as the maximal mapping from each $({}^{\dot{C}}x)$ to every $x \overset{\lessdot c}{=} x'$ in the CFG. Since this mapping is maximal, all of the Π conditions in $\overset{SC}{\mathbb{L}}$ will vacuously hold, neutralizing them. We then may easily show the following.

Lemma 5.9. ${}^{\dot{C}}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi)$ *iff* ${}^{\dot{C}}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi^{max})$.

5.5.3 Constructing values from constraints. By this point, we have removed two key differences between the concrete and symbolic lookup: the relative stack $\dot{C}$ and the search path Π. The only remaining differences are the constraints Φ and the fact that DDSE lookup returns a (constrained) variable rather than a value.

Since the equational constraints of DDSE have the same actual equivalences in $\omega\iota$DDPAc and by Lemma 3.3, we know that ${}^{C}x$ variable names will never collide. In the following, we rely upon this non-collision property to implicitly convert between ${}^{C}x$ (stack annotations) and ${}^{[]?C}x$ (stack with empty co-stack annotations) when relating the two lookup functions. Since the co-stack is always empty, we will use ${}^{C}x$ below for both systems. Recall that notation used below is defined in Section 5.3.

Lemma 5.10.

(1) *If* ${}^{C}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi^{max})$ *then for all* $M \in \text{SATs}(\Phi)$*, if* ${}^{C}x \mapsto v \in M$ *then* $v \in \overset{S}{\mathbb{L}}(G, X, a_0, C, M)$.

(2) $v \in \overset{\omega}{\mathbb{L}}(G, X, a_0, C, \iota)$ *implies* ${}^{C}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi^{max})$ *for some* Φ *such that for some* $M \in \text{SATs}(\Phi)$*,* $\iota \subseteq M$ *and* ${}^{C}x \mapsto v \in M$.

Proof. For (1), proceed by induction on the depth of the definition of ${}^{C}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi^{\max})$ and pick arbitrary M with ${}^{C}x \mapsto v \in M$. Proceed by cases of which clause matched in $\overset{S}{\mathbb{L}}$. For the base case of $\boxed{\text{Value Discovery}}$, $v \in V$ is immediate by the $\overset{S}{\mathbb{L}}$ requirements that ${}^{C}x \in \mathcal{X}$ and $({}^{C}x = v) \in \Phi$. All other cases have similar direct analogues between the two definitions, symbolic returning a variable and requiring a constraint, and concrete inlining that constraint. Observe that the co-stack is always empty and the stack operations coincide in this case.

For (2), proceed by induction on the depth of the definition of $v \in \overset{\omega}{\mathbb{L}}(G, X, a_0, C, \iota)$ to both incrementally construct a Φ/M and show ${}^{C}x \in \overset{S}{\mathbb{L}}(G, X, a_0, []?C, \Phi, \Pi^{\max})$. In other words, we strengthen the induction hypothesis to also assume a Φ_0 and M_0 has been constructed in each inductive reference, and will produce a new extended Φ and M. For the $\boxed{\text{Value Discovery}}$ base case we simply construct $\{({}^{C}x = v)\} = \Phi$ and $\{({}^{C}x \mapsto v)\} = M$. Considering the $\boxed{\text{Input}}$ case, it only adds $\varsigma_{\texttt{true}} = ({}^{C}x = {}^{C}x)$ to Φ and nothing about actual input v, but since it is unconstrained (other than being an integer by the previous) we can add ${}^{C}x \mapsto v$ to M and it will be a satisfying assignment. This also preserves the requirement that $\iota \subseteq M$. The other cases proceed similarly. □

From the chain of Lemmas defined above, we may now directly conclude that test generation is sound and complete.

Lemma 5.11. *Test generation is* sound *and* complete*:*

- *If* $\mathbb{T}(e, a, G, \Phi, \Pi)$ *then* $\overset{\omega}{\mathbb{L}}(G, a, C, \iota)$ *is non-empty for some* ι *satisfying* Φ*, and some* C.
- *If* $\overset{\omega}{\mathbb{L}}(G, a, C, \iota)$ *is non-empty for some* ι *and* C*, then* $\mathbb{T}(e, a, G, \Phi, \Pi)$ *holds for some* Φ *satisfied by* ι*, and some* Π.

6 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 outline its performance on some known challenging test generation examples.

6.1 Methodology

Definition 5.1 gives the lookup function which symbolically executes a program in a demand-driven fashion. This definition is declarative, treating the logical formulae Φ and the search path Π as oracular, to improve readability. For a feasible implementation, however, we must construct Φ and Π as we search for control flows which satisfy the lookup rules. Designing this algorithm presented three key challenges – nondeterminism, nontermination, and caching – which we discuss here.

Our implementation searches for paths through a program to a desired destination by moving backward through the control flow graph and applying the rules of Definition 5.1 as possible. 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 Φ containing unsatisfiable formulae) are discarded. This application of nondeterminism is common in proof

Benchmark	Time	Cache size
`ack`	0.86s	2564
`tak`	44.12s	14092
`cpstak`	11.61s	8374
`blur`	0.12s	364
`facehugger`	0.19s	840
`ack'`	63.60s	5626
`tak'`	timeout (600s)	
`cpstak'`	timeout (600s)	
`blur'`	timeout (600s)	
`facehugger'`	3.65s	2113

Table 1. DDSE Scheme benchmarks

`mt`	`nk`	Time	Cache size	Inferred input
1	1	2.16s	2277	[1, 2]
2	1	33.54s	4608	[2, 2]
3	1	timeout (600s)		
1	2	7.74s	3042	[1, 4]
2	2	50.36s	4705	[2, 4]
3	2	timeout (600s)		
1	3	218.21s	7616	[1, 8]
2	3	253.28s	7616	[2, 8]
3	3	timeout (600s)		
1	4	timeout (600s)		

Table 2. DDSE `backotter` C benchmark

searching algorithms, transitive closures, and similar domains ; however, nondeterminism is nontrivial to combine with other features of computation [Zwart and Marsden 2018].

One example of this poor interaction is with nonterminating computations. As stated in Section 5.4, lookup is recursively enumerable (Lemma 5.6) 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, we observe that most lookup rules include multiple child lookups and that the descendants of those children often overlap. In the Skip Rule, for instance, the lookup of both x'' and x will often perform the same operation on a_1's predecessors. 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. Of note, 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 4.07.1 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 5.1 as straight-line imperative code. This design permits additional language features to be supported with minimal effort.

6.2 Preliminary Evaluation

We have performed a preliminary evaluation of our reference implementation using some standard Scheme benchmarks[5] as well as some C examples challenging for symbolic execution [Ma et al. 2011]. The C examples do not use higher-order programming, but they are a good stress test of our

[5]From http://www.larcenists.org/benchmarksAboutR7.html and https://bitbucket.org/ucombinator/p4f-prototype/src/master/benchmarks/.

implementation. Our implementation does not support data structures or mutable state, limiting the number of existing benchmarks which it can operate on in it's current state.

The formal ANF syntax of Figure 5 is difficult to code in, so we implemented a translator which allows direct coding in an ML-like syntax; this syntax is seen in the Section 2 examples. Recursive functions or loops appearing in the original Scheme and C benchmarks are encoded via self-application in the ANF.

The Scheme benchmarks supported by our implementation include `ack` (Ackermann function), `tak` (Takeuchi function), `cpstak` (CPS-converted tak), `blur`, and `facehugger`. We modified the benchmarks to use `inputs` instead of hard-coded constants. The first part of Table 1 shows the time required to find control flow to the top of each benchmark's recursive calls; DDSE finds these cases very quickly. The cache size numbers indicate the number of different lookup operations requested (identified by variable, program point, and stack); some of these lookups may have been in-progress and not yet complete when a correct input was found.

We also evaluated DDSE by finding control flow to the *bottom* of each benchmark's recursive calls; the `ack'` benchmark, for instance, shows the time required to find control flow to the `target'` variable in Figure 10. The `tak'`, `cpstak'`, and `blur'` benchmarks time out. These cases contain numerous recursive call sites within the same function and we believe the timeout to be a weakness in the current implementation as it pursues sublookups without respect to the accumulated constraints of the parent lookups which depend upon them.

```
1  let xi = input
2  in let yi = input
3  in let rec ack m n =
4    if m == 0
5    then
6        n+1
7    else
8      if n == 0
9      then
10         ack (m-1) 1
11     else
12       let target = 0 in
13       let r =
14          ack (m-1) (ack m (n-1)) in
15       let target' = 0 in r
16 in
17   if 0 <= xi and 0 <= yi
18     then ack xi yi
19     else 0
```

Fig. 10. Ackermann function

For the C examples, we ran two synthetic programs from a C-based symbolic interpreter [Ma et al. 2011]: `pro-backotter-3.c` (Figure 4 in that paper) and `pro-mix-1.c` (Figure 6 in that paper); here we only focus on `pro-backotter-3.c`. The corresponding ML code we used appears in Figure 11. This program takes integer inputs `m` and `n`. The `main_loop` function in the C code is a for-loop from 0 to 1000; it calls `f` when the loop index equals input `m`. Function `f` in C starts with a for-loop from 0 to 6 which returns 0 only when the last six bits of the input `n` are zero. After the for-loop is a while-true-loop which aborts only if the number of 1's is 0 and `m` equals the constant `mk`, 7 here. Table 1 of [Ma et al. 2011] shows how standard forward symbolic interpreters get stuck on this example, since they will spend most of their work time stuck in the infinite while-loop.

The backward execution starts at the `target` variable in line 12. It immediately infers the goal formula that `m` must equal `mt`, so the outer for-loop search will be bounded by `mt` and not 1000. The inner for-loop needs to be fully unwound to determine how `nt` constrains `n`. The search strategy prioritizes relative stacks with the least repetition, which will allow the search to escape the while-true-loop quickly. So, we avoid the pitfalls of standard forward symbolic execution.

Our current implementation times out after over an hour on the Figure 11 example as written, but if we reduce the value of `nk` and `mt` we can successfully generate input examples. Table 2 shows the results for various values of `nk` and `mt`. Changing the for-loop index from 1000 does not affect our results as the loop only unrolls on demand and only needs `mt` unrollings total.

While the wall times of this reference implementation are slow, the size of the lookup cache is not growing unreasonably. Our reference implementation is intelligent in that avoids getting stuck in extremely deep search paths, but has a very high overhead since the lookup function is more or less implemented directly as specified in Definition 5.1. For example, looking up a variable defined in the current basic block is achieved by manually checking each previous line in turn for its definition; since the definition point is lexically known this is very wasteful.

```
let f m n =
  let rec f_loop n sum i =
  let nk = 6 in
  if i == nk
    then sum
    else (
      let a = n % 2
      in let sum_next = sum +
       (if a == 0 then 0 else (a + 1))
      in f_loop (n/2) sum_next (i+1))
  in let loop_result = f_loop n 0 0
  in let rec f_while c =
    let mt = 7 in
    if (loop_result == 0) and (m == mk)
    then let target = false in 1
    else (
      if c
        then f_while c
        else 0)
  in f_while true
in let rec main_loop i =
  if i == 1000
  then 0
  else (
    let _ =
      if m == i
      then f m n
      else 0
    in main_loop (i+1))
in main_loop 0
```

Fig. 11. Translated backotter

7 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) [Chandra et al. 2009; Charreteur and Gotlieb 2010; Dinges and Agha 2014]. 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. 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 $\mathbb{L}^{\mathbb{S}}$ 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 5.1), and it is a direct generalization of a non-symbolic interpreter (Definition 4.2) to symbolic data.

The demand-driven interpreter which we symbolize here is based on the ωDDPAc interpreter of [Facchinetti et al. 2019; Palmer and Smith 2016], where it was developed solely to show soundness of a demand-driven program analysis.

While we address 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; Palmer and Smith 2016].

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. DDSE complements these in that it is more suited to goal-directed reasoning.

8 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 several fronts on which this approach can 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: it's goal-directed nature naturally aligns with theorem provers.

REFERENCES

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).

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 Charreteur 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 International Conference on Functional Programming (ICFP '00).* ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/351240.351266

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

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

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

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.