Higher-Order Demand-Driven Symbolic Evaluation

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

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

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

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

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

ACM Reference Format:
Zachary Palmer, Theodore Park, Scott Smith, and Shiwei Weng. 2020. Higher-Order Demand-Driven Symbolic Evaluation. *Proc. ACM Program. Lang.* 4, ICFP, Article 102 (August 2020), 53 pages. https://doi.org/10.1145/3408984

1 INTRODUCTION

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

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

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

2475-1421/2020/8-ART102
https://doi.org/10.1145/3408984

SBEs have been developed for imperative languages; examples include [Chandra et al. 2009; Charreteur and Gotlieb 2010; Dinges and Agha 2014; Ma et al. 2011]. These reverse techniques are useful for goal-directed reasoning about paths leading to a particular program point: if a condition at a program point can be propagated back to the program start, this will deduce its validity. The 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. To be clear: SBE does not single-handedly solve the problem of symbolic execution performance, but it is a fundamentally different approach that has advantages in some contexts.

These imperative language systems do not directly generalize to functional languages. Functional languages have a combination of non-local variables and a control flow that can itself depend on (function) data flow which makes this gap non-trivial. In this paper, we develop DDSE: a **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 possible to formally prove DDSE is correct, something not previously proven for any SBE. In order to show applicability of DDSE, we develop a theory and implementation of test generation for a functional language. While the paper focuses on the test generation application to show that concrete results are possible, DDSE is also applicable to other goal-directed problems that SBEs can address.

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

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

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.

The goal of this paper is to show how a demand-driven symbolic evaluator can be developed for higher-order functional languages. Weakest precondition logic was designed for first-order stateful programs, and we aim to design a similar reverse propagation for functional programs.

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

2.1 Demand-Driven Functional Evaluators

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

```
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

For the `f y` call on line 5, a standard evaluator would pass in the actual value `0` by some means. In our truly demand-driven evaluator, however, the body `x + 1` executes *without any binding* for `x`, only knowing that the function was called from line 5. When `x`'s value is needed in the body to add 1 to it, we rely on the fact that the call site `fy` was recorded. (We use unique variables in our A-normalized programs, so variable definitions serve to uniquely identify program points.) We know that `x`'s value will take on the value of `y` at that call site, which in turn can be seen to be `0`. Although there is a call `f 1` on line 6, we know that our parameter `x` does not have the argument value 1 here because the call site `fy` (and not the call site `f1`) was 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 will formally define a variable lookup assertion $\mathbb{L}([x], n, C) \equiv v$ to mean that v is the result of a lookup of variable x, starting the (reverse) search from program line n, and assuming the current call site stack context is C. We will pun this relation as a function since lookup is deterministic, writing it as a function $\mathbb{L}([x], n, C)$ returning v equivalently. The $[x]$ is just a singleton list; it can in general be a non-singleton for looking up non-local variables, a topic we address shortly in Section 2.2. The call stack C is not the forward execution stack, since we are *wp*-style walking the program in reverse; it merely denotes the calls entered and not yet exited in this *reverse*-order sequence. We now illustrate lookup in detail by showing how the value of `ret` in the above program is looked up from the program end and empty call stack (since we are starting outside any function calls), $\mathbb{L}([\texttt{ret}], 7, [])$.

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

Fig. 2. Simple demand-driven evaluation example

[1] This section uses an informal OCaml-like syntax. In our formal presentation below, we A-normalize our programs to clarify operator ordering; we give the formal grammar for our A-normalized form (ANF) in the following section.

(1) $\mathbb{L}([\texttt{ret}], 7, []) \equiv \mathbb{L}([\texttt{fy}], 6, []) + \mathbb{L}([\texttt{f1}], 6, [])$: Line 7 defines `ret` in terms of two other variables, so we now have two lookup sub-goals to find `ret`'s value: looking up `fy` and `f1` starting from the previous line. We will trace only `fy` in this example since `f1` is similar.
(2) $\mathbb{L}([\texttt{fy}], 6, []) \equiv \mathbb{L}([\texttt{fy}], 5, [])$: We may skip over `f1`'s definition as we do not find `fy` on that line, making it irrelevant to `fy`'s lookup. (An astute reader may notice that skipping is not sound if the call does not terminate, so skipping is not always correct; we address this case at the end of this subsection.) In line 5, we find the definition of `fy` is not yet a value but is a function call: `fy`'s value is in fact the *result* of the application `f y`. To obtain value, we first need to look up the definition of `f`.
(3) $\mathbb{L}([\texttt{f}], 2, []) \equiv$ `fun x -> ...`. We have looked up the definition of `f`, so now we need to search for the *result value* of the function body, the contents of `fret`. We perform the lookup $\mathbb{L}([\texttt{fret}], 4, [\texttt{fy}])$, pushing the call site `fy` onto the call stack since the search has entered that function body.[2]
(4) $\mathbb{L}([\texttt{fret}], 4, [\texttt{fy}]) \equiv \mathbb{L}([\texttt{x}], 3, [\texttt{fy}]) + 1$: When we perform the lookup, we see that `fret` is defined in the current line as the expression `x + 1`, so we next lookup `x` from the previous line.
(5) $\mathbb{L}([\texttt{x}], 3, [\texttt{fy}]) \equiv \mathbb{L}([\texttt{y}], 2, [])$: `x` is immediately seen as a parameter to the `f`, so we want to look up the *value* of this parameter at the original call site. We know to examine call site `fy` since it was recorded on the call stack. So we induce a lookup of the argument `y` from the main program line right before `fy`, which is line 2.
(6) $\mathbb{L}([\texttt{y}], 2, []) \equiv \mathbb{L}([\texttt{y}], 1, []) \equiv \texttt{0}$: The lookup fails to find `y` on line 2, so we skip to line 1, where `y` is observed to be `0`.
(7) Now that we have found our values, we can pop off lookup obligations. We have seen that $\mathbb{L}([\texttt{x}], 3, [\texttt{fy}]) \equiv \mathbb{L}([\texttt{y}], 2, []) \equiv \texttt{0}$, so $\mathbb{L}([\texttt{fret}], 3, [\texttt{fy}]) \equiv \texttt{0 + 1} = 1$, so $\mathbb{L}([\texttt{fy}], 6, []) \equiv 1$.
(8) A similar lookup of `f1`, yields $\mathbb{L}([\texttt{f1}], 6, []) \equiv 2$, so $\mathbb{L}([\texttt{ret}], 7, []) \equiv \texttt{1 + 2} = 3$.

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

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

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

Fig. 3. Non-local variable example

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

2.2 Non-Local Variables

Non-local variables are variables *used* in a function but whose *definitions* lie outside of that function; they must be placed in closures in standard functional language evaluators. To support non-local variable lookup in a reverse evaluator, we cannot rely on closures as they are a forward-passed structure. So, we instead use a stack to record the chain of call frames we need to walk back through to find where a non-local variable is local. This process is related to access links in compiler implementations of non-local variable lookup. This is why the variable looked up in the previous examples were singleton lists [x] – in the general case it can be multiple function definitions followed by the variable, $X = [\mathtt{f}_1, \ldots, \mathtt{f}_n, \mathtt{x}]$ and lookup is generally $\mathbb{L}(X, n, C) = v$. The lookup stack gets longer than two elements when the non-locals are themselves functions – the stack in effect is increasing as we climb the type hierarchy of non-local atomic values, functions, functionals, functions on functionals, etc.

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

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

The above examples give a basic idea of the process; for even deeper lexically nested variables there will be further chaining back through a series of function definitions. For recursive functions, definable here via self-passing, the context stack may grow unboundedly but there is no need for any special handling in the lookup definition. The demand-driven evaluator described here is novel

but is based on ideas in [Facchinetti et al. 2019] (the ωDDPAc evaluator there), where it was used for the purpose of proving a program analysis sound.

2.3 Demand-Driven Symbolic Execution: DDSE

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

2.3.1 Constraint-Based execution. First we show how the demand-driven evaluator can be extended to a symbolic demand-driven evaluator. The basic idea is simple: to accumulate all constraints on variable values in a global formula Φ which must remain satisfiable, and to define lookup to return a *variable* over which constraints are constructed. If a variable is directly receiving input, it obviously can't have a concrete value. But additionally, any variable *depending* on input cannot have a concrete value; so, for uniformity, the symbolic evaluator always produces (constrained) variables. Still, there are subtleties on how to name variables given there may be many activations of the same variable at runtime. Fortunately, the pair of variable name and current call site stack serves as a unique reference into the runtime heap, assuming that the initial program had any duplicate variable definitions renamed (i.e. the program was alphatized). This property is formally established in the Supplementary Appendix as Lemma C.8. We use notation ${}^{C}x$ 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 (equivalently, their heap locations) and not their uses. Every variable use is invariably a chain of variable aliases back to a definition of the variable; such chains may go through function calls. The symbolic lookup relation is of the form $\mathbb{L}^{\mathbb{S}}([x], n, C) \equiv {}^{C_0}x_0$, the S standing for "symbolic". Analogously with concrete lookup, we will equivalently write this in function form as $\mathbb{L}^{\mathbb{S}}([x], n, C)$ returning ${}^{C_0}x_0$. We will additionally produce a global set of constraints Φ over all lookups which is an implicit result.

Let us re-evaluate the Figure 2 example symbolically to illustrate the differences, specifically by looking up the variable `ret` from line 7.

(1) $\mathbb{L}^{\mathbb{S}}([\texttt{ret}], 7, []) \equiv {}^{[]}\texttt{ret}$ and ${}^{[]}\texttt{ret} = \mathbb{L}^{\mathbb{S}}([\texttt{fy}], 6, []) + \mathbb{L}^{\mathbb{S}}([\texttt{f1}], 6, [])$: We proceed with looking up `fy` and `f1` to find their variable definitions, which we need in order to complete the second equation constraining ${}^{[]}\texttt{ret}$. We trace only $\mathbb{L}^{\mathbb{S}}([\texttt{fy}], 6, [])$ since looking up `f1` is similar.
(2) $\mathbb{L}^{\mathbb{S}}([\texttt{fy}], 6, []) \equiv \mathbb{L}^{\mathbb{S}}([\texttt{fret}], 4, [\texttt{fy}])$: Similar to how we looked up `fy` in the demand evaluator, we skip `f1`'s definition and find that we need to look up `f`'s result value `fret`.
(3) $\mathbb{L}^{\mathbb{S}}([\texttt{fret}], 4, [\texttt{fy}]) \equiv {}^{[\texttt{fy}]}\texttt{fret}$ and ${}^{[\texttt{fy}]}\texttt{fret} = \mathbb{L}^{\mathbb{S}}([\texttt{x}], 3, [\texttt{fy}]) + 1$: We have reached a point where a concrete value is constructed, so the latter equation (after the remaining lookup has completed) will be added to Φ. Note that here, ${}^{[\texttt{fy}]}\texttt{fret}$ is the defining variable: the stack annotation `[fy]` disambiguates to mean the `fy` call site allocation of `fret`, as opposed to the `f1` allocation - we have succeeded in avoiding variable name clashes.

(4) $\mathbb{L}^{\mathbb{S}}([\mathtt{x}], 3, [\mathtt{fy}]) \equiv \mathbb{L}^{\mathbb{S}}([\mathtt{y}], 2, []) \equiv {}^{[]}\mathtt{y}$ and ${}^{[]}\mathtt{y} = \mathtt{0}$: After tracing back through the lookups, we find that we have reached a value definition with ${}^{[]}\mathtt{y} = \mathtt{0}$, which we can add to Φ without any further lookup. This also entails that both $\mathbb{L}^{\mathbb{S}}([\mathtt{y}], 2, [])$ and $\mathbb{L}^{\mathbb{S}}([\mathtt{x}], 3, [\mathtt{fy}])$ will return ${}^{[]}\mathtt{y}$.
(5) ${}^{[\mathtt{fy}]}\mathtt{fret} = {}^{[]}\mathtt{y} + 1$: Given the above results, we can construct this equation and add it to Φ. We can then return ${}^{[\mathtt{fy}]}\mathtt{fret}$ as the defining variable for `fy`.
(6) ${}^{[]}\mathtt{ret} = {}^{[\mathtt{fy}]}\mathtt{fret} + {}^{[\mathtt{f1}]}\mathtt{fret}$: Here, we complete the lookup not only for `fy`, but also for `f1`, which completes the equation in step 1 and allows us to add it to Φ.

The final constraint set Φ for this lookup (including constraints added when looking up `f1`) is:

$$\Phi = \{{}^{[]}\mathtt{ret} = {}^{[\mathtt{fy}]}\mathtt{fret} + {}^{[\mathtt{f1}]}\mathtt{fret}, {}^{[\mathtt{fy}]}\mathtt{fret} = {}^{[]}\mathtt{y} + 1, {}^{[\mathtt{f1}]}\mathtt{fret} = 1 \ + \ 1, {}^{[]}\mathtt{y} = \mathtt{0}\}$$

By basic arithmetic we can conclude that ${}^{[]}\mathtt{ret} = 3$ 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 5 below.

2.3.2 Adding Input. While relatively easy in forward evaluators, adding input in this backward evaluation model is somewhat challenging. Forward evaluators simply process input in sequence as they execute the program. In the demand-driven process described above, however, values are looked up in the (reverse) order in which they are *used* rather than the order in which they are *defined*. For example, consider the program in Figure 4. Here, the `input` keyword reads an integer off standard input. When looking up `ifret`, 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. We formally establish that the two notions of input are isomorphic.

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

Fig. 4. Input Example

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

2.3.3 Test Generation. We may finally consider how to generate a test exercising any particular line of the program. Consider again the Figure 4 input example and suppose our goal is to find inputs which reach (i.e., cover) line 7. Lookup of `fretp` from line 7 is non-trivial: this search begins inside the body of `f`, there are two call sites to `f`, and either could have been the calling point. So, a search is fundamentally required. A conservative approach would be that *any* call site could have called `f`, but a simple program analysis can build a conservative call graph to winnow out nearly all

call sites from contention. For this search process one additional global structure is present, along with formula Φ: a path choice, Π, which records which call site was chosen in the current search attempt.

Suppose here we arbitrarily guessed `f i2` in line 14 as the call site that got us to line 7; let us at a high level describe the lookup constraints 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 4.1 for details.

We began our search in the `then` branch, so we know the conditional expression to be true; we will ultimately express this with a constraint of the shape $\mathbb{L}^{\mathbb{S}}([\texttt{x}], 5, [\texttt{fi2}]) = 0$ where $\mathbb{L}^{\mathbb{S}}([\texttt{x}], 5, [\texttt{fi2}])$ 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 to determining $\mathbb{L}^{\mathbb{S}}([\texttt{i2}], 12, [])$. Since we came from the `else` branch, we will build a constraint of the shape $\mathbb{L}^{\mathbb{S}}([\texttt{i2}], 12, []) \neq 0$ to remember that this condition must have failed. Continuing with $\mathbb{L}^{\mathbb{S}}([\texttt{i2}], 12, [])$, we finally arrive at its definition in line 2. Since `i2` is defined as an arbitrary `input`, it is only constrained to be an integer. Having completed lookup, we can fill in the unresolved lookups in the previous constraints: the `else` constraint is $[]\texttt{i2} \neq 0$ (because $\mathbb{L}^{\mathbb{S}}([\texttt{i2}], 12, []) = []\texttt{i2}$) and the original `then` constraint is $[]\texttt{i2} = 0$ (since $\mathbb{L}^{\mathbb{S}}([\texttt{x}], 5, [\texttt{fi2}]) = \mathbb{L}^{\mathbb{S}}([\texttt{i2}], 12, []) = []\texttt{i2}$). These two constraints, $[]\texttt{i2} \neq 0$ and $[]\texttt{i2} = 0$, are immediately contradictory, meaning that this path will never happen and so the `fi2` call site choice can be 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 $\Phi = \{[]\texttt{i2} = 0\}$. Any input mapping conforming to these constraints, such as $\{[]\texttt{i1} \mapsto 0, []\texttt{i2} \mapsto 0, []\texttt{i3} \mapsto 0\}$, will exercise line 6, and so we have successfully deduced a test case.

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

3 A REVERSE CONSTRUCTION INTERPRETER

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

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

Definition 3.1 Notation and well-formedness.

(1) *We assume there is a fixed program e_{glob} we are working over in this section.*
(2) *We use notation $[x_1, \ldots, x_n]$ for lists and $||$ for list concatenation.*

(3) *We require that e_{glob} is closed and that each clause c in e_{glob} is uniquely identified by the variable it defines (i.e., our programs are* alphatized*).*
(4) *For technical reasons, e_{glob} cannot begin with a function definition clause. (W.o.l.o.g. the reader can assume there is a* `dummy = 0` *first clause.)*
(5) $\textsc{Cl}(x)$ *is the unique c where $c = (x = b)$ occurs in e_{glob} (there is at most one such clause in e_{glob} by alphatization).*
(6) *Clauses* `fun` x `->` *may* only *be used as the initial clause in an f.*
(7) *Clauses x `!` β may* only *appear at the beginning of a conditional expression's branch and all conditional branches must start with such a clause. Variable x must match the variable defined by the conditional; boolean value β must match the condition (*`true` *or* `false`*) used to enter the branch.*

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

```
[o = 1, y = 0, f = [fun x ->,fret = x + o], fy = f y, f1 = f o, ret = fy + f1]
```

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

```
[bv = x = 0, fret = bv ? [ fret ! true, fretp = x + 1 ] : [ fret ! false, fretm = x - 1 ]]
```

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

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

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

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

e	::=	$[c, \ldots]$	*expressions*
c	::=	$x = b \mid \texttt{fun}\ x\ \texttt{->} \mid x\ \texttt{!}\ \beta$	*clauses*
x	::=	*(identifiers)*	*variables*
b	::=	$v \mid x \mid \texttt{input} \mid x\ x$	*bodies*
		$\mid x\ \texttt{?}\ e : e \mid x \odot x$	
$\odot$	::=	`+` $\mid$ `-` $\mid$ `<` $\mid$ `=` $\mid$ `and` $\mid$ `or` $\mid$ `xor`	*binops*
v	::=	$f \mid n \mid \beta$	*values*
f	::=	$[\texttt{fun}\ x\ \texttt{->}] \,||\, e$	*functions*
n	::=	$0 \mid 1 \mid -1 \mid \ldots$	*integers*
β	::=	`true` $\mid$ `false`	*booleans*
C	::=	$[c, \ldots]$	*contexts*
X	::=	$[x, \ldots]$	*lookup stacks*
ι	::=	$\{{}^{C}x \mapsto n, \ldots, {}^{C}x \mapsto n\}$	*inputs*

Fig. 5. Language grammar and interpreter structures

[3]Lemma C.9 shows how this is a sufficient notion of freshening in a forward interpreter.

3.1 Variable value lookup

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

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

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

Definition 3.2 Clause operations.

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

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

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

Definition 3.3. *Given fixed program* e_{glob} *and input mapping* ι*, the relation* $\mathbb{L}(X, c, C) \equiv v$ *holds iff there is a proof using the rules of Figure 6.*
$\textsc{First}(x, c, C)$ *used in the Figure holds iff we can look up the very first variable in the program from the current clause/stack:* $x \neq \textsc{FirstV}(e_{glob})$ *implies* $\exists v'.\ \mathbb{L}([\textsc{FirstV}(e_{glob})], \textsc{Pred}(c), C) \equiv v' \wedge x = \textsc{FirstV}(e_{glob})$ *implies* $C = []$.

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

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

The proof appears in Appendix A.

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

The intuitions for Definition 3.3 were given in examples in Figures 2 and 3 in Section 2.1. There informally we used line numbers, and formally the clause in that line is used; additionally, for context stack elements in the overview we used the defining variable of the clause and here we

$$\textsc{Value Discovery}\ \frac{\textsc{First}(x, \textsc{Cl}(x), C)}{\mathbb{L}([x], (x = v), C) \equiv v} \qquad \textsc{Input}\ \frac{\iota(^{C}x) = v \qquad \textsc{First}(x, \textsc{Cl}(x), C)}{\mathbb{L}([x], (x = \texttt{input}), C) \equiv v}$$

$$\textsc{Value Discard}\ \frac{\mathbb{L}(X, \textsc{Pred}(x), C) \equiv v}{\mathbb{L}([x] \,||\, X, (x = f), C) \equiv v} \qquad \textsc{Alias}\ \frac{\mathbb{L}([x'] \,||\, X, \textsc{Pred}(x), C) \equiv v}{\mathbb{L}([x] \,||\, X, (x = x'), C) \equiv v}$$

$$\textsc{Binop}\ \frac{\mathbb{L}([x'], \textsc{Pred}(x), C) \equiv v' \qquad \mathbb{L}([x''], \textsc{Pred}(x), C) \equiv v''}{\mathbb{L}([x], (x = x' \odot x''), C) \equiv v' \odot v''}$$

$$\textsc{Function Enter Parameter}\ \frac{c = (x_r = x_f\ x_v) \qquad \mathbb{L}([x_v] \,||\, X, \textsc{Pred}(c), C) \equiv v \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x\ \texttt{->}] \,||\, e}{\mathbb{L}([x] \,||\, X, (\texttt{fun}\ x\ \texttt{->}), [c] \,||\, C) \equiv v}$$

$$\textsc{Function Enter Non-Local}\ \frac{x'' \neq x \qquad c = (x_r = x_f\ x_v) \qquad \mathbb{L}([x_f, x] \,||\, X, \textsc{Pred}(c), C) \equiv v \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x''\ \texttt{->}] \,||\, e}{\mathbb{L}([x] \,||\, X, (\texttt{fun}\ x''\ \texttt{->}), [c] \,||\, C) \equiv v}$$

$$\textsc{Function Exit}\ \frac{\mathbb{L}([x'] \,||\, X, (x' = b), [\textsc{Cl}(x)] \,||\, C) \equiv v \qquad \textsc{RetCl}(e) = (x' = b) \qquad \mathbb{L}([x_f], \textsc{Pred}(c), C) \equiv [\texttt{fun}\ x''\ \texttt{->}] \,||\, e}{\mathbb{L}([x] \,||\, X, (x = x_f\ x_v), C) \equiv v}$$

$$\textsc{Skip}\ \frac{x'' \neq x \qquad \mathbb{L}([x] \,||\, X, \textsc{Pred}(x''), C) \equiv v \qquad \exists v_0.\, \mathbb{L}([x''], \textsc{Cl}(x''), C) \equiv v_0}{\mathbb{L}([x] \,||\, X, (x'' = b), C) \equiv v}$$

$$\textsc{Conditional Top}\ \frac{\textsc{Cl}(x_1) = (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}) \qquad \mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv \beta \qquad \mathbb{L}(X, \textsc{Pred}(x_1), C) \equiv v}{\mathbb{L}(X, (x_1\ !\ \beta), C) \equiv v}$$

$$\textsc{Conditional Bottom}\ \frac{\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv \beta \qquad \mathbb{L}([x'] \,||\, X, (x' = b), C) \equiv v \qquad \textsc{RetCl}(e_\beta) = (x' = b)}{\mathbb{L}([x_1] \,||\, X, (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}), C) \equiv v}$$

Fig. 6. Value Lookup Rules

use the full clause. For example, the informal lookup $\mathbb{L}([\texttt{x}], 3, [\texttt{fy}])$ from the overview is formally $\mathbb{L}([\texttt{x}], (\texttt{fun x ->}), [(\texttt{fy = f y})])$.

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

The Conditional rules are similar in several ways to the Function rules: Conditional Top transitions out of the front of the conditional, and Conditional Bottom transitions into the last clause in one of the two branches. This latter rule shows the purpose of the $x_1\ !\ \beta$ clause: it serves to mark which branch of the conditional we are about to exit out the front of. In both rules we

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

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

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

(1) $\mathbb{L}([\texttt{fy}], (\texttt{f1 = f 1}), [])$: Apply the Skip rule as $\texttt{fy} \neq \texttt{f1}$. This induces a lookup of `fy` from the clause $\textsc{Pred}(\texttt{f1 = f 1}) = (\texttt{fy = f y})$. The last precondition of Skip is needed to ensure that the computation we skipped over (`f1`'s in this case) is not diverging; that is, there exists *some* value that could have been looked up for `f1`. In many practical cases non-termination checks are not needed so this can be skipped. For brevity we will not prove that precondition now, but the implementation currently always performs this check.
(2) $\mathbb{L}([\texttt{fy}], (\texttt{fy = f y}), [])$: Apply the Function Exit rule. This induces a lookup on `fret` which pushes this call site onto the call stack. In addition, we need to satisfy these preconditions:
   (a) Look up `f` to ensure that it is indeed a function. By Value Discovery we see it holds: $\mathbb{L}([\texttt{f}], (\texttt{f = [fun x ->, ...]}), []) = (\texttt{f = [fun x ->, fret = x + o]})$.
   (b) Look up `f`'s return variable `fret`, the last variable defined in the body: $\textsc{RetCl}([\texttt{fun x ->, fret = x + o }]) = (\texttt{fret = x + o})$.
(3) $\mathbb{L}([\texttt{fret}], (\texttt{fret = x + o }), [(\texttt{fy = f y})])$: Apply the Binop rule, looking up both terms in the addition clause.

We will first look up `x`:

(4) $\mathbb{L}([\texttt{x}], (\texttt{fun x ->}), [(\texttt{fy = f y})])$: Apply the Function Enter Parameter rule. We are looking for the variable `x`, which is the parameter passed into our function, so we pop out of the function, pop the call site `fy = f y` from the stack, and search for the formal parameter `y` from $\textsc{Pred}(\texttt{fy = f y}) = (\texttt{f = ...})$.
(5) $\mathbb{L}([\texttt{y}], (\texttt{f = ...}), [])$: Apply the Skip rule.
(6) $\mathbb{L}([\texttt{y}], (\texttt{y = 0}), [])$: Apply the Value Discovery rule to obtain `0` as the result for `y` and thus for `x`.

Now we look up `o` from the Binop:

(7) $\mathbb{L}([\texttt{o}], (\texttt{fun x ->}), [(\texttt{fy = f y})])$: Apply the Function Enter Non-Local rule. In this stringent ANF version of the example, `o` is a non-local variable in the function, giving us an opportunity to exercise this rule. For the next lookup we will need to push `f` onto the lookup stack to first find the definition point of `o`, and pop the current call site from the call stack since we are exiting the function body.
(8) $\mathbb{L}([\texttt{f}, \texttt{o}], (\texttt{f = ...}), [])$: Apply the Value Discard rule, since we are sitting right on `f`'s definition. We can now pop `f` off the lookup stack and continue lookup for `o` at the predecessor clause.
(9) $\mathbb{L}([\texttt{o}], (\texttt{y = 0}), [])$: Apply the Skip rule.
(10) $\mathbb{L}([\texttt{o}], (\texttt{o = 1}), [])$: Apply the Value Discovery rule, returning `1`.

At this point both preconditions of Binop have completed and the result there is known to be `0 + 1 = 1`, and chaining back that means the above `fret` and `fy` lookups are also `1`.

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

3.2 Equivalence of Demand and a Forward Interpreter

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

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

4 A SYMBOLIC DEMAND-DRIVEN EVALUATOR

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

We will use a global cache of constraints Φ to simultaneously constrain all run-time variables in the program. In order to disambiguate different runtime versions of the same variable due to recursion, we index each variable by its call stack: variables in Φ are pairs of the form ${}^{\dot{C}}x$ (where the relative stacks $\dot{C}$ are explained below). 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 often no single path of execution. First, since we can start lookup mid-program, we may start deep inside an (unknown) call stack and, as the correct caller is not known, we must search through all potential callers. We address this issue by adding a parameter Π to lookup which is an oracle to consult for which calling sites to choose in a particular lookup. Second, conditionals could have both true and false branches satisfiable if we had not (yet) accumulated any constraints in 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.

4.1 Relative Stacks

Since our symbolic interpreter does not know the program stack at the time that lookup starts, it uses a *relative* stack $\dot{C}$ to characterize the stack state. At the beginning of symbolic lookup, the stack is *completely unknown*; if we start within a function body, for instance, we do not know from where that function was called. As we move backward through the program using decisions in the aforementioned Π, we also *retrospectively* learn what the stack was when lookup began.

The relative stack $\dot{C}$ is thus a pair (co-stack, concrete stack) written $C?C$: the *co-stack* represents those frames which have been *popped* since we started a lookup from inside a function body, and the *concrete stack* represents those frames which have been pushed since we began lookup. Once execution has reached the start of the program we can in retrospect transform all relative stacks $\dot{C}$ into absolute stacks C.

Before giving the formal definitions we provide some motivation. Consider the example of Figure 4 where we wanted to find inputs exercising line 7. When lookup begins the relative stack will be $\dot{C} = []?[]$: we have not yet exited or entered any calls. Tracing back as with the interpreter rules we will arrive at the start of `f` looking for parameter `x` but there are two call sites for the function: in line 13 and in line 14. So, Π will contain one of those two as its arbitrary choice, say line 13. When the search for `x` turns into a search for the parameter value `i1` there, the relative stack will now be `[(fi1 = f i1)]?[]` since that call site has been popped upon exiting `f`. We are now at the top of the program, and when we learn that fact we can in retrospect convert relative stack `[]?[]` that we started lookup with to absolute stack `[(fi1 = f i1)]`, as the pops to get from line 7 in reverse to the start of the program must be the same as the pushes in a forward run to get from the front of the program to line 7.

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

4.2 Notation for Symbolic Lookup

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

Definition 4.1. *Notation for pushing, popping, and concretizing relative stacks is 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).*
(5) $\textsc{Concretize}(C?[]) = \textsc{Reverse}(C)$

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

$\dot{C}x$			*annotated vars*
$\mathcal{X}$			*annotated var sets*
$\dot{C}$	::=	$C?C$	*relative stacks*
ς	::=	$\dot{C}x \mid \varsigma_{\texttt{true}}$	*formulae symbols*
ϕ	::=	$\varsigma = \varsigma \odot \varsigma \mid \varsigma = \varsigma$	*formulae atoms*
		$\mid \varsigma = v \mid \texttt{stack} = C$	
Φ	::=	$\phi \wedge \ldots \wedge \phi$	*formulae*
Π	::=	$\{\dot{C} \mapsto c, \ldots\}$	*search paths*

Fig. 7. New Constructs for Symbolic Lookup

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

Above, we described how non-determinism in the lookup function needs to be addressed and how an additional lookup parameter Π is used to represent the call site choices made. Concretely, in the grammar Π ranges over mappings from relative stacks $\dot{C}$ to function call sites $c = (x_r = x_f\ x_v)$. Mapping key $\dot{C}$ is the current relative stack and the mapped call site is the one which should be stepped (backwards) to by the lookup function under this call stack. The lookup function takes a Π oracle map as parameter; our implementation will need to search through the space of potential Π mappings. Since in large programs it is usually the case that the space of call sites paired with what function called there is sparse (most call sites usually call only a very limited number of functions in the whole program), the implementation in fact optimizes Π search by using an initial analysis pass to remove many provably-invalid cases.

Formulae Φ of Figure 7 are similarly oracular in the specification, and the implementation for the most part builds Φ 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.

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

(1) $\Phi = \phi_1 \wedge \cdots \wedge \phi_n$ *in some contexts is punned as the set of its atomic conjunctions,* $\Phi = \{\phi_1, \ldots, \phi_n\}$.
(2) $\textsc{isSAT}(\Phi)$ *holds if there is a satisfying assignment for* Φ.
(3) $\textsc{SATs}(\Phi)$ *is the set of all satisfying assignments* M *that map (annotated) variables in* Φ *to values* v.

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

4.3 Variable Lookup Defined

We are now ready to define symbolic variable lookup.

Definition 4.3. *For implicit fixed program* e_{glob}, Φ *with* $\textsc{isSAT}(\Phi)$ *holding, and path mapping* Π, *DDSE variable lookup,* $\mathbb{L}^S(X, \Phi, \Pi, c, \dot{C}) \equiv {}^{\dot{C}}x$ *holds iff there is a proof using the rules of Figure 8. Since* Φ *and* Π *are fixed in most places we take them as implicit parameters in the Figure and elsewhere, writing the equivalent shorthand* $\mathbb{L}^S(X, c, \dot{C}) \equiv {}^{\dot{C}}x$. *In this Figure, we use a few additional notational abbreviations:*

- $\textsc{First}_V([x_1 = b_1, \ldots]) = x_1$ *extracts the first variable defined in the program* $e_{glob} = [x_1 = b_1, \ldots]$.
- $\mathbb{L}^S(X, c, \dot{C}) \equiv v$ *abbreviates* $\exists {}^{\dot{C}_0}x_0.\ \mathbb{L}^S(X, c, \dot{C}) \equiv {}^{\dot{C}_0}x_0 \wedge ({}^{\dot{C}_0}x_0 = v) \in \Phi$
- $\mathbb{L}^S(X, c, \dot{C}) \equiv \_$ *abbreviates* $\mathbb{L}^S(X, c, \dot{C}) \equiv {}^{\dot{C}_0}x_0$ *for some* ${}^{\dot{C}_0}x_0$.
- $\textsc{First}^S(x, c, \dot{C})$ *holds iff* $x \neq \textsc{First}_V(e_{glob})$ *implies* $\mathbb{L}^S([\textsc{First}_V(e_{glob})], \textsc{Pred}(c), \dot{C}) \equiv \_$

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

Instead of returning a value v as the result we return the *defining variable of the value*, allowing us to return *symbolic* constraints in Φ instead of concrete values. For example, the Value Discovery rule directly returns the defining variable x of $(x = v)$ paired with the stack, ${}^{\dot{C}}x$. Note that some of the lookup assertions have non-variables on the right, but that is just a notational shorthand, described in Definition 4.3. This defining variable may then be used by other rules such as the Binop rule, which invokes lookup on both operator parameters and uses the defining variables to build the equation constraining the binary operator behavior in Φ. By using stack-indexed variable definitions ${}^{\dot{C}}x$ in the Φ constraints, we have a guarantee that there are no collisions of different

$$\textsc{Value Discovery}\ \frac{(\dot{C}x = v) \in \Phi \qquad x \neq \textsc{FirstV}(e_{\text{glob}}) \vee (\texttt{stack} = \textsc{Concretize}(\dot{C})) \in \Phi \qquad \textsc{First}^S(x, c, \dot{C})}{\mathbb{L}^S([x], (x = v), \dot{C}) \equiv {}^{\dot{C}}x}$$

$$\textsc{Input}\ \frac{\varsigma_{\texttt{true}} = ({}^{\dot{C}}x = {}^{\dot{C}}x) \in \Phi \qquad x \neq \textsc{FirstV}(e_{\text{glob}}) \vee (\texttt{stack} = \textsc{Concretize}(\dot{C})) \in \Phi \qquad \textsc{First}^S(x, c, \dot{C})}{\mathbb{L}^S([x], (x = \texttt{input}), \dot{C}) \equiv {}^{\dot{C}}x}$$

$$\textsc{Value Discard}\ \frac{\mathbb{L}^S(X, \textsc{Pred}(x), \dot{C}) \equiv {}^{\dot{C}_0}x_0}{\mathbb{L}^S([x] \,||\, X, (x = f), \dot{C}) \equiv {}^{\dot{C}_0}x_0} \qquad \textsc{Alias}\ \frac{\mathbb{L}^S([x'] \,||\, X, \textsc{Pred}(x), \dot{C}) \equiv {}^{\dot{C}_0}x_0}{\mathbb{L}^S([x] \,||\, X, (x = x'), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Binop}\ \frac{\mathbb{L}^S([x'], \textsc{Pred}(x), \dot{C}) \equiv {}^{\dot{C}}x' \qquad \mathbb{L}^S([x''], \textsc{Pred}(x), \dot{C}) \equiv {}^{\dot{C}}x'' \qquad {}^{\dot{C}_0}x_0 = {}^{\dot{C}}x' \odot {}^{\dot{C}}x'' \in \Phi}{\mathbb{L}^S([x], (x = x' \odot x''), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Function Enter Parameter}\ \frac{\dot{C}' = \textsc{Pop}(\dot{C}, c) \qquad \mathbb{L}^S([x_v] \,||\, X, \textsc{Pred}(c), \dot{C}') \equiv {}^{\dot{C}_0}x_0 \qquad c = (x_r = x_f\ x_v) \qquad \Pi(\dot{C}) = c \qquad \mathbb{L}^S([x_f], \textsc{Pred}(c), \dot{C}') \equiv (\texttt{fun}\ x \texttt{->}\ (\ e\ ))}{\mathbb{L}^S([x] \,||\, X, (\texttt{fun}\ x \texttt{->}), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Function Enter Non-Local}\ \frac{\dot{C}' = \textsc{Pop}(\dot{C}, c) \qquad x'' \neq x \qquad c = (x_r = x_f\ x_v) \qquad \Pi(\dot{C}) = c \qquad \mathbb{L}^S([x_f, x] \,||\, X, \textsc{Pred}(c), \dot{C}') \equiv {}^{\dot{C}_0}x_0 \qquad \mathbb{L}^S([x_f], \textsc{Pred}(c), \dot{C}') \equiv (\texttt{fun}\ x'' \texttt{->}\ (\ e\ ))}{\mathbb{L}^S([x] \,||\, X, (\texttt{fun}\ x'' \texttt{->}), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Function Exit}\ \frac{\mathbb{L}^S([x'] \,||\, X, (x' = b), \textsc{Push}(\dot{C}, \textsc{Cl}(x))) \equiv {}^{\dot{C}_0}x_0 \qquad \textsc{RetCl}(e) = (x' = b) \qquad \mathbb{L}^S([x_f], \textsc{Pred}(x), \dot{C}) \equiv (\texttt{fun}\ x'' \texttt{->}\ (\ e\ ))}{\mathbb{L}^S([x] \,||\, X, (x = x_f\ x_v), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Skip}\ \frac{x'' \neq x \qquad \mathbb{L}^S([x] \,||\, X, \textsc{Pred}(x''), \dot{C}) \equiv {}^{\dot{C}_0}x_0 \qquad \mathbb{L}^S([x''], \textsc{Cl}(x''), \dot{C}) \equiv \_}{\mathbb{L}^S([x] \,||\, X, (x'' = b), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Conditional Top}\ \frac{\textsc{Cl}(x_1) = (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}) \qquad \mathbb{L}^S([x_2], \textsc{Pred}(x_1), \dot{C}) \equiv \beta \qquad \mathbb{L}^S(X, \textsc{Pred}(x_1), \dot{C}) \equiv {}^{\dot{C}_0}x_0}{\mathbb{L}^S(X, (x_1\ !\ \beta), \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

$$\textsc{Conditional Bottom}\ \frac{\mathbb{L}^S([x_2], \textsc{Pred}(x_1), \dot{C}) \equiv \beta \qquad \mathbb{L}^S([x'] \,||\, X, (x' = b), \dot{C}) \equiv {}^{\dot{C}_0}x_0 \qquad \textsc{RetCl}(e_\beta) = (x' = b)}{\mathbb{L}^S([x_1] \,||\, X, (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}, \dot{C}) \equiv {}^{\dot{C}_0}x_0}$$

Fig. 8. Symbolic Lookup Rules

activations of the same variable; they are relative stacks in the rules but upon completion of lookup can be converted to absolute stacks as we describe below.

The Function Enter ... rules are extended to support the case that a search is started lexically within a function body. In this case Π is consulted for the choice to make and a call site frame is added to the co-stack by the Pop. The rules contain an additional precondition to look up the

function definition variable x_f which is not present in the reverse interpreter; there the function was already looked up by Function Exit and so it is unnecessary to look up the function again.

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

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

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

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

4.4 Defining Test Generation and Showing Computability

This section uses the symbolic lookup definition to formally define a function $\mathbb{T}$ which generates a test input exercising a particular clause in a program. 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 the determinism of symbolic lookup, in analogy to the determinism of the reverse interpreter. To be clear, test generation is non-deterministic as the input choices in Φ and branch choices in Π are not fixed, but for fixed Φ/Π there can be only one result.

Lemma 4.4. *$\mathbb{L}^S([x], \Phi, \Pi, c, []?[]) \equiv {}^{\dot{C}_0}x_0$ is deterministic: given e_{glob}, x, c, Π and satisfiable Φ, there is at most one ${}^{\dot{C}_0}x_0$ such that a proof can be constructed.*

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

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

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

- *$\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ holds if $\mathbb{L}^S([\textsc{FirstV}(e_{glob})], \Phi, \Pi, c, []?[]) \equiv \_$.*
- *$\mathbb{T}(e_{glob}, c)$ holds iff $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ for some Φ, Π.*

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

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

Lemma 4.6. *Given expression e_{glob} and clause c, if $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ 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 4.7. *Given e_{glob} and c, finding a Φ, Π for which $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ holds is recursively enumerable.*

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

4.5 Correctness

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

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

Lemma 4.8. *If $\mathbb{L}^{\mathbb{S}}([\textsc{FirstV}(e_{glob})], \Phi, \Pi, c_0, \dot{C}_0) \equiv {}^{\dot{C}}x$ then there is exactly one constraint of the form* (`stack` = C) *in Φ for some C.*

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

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

- $|C_c?C_s|_C = C_s \,||\, C'$ *where* $C = \textsc{Reverse}(C_c) \,||\, C'$ *for some C'; this operator is undefined if the equation fails for all C';*
- $|{}^{\dot{C}}x|_C = {}^{|\dot{C}|_C}x$*;*
- $|\mathcal{X}|_C = \{|{}^{\dot{C}}x|_C \mid {}^{\dot{C}}x \in \mathcal{X}\}$*; and*
- $|\Phi|_C = \Phi[(|\mathcal{X}|_C)/\mathcal{X}]$*, for $\mathcal{X}$ being the set of all variables in Φ, and in addition replacing constraint* (`stack` = C) *in Φ with* (`stack` = `[]`).

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

Lemma 4.10 Equivalence of relative and absolute stacks.

(1) *If $\mathbb{L}^{\mathbb{S}}(X, \Phi, \Pi, c_0, \dot{C}_0) \equiv {}^{\dot{C}}x$ and* (`stack` = C) $\in \Phi$ *then $|\dot{C}|_C$ and $|\dot{C}_0|_C$ are defined and $\mathbb{L}^{\mathbb{S}}(X, |\Phi|_C, \Pi, c_0, []?|\dot{C}_0|_C) \equiv {}^{|\dot{C}|_C}x$.*
(2) *If $\mathbb{L}^{\mathbb{S}}([\textsc{FirstV}(e_{glob})], \Phi, \Pi, c_0, []?C) \equiv {}^{[]?C'}x$ then $\mathbb{L}^{\mathbb{S}}([\textsc{FirstV}(e_{glob})], \Phi', \Pi, c_0, []?[]) \equiv {}^{\dot{C}}x$ for some $\dot{C}$ with $C' = |\dot{C}|_C$,* (`stack` = C) $\in \Phi'$*, and $\Phi = |\Phi'|_C$.*

4.5.2 Eliminating the Search Path. Now that we have an absolute stack, we no longer need Π: the call site that invoked the function we wish to exit is always on top of the context stack. Formally, we implement this by replacing mapping Π with a *multi*-mapping $\Pi^{\max}$ which maps each ${}^{\dot{C}}x$ to

every call site c in the program. The effect of this is to neutralize any Π conditions in symbolic lookup and bring lookup closer to the concrete interpreter.

In the following two Lemmas and proofs we are always dealing with relative stacks $\dot{C} = []?C$, i.e. the co-stack portion is empty. So, we will sometimes abbreviate this $\dot{C}$ as just C, in particular for variables we let ${}^{C}x$ abbreviate ${}^{[]?C}x$ in the context of the symbolic system.

Lemma 4.11 Elimination of search paths.

(1) $\mathbb{L}^{S}(X, \Phi, \Pi, c_0, []?C) \equiv {}^{\dot{C}}x$ *implies* $\mathbb{L}^{S}(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^{\dot{C}}x$.
(2) $\mathbb{L}^{S}(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^{\dot{C}}x$ *implies* $\mathbb{L}^{S}(X, \Phi, \Pi, c_0, []?C) \equiv {}^{\dot{C}}x$ *for some (non-multi-) mapping* Π.

4.5.3 Relating Symbolic and Concrete Interpreters. At 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 significant remaining differences are the constraints Φ and the fact that DDSE lookup returns a (constrained) variable rather than a value, but the constraints in Φ can be shown to be isomorphic to the values produced by the concrete interpreter.

Lemma 4.12 Relating symbolic and concrete interpreters.

(1) *If* $\mathbb{L}^{S}(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^{C}x$ *then for all* $M \in \text{SATs}(\Phi)$ *with* ${}^{C}x \mapsto v \in M$, *setting* ι *to* M *we have* $\mathbb{L}(X, c_0, C) \equiv v$.
(2) $\mathbb{L}(X, c_0, C) \equiv v$ *implies* $\mathbb{L}^{S}(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^{C}x$ *for some* Φ *such that for some* $M \in \text{SATs}(\Phi)$, $\iota \subseteq M$ *and* ${}^{C}x \mapsto v \in M$.

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

Theorem 4.13. *Test generation is* sound *and* complete*:*

(1) *If* $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ *then* $\mathbb{L}([\text{FirstV}(e_{glob})], c, C) \equiv v$ *for some* v, *some* ι *satisfying* Φ, *and some* C.
(2) *If* $\mathbb{L}([\text{FirstV}(e_{glob})], c, C) \equiv v$ *for some* v, ι *and* C, *then* $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ *holds for some* Φ *satisfied by* ι, *and some* Π.

5 IMPLEMENTATION

In this section we describe the reference implementation of DDSE. This implementation closely follows the specification, and is primarily designed to confirm correctness of the specification; while it includes some optimizations, many more are needed for good performance. First we describe the implementation and then we describe its performance on example programs. The language implemented is the grammar of Figure 5 with additional syntax for records and projection to allow us to re-use a wider range of benchmarks.

5.1 The OCaml Implementation of DDSE

Definition 4.3 gives the lookup function which symbolically executes a program in a demand-driven fashion. This definition is declarative, treating the logical formulae 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 now discuss.

Our implementation searches for paths through a program to a desired destination by moving backward and applying the rules of Definition 4.3. This naturally gives rise to a nondeterministic algorithm: if multiple rules or uses of a rule apply, each of those choices is attempted. As choices are made, incoherent universes (e.g. with Φ containing unsatisfiable formulae) are discarded. This application of nondeterminism is common in proof searching algorithms, transitive closures,

and similar domains; however, nondeterminism is non-trivial to combine with other features of computation [Zwart and Marsden 2018].

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

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

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

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

5.2 Methodology

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

The SMBC benchmarks are SMT programs in SMT-LIB syntax [Barrett et al. 2017]. The original programs consist of function and datatype definitions, assertions and a goal. We can directly translate the functions to our syntax. We put the assertions and the goal in an expression `if (assertions and goal) then target else nothing` at the end of our program and start test generation from `target`. We don't directly support datatype declarations or solve for uninterpreted functions, so those features of their benchmarks we encode by writing helper functions which

[4]From https://cedeela.fr/~simon/files/cade_17.tar.gz, and https://github.com/c-cube/smbc
[5]From http://www.larcenists.org/benchmarksAboutR7.html and https://github.com/adamsmd/paper-push-down-for-free-prototype/tree/master/benchmarks.

generate them. For instance, to generate an arbitrary list of integers, we use the code `let rec gen_list () = let t = input in if 0 == t then nil else t::(gen_list ())`; to express an uninterpreted function on booleans, we can use `let bt = input in let bf = input in fun x -> if x then (bt == 0) else (bf == 0)`. Using such encodings, we can convert the original SMT programs with unknown terms to closed programs with an input-dependent goal.

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

The formal ANF syntax of Figure 5 is difficult to code in, so we also implemented a translator which allows direct coding in an ML-like syntax; this syntax was used in the Section 2 examples. Recursive functions or loops appearing in the original benchmarks are encoded via self-application in the ANF. Lists are encoded as records, for example `[1; 2]` is

```
{last = false, elem = 1, next = {last = false, elem = 2, next = {last = true}}}
```

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

5.3 Evaluation

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

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

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

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

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

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

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

facehugger The modifications to this benchmark appear in Figure 9(a) and were described above.

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

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

```
let x = input in
let id x = x in
let rec f n =
    if n <= 1 then
        1
    else
        n * (f (n-1))
    in
let rec g n =
    if n <= 1 then
        1
    else
        n * (g (n-1))
    in
# original benchmark
# ((id f) 3) + ((id g) 4)
# our benchmark
let sum = ((id f) 3) + ((id g) x) in
if (sum == 30) then
    let target = 1 in 1
else
    0
```

(a) Facehugger function

```
let y = input in
let rec list_build acc =
    let x = input in
    if 0 == x then
        acc
    else
        list_build (x :: acc)
    in
let rec list_map f l =
    let rec lp lst =
        match lst with
        | [] -> []
        | x::xs -> (f x)::(lp xs)
        in
    lp l
    in
let rec sum_list l =
    match lst with
    | [] -> 0
    | x::xs -> x + (sum_list xs)
    in
let lst = list_build [] in
let make_adder s a = a + s in
let adder = make_adder y in
let lst2 = list_map adder lst in
let s = sum_list lst2 in
if (s == 10) then
    let target = 1 in 1
else
    0
```

(b) List build, map and sum

Fig. 9. Benchmark Source Code Examples

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

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

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

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

All of these benchmarks run successfully in our test generator, and generate satisfying inputs (if any). For each benchmark, Table 1 lists the elapsed time, taken as the average of three runs. The total *steps* is an internal count of the number of coroutine monad binds and should only be viewed relatively – the vast majority of these binds are trivially discharged, allowing the code to be highly compositional. All of the times for these short programs are reasonably fast, in spite of the lack of optimizations in our current implementation. `sat-1-direct` is much faster than `sat-1` because the satisfiability search is entirely within Z3 whereas `sat-1` is a naive SAT solver checking cases

one by one and needing a different control flow path for each. The `fold` benchmark also has to do a control flow space search as we have to encode an uninterpreted boolean function by conditioning on integer inputs; the SMBC system has native support for uninterpreted functions.

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

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

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

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

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

Benchmark	Time	Steps
fold	20.51s	77587
palindrome	4.01s	20585
pigeon	0.55s	8055
sorted	2.46s	18377
blur	0.24s	2880
eta	0.04s	268
facehugger	0.98s	12226
flatten	0.85s	8157
k-cfa-2	0.09s	1537
k-cfa-3	0.25s	4308
map	1.19s	10631
mj09	0.07s	767
sat-1	4.42s	20213
sat-1 direct	0.07s	861

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

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

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

For example, consider the example of Figure 10. For this program, pure forward symbolic execution doesn't

```
1  let rec count d =
2      let x = input in
3      if x == 0 then 0
4      else 1 + (count d)
5      in
6  let ca = count 0 in
7  let cb = count 0 in
8  if ca == 2 * cb and 2 < cb then
9      let target = 1 in 1
10 else 0
```

Fig. 10. Double count program

have any guidance on how many times the count loop
should run in the two calls to `count`, and so `ca` and `cb` do not initially get constrained. However, the later code requires `ca` to be twice the value of `cb` and a forward symbolic interpreter must perform a search of all possible iterations for both loops until it luckily picks one that is twice the other, making many fruitless attempts before reaching this goal. DDSE on the other hand starts from the target point, which immediately gives it a constraint as to what previous control flow paths will work, and significantly shrinks the search space for control flows – DDSE successfully infers input stream `[1,1,1,1,1,1,0,1,1,1,0]` with 69823 steps required. This code is a simplified example of common real-world cases which contain an assertion deep inside a program but which may have a massive number of forward paths to explore to reach it.

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

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

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

6 RELATED WORK

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

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

some constraints forward which would preclude those paths from ever being considered in the backward phase.

Many of the issues and challenges of these systems we also share. The key problem in symbolic execution is the well-known *path explosion problem* [Anand et al. 2013, 2008]: the search space grows far too rapidly and the algorithm founders. We currently use a simple cache of the lookup function $\mathbb{L}^{\mathsf{S}}$ 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 4.3), and it is a direct generalization of a non-symbolic interpreter (Definition 3.3) to symbolic data.

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

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

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

Forward symbolic evaluators have been developed for extended functional languages, e.g. Rosette [Torlak and Bodik 2013] for Racket and Kaplan [Köksal et al. 2012] for Scala. Higher-order contract verification [Tobin-Hochstadt and Van Horn 2012] is a generalized form of higher-order forward

symbolic execution. Contract verification is a step closer to program logic; one particular reason why this current study excites us is for the potential applications as a more general logic.

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

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

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

7 CONCLUSIONS

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

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

ACKNOWLEDGEMENTS

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

REFERENCES

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

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

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

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

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

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

François Bourdoncle. 1993. Abstract Debugging of Higher-order Imperative Languages. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 46–55. https://doi.org/10.1145/155090.155095

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

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

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

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

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

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

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

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

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

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

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

Peter W. O'Hearn. 2019. Incorrectness Logic. *Proc. ACM Program. Lang.* 4, POPL, Article Article 10 (Dec. 2019), 32 pages. https://doi.org/10.1145/3371078

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

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

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

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

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

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

A PROOFS FOR SECTION 3 (A REVERSE CONSTRUCTION INTERPRETER)

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

Proof. We show by induction on n that all proof trees constructed from Definition 3.3 of height up to n from some X, c, C must have identical result value v. Assume trees of height less than n have the above property, and show for trees of height n. This tree has a fixed X, c, C in its conclusion.

It is a fact that at most one of the rules can apply at the conclusion of the tree: All the rules have a different c below the line, which will make the choice unique, except in a few cases we now consider. For values, since the two Value rules apply to lookup stacks of different lengths (Value Discovery for singleton X and Value Discard for non-singleton X) so only one of those rules applies. For the two Function Enter rules, in the Parameter rule the function parameter must be on the top of the lookup stack, and for the Non-Local must not be, so they are mutually exclusive. The Skip rule only applies when the clause contains a variable x'' that is *not* the top of the lookup stack; for all the other rules that condition fails.

Now, by inspection of all these rules, in each case but Function Exit or Conditional Bottom the subordinate lookup parameters are deterministically calculable from the information in the rule. So, by induction each of those subordinate lookup proof trees must have unique values. In the Function Exit rule, the x' lookup depends on the result of the x_f lookup so we first can show x_f has a unique value by induction, and then show the same for x'. The final value v is calculable from x_f and x' by direct inspection of the Function Exit rule, so only one v can be constructed, establishing the claim in this case. Lastly, for Conditional Bottom by induction the lookup for x_2 must return a unique value which must be either `true` or `false`; it cannot return both as that would violate the uniqueness of value requirement in the induction hypothesis. Thus the conditional branch chosen is fixed and the case follows by induction. □

B PROOFS FOR SECTION 4 (A SYMBOLIC DEMAND-DRIVEN EVALUATOR)

Lemma 4.4. *$\mathbb{L}^{\mathcal{S}}([x], \Phi, \Pi, c, []?[]) \equiv {}^{\dot{C}_0}x_0$ is deterministic: given e_{glob}, x, c, Π and satisfiable Φ, there is at most one ${}^{\dot{C}_0}x_0$ such that a proof can be constructed.*

Proof. We show by induction on n that all proof trees constructed from Definition 4.3 of height up to n from some $X, c, \dot{C}$ must have identical result ${}^{\dot{C}_0}x_0$. Assume trees of height less than n have the above property, and show for trees of height n. This tree has a fixed $X, c, \dot{C}$ in its conclusion.

It is a fact that at most one of the rules can apply at the conclusion of the tree: All the rules have a different c below the line, which will make the choice unique, except in a few cases we now consider. For values, since the two Value rules apply to X of different lengths, only one of them can apply. For the two Function Enter rules, in the Parameter rule the function parameter must be on the top of the lookup stack, and for the Non-Local must not be, so they are mutually exclusive. The Skip rule only applies when the clause contains a variable x'' that is *not* the top of the lookup stack; for all the other rules that condition fails.

Now, by inspection of all these rules, in each case but Function Exit or Conditional Bottom the three subordinate lookup parameters are deterministically calculable from information in the rule (in the Function Enter rules in particular, note how Π pre-determines a fixed c for subordinate lookups, forcing determinism). So, by induction each of those subordinate lookup proof trees must have unique values. In the Function Exit rule, the x' lookup depends on the result of the x_f lookup; we first can show x_f is constrained to a unique value by induction, and then show the same for x'. Thus the final value ${}^{\dot{C}_0}x_0$ is calculable from x_f and x' by direct inspection of the rule, so only one ${}^{\dot{C}_0}x_0$ can be constructed, establishing the claim in this case. Lastly, for Conditional Bottom by

induction the x_2 lookup must result in constraining the variable to either `true` or `false`; it cannot be constrained to both as that would result in an unsatisfiable Φ and also violate the uniqueness of value requirement in the induction hypothesis. Thus the conditional branch chosen is fixed and the case follows by induction. □

Lemma 4.6. *Given expression e_{glob} and clause c, if $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ for some Φ, Π then there exists an ι such that ι satisfies Φ.*

Proof. Suppose $\mathbb{T}(e_{\text{glob}}, c, \Phi, \Pi)$ for some satisfiable Φ and Π, i.e. expanding definitions we have $\mathbb{L}^S([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c, []?[]) \equiv \_$. Inspecting the rules, for each input action a constraint of the form $\varsigma_{\texttt{true}} = (\dot{C}x = \dot{C}x) \in \Phi$, and no other, is placed on input variable $\dot{C}x$, requiring it to be an integer. So, to construct ι, pick its domain to be all variables $\dot{C}x$ such that a constraint of the above form appears in Φ; those are all of the dynamic inputs occurring over the program run. Now, take any satisfying assignment to Φ (which we know must exist by virtue of it being satisfiable), and restrict its domain to these $\dot{C}x$; since is it a sub-mapping of the satisfiable mapping it must satisfy Φ. □

Lemma 4.7. *Given e_{glob} and c, finding a Φ, Π for which $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ holds is recursively enumerable.*

Proof. The space of (Φ, Π) pairs is recursively enumerable since each individually is r.e. by inspection of the grammar of Figure 7, and pairing preserves r.e.-ness. Lookup itself is recursively enumerable as it is a proof system with decidable auxiliary predicates. Thus, dovetailing lookups over the Φ, Π enumeration will enumerate the valid Φ, Π. □

Lemma 4.8. *If $\mathbb{L}^S([\textsc{FirstV}(e_{glob})], \Phi, \Pi, c_0, \dot{C}_0) \equiv \dot{C}x$ then there is exactly one constraint of the form $(\texttt{stack} = C)$ in Φ for some C.*

Proof. We need to show that looking up $\textsc{FirstV}(e_{\text{glob}})$ anywhere in the program must eventually lead to applying the Value Discovery or Input rule to $\textsc{FirstV}(e_{\text{glob}})$ at the start of the program. Consider the structure of the proof tree for $\mathbb{L}^S([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c_0, \dot{C}_0) \equiv \dot{C}x$. Note that the first clause in the program must either be a value or an input clause, for all of the other clauses would contain a free variable and the program would not be well-formed.

By inspection of the rules, the only leaves of a valid proof tree are in fact lookups of $[\textsc{FirstV}(e_{\text{glob}})]$ in either Value Discovery or Input: all the other rules and the other cases of those two rules are not leaves and force one or more sub-lookups. So, there must be at least one constraint of the form $(\texttt{stack} = C)$ in Φ.

All that remains is to show there can be no $(\texttt{stack} = C')$ for $C' \neq C$ in Φ; this is trivial from the requirement that Φ be satisfiable. □

Lemma 4.10 Equivalence of relative and absolute stacks.

(1) *If $\mathbb{L}^S(X, \Phi, \Pi, c_0, \dot{C}_0) \equiv \dot{C}x$ and $(\texttt{stack} = C) \in \Phi$ then $|\dot{C}|_C$ and $|\dot{C}_0|_C$ are defined and $\mathbb{L}^S(X, |\Phi|_C, \Pi, c_0, []?|\dot{C}_0|_C) \equiv {}^{|\dot{C}|_C}x$.*
(2) *If $\mathbb{L}^S([\textsc{FirstV}(e_{glob})], \Phi, \Pi, c_0, []?C) \equiv {}^{[]?C'}x$ then $\mathbb{L}^S([\textsc{FirstV}(e_{glob})], \Phi', \Pi, c_0, []?[]) \equiv \dot{C}x$ for some $\dot{C}$ with $C' = |\dot{C}|_C$, $(\texttt{stack} = C) \in \Phi'$, and $\Phi = |\Phi'|_C$.*

Proof. Clause (1): Proceed by induction on the height of the proof tree for $\mathbb{L}^S(X, \Phi, \Pi, c_0, \dot{C}_0) \equiv \dot{C}x$, assuming $(\texttt{stack} = C) \in \Phi$. Assume the result holds for trees of height shorter than n, show for height n. For Value Discovery and Input, uniformly replacing $\dot{C}_0$ with $|\dot{C}_0|_C$ will produce a valid proof directly by induction if the case where FirstV holds is ignored. And, if the FirstV assertion holds, it must be that $\textsc{Concretize}(\dot{C}_0) = C$ in the assumption since $(\texttt{stack} = C) \in \Phi$, meaning

$\dot{C}_0 = \textsc{Reverse}(C)?[]$; in this case $|\dot{C}_0|_C = []$ by the definition of $|\cdot|_C$; so, by Definition 4.9 for formulae Φ we have $(\texttt{stack} = []) \in |\Phi|_C$ which will allow the FirstV condition to hold in the absolute stack derivation.

For the other rules most do not change the context stack so the result follows immediately by induction. Only the Function rules change the stack, and we address those cases in turn.

For the Function Enter Parameter rule, in the assumed relative stack derivation, the sub-lookups use stack $\textsc{Pop}(\dot{C}_0, c)$, where $\Pi(\dot{C}_0) = c$. Now, by induction hypothesis we have sub-lookups for some popped stack $[]?C_0'$ and then the stack in the final proof node we need to construct is $|\dot{C}_0|_C = C_0 = [c] \,||\, C_0'$ (recall that Pop functions as a concrete pop if there is no co-stack present).

At this point, we proceed by cases on whether $\dot{C}_0$ has a non-empty concrete stack component. If the concrete stack is empty, then $\dot{C}_0 = C_{00}?[]$. In this case, $\textsc{Pop}(\dot{C}_0, c) = \textsc{Pop}(C_{00}?[], c) = ([c] \,||\, C_{00})?[]$. By the induction hypothesis on the sub-derivations, we know that $|([c] \,||\, C_{00})?[]|_C = C_0'$. Now, expanding $|\cdot|_C$, this means $C = (\textsc{Reverse}(C_{00}) \,||[c]) \,||\, C''$ for some C'' with $|\textsc{Pop}(\dot{C}_0, c)|_C = C_0'$, and so by the definition $C_0' = [] \,||\, C'' = C''$. We now have the justifications at hand to show the stacks still align, i.e. we can show $|\dot{C}_0|_C = C_0$, by the following chain of equivalences: $|\dot{C}_0|_C = |C_{00}?[]|_C$ which expanding the definition means $C = \textsc{Reverse}(C_{00}) \,||([c] \,||\, C'')$ and so $|C_{00}?[]|_C = [c] \,||\, C''$; since $C_0' = C''$ was shown above, it is also equivalent to $[c] \,||\, C_0'$ which is then C_0, establishing the result for the empty concrete stack sub-case.

Next we consider the case that the concrete stack component is non-empty. Here the co-stack is not used as Pop can directly pop from the concrete stack, and the case follows directly by induction.

The Function Enter Non-Local rule has identical stack operations as with the previous case and the proof proceeds analogously.

Lastly consider the Function Exit rule. Here the stack operations in both assumption and conclusion are concrete pushes which uniformly work on a concrete stack, and so it follows directly by induction.

Clause (2): Assume $\mathbb{L}^{\mathbb{S}}([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c_0, []?C) \equiv {}^{[]?C'}x$. This assertion is only for lookup of the first variable of the program; to establish the result by induction we strengthen the statement we wish to prove to: If $\mathbb{L}^{\mathbb{S}}(X, \Phi, \Pi, c_0, []?C_0) \equiv {}^{[]?C'}x$ then $\mathbb{L}^{\mathbb{S}}(X, \Phi', \Pi, c_0, \dot{C}_0) \equiv {}^{\dot{C}}x$ for some $\dot{C}_0, \dot{C}$ with $C' = |\dot{C}|_C$, $C_0 = |\dot{C}_0|_C$, $(\texttt{stack} = C) \in \Phi'$, and $\Phi' = |\Phi|_C$. (Note that $C = |[]?[]|_C$ so this is indeed a generalization of the Lemma statement).

So, assume proof trees of height less than n have the above property, and show for trees of height n. For Value Discovery and Input, if the FirstV clause does not apply those cases are direct by induction. For the case $x = \textsc{FirstV}(e_{\text{glob}})$, we are at the top of the program inside no function calls, so $C_0 = []$. So, by the assumed derivation holding we must then have $\texttt{stack} = [] \in \Phi$. Define Φ' to include $\texttt{stack} = C \in \Phi$ so that $|\Phi|_C = \Phi'$ holds, and this then satisfies the preconditions of the relative stack derivation, completing this case.

Most of the remaining rules only use the $C/\dot{C}$ to tag variables, and for those variables we may directly map a stack such as C_1 to a stack $\dot{C}_1$ where $C_1 = |\dot{C}_1|_C$ by the induction hypothesis. By inspection of the rules, all but the Function rules produce a proof tree falling under this case and are direct by induction. We now consider the Function rules in turn/

Consider the case where Function Enter Parameter is the root of the proof tree. In our assumption $\mathbb{L}^{\mathbb{S}}(X, \Phi, \Pi, c_0, []?C_0) \equiv {}^{[]?C'}x$ this means the "$\dot{C}$" of the rule is $[]?C_0$, and $\textsc{Pop}([]?C_0, c)$ by inspection of Definition 4.1 is $[]?C_0'$ where $C_0 = [c] \,||\, C_0'$, with $c = \Pi([]?C_0)$. The sub-lookups in this rule all use this popped stack, $[]?C_0'$. And, on those sub-lookups we can apply our induction hypothesis to obtain lookups on stack $\dot{C}_0$ defined by $C_0' = |\dot{C}_0|_C$. Now, to construct the proof of $\mathbb{L}^{\mathbb{S}}(X, \Phi', \Pi, c_0, \dot{C}_0) \equiv {}^{\dot{C}}x$ we proceed by cases on the form of $\dot{C}_0$. In the first case, the concrete stack component is empty in the sub-lookups: $\dot{C} = C_{00}?[]$ for some C_{00}. In this case, the concrete

derivation invokes sub-lookups on the c-popped stack so we have $C_0 = [c] \,||\, C_0'$. At this point we are at the exact same point as in the clause (1) Function Enter Parameter case and so by identical algebraic reasoning on relative stacks can obtain $|\dot{C}_0|_C = C_0$, allowing us to construct the desired derivation concluding with stack $\dot{C}_0$.

The Function Enter Non-Local rule has identical stack manipulations as the previous case and proceeds analogously.

Lastly, Function Exit is only performing concrete pushes to the stack in either derivation and so follows directly by induction. □

Lemma 4.11 Elimination of search paths.

(1) $\mathbb{L}^S(X, \Phi, \Pi, c_0, []?C) \equiv \dot{C}x$ *implies* $\mathbb{L}^S(X, \Phi, \Pi^{max}, c_0, []?C) \equiv \dot{C}x$.
(2) $\mathbb{L}^S(X, \Phi, \Pi^{max}, c_0, []?C) \equiv \dot{C}x$ *implies* $\mathbb{L}^S(X, \Phi, \Pi, c_0, []?C) \equiv \dot{C}x$ *for some (non-multi-) mapping* Π.

Proof. Clause (1): Assuming $\mathbb{L}^S(X, \Phi, \Pi, c_0, []?C) \equiv \dot{C}x$, proceed by induction on the depth of the proof tree to show $\mathbb{L}^S(X, \Phi, \Pi^{\max}, c_0, []?C) \equiv \dot{C}x$. Notice that these lookups are identical in all parameters except for Π. Assume the statement holds for proofs shorter than height n, prove for height n. For all but the Function Enter rules Π is not used so it follows trivially by induction in those cases. Consider the Function Enter Parameter rule. By induction all the sub-lookups and conditions must be identical, the only difference is in the assumed proof tree we have constraint $\Pi(\dot{C}) = c$ and in the newly constructed proof we need to instead have $\Pi^{\max}(\dot{C}) = c$. But, since $\Pi^{\max}$ is maximal by construction this is trivial, completing this case. The Function Enter Non-Local rule proceeds by an identical argument.

Clause (2): Suppose $\mathbb{L}^S(X, \Phi, \Pi^{\max}, c_0, []?C) \equiv \dot{C}x$, and show there is a Π mapping for which $\mathbb{L}^S(X, \Phi, \Pi, c_0, []?C) \equiv \dot{C}x$. Consider the partial proof which has the identical structure with the assumed proof tree, but which has a collection of outstanding constraints of the form $\{\Pi(\dot{C}_1) = c_1, \ldots \Pi(\dot{C}_n) = c_n\}$ taken by collecting all of these constraints from all the Function Enter rule instantiations in the proof tree. The only thing we need to show is this is a mapping, i.e. each $\dot{C}$ relates to a unique c in this set, and it will then induce the Π for the desired proof.

Now, since the context stack always has an empty co-stack in these proof trees (recall we removed the co-stack portion with the previous Lemma), we know that Pop is invariably a concrete pop in the Function Enter rules, meaning the c in those rules is always the top concrete stack frame, i.e. the $\dot{C}$ there is of the form $[]?([c] \,||\, C')$ for some C'. So, we can simply define $\Pi([]?([c] \,||\, C')) = c$, a mapping returning the top of the concrete stack, which will satisfy all Function Enter constraints collected above and produce a complete proof tree. □

Lemma 4.12 Relating symbolic and concrete interpreters.

(1) *If* $\mathbb{L}^S(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^C x$ *then for all* $M \in \text{SATs}(\Phi)$ *with* ${}^C x \mapsto v \in M$, *setting* ι *to* M *we have* $\mathbb{L}(X, c_0, C) \equiv v$.
(2) $\mathbb{L}(X, c_0, C) \equiv v$ *implies* $\mathbb{L}^S(X, \Phi, \Pi^{max}, c_0, []?C) \equiv {}^C x$ *for some* Φ *such that for some* $M \in \text{SATs}(\Phi)$, $\iota \subseteq M$ *and* ${}^C x \mapsto v \in M$.

Proof. Clause (1): Assuming the antecedent with $M \in \text{SATs}(\Phi)$, ${}^C x \mapsto v \in M$, and set ι to be this M. Proceed by induction on the depth of the proof of $\mathbb{L}^S(X, \Phi, \Pi^{\max}, c_0, []?C) \equiv {}^C x$ with the goal to show $\mathbb{L}(X, c_0, C) \equiv v$.

Assume the statement holds for proofs shorter than height n, prove for height n. If the antecedent constructs a proof with Value Discovery, it follows immediately: $M(x) = v$ is required since it is directly a constraint placed in Φ by the rule. For Input, since $M = \iota$, $M(x) = v$ again directly holds, and the First condition in these rules follows immediately by induction.

For the remaining rules, observe that the symbolic and concrete rules are identical except in a few dimensions which we carefully address one-by-one. (1) Symbolic rules relate to a variable ${}^{C}x$ whereas concrete rules relate to a value v. But we can translate the symbolic to concrete by replacing ${}^{C}x$ with $M({}^{C}x)$. (2) Symbolic rules use stacks $\dot{C}$ and concrete rules use stack C. But, here the symbolic stacks are always of the form $\dot{C} = []?C$, defining an isomorphism between the stack forms. And, by inspection of the definitions of pushing and popping in Definition 4.1, the co-stack operations degenerate into the same operations in the concrete system in this case. So for example in Function Enter in the symbolic system we have requirement $\textsc{Pop}(\dot{C}, c) = \dot{C}'$ which given the lack of co-stack may be simplified to $\textsc{Pop}([]?([c] \,||\, C), c) = []?C$, which aligns symbolic $\dot{C}$ precisely with the context $[c] \,||\, C$ of the concrete Function Enter. (3) Symbolic rules include additional $\Pi^{\max}$ constraints; they can simply be removed when constructing the concrete proof tree. So, we have succeeded in translating each symbolic node to a concrete node, establishing the result.

Clause (2): Assume $\mathbb{L}(X, c_0, C) \equiv v$. Proceed by induction on the depth of the proof tree to both incrementally construct a Φ/M and to show that $\mathbb{L}^{\text{S}}(X, \Phi, \Pi^{\max}, c_0, []?C) \equiv {}^{C}x$. In other words, we strengthen the induction hypothesis to also assume a Φ_0 and satisfying assignment M_0 has been constructed in each sub-lookup of the symbolic system, and will produce a new extension of these, Φ and M, which are supersets of all sub-lookup formulae and mappings. Since we are constructing satisfying assignment M in parallel, by construction we will have the final Φ satisfiable (by the final M). Observe that all requirements upon Φ placed by the rules in the symbolic system are *positive*, i.e. they require certain constraints only be present and nothing is required to be absent from Φ, so it will be sound to re-play the proof tree with the final fixed Φ/M reaching the root to produce a fixed Φ over the entire derivation. We assume initially that Φ consists of only the constraint $(\texttt{stack} = [])$ (this constraint was only needed to absolutize the relative stacks which we have already done, we need to introduce it here only so the base case lookups on the first variable will satisfy the rule set).

Assume the strengthened statement holds for proofs shorter than height n, prove for height n. For Value Discovery we simply add $\Phi = \{({}^{C}x = v)\}$ and $M = \{({}^{C}x \mapsto v)\}$ and the claim then follows directly by induction. For Input, it only adds $\varsigma_{\texttt{true}} = ({}^{C}x = {}^{C}x)$ to Φ, so we can directly add ${}^{C}x \mapsto \iota({}^{C}x)$ to M and it will be a satisfying assignment. This also meets the requirement that $\iota \subseteq M$.

Now consider the inductive cases. In the proof of clause (1) above we carefully enumerated all differences between the rule sets excepting Value Discovery and Input, and we will re-visit those same differences, (1)-(3), for this direction.

For difference (1), observe by inspection of the symbolic rules that return variables ${}^{C}x$ in lookup are ultimately only constructed in the base case rules; all the other rules just "forward" ${}^{C}x$ from some subordinate lookup result. These variables are formed by pairing the defining program variable x with the current call stack $\dot{C} = []?C$. As such, they are in fact isomorphic to such base lookups, the variable *completely defines* the lookup: supposing the result of a base case lookup was ${}^{C}x$; this result by inspection of the base case rules must have come from a lookup of the form $\mathbb{L}^{\text{S}}([x], \Phi, \Pi^{\max}, (x = v \text{ or } \texttt{input}), []?C) \equiv {}^{C}x$ – the lookup stack must be $[x]$, the clause must be the (sole) defining clause of x in the program, and the stack must be the variable annotation, $[]?C$. This thus fixes all parameters of lookup. So, there will by definition be no variable collisions.

Difference (1) also adds constraints to Φ, let us enumerate these cases and define how satisfying assignments from the sub-lookups can be combined and extended to produce an overall satisfying assignment M. Rule Binop adds ${}^{C}x = {}^{C}x' \odot {}^{C}x''$ to Φ. By assumption from the concrete system we have final result $v' \odot v''$ where v' and v'' are the results returned from the two sub-lookups. By induction we can assume ${}^{C}x' \mapsto v'$ and ${}^{C}x'' \mapsto v''$ are in M' and M'' respectively (mappings produced from the left and right operator lookups, respectively), and so we can consistently define the extended mapping M as $M' \cup M'' \cup {}^{C}x \mapsto v' \odot v''$. Observe that this extends assignment M to

cover the new variable ${}^{C}x''$ added to Φ and so is a satisfying assignment for Φ. The other rules adding constraints are similar, here is a rundown of those cases. For the Conditional rules, expanding the notation abbreviation used in the rule from Definition 4.3 there must be either a ${}^{C}x_2 = \texttt{true}$ or ${}^{C}x_2 = \texttt{false}$ constraint added to Φ; we thus also add mapping ${}^{C}x_2 \mapsto \texttt{true}$ or ${}^{C}x_2 \mapsto \texttt{false}$, respectively, to M. The Function rules similarly extend Φ and M to constrain ${}^{C}x_f$ to a function. For rules adding no constraints to Φ, we only union the constraints Φ and assignments M.

For difference (2), in clause (1) we defined a trivial isomorphism between the two stack forms and operations which we can just apply in reverse here. For difference (3), the additional constraints on $\Pi^{\max}$ are all vacuously true as $\Pi^{\max}$ maps to all call sites – it was previously neutralized in Lemma 4.11 above. □

Theorem 4.13. *Test generation is* sound *and* complete*:*

(1) *If* $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ *then* $\mathbb{L}([\textsc{FirstV}(e_{glob})], c, C) \equiv v$ *for some* v*, some* ι *satisfying* Φ*, and some* C*.*
(2) *If* $\mathbb{L}([\textsc{FirstV}(e_{glob})], c, C) \equiv v$ *for some* v*,* ι *and* C*, then* $\mathbb{T}(e_{glob}, c, \Phi, \Pi)$ *holds for some* Φ *satisfied by* ι*, and some* Π*.*

Proof. Clause (1): Assume $\mathbb{T}(e_{\text{glob}}, c, \Phi, \Pi)$. Expanding the definition of the test generation predicate, we have $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c, []?[]) \equiv \_$. By Lemma 4.8, we must have a (single) constraint $(\texttt{stack} = C) \in \Phi$. Now, applying clause (1) of Lemma 4.10 we can absolutize the stack to obtain $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], |\Phi|_C, \Pi, c, []?C) \equiv {}^{|\dot{C}|_C}x$ observing $|[]?[]|_C = C$. Then, since the stack is absolutized we may apply clause (1) of Lemma 4.11 to obtain $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], |\Phi|_C, \Pi^{\max}, c, []?C) \equiv {}^{|\dot{C}|_C}x$. Next we may apply clause (1) of Lemma 4.12 to obtain $\mathbb{L}([\textsc{FirstV}(e_{\text{glob}})], c, C) \equiv v$.

Clause (2): Assume $\mathbb{L}([\textsc{FirstV}(e_{\text{glob}})], c, C) \equiv v$. By clause (2) of Lemma 4.12, we then must have $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi^{\max}, c, []?C) \equiv \_$. Then by clause (2) of Lemma 4.11 it follows that we have $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c, []?C) \equiv \_$ for some mapping Π. Lastly, by clause (2) of Lemma 4.10 we have $\mathbb{L}^{S}([\textsc{FirstV}(e_{\text{glob}})], \Phi, \Pi, c, []?[]) \equiv \_$ and thus $\mathbb{T}(e_{\text{glob}}, c, \Phi, \Pi)$. □

C CORRECTNESS OF REVERSE OPERATIONAL SEMANTICS

We now demonstrate the correctness of the lookup system given in Definition 3.3. In general terms, we aim to prove a program can run forward in a standard forward operational semantics to a particular program point if and only if the reverse lookup relation provides correct values for that program point. We demonstrate this first by defining a forward operational semantics for the grammar appearing in Figure 5 and then proving its equivalence with the lookup relation. This proof is delicate as we are showing a forward run is equivalent to a backward which is the exact opposite of an induction argument; significant strengthenings are thus required to prove the result. The proof here is a completely new proof, but a previous proof of a similar property is found in [Facchinetti et al. 2019] as Theorem 5.19; that result requires a control flow graph construction and is for a language with no conditionals or inputs.

In the process, we formally show how a sequence of inputs can be translated to an equivalent mapping form as was used in the paper body in Section 3.

C.1 Forward Operational Semantics

In this subsection we present a forward operational semantics which we subsequently show equivalent with the reverse semantics.

In order to make the forward-reverse proof simpler, we first factor out one issue in the forward operational semantics alone: we show how the view of inputs as a mapping ι is isomorphic to a

$$
\begin{array}{rcll}
E & ::= & [x \mapsto \dot{v}, \ldots] & \textit{environments} \\
\dot{v} & ::= & \kappa \mid n \mid \beta & \textit{environment values} \\
\kappa & ::= & \langle f, E \rangle & \textit{closures} \\
I & ::= & [n, \ldots] & \textit{input sequences} \\
\sigma & ::= & \langle E, e \rangle & \textit{program stack frames} \\
\Sigma & ::= & [\sigma, \ldots] & \textit{program stacks}
\end{array}
$$

Fig. 11. Forward Operational Semantics Grammar

$$\textsc{Definition}\ \frac{x \text{ not of form } f}{[\langle E, [x = v] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E \,|||\, [x \mapsto v], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Closure}\ \frac{}{[\langle E, [x = f] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E \,|||\, [x \mapsto \langle f, E\rangle], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Input}\ \frac{}{[\langle E, [x = \texttt{input}] \,||\, e\rangle] \,||\, \Sigma, [n] \,||\, I \longrightarrow^1 [\langle E \,|||\, [x \mapsto n], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Alias}\ \frac{E(x') = \dot{v}}{[\langle E, [x = x'] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E \,|||\, [x \mapsto \dot{v}], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Binop}\ \frac{E(x_2) = \dot{v}_2 \qquad E(x_3) = \dot{v}_3 \qquad \dot{v}_1 = \dot{v}_2 \odot \dot{v}_3}{[\langle E, [x_1 = x_2 \odot x_3] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E \,|||\, [x_1 = \dot{v}_1], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Call}\ \frac{\Sigma = [\langle E, [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma' \qquad E(x_2) = \langle [\texttt{fun}\ x_4\ \texttt{->}] \,||\, e', E'\rangle \qquad E(x_3) = \dot{v}}{\Sigma, I \longrightarrow^1 [\langle E' \,|||\, [x_4 \mapsto \dot{v}], e'\rangle] \,||\, \Sigma, I}$$

$$\textsc{Return}\ \frac{}{[\langle E \,|||\, [x \mapsto \dot{v}], []\rangle, \langle E', [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E' \,|||\, [x_1 = \dot{v}], e\rangle] \,||\, \Sigma, I}$$

$$\textsc{Conditional Start}\ \frac{\Sigma = [\langle E, [x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}] \,||\, e\rangle] \,||\, \Sigma' \qquad E(x_2) = \beta \qquad e_\beta = [x_1\ !\ \beta] \,||\, e'}{\Sigma, I \longrightarrow^1 [\langle E, e'\rangle] \,||\, \Sigma, I}$$

$$\textsc{Conditional End}\ \frac{}{[\langle E \,|||\, [x \mapsto \dot{v}], []\rangle, \langle E', [x_1 = x_2\ ?\ e_T : e_F] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1 [\langle E \,|||\, [x_1 \mapsto \dot{v}], e\rangle] \,||\, \Sigma, I}$$

Fig. 12. Input-as-list Forward Operational Semantics Rules

standard list view of inputs, I. So, we first present a list-based-input forward semantics, then a map-based-input forward semantics, and then show the two are equivalent.

The operational semantics is a more or less standard environment/closure/stack based presentation. Grammatical entities for environments, closures, and stacks are defined in Figure 11.

Environment lookup is standard.

Definition C.1. *We define lookup of a variable x in an environment E, denoted $E(x)$, as follows:*

- *If $E = E' \,|||\, [x \mapsto \dot{v}]$ then $E(x) = \dot{v}$.*
- *If $E = E' \,|||\, [x' \mapsto \dot{v}]$ where $x \neq x'$ then $E(x) = E'(x)$.*

Note that lookup of an unbound variable is undefined; that is, if $E = []$, then $E(x)$ is undefined for all x.

C.1.1 Input List Operational Semantics. We first present an operational semantics where inputs are read off of a list I.

Definition C.2. *The relation* $\Sigma, I \longrightarrow^1 \Sigma', I'$ *holds iff there is a proof using the rules of Figure 12. We inductively define* $\Sigma, I \longrightarrow^* \Sigma', I'$ *to hold iff either* $\Sigma, I \longrightarrow^1 \Sigma', I'$ *or both* $\Sigma, I \longrightarrow^* \Sigma'', I''$ *and* $\Sigma'', I'' \longrightarrow^1 \Sigma', I'$.

There is nothing too surprising in the rules, but note that the stack is not only for function calls but also for conditional bodies. This presentation is designed to better align with the reverse system. Input list I is part of the state, and each Input rule reads and removes one element from the list.

C.1.2 Mapping Input. Input is naturally a list that is read from incrementally as computation proceeds, but a demand-driven operational semantics looks up inputs in the order they are *used in reverse*, not how they are *defined going forward*, and we need to connect these two views. In the reverse semantics of the paper we used an order-free notion of inputs, ι, taking "heap locations" (uniquely identified by the call stack) onto values. So, next define an operational semantics where instead of reading inputs off of a list, we assume a fixed mapping ι of context-tagged variables ${}^C x$ to integers, as was used in the main paper body (see Definition 5). This is the forward operational semantics we will be able to align to the reverse system. The rules are nearly identical: only the input rule differs.

Recall that in the reverse system we support code which iteratively inputs an unbounded amount of data, and so a single program input clause $x = \texttt{input}$ may read in many input values. So, the domain of ι cannot just be variables x. In the reverse system, we added context C to make ${}^C x$ as the domain of ι, and we can do the same here. But, to do that we need to be able to extract the call stack from the Σ of our operational semantics. Fortunately, that is easy.

Definition C.3. *We define* $\textsc{StackName}(\Sigma) = C$ *inductively as follows:*

- $\textsc{StackName}([]) = []$
- $\textsc{StackName}([\langle E, [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma) = [x_1 = x_2\ x_3] \,||\, \textsc{StackName}(\Sigma)$
- $\textsc{StackName}([\langle E, [x_1 = x_2\ ?\ e_T\ :\ e_F] \,||\, e\rangle] \,||\, \Sigma) = \textsc{StackName}(\Sigma)$

Notice that we are only accumulating the function call sites and passing over the conditionals on the stack frames, since C contains only the function call portion. The main challenge in proving the list- and mapping-input views are equivalent is showing that the ${}^C x$ names do not collide: the call stack is sufficient to differentiate distinct runtime input clauses.

Definition C.4. *The relation* $\Sigma \longrightarrow^1_\iota \Sigma'$ *holds iff there is a proof using the rules of Figure 13. We inductively define* $\Sigma \longrightarrow^*_\iota \Sigma'$ *to hold iff either* $\Sigma \longrightarrow^1_\iota \Sigma'$ *or both* $\Sigma \longrightarrow^*_\iota \Sigma''$ *and* $\Sigma'' \longrightarrow^1_\iota \Sigma'$.

Inspecting the rules of Figure 13, they are nearly identical to the list-based input system. One difference is that ι is not part of the configuration state, it is fixed. Also, in the Input rule we extract the current calling context C from Σ and use that to construct pair ${}^C x$ which can be looked up in the mapping.

C.1.3 Proving the Two Input Representations Equivalent. We need to prove these two approaches to input are isomorphic. For this we construct two variations on the systems above, representing two different hybrids of them.

Our first is an instrumented version of the forward-running operational semantics that constructs ι from I. Each evaluation of a clause ${}^C x = \texttt{input}$ adds the fresh mapping ${}^C x \mapsto v$ to the input mapping ι, which is not fixed but is increasing; otherwise, every rule exhibits the same behavior as the list semantics of Figure 12.

The second instrumented system is a variation on the mapping semantics of Figure 13 which given fixed mapping ι can incrementally reconstruct the equivalent I. The relationship between these operational semantics in our equivalence proof is depicted in Figure 14: we prove each arrow in that Figure as an implication, showing the systems are all equivalent.

$$\textsc{Definition}\ \frac{x \text{ not of form } f}{[\langle E, [x = v] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E\,|||[x \mapsto v], e\rangle] \,||\, \Sigma}$$

$$\textsc{Closure}\ \frac{}{[\langle E, [x = f] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E\,|||[x \mapsto \langle f, E\rangle], e\rangle] \,||\, \Sigma}$$

$$\textsc{Input}\ \frac{C = \textsc{StackName}(\Sigma) \qquad \iota({}^C x) = n}{[\langle E, [x = \texttt{input}] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E\,|||[x \mapsto n], e\rangle] \,||\, \Sigma}$$

$$\textsc{Alias}\ \frac{E(x') = \dot{v}}{[\langle E, [x = x'] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E\,|||[x \mapsto \dot{v}], e\rangle] \,||\, \Sigma}$$

$$\textsc{Binop}\ \frac{E(x_2) = \dot{v}_2 \qquad E(x_3) = \dot{v}_3 \qquad \dot{v}_1 = \dot{v}_2 \odot \dot{v}_3}{[\langle E, [x_1 = x_2 \odot x_3] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E\,|||[x_1 = \dot{v}_1], e\rangle] \,||\, \Sigma}$$

$$\textsc{Call}\ \frac{\Sigma = [\langle E, [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma' \qquad E(x_2) = \langle [\texttt{fun}\ x_4\ \texttt{->}] \,||\, e', E'\rangle \qquad E(x_3) = \dot{v}}{\Sigma \longrightarrow_\iota^1 [\langle E'\,|||[x_4 \mapsto \dot{v}], e'\rangle] \,||\, \Sigma}$$

$$\textsc{Return}\ \frac{}{[\langle E\,|||[x \mapsto \dot{v}], []\rangle, \langle E', [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E'\,|||[x_1 = \dot{v}], e\rangle] \,||\, \Sigma}$$

$$\textsc{Conditional Start}\ \frac{\Sigma = [\langle E, [x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}] \,||\, e\rangle] \,||\, \Sigma' \qquad E(x_2) = \beta \qquad e_\beta = [x_1\ !\ \beta] \,||\, e'}{\Sigma \longrightarrow_\iota^1 [\langle E, e'\rangle] \,||\, \Sigma}$$

$$\textsc{Conditional End}\ \frac{}{[\langle E\,|||[x \mapsto \dot{v}], []\rangle, \langle E', [x_1 = x_2\ ?\ e_T : e_F] \,||\, e\rangle] \,||\, \Sigma \longrightarrow_\iota^1 [\langle E'\,|||[x_1 \mapsto \dot{v}], e\rangle] \,||\, \Sigma}$$

Fig. 13. Input-as-Mapping Forward Operational Semantics Rules

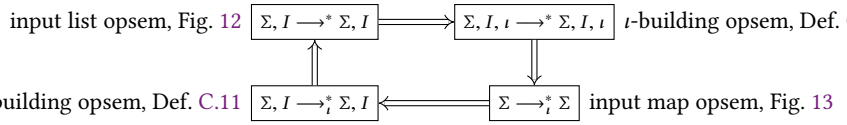

Fig. 14. Relating Input-as-list (I) to Input-as-Mapping (ι) interpreters; $\Longrightarrow$ are transformations

We now define $\Sigma, I, \iota \longrightarrow^1 \Sigma, I, \iota$.

Definition C.5. *Define $\Sigma, I, \iota \longrightarrow^1 \Sigma', I', \iota'$ as the relation defined by the rules of Figure 12, but with additional ι added to each state and for each non-Input rule mapping ι to itself. The new Input rule in this system is as follows:*

$$\textsc{Input}\ \frac{C = \textsc{StackName}(\Sigma)}{[\langle E, [x = \texttt{input}] \,||\, \Sigma\rangle] \,||\, \Sigma, ([n] \,||\, I), \iota \longrightarrow^1 [\langle E\,|||[x \mapsto n], \Sigma\rangle] \,||\, \Sigma, I, (\iota \cup \{{}^C x \mapsto n\})}$$

We inductively define $\Sigma, I, \iota \longrightarrow^* \Sigma', I', \iota'$ to hold iff either $\Sigma, I, \iota \longrightarrow^1 \Sigma', I', \iota'$ or both $\Sigma, I, \iota \longrightarrow^* \Sigma'', I'', \iota''$ and $\Sigma'', I'', \iota'' \longrightarrow^1 \Sigma', I', \iota'$.

In practice, we work with global derivations in which the initial mapping ι is empty. Since this system is doing nothing new other than accumulating constraints it is trivial to show that it is equivalent to the input-as-list system.

Lemma C.6. *$\Sigma, I \longrightarrow^* \Sigma', I'$ iff $\Sigma, I, \{\} \longrightarrow^* \Sigma, I, \iota$.*

Proof. Immediate by induction. □

The more meaningful result is that the accumulated mapping can be extracted from the ι-building semantics and then produce an equivalent run in the input-as-mapping semantics. Since the configurations are otherwise identical in the two systems, the only case where equivalence could fail is if there was a name collision: a derivation contained two different input steps which both used mapping key ${}^{C}x$. So, we first establish uniqueness of the ${}^{C}x$ keys as a Lemma.

The Lemma requires some auxiliary notation that we now define.

Definition C.7. (1) *Let $\dot{c}$, $\dot{e}$, $\dot{\sigma}$, and $\dot{\Sigma}$ be minor extensions to their respective grammars of figure 11 with additional clause $\dot{c} = \bullet$, and where $\dot{e}$, $\dot{\sigma}$, and $\dot{\Sigma}$ are in turn extended to use this new notion of clause. Additionally, there must be at most* one *occurrence of $\bullet$ in any $\dot{e}/\dot{\sigma}/\dot{\Sigma}$.*

(2) *We define hole filling, $\dot{\Sigma}[c]$, as the replacement of the (single) $\bullet$ in $\dot{\Sigma}$ with c. The result of this replacement is clearly a Σ.*

(3) *We can now use this notation to define the occurrence of a variable: let pair $(c, \dot{\Sigma})$ define an* occurrence *of c in Σ if $\dot{\Sigma}[c] = \Sigma$.*

(4) *Given a clause occurrence $(c, \dot{\Sigma})$, Define* ContFrames$(c, \dot{\Sigma})$ *to be Σ_1 where $\dot{\Sigma} = \Sigma_0 \,||\, \dot{\sigma} \,||\, \Sigma_1$.*

(5) *We overload* StackName$(c, \dot{\Sigma})$ *to mean* StackName(ContFrames$(c, \dot{\Sigma})$).

(6) *Given a step $\Sigma_0, I, \iota_0 \longrightarrow^1 \Sigma_1, I_1, \iota_1$ and some clause c with $\Sigma_0 = \dot{\Sigma}_0[c]$ and $\Sigma_1 = \dot{\Sigma}_1[c]$, we say c is* 1-common *to this step 6-tuple if $\dot{\Sigma}_0[c'], I, \iota_0 \longrightarrow^1 \Sigma_1[c'], I_1, \iota_1$ for any c': it is the same occurence across the step.*

(7) *Given an n-step computation sequence $\Sigma_0, I, \iota_0 \longrightarrow^* \Sigma_n, I_n, \iota_n$ and some clause c with $\Sigma_0 = \dot{\Sigma}_0[c]$ and $\Sigma_1 = \dot{\Sigma}_1[c]$, we say c is* common *to this sequence if it is 1-common to each of the $n-1$ steps in this computation sequence.*

We need to prove a stronger result for which uniqueness of input keys follows as an immediate corollary.

Lemma C.8. *given any computation sequence $\Sigma_0, I_0, \iota_0 \longrightarrow^* \Sigma_n, I_n, \iota_n$, if two distinct occurrences of the same c, $\Sigma_i = \dot{\Sigma}[c]$ and $\Sigma_j = \dot{\Sigma}'[c]$) (for which i may or not be equal to j) are not common to this sequence, that it must be the case that* StackName$(c, \dot{\Sigma})$ *and* StackName$(c, \dot{\Sigma}')$ *differ.*

Proof. Proceed by induction the length of the derivation, assuming the fact holds for shorter derivations. So, assume we have some derivation $\Sigma_0, I_0, \iota_0 \longrightarrow^* \Sigma_n, I_n, \iota_n$ with the above property, and show it holds this derivation extended by one additional step, $\Sigma_n, I_n, \iota_n \longrightarrow^1 \Sigma_{n+1}, I_{n+1}, \iota_{n+1}$. Proceed by cases on which rule fires in this nth-step.

All of the rules which do not change the stack have the property trivially holding by induction, as they only remove one clause and change nothing else. The conditional rules change the stack but do not add any new clauses, and the stack changes are also no-ops as far as StackName is concerned as it only is counting call frames. The Return rule also adds no new clauses.

Only Call adds clauses and so introduces the possibility of name clashes; let us consider that case in detail. By inspection of the rule, the step must be of the form

$$\Sigma_n, I_n, \iota_n \longrightarrow^1 [\langle E' \,|||\, [x_4 \mapsto \dot{v}], [c_1, \ldots, c_m]\rangle] \,||\, \Sigma_n, I_{n+1}, \iota_{n+1}$$

for $\Sigma_n = [\langle E, [x_1 = x_2\ x_3] \,||\, e\rangle] \,||\, \Sigma'$ and $E(x_2) = \langle [\texttt{fun}\ x_4\ \texttt{->}] \,|||\, [c_1, \ldots, c_m], E'\rangle$. Let us compute the StackName of $x_1 = x_2\ x_3$ in this redex: letting $\dot{\Sigma}'' = [\langle E, \bullet \,||\, e\rangle] \,||\, \Sigma'$, we have $\Sigma_n = \dot{\Sigma}''[x_1 = x_2\ x_3]$. So for this occurrence StackName$(x_1 = x_2\ x_3, \dot{\Sigma}'') = C$ for some C which by the induction hypothesis is unique for all non-common occurrences of $x_1 = x_2\ x_3$ in the derivation. We have expanded the e' in the rule to $[c_1, \ldots, c_m]$ to show the individual clauses added; we must establish for each of these clause occurrences c_i that their StackName is unique. The occurrence of some arbitrary c_i in the $(n+1)$-th stack is the pair $(c_i, \dot{\Sigma})$ for $\dot{\Sigma} = [\langle E' \,|||\, [x_4 \mapsto \dot{v}], [c_1, c_{i-1}, \bullet, c_{i+1}, \ldots, c_m]\rangle]$.

Computing, $\textsc{StackName}(c_i, \dot{\Sigma}) = [x_1 = x_2\ x_3] \,||\, \textsc{StackName}(\Sigma')$, which by inspection of the definitions equals $[x_1 = x_2\ x_3] \,||\, C$. Now, consider some other non-common occurrence of the same clause c_i in the derivation, i.e. some $\dot{\Sigma}'[c_i] = \Sigma_j$. We assert $\textsc{StackName}(c_i, \dot{\Sigma}')$ cannot be $[x_1 = x_2\ x_3] \,||\, C$: we know C was unique for clause $[x_1 = x_2\ x_3]$, and context $[x_1 = x_2\ x_3] \,||\, C$ can only arise by a call at call site $x_1 = x_2\ x_3$ from context C; so, $[x_1 = x_2\ x_3] \,||\, C$ must be unique for c_i since any previous Call at C cannot be at site $x_1 = x_2\ x_3$ by uniqueness of C for the call site and by inspection of the rules: once a call site occurrence fires, it cannot ever re-fire. □

Lemma C.9. *Given any computation sequence $\Sigma_0, I_0, \iota_0 \longrightarrow^* \Sigma_n, I_n, \iota_n$, for each Input step in the sequence, $\iota \cup \{{}^C x \mapsto n\}$ is a disjoint union: ${}^C x \notin \textsc{Dom}(\iota)$.*

Proof. This folllows directly from Lemma C.8: let the clause c of that Lemma above be an input $x = \texttt{input}$ occurring in steps $\Sigma_i = \dot{\Sigma}[c]$ and $\Sigma_j = \dot{\Sigma}'[c]$) which are not common occurrences. By the above property, $\textsc{StackName}(c, \dot{\Sigma})$ and $\textsc{StackName}(c, \dot{\Sigma}')$ differ. So, the ${}^C x$ produced by the two will be unique. □

Lemma C.10. *If $\Sigma_0, I_0, \{\} \longrightarrow^* \Sigma_n, I_n, \iota_n$ then $\Sigma_0 \longrightarrow^*_{\iota_n} \Sigma_n$.*

Proof. Proceed by induction on the number of steps in the assumption to establish the goal. Assume the property holds for computations of length less than k, show for k. Before getting into the proof, observe that $\iota_i \subseteq \iota_n$ for all i by inspection of the rules: ι is monotonically increasing.

For the non-Input case the result is trivial by induction.

Now consider Input. In the hybrid system rule the mapping portion steps from ι_{k-1} to $\iota_k = \iota_{k-1} \cup \{{}^C x \mapsto n\}$. By the induction hypothesis, in the ι system we can step to the point before this point with identical Σ. The Input rule step in that system then is also identical but needs $\iota_n({}^C x) = n$. And, this follows trivially from the observation that $\iota_k({}^C x) = n$ from the hybrid system conclusion the fact that $\iota_k \subseteq \iota_n$, and Lemma C.9 which requires ι_n to indeed be a (deterministic) mapping. □

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.

Definition C.11. *Define $\Sigma, I \longrightarrow^1_\iota \Sigma', I'$ as the relation defined by the rules of Figure 13, but with additional I parameter added to each state and for each non-Input rule mapping I to itself. The new Input rule in this system is then as follows:*

$$\textsc{Input}\ \frac{C = \textsc{StackName}(\Sigma) \qquad \iota({}^C x) = n}{[\langle E, [x = \texttt{input}] \,||\, e\rangle] \,||\, \Sigma, I \longrightarrow^1_\iota [\langle E \,||[x \mapsto n], e\rangle] \,||\, \Sigma, (I \,||[n])}$$

We inductively define $\Sigma, I \longrightarrow^*_\iota \Sigma', I'$ to hold iff either $\Sigma, I \longrightarrow^1_\iota \Sigma', I'$ or both $\Sigma, I \longrightarrow^*_\iota \Sigma'', I'', \iota''$ and $\Sigma'', I' \longrightarrow^1_\iota \Sigma', I'$. In a global derivation we begin with $I = []$.

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

Lemma C.12. *If $e_0, [] \longrightarrow^*_\iota e_n, I$ then $e_0, I \longrightarrow^* e_n, []$.*

Proof. Suppose $e_0, [] \longrightarrow^*_\iota e_n, I$. In order to capture the intermediate states of this run we strengthen the induction hypothesis to show $e_0, [] \longrightarrow^*_\iota e_n, I_l$ for $I = I_l \,||\, I_r$ implies $e_0, I \longrightarrow^* e_n, I_r$. Proceed by induction on the length of the derivation, assuming the property holds for less than k

$$\begin{array}{lcll} \tilde{c} & ::= & c \mid \epsilon & \textit{optional clauses} \\ \sigma & ::= & \langle E, e, \tilde{c} \rangle & \textit{tracking stack frames} \\ S & ::= & [\varsigma, \ldots] & \textit{tracking stacks} \end{array}$$

Fig. 15. Tracking Operational Semantics Grammar

steps and show for the k-th step. As before, non-Input rules follow trivially by induction. Consider the case where the last step of the hybrid system was an Input. In this case, transition in the rule was from input list I' to $I_l = (I' \,||[n])$. In the input list system the previous step must then have had input list $[n] \,||\, I_r$ by induction, and so by inspection of the Input rule for the input list system it can then step to a state with list I_r after reading off input $[n]$, establishing the result. □

C.1.4 A Semantics Tracking the Previous Clause. Now that we have shown that input can be represented as either a list or a mapping, we prepare to show the correctness of reverse lookup with respect to an input-as-mapping operational semantics. In preparation for this, we must make a small change to that operational semantics to align it with reverse lookup: we must keep track of the most recently evaluated clause in each stack frame. We address this by defining an equivalent and nearly identical operational semantics here. The relevant grammar appears in Figure 15.

We must also provide auxiliary definitions for relating the two systems and their operations:

Definition C.13. *Let* $\textsc{Forget}(\langle E, e, \tilde{c} \rangle) = \langle E, e \rangle$. *We overload this definition to lists of stack frames homomorphically:* $\textsc{Forget}([]) = []$ *and* $\textsc{Forget}([\varsigma] \,||\, S) = [\textsc{Forget}(\varsigma)] \,||\, \textsc{Forget}(S)$.

Definition C.14. *We define* $\textsc{StackName}(S) = \textsc{StackName}(\textsc{Forget}(S))$.

Given these definitions, we then formally define this tracking operational semantics as follows:

Definition C.15. *The relation* $S \longrightarrow_\iota^1 S'$ *holds iff there is a proof using the rules of Figure 16. We inductively define* $S \longrightarrow_\iota^* S'$ *to hold iff either* $S \longrightarrow_\iota^1 S'$ *or both* $S \longrightarrow_\iota^* S''$ *and* $S'' \longrightarrow_\iota^1 S'$.

Note that the changes in this operational semantics are largely notational. In each rule, the just-processed clause is moved from the expression to the new third position in the stack rather than being discarded entirely. In the Call and Conditional Start rules, the underlying stack frame is not affected; in the Call case, for instance, the function header is removed from the function body. The header is treated as the just-processed clause (as it has been bound to the argument value) and the body of the function is placed in the stack frame for execution.

None of these changes impact how the program actually runs, so these operational semantics are equivalent. We formally state and prove equivalence between the two systems as follows:

Lemma C.16. $[\langle [], e_{glob} \rangle] \longrightarrow_\iota^* \Sigma$ *iff* $[\langle [], e_{glob}, \epsilon \rangle] \longrightarrow_\iota^* S$ *such that* $\textsc{Forget}(S) = \Sigma$.

Proof. By proving each implication of the equivalence separately, first by induction on the height of the proof tree and then by the proof rule used. In each case, the preconditions provided by one proof tree are together with the inductive hypothesis sufficient to construct the other proof tree. □

C.2 Auxiliary Lemmas for Soundness and Completeness

Our next goal is to demonstrate the equivalence of the tracking operational semantics with the reverse lookup relation in Definition 3.3. To do so, we must first prove a number of properties about this forward operational semantics which are instrumental in showing this equivalence.

$$\textsc{Definition}\ \frac{c = (x = v) \qquad x \text{ not of form } f}{[\langle E, [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E\,||[x \mapsto v], e, c\rangle] \,||\, S}$$

$$\textsc{Closure}\ \frac{c = (x = f)}{[\langle E, [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E\,||[x \mapsto \langle f, E\rangle], e, c\rangle] \,||\, S}$$

$$\textsc{Input}\ \frac{c = (x = \texttt{input}) \qquad C = \textsc{StackName}(S) \qquad \iota(^C x) = n}{[\langle E, [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E\,||[x \mapsto n], e, c\rangle] \,||\, S}$$

$$\textsc{Alias}\ \frac{c = (x = x') \qquad E(x') = \dot{v}}{[\langle E, [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E\,||[x \mapsto \dot{v}], e, c\rangle] \,||\, S}$$

$$\textsc{Binop}\ \frac{c = (x_1 = x_2 \odot x_3) \qquad E(x_2) = \dot{v}_2 \qquad E(x_3) = \dot{v}_3 \qquad \dot{v}_1 = \dot{v}_2 \odot \dot{v}_3}{[\langle E, [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E\,||[x_1 = \dot{v}_1], e, c\rangle] \,||\, S}$$

$$\textsc{Call}\ \frac{S = [\langle E, [x_1 = x_2\ x_3] \,||\, e, \tilde{c}\rangle] \,||\, S' \qquad E(x_2) = \langle [\texttt{fun}\ x_4\ \texttt{->}] \,||\, e', E'\rangle \qquad E(x_3) = \dot{v}}{S \longrightarrow_\iota^1 [\langle E'\,||[x_4 \mapsto \dot{v}], e', \texttt{fun}\ x_4\ \texttt{->}\rangle] \,||\, S}$$

$$\textsc{Return}\ \frac{c = (x_1 = x_2\ x_3)}{[\langle E\,||[x \mapsto \dot{v}], [], \tilde{c}'\rangle, \langle E', [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E'\,||[x_1 = \dot{v}], e, c\rangle] \,||\, S}$$

$$\textsc{Conditional Start}\ \frac{\begin{array}{c} S = [\langle E, [x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}] \,||\, e, \tilde{c}\rangle] \,||\, S' \\ E(x_2) = \beta \qquad e_\beta = [x_1\ !\ \beta] \,||\, e' \end{array}}{S \longrightarrow_\iota^1 [\langle E, e', x_1\ !\ \beta\rangle] \,||\, S}$$

$$\textsc{Conditional End}\ \frac{c = x_1 = x_2\ ?\ e_T : e_F}{[\langle E\,||[x \mapsto \dot{v}], [], \tilde{c}'\rangle, \langle E', [c] \,||\, e, \tilde{c}\rangle] \,||\, S \longrightarrow_\iota^1 [\langle E'\,||[x_1 \mapsto \dot{v}], e, c\rangle] \,||\, S}$$

Fig. 16. Input-as-Mapping Forward Operational Semantics Rules

C.2.1 Properties of Evaluation. We begin with a simple property: demonstrating that any proof of evaluation of a well-formed expression uses one of only two of the operational semantics rules.

Lemma C.17. *For any well-formed e_{glob} and any ι, any proof of $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^1 S$ uses either the Definition rule or Input rule from Figure 16.*

Proof. By case analysis to exclude all other cases. The Closure rule is not used because a well-formed e_{glob} does not start with a function definition clause (Definition 3.1). All other rules other than Definition and Input operate on clauses which contain free variables. If e_{glob} is closed, then no free variables appear in its first clause; thus, those other rules are also excluded. □

We next demonstrate that the first variable of the program appears in *every* environment which is created during evaluation. This is especially helpful when proving equivalence with the reverse lookup system, as it relies heavily upon being able to look up the first program variable (Definition 3.3.

Lemma C.18. *Suppose a well-formed e_{glob} such that $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^* S$. For any E appearing within S (including within the closures of other environments), $(\textsc{FirstV}(e_{glob}) \mapsto \dot{v}) \in E$ for some $\dot{v}$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon \rangle \longrightarrow_\iota^* S$. Lemma C.17 gives us that the base case only contends with the Definition and Input rules. In the base case, each of these rules directly adds a mapping for the program's first variable to the resulting environment.

In the inductive step, we proceed by case analysis on the proof rule used. In the Definition, Input, Alias, and Binop rules, the only change from the previous stack (which maintains this property by the inductive hypothesis) is that a new mapping is added to the environment of the topmost stack frame. This mapping is either not a closure or is a pre-existing closure (and so already satisfies this property), so these cases are finished.

If the Closure rule is used, a new closure is added to the top stack frame's environment. This closure contains a copy of the top stack frame's previous environment. By the inductive hypothesis, this new closure contains a mapping for $\textsc{FirstV}(e_{\text{glob}})$ and this case is finished.

If the Call rule is used, a new stack frame is added to the top of the stack with a new environment. This environment is an extension of one appearing in a closure in the environment of the old top stack frame so, by the inductive hypothesis, it also contains a $\textsc{FirstV}(e_{\text{glob}})$ mapping and this case is finished.

If the Conditional Start rule is used, a new stack frame is added to the top of the stack with a copy of the environment from the old top stack frame. By the inductive hypothesis, this environment also contains an appropriate mapping, so these cases are finished.

Otherwise, either the Return rule or the Conditional End rule is used. These rules remove a frame from the top of the stack and extend the environment stack frame beneath it. Thus, by the inductive hypothesis, this extended environment contains a mapping for $\textsc{FirstV}(e)$ and these cases are also finished. □

A number of lemmas rely upon the fact that new expressions are not synthesized during evaluation. We formalize that property as follows:

Lemma C.19. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon \rangle \longrightarrow_\iota^* S$. Then any e appearing in S either appears in e_{glob} or is a suffix of an expression that appears in e_{glob}.*

Proof. By induction on the length of $\langle [], e_{\text{glob}}, \epsilon \rangle \longrightarrow_\iota^* S$. In the base case, Lemma C.17 gives us that $S = [\langle [x \mapsto v], e, \textsc{Cl}(x) \rangle]$ where $[\textsc{Cl}(x)] \,||\, e = e_{\text{glob}}$; the only expression appearing here is e, which is a suffix of e_{glob} itself.

In the inductive case, we proceed by case analysis on the rule used. In the case of the Definition, Input, Alias, and Binop rules, the only change to the expressions in the stack is the removal of a clause from the expression in the top stack frame, so this property holds by the inductive hypothesis. The Closure rule introduces a new closure to the environment containing a function's body; that function appeared in the top stack frame's expression previously and so this property holds by the inductive hypothesis.

In the Call case, a new stack frame is created containing an expression and environment drawn from a closure in the previous top stack frame's environment; thus, by the inductive hypothesis, any expressions appearing in this closure (and therefore in the new top stack frame) appear in e_{glob}. The Conditional Start case is similar except that the new top stack frame's expression is part of the previous top stack frame's expression rather than part of its environment.

In the Return and Conditional End cases, a top stack frame is discard which induces no proof obligation. The new top stack frame contains an expression which is a suffix of one appearing in the previous stack, so this case is also addressed by the inductive hypothesis. □

We also rely upon the fact that expressions are closed by their respective environments throughout evaluation. This is particularly helpful when proving that variables which are free in the next-to-be-executed clause are bound in the current environment.

Lemma C.20. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^1 S$ such that $S = S_1 \,||[\langle E, e, \tilde{c}\rangle]\,||\, S_2$. For any free variable x appearing in e, a binding $x \mapsto \dot{v}$ for some $\dot{v}$ appears in E. Likewise, for any closure $\langle f, E'\rangle$ appearing recursively in E and any free variable x in f, a binding $x' \mapsto \dot{v}$ for some $\dot{v}$ appears in E'.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^1 S$. The Definition, Input, Alias, and Binop rules immediately remove a variable-binding expression ($x = b$) and add a non-closure binding for x to the environment. Removing this expression makes occurrences of x free in the remainder of the expression; the new x binding in the environment preserves the invariant.

The Closure rule operates similarly but adds a closure to the environment. This imposes an additional proof obligation: that the free variables of the closure's function are bound by its environment. This is satisfied by the inductive hypothesis: the function appeared in the current stack frame's expression, that expression's free variables are bound by the stack frame's environment, and that environment is the environment of the new closure.

The Call and Conditional Start rules are similar to the non-closure cases above except that they create a new stack frame. In the Call rule, this new stack frame's environment and expression are taken from a closure (and then the first clause is processed as above), so the closure property of the inductive hypothesis satisfies the proof obligation for this new stack frame. Conditional Start rule copies the current environment as well as a subexpression of the current expression, immediately satisfying the proof obligation.

The Return and Conditional End rules are similar to the non-closure cases above except that they discard a stack frame. Discarding a stack frame imposes no additional proof obligation. The single clause processed by these rules satisfies the proof obligation in the same fashion as the non-closure cases above. □

C.2.2 Properties of Predecessors and Tracking. Next, we demonstrate that, during evaluation, each clause in an expression on the stack is the predecessor of the one that follows it. This is already true of a well-formed program; we here demonstrate that it is true of the subexpressions after each step of evaluation.

Lemma C.21. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^* S$. For any e appearing in S (in an environment, an expression, or a stack frame), if $e = [\ldots, x_1 = b_1, x_2 = b_2, \ldots]$ then $\textsc{Pred}(x_2) = x_1$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^* S$. In particular, no rule synthesizes a new sequence of clauses; all expressions which appear during evaluation are substrings of the expressions which appeared in e_{glob}, so no new Pred relations need to be proven. □

Further, after the first step of evaluation, the tracking clause is never the placeholder ϵ; it is always a valid clause.

Lemma C.22. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^* S$. For any stack frame $\langle E, e, \tilde{c}\rangle$ appearing in S, $\tilde{c} \neq \epsilon$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^* S$. This is immediate by inspection of the rules: ϵ does not appear on the right side of any small step. □

We also demonstrate the predecessor relationship between the new tracking clause at the current expression for each stack frame.

Lemma C.23. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^* S$. For any stack frame $\langle E, e, \tilde{c}\rangle$ appearing in S, $\tilde{c} = c'$ for some non-ϵ c' and $e = [c]\,||\,e'$ iff $\textsc{Pred}(c) = c'$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon \rangle \longrightarrow_\iota^* S$. We have $\tilde{c} = c'$ by Lemma C.22. The predecessor relationship is immediate by Lemma C.21 in each rule. Note that, for each rule, we must determine if the resulting expression is non-empty or not before applying Lemma C.21; if the resulting expression is empty, we have no proof obligation for that stack frame. □

We also show that a given clause in e_{glob} may either be a return clause or a predecessor of another clause; it cannot be both.

Lemma C.24. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon \rangle \longrightarrow_\iota^1 S$. For any e appearing in S (including in closures within environments), there is no c such that* $\textsc{Pred}(c) = \textsc{RetCl}(e)$.

Proof. By induction on the length of $\langle [], e_{\text{glob}}, \epsilon \rangle \longrightarrow_\iota^1 S$. In the base case, only one such expression appears – e_{glob} – and, by Definition 3.2, this property holds for all expressions in it. In the inductive step, the only expressions added to stack frames are (1) those drawn from previous stack frames or (2) suffixes of them. The set of return clauses for all expressions in the stack is therefore fixed throughout execution and so the property established in the base case holds at each inductive step. □

C.2.3 Properties of Application and Conditionals. We now demonstrate the coherence of stack frames throughout execution. As we evaluate the program, we add stack frames at the beginning of function calls and conditionals and remove stack frames at the end of function calls and conditionals. Intuitively, the stack frame appearing above e.g. a call is the body of the function that is called. While this is immediately evident when the function is entered, we provide these lemmas to reinforce that property as we finish the stack frame. These lemmas are specifically tailored toward requirements that appear during the proof of completeness.

Lemma C.25. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon \rangle \longrightarrow_\iota^* S$. Suppose also that $S = S_3 \,||[\varsigma_2, \varsigma_1]||\, S_0$ where $c_1 = (x_1 = x_2\ x_3)$, $\varsigma_2 = \langle E_2, e_2, c_L' \rangle$, and $\varsigma_1 = \langle E_1, [c_1] \,||\, e_1', c_L \rangle$. If $e_2 = []$ then let $c' = c_L'$; otherwise, let $c' = \textsc{RetCl}(e_2)$. From this, we can conclude $E_1(x_2) = \langle f, E' \rangle$ and $\textsc{RetCl}(f) = c'$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon \rangle \longrightarrow_\iota^1 S$. In the base case, Lemma C.17 gives us that $S = [\langle E, e', \tilde{c} \rangle]$, so this property (which universally quantifies over adjacent pairs of stack frames) is vacuously true.

In the inductive step, all stack frames except for the top two are satisfied as necessary by the inductive hypothesis. We proceed by case analysis of the rule used.

In the Definition, Closure, Input, Alias, and Binop rules, we observe that the size of the stack does not change. A single clause is removed from the expression and its defined variable is mapped in the environment. If this clause is the last clause in the expression, then the inductive hypothesis (in the $\textsc{RetCl}(e_2)$ case) shows this property (in the c_L' case) for the top two stack frames. Otherwise, both the inductive hypothesis and the property use the $\textsc{RetCl}(e_2)$ case for the top two stack frames. All other adjacent stack frames use the inductive hypothesis directly.

In the Call case, a new stack frame is introduced. Its expression taken from the body of the called function; this establishes the property between the new topmost stack frame and the stack frame beneath it. All other adjacent stack frames use the inductive hypothesis directly.

In the Conditional Start case, a new stack frame is added but the stack frame immediately beneath it does not have an expression with a first clause in the form of an application. Thus, this property is established vacuously for that pair of adjacent stack frames.

In the Return and Conditional End cases, an existing stack frame is removed. This induces no proof obligations and all other adjacent stack frames use the inductive hypothesis directly. □

We must similarly formalize the coherence of conditional stack frames.

Lemma C.26. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^1 S$. Suppose also that $S = S_1 \,||[\langle E, e, c_L\rangle, \langle E', [c] \,||\, e', c'_L\rangle] \,||\, S_2$ where $c = (x_1 = x_2 \,?\, e_{\texttt{true}} : e_{\texttt{false}})$. Then $E'(x_2) = \beta$ for some β. Further: if $e \neq []$ then $\textsc{RetCl}(e) = \textsc{RetCl}(e_\beta)$; otherwise, $e = []$ and $c_L = \textsc{RetCl}(e_\beta)$.*

Proof. By induction on the length of the proof of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^1 S$. This proof proceeds exactly as in Lemma C.25 above using a boolean value rather than a function and using the two branches of the conditional rather than a function's body. □

We show a similar property on functions and conditionals to satisfy a corresponding proof obligation during the proof of soundness. These proofs have similar (but not identical) properties and preconditions.

Lemma C.27. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^* S$ such that $S = S_1 \,||[\langle E, e, c\rangle] \,||\, S_2$. If $e = []$, then let $c'' = c$; otherwise, let $c'' = \textsc{RetCl}(e)$. If $c'' = \textsc{RetCl}(f)$ for f appearing in e_{glob} then $S_2 = [\langle E', e', c'\rangle] \,||\, S_3$ where $e' = [x_1 = x_2\ x_3] \,||\, e''$ and $E'(x_2) = \langle f, E''\rangle$.*

Proof. By induction on the length of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^1 S$. In the base case, Lemma C.17 gives us that $S = [\langle E, e, c\rangle]$ and Lemma C.23 together with Definition 3.2 gives us that $e_{\text{glob}} = [c] \,||\, e$; thus, c does not appear within a function and this condition is vacuously satisfied.

In the inductive step, we have two cases: the topmost stack frame's return clause (either the return of its current expression or, if that expression is empty, the last executed clause) is the return clause of a function in e_{glob} or it is not. We begin by considering the case in which it is not.

When the top stack frame's return clause is not the return clause for a function in e_{glob}, the Definition, Closure, Input, Alias, and Binop rules satisfy this property directly from the inductive hypothesis: only the topmost stack frame changes and it induces no proof obligation.

When the Call rule is used, a new stack frame is added. This new stack frame contains an expression taken from a function in the old top stack frame's environment. By Lemma C.19, this function appears in e_{glob}, so the stack frame beneath it must have an expression beginning with an application clause. Because the Call rule applies, we know this to be true.

When the Conditional Start rule is used, a new stack frame is added which contains an expression taken from a conditional in the old top stack frame's environment. By Lemma C.19, this conditional appears in e_{glob}. Because e_{glob} is well-formed (and therefore alphatized), this expression cannot appear as the return clause of a function in e_{glob} and so no proof obligation is induced.

The Return and Conditional End rules remove a stack frame and therefore induce no new proof obligations. The obligations imposed by the remainder of the stack are satisfied by the inductive hypothesis.

Otherwise, the top stack frame's return clause *is* the return clause for a function appearing in e_{glob}. We proceed again by case analysis on the rule used. These cases proceed exactly as above but with an additional observation: each step of evaluation either does not change the expression of the topmost stack frame or changes it by removing the first clause and moving it to the position of the last executed clause. When the expression is empty, a step will remove the stack frame; when an expression is a singleton, its return clause is moved to the last executed clause; and when an expression is not a singleton, its return clause does not change. In each of these cases, the property is preserved on the topmost stack frame; the property is preserved on the remaining stack frames by the inductive hypothesis because they do not change (or, in the case of Return and Conditional End, they change in the exact same fashion). □

We have a similar property for satisfying the soundness proof obligation for conditionals.

Lemma C.28. *For any well-formed e_{glob} and any ι, suppose $\langle [], e_{glob}, \epsilon\rangle \longrightarrow_\iota^1 S$ such that $S = S_1 \,||[\langle E, e, c\rangle]\,||\, S_2$. If $e = []$, then let $c'' = c$; otherwise, let $c'' = \textsc{RetCl}(e)$. If $c'' = \textsc{RetCl}(e_\beta)$ for $c''' = (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}})$ appearing in e_{glob} then $S_2 = [\langle E', e', c'\rangle]\,||\, S_3$ where $e' = [c''']\,||\, e''$.*

Proof. By induction on the length of $\langle [], e_{\text{glob}}, \epsilon\rangle \longrightarrow_\iota^1 S$. This proof proceeds exactly as Lemma C.27 using a boolean value rather than a function. □

C.2.4 Properties of Determinism. In discussing the deterministic properties of the tracking operational semantics, we begin by proving simple determinism on that relation.

Lemma C.29. *The relation $S \longrightarrow_\iota^1 S'$ is deterministic for well-formed expressions. That is, if $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow^* S$, $S \longrightarrow_\iota^1 S_1'$, and $S \longrightarrow_\iota^1 S_2'$ then $S_1' = S_2'$.*

Proof. By case analysis on the rules used in the proofs of $S \longrightarrow_\iota^1 S_1'$ and $S \longrightarrow_\iota^1 S_2'$. □

More precisely, however, we demonstrate that each of the deterministic steps taken by this relation can be uniquely identified by the $\tilde{c}$ in the topmost stack frame and the C name of the remainder of the stack.

Lemma C.30. *Suppose $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_1$ and $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_2$ such that $S_1 = [\langle E_1, e_1, c\rangle]\,||\, S_1'$ and $S_2 = [\langle E_2, e_2, c\rangle]\,||\, S_2'$. Suppose further that $\textsc{StackName}(S_1') = \textsc{StackName}(S_2')$. Then $S_1 = S_2$.*

Proof. By Lemma C.29, there is only one computation sequence with fixed start state $[\langle [], e_{\text{glob}}, \epsilon\rangle]$ and so S_1 is either before or after S_2 in this sequence; if they are the same (neither before nor after), it means we have our desired result, $S_1 = S_2$, so let us assume $S_1 \neq S_2$ and derive a contradiction. Suppose w.o.l.o.g. then that $S_1 \longrightarrow_\iota^* S_2$. By inspection of the rules and Lemma C.23, we know that clause c fired twice in this computation, once immediately before S_1 and once before S_2. Since in each case the clause c was removed, it means that two occurrences cannot be common (as per Definition C.7). So, by Lemma C.8, $\textsc{StackName}(S_1') \neq \textsc{StackName}(S_2')$, contradicting our hypothesis and thus establishing the result. □

Lemma C.31. *Let $S = [\varsigma_n, \ldots, \varsigma_0]$ and let $S' = [\varsigma_m', \ldots, \varsigma_0']$. If $S \longrightarrow_\iota^1 S'$ then, for each $i \in \{0, \ldots, n\}$, one of the following is true:*

- *$\varsigma_i = \varsigma_i'$ (when $i < m$)*
- *$\varsigma_i = \langle E, [\textsc{Cl}(x)]\,||\, e, \tilde{c}\rangle$ and $\varsigma_i' = \langle E\,||[x \mapsto \dot{v}], e, \textsc{Cl}(x)\rangle$ for some x (when $i = m$)*
- *$i > m$ (and so the stack frame is removed)*

Proof. By case analysis on the proof rule used. □

Lemma C.32. *Suppose $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_1$ and $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_2$ such that $S_1 = [\langle E_1, e_1, \textsc{Pred}(c)\rangle]\,||\, S_1'$ and $S_2 = [\langle E_2, e_2, c\rangle]\,||\, S_2'$ such that $\textsc{StackName}(S_1') = \textsc{StackName}(S_2')$. Then $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_1 \longrightarrow_\iota^* S_2$.*

Proof. By Lemma C.23 we have $e_1 = [c]\,||\, e_2$. We conclude $S_1 \neq S_2$ immediately from this. Because the operational semantics is deterministic (Lemma C.29, we have either $S_1 \longrightarrow_\iota^* S_2$ or $S_2 \longrightarrow_\iota^* S_1$. Assume for contradiction that $S_2 \longrightarrow_\iota^* S_1$.

By induction on Lemma C.31, we have that expressions within a stack frame become monotonically smaller at each step; thus, a stack frame containing e_2 will never contain e_1 in a future step. By Lemma C.30, we will not encounter a stack frame with the same name containing this expression again in the future. This prevents S_2 from stepping to S_1, a stack with a strictly larger expression in the same context. By this contradiction, we conclude $S_1 \longrightarrow_\iota^* S_2$. □

Finally, we use these properties to demonstrate the relationship between two different environments appearing in the same stack name of execution but at different clauses (no matter how many steps or stack frames occur between them).

Lemma C.33. *Suppose* $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_1$ *and* $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S_2$ *such that* $S_1 = [\langle E_1, e_1, c_1\rangle] \,||\, S_1'$ *and* $S_2 = [\langle E_2, e_2, c_2\rangle] \,||\, S_2'$. *Suppose further that* $\textsc{StackName}(S_1') = \textsc{StackName}(S_2')$ *and that* $c_1 = \textsc{Pred}(c_2)$. *Then* $E_2 = E_1 \,||[x \mapsto \dot{v}]$ *such that* $\textsc{Cl}(x) = c_2$.

Proof. By Lemma C.32 we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_1 \longrightarrow_\iota^* S_2$. Because $c_1 = \textsc{Pred}(c_2)$ and by Lemma C.23, we have that $e_1 = [c_2] \,||\, e_2$. Note that e_1 is non-empty. Using Lemma C.31 by induction on length of $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_2$, we have that each step either does not affect the stack frame $\langle E_1, e_1, c_1\rangle$ or reduces its expression by one clause to e_2 while adding a mapping $(x \mapsto \dot{v})$ to E_1 to produce E_2. (This stack frame cannot be removed because e_1 is non-empty, as per Lemma C.31.) □

C.3 Completeness of Reverse Lookup

We are now prepared to prove the completeness of reverse lookup. We begin by giving a definition of a function called Extract which is designed to retrieve a value from a potentially nested sequence of environments. The extraction function uses the lookup stacks X of the reverse lookup system effectively to name the closure in which a particular value appears. This matches helpfully with the manner in which the reverse lookup system uses the lookup stack to refer to non-local variables (which are captured in a closure in a forward-running system).

Definition C.34. *We denote the* extraction *of a value from an environment E using a location X as* $\textsc{Extract}(E, X)$. *We define this function as follows:*

- *If* $X = [x]$ *then* $\textsc{Extract}(E, X) = E(x)$.
- *If* $X = [x] \,||\, X'$ *and* $E(x) = \langle f, E'\rangle$ *then* $\textsc{Extract}(E, X) = \textsc{Extract}(E', X')$.

We also provide a helpful definition for relating environment values to program values.

Definition C.35. *We define* $\textsc{RawVal}(\dot{v}) = v$ *as follows:*

- $\textsc{RawVal}(v) = v$
- $\textsc{RawVal}(\langle f, E\rangle) = f$

We begin by proving completeness for the first step of evaluation (to serve as a base case for induction later). This serves as a simple warm-up to how these systems relates but, more importantly, addresses several corner cases which will simplify the inductive step below.

Lemma C.36. *Suppose* $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^1 S$ *such that* $e_{glob} = [c] \,||\, e$ *and* $c = (x = b)$. *Then* $S = [\langle E, e, c\rangle]$ *such that, for all* $\textsc{Extract}(E, X) = \dot{v}$, *we have* $\mathbb{L}(X, c, []) \equiv \textsc{RawVal}(\dot{v})$.

Proof. By case analysis on the rule used in $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^1 S$. By Lemma C.17, this will be either the Definition rule or the Input rule.

If the Definition rule is used, then $c = (x = v)$. Then $S = [\langle [x \mapsto v], e, c\rangle]$. By Definition C.1, $E(x')$ is defined only when $x' = x$. Note also that $E(x) = v$ and v is not a closure. Thus, by Definition C.34, $\textsc{Extract}(E, X)$ is only defined when $X = [x]$. It therefore suffices to show that $\mathbb{L}([x], c, []) \equiv v$. Because $x = \textsc{FirstV}(e_{\text{glob}})$, this is true by the Value Discovery rule in Figure 6 and so this case is finished.

Otherwise, the Input rule is used and so $c = (x = \texttt{input})$. The argument proceeds exactly as in the Definition rule case, observing that the assigned value v is an integer (and therefore not a function) and using the Input rule of Figure 6. □

To prove the inductive step, we must formally relate the state of a forward-running program with the properties demonstrated by the reverse lookup system. This relationship must span the particular gap that reverse lookup is concerned only with one lookup at a time while the operational semantics maintains entire environments. We establish this as follows:

Definition C.37. *A program state S* agrees with reverse lookup *if, for all S_1, S_2, X, the conditions*

- $S = S_1 \,||[\langle E, e, c\rangle]\,||\, S_2$,
- $\textsc{Extract}(E, X) = \dot{v}$, *and*
- $C = \textsc{StackName}(S_2)$

imply that $\mathbb{L}(X, c, C) \equiv \textsc{RawVal}(\dot{v})$.

Note that this property is a generalization of the conclusions of the base case lemma above.

We are now prepared to demonstrate that this property is preserved by evaluation.

Lemma C.38. *For any well-formed e_{glob} such that $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_t^* S$ and S agrees with reverse lookup, if $S \longrightarrow_t^1 S'$ then S' agrees with reverse lookup.*

Proof. By case analysis on the rule used to show $S \longrightarrow_t^1 S'$.

Definition: If the Definition rule is used, then $S = [\langle E, [c]\,||\,e', \tilde{c}\rangle]\,||\,S_0$ where $c = (x = v)$ and $S' = [\langle E', e', c\rangle]\,||\,S_0$ where $E' = E\,||[x \mapsto v]$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Because only the top stack frame changes between S and S', we only need to show the agreement property of Definition C.37 for the topmost stack frame. Let $C = \textsc{StackName}(S_0)$. We begin by selecting without loss of generality some X such that $\textsc{Extract}(E', X) = \dot{v}$. Either X begins with x or it does not.

Suppose X begins with x. Note that v is not a closure. Therefore by Definition C.34, $\dot{v} = v$ and $X = [x]$. Let $x' = \textsc{FirstV}(e_{\text{glob}})$. By Lemma C.18, we have that $E(x')$ is defined; therefore, $\textsc{Extract}(E, [x'])$ is defined. Because S agrees with reverse lookup, we have $\mathbb{L}([x'], c_L, C) \equiv v'$. By Lemma C.23, $c_L = \textsc{Pred}(x)$, so $\mathbb{L}([x'], \textsc{Pred}(x), C) \equiv v'$. By Definition 3.3, this satisfies $\textsc{First}(x, c, C)$. Since $\textsc{RawVal}(\dot{v}) = \dot{v} = v$, we have by the Value Discovery rule of Figure 6 that $\mathbb{L}([x], c, C) \equiv \textsc{RawVal}(\dot{v})$ and this case is finished.

Otherwise, the Definition rule is used and X does not start with x. We observe that $\textsc{Extract}(E', X) = \dot{v} = \textsc{Extract}(E, X)$ by induction on the height of the proof of $\textsc{Extract}(E', X)$. Next, we observe that, because S agrees with reverse lookup, we have $\mathbb{L}(X, c_L, C) \equiv \textsc{RawVal}(\dot{v})$. By Lemma C.23, we have $c_L = \textsc{Pred}(x)$ and so $\mathbb{L}(X, \textsc{Pred}(x), C) \equiv \textsc{RawVal}(\dot{v})$. This is the first premise of the Skip rule; we must now demonstrate that we can look up x from $\textsc{Cl}(x)$. This proceeds exactly as in the case above when X starts with x. Therefore, by the Skip rule, $\mathbb{L}([x], c, C) \equiv \textsc{RawVal}(\dot{v})$ and this case is finished.

Closure: If the Closure rule is used, then $S = [\langle E, [c]\,||\,e', \tilde{c}\rangle]\,||\,S_0$ where $c = (x = f)$ and $S' = [\langle E', e', c\rangle]\,||\,S_0$ where $E' = E\,||[x \mapsto \langle f, E\rangle]$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Let $C = \textsc{StackName}(S_0)$. As before, we only need to show the agreement property for the topmost stack frame. We again select without loss of generality some X such that $\textsc{Extract}(E', X) = \dot{v}$; it either begins with x or it does not.

Suppose X begins with x; that is, $X = [x]\,||\,X'$. There are two subcases: either $X' = []$ or not.

If $X' = []$, then $\dot{v} = \langle f, E\rangle$. We aim to show $\mathbb{L}([x], c, C) \equiv f$; because $\textsc{RawVal}(\dot{v}) = f$, it suffices to show by the Value Discovery rule of Figure 6 that $\textsc{First}(x, \textsc{Cl}(x), C)$. Let $x' = \textsc{FirstV}(e_{\text{glob}})$. We have from Lemma C.23 that $\textsc{Pred}(x) = c_L$; as $\textsc{Cl}(x')$ has no predecessor, we have $x \neq x'$ and so by Definition 3.3 it suffices to show that $\mathbb{L}([x'], \textsc{Pred}(x), C) \equiv v'$. By Lemma C.18, we have that $E(x')$ is defined; therefore, $\textsc{Extract}(E, [x'])$ is defined. Because S agrees with reverse lookup, we have $\mathbb{L}([x'], \textsc{Pred}(x), C) \equiv v'$. Thus, by the Value Discovery rule, $\mathbb{L}([x], c, C) \equiv \textsc{RawVal}(\dot{v})$ and the subcase of $X' = []$ is finished.

If $X' \neq []$, then recall that $\textsc{Extract}(E', X) = \dot{v}$. By Definition C.34 and because X' is non-empty and $E'(x) = \langle f, E\rangle$, we have $\textsc{Extract}(E, X') = \dot{v}$. S agrees with reverse lookup, so $\mathbb{L}(X', c_L, C) \equiv$

$\textsc{RawVal}(\dot{v})$. Then by the Value Discard rule of Figure 6, we have $\mathbb{L}(X, c, C) \equiv \dot{v}$ and the subcase of $X' \neq []$ (and thus the case of X beginning with x) is finished.

Otherwise, X does not begin with x. This case proceeds exactly as in the Definition case above, applying the Skip rule and appealing to the fact that other variables' values have not changed in this execution step.

Input: If the Input rule is used, then $S = [\langle E, [c] \,||\, e', \tilde{c}\rangle] \,||\, S_0$ where $c = (x = \texttt{input})$ and $S' = [\langle E', e', c\rangle] \,||\, S_0$ where $E' = E \,|||[x \mapsto \iota(^{C}x)]$. Because $\iota(^{C}x)$ is a non-function value, this case proceeds exactly as in the Definition case above.

Alias: If the Alias rule is used, then $S = [\langle E, [c] \,||\, e', \tilde{c}\rangle] \,||\, S_0$ where $c = (x = x')$ and $S' = [\langle E', e', c\rangle] \,||\, S_0$ where $E' = E \,|||[x \mapsto E(x')]$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Let $C = \textsc{StackName}(S_0)$. As above, both stacks share S_0 and so it suffices to show the agreement property for the topmost stack frame. We again select without loss of generality some X such that $\textsc{Extract}(E', X) = \dot{v}$; it either begins with x or it does not.

If $X = [x] \,||\, X'$ then by Definition C.34 and induction on the height of $\textsc{Extract}(E', X)$ we have $\dot{v} = \textsc{Extract}(E, [x'] \,||\, X')$. Because S agrees with reverse lookup, we have $\mathbb{L}([x'] \,||\, X', c_L, C) \equiv \textsc{RawVal}(\dot{v})$. By Lemma C.23, $c_L = \textsc{Pred}(x)$. From this and the Alias rule in Figure 6, we conclude $\mathbb{L}(X, c, C) \equiv \textsc{RawVal}(\dot{v})$. So in this case, S' agrees with reverse lookup.

Otherwise, X does not start with x. The Skip rule applies exactly as in the Definition case above.

Binop: If the Binop rule is used, then $S = [\langle E, [c] \,||\, e', \tilde{c}\rangle] \,||\, S_0$ where $c = (x_1 = x_2 \odot x_3)$ and $S' = [\langle E', e', c\rangle] \,||\, S_0$ where $E' = E \,|||[x_1 \mapsto \dot{v}_1]$, $\dot{v}_1 = \dot{v}_2 \odot \dot{v}_3$, $E(x_2) = \dot{v}_2$, and $E(x_3) = \dot{v}_3$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Let $C = \textsc{StackName}(S_0)$. Note also that none of $\dot{v}_1$, $\dot{v}_2$, or $\dot{v}_3$ are closures as our primitive binary operators do not include closures in their domains or codomains. Let $C = \textsc{StackName}(S_0)$. Both stacks share S_0, so it suffices to show the agreement property for the topmost stack frame. We again select without loss of generality some X such that $\textsc{Extract}(E', X) = \dot{v}$; it either begins with x or it does not.

Suppose $X = [x_1] \,||\, X'$. Because $\dot{v}_1$ is not a closure, Definition C.34 gives us that $X = [x_1]$; that is, $X' = []$. Because S agrees with reverse lookup, we have $\mathbb{L}([x_2], c_L, C) \equiv \textsc{RawVal}(\dot{v}_2)$ and $\mathbb{L}([x_3], c_L, C) \equiv \textsc{RawVal}(\dot{v}_3)$. By Lemma C.23, $c_L = \textsc{Pred}(x)$. Because $\dot{v}_2$ and $\dot{v}_3$ are not closures, $\textsc{RawVal}(\dot{v}_2) = \dot{v}_2$ and $\textsc{RawVal}(\dot{v}_3) = \dot{v}_3$. By the Binop rule of Figure 6, we have $\mathbb{L}(X, c, C) \equiv \dot{v}_1$ and this case is finished.

Otherwise, X does not begin with x_1 and the Skip rule applies as in the Definition case above.

Call: If the Call rule is used, then $S = [\langle E, [c] \,||\, e', \tilde{c}\rangle] \,||\, S_0$ where $c = (x_1 = x_2\ x_3)$ and $S' = [\langle E'', e', (\texttt{fun}\ \ x_4\ \texttt{->})\rangle] \,||\, S$ where $E(x_2) = \langle f, E'\rangle$, $f = [\texttt{fun}\ \ x_4\ \texttt{->}] \,||\, e'$, $E(x_3) = \dot{v}$, and $E'' = E' \,|||[x_4 \mapsto \dot{v}]$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Because S agrees with reverse lookup, it suffices to show that the agreement property for the new topmost stack frame. We again select without loss of generality some X such that $\textsc{Extract}(E'', X) = \dot{v}'$.

Before performing case analysis as above, we will show a general properly necessary for handling all calls. Recall from above that $E(x_2) = \langle f, E'\rangle$. By Definition C.34, we have $\textsc{Extract}(E, [x_2]) = \langle f, E'\rangle$. By the inductive hypothesis, we have that $\mathbb{L}([x_2], \textsc{Pred}(c), \textsc{StackName}(S_0)) \equiv f$. We now consider two cases: either X begins with the parameter x_4 or it does not.

Suppose $X = [x_4] \,||\, X'$. Note that $E(x_3) = \dot{v} = E''(x_4)$. By induction on the height of $\textsc{Extract}(E'', X)$ we have $\textsc{Extract}(E, [x_3] \,||\, X') = \dot{v}'$. Because S agrees with reverse lookup, we have $\mathbb{L}([x_3] \,||\, X', c_L, \textsc{StackName}(S_0)) \equiv \textsc{RawVal}(\dot{v}')$. By Lemma C.23, we have $c_L = \textsc{Pred}(x)$ and so by the Function Enter Parameter rule of Figure 6, we have $\mathbb{L}(X, c, \textsc{StackName}(S)) \equiv \textsc{RawVal}(\dot{v}')$. This demonstrates the agreement property.

Otherwise, X does not start with x_4. Let $X' = [x_2] \,||\, X$. Because S agrees with reverse lookup and by Definition C.34, we have $\mathbb{L}(X', c_L, \textsc{StackName}(S_0)) \equiv \textsc{RawVal}(\dot{v}')$. By Lemma C.23, we have $c_L =$

$\textsc{Pred}(x)$ and so by the Function Enter Non-Local rule of Figure 6, we have $\mathbb{L}(X, c, \textsc{StackName}(S)) \equiv \textsc{RawVal}(\dot{v}')$. This again demonstrates the agreement property and so this case is complete.

Return: If the Return rule is used, then $S = [\varsigma_2, \varsigma_1] \,||\, S_0$ where $\varsigma_2 = \langle E_2', [], \tilde{c}' \rangle$, $\varsigma_1 = \langle E_1, [c] \,||\, e', \tilde{c} \rangle$, $c = (x_1 = x_2\ x_3)$, and $E_2' = E_2 \,|||[x \mapsto \dot{v}]$. Also $S' = [\langle E_1', e', c \rangle] \,||\, S_0$ where $E_1' = E_1 \,|||[x_1 \mapsto \dot{v}]$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L and $\tilde{c}'$ is some non-ϵ c_L'. Because both stacks share S_0, it is sufficient to show the agreement property for the topmost stack frame in S'. Without loss of generality, we select X such that $\textsc{Extract}(E_1', X) = \dot{v}'$. Either X begins with x_1 or it does not.

Suppose $X = [x_1] \,||\, X'$. By induction on the height of $\textsc{Extract}(E_1', X)$ we have $\textsc{Extract}(E_2', [x] \,||\, X') = \dot{v}'$. Because S agrees with reverse lookup, we have $\mathbb{L}([x] \,||\, X', c_L', \textsc{StackName}([\varsigma_1] \,||\, S_0)) \equiv \textsc{RawVal}(\dot{v}')$. By Lemma C.25, we have $c_L' = \textsc{RetCl}(f)$ where $E_1(x_2) = \langle f, E' \rangle$. This is the first lookup premise of the Function Exit rule in Figure 6. Because S agrees with reverse lookup, we have $\mathbb{L}([x_2], c_L, \textsc{StackName}(S_0)) \equiv f$. By Lemma C.23, $c_L = \textsc{Pred}(x_1)$. These two lookups are the premises of the Function Exit rule, so we conclude $\mathbb{L}(X, c, \textsc{StackName}(S_0)) \equiv \textsc{RawVal}(\dot{v}')$ and this case is finished.

Otherwise, X does not begin with x_1. Because $[\varsigma_1] \,||\, S_0$ agrees with reverse lookup, this case proceeds by using the Skip rule as in the Definition case above.

Conditional Start: If the Conditional Start rule is used, then $S = [\langle E, [c] \,||\, e, \tilde{c} \rangle] \,||\, S'$ where $c = (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}})$ and $S' = [\langle E, e', c' \rangle] \,||\, S$ where $c' = (x_1\ !\ \beta)$ and $E(x_2) = \beta$. Note by Lemma C.22 that $\tilde{c}$ is some non-ϵ c_L. Because S agrees with reverse lookup, it suffices to show that the agreement property for the new topmost stack frame. We again select without loss of generality some X such that $\textsc{Extract}(E, X) = \dot{v}$.

Because $\textsc{Extract}(E, X) = \dot{v}$ and S agrees with reverse lookup, we have $\mathbb{L}(X, c_L, C) \equiv \textsc{RawVal}(\dot{v})$. By Lemma C.23, we have $c_L = \textsc{Pred}(c)$ and so $\mathbb{L}(X, \textsc{Pred}(c), C) \equiv \textsc{RawVal}(\dot{v})$. Because $E(x_2) = \beta$ we have by Definition C.34 that $\textsc{Extract}(E, \beta)$ and so, as S agrees with reverse lookup, we have $\mathbb{L}(X, \textsc{Pred}(c), C) \equiv \beta$. These two lookups satisfy the premises of the Conditional Top rule of Figure 6, so we conclude $\mathbb{L}(X, c', C) \equiv \textsc{RawVal}(\dot{v})$ and this case is finished.

Conditional End: If the Conditional End rule is used, then $S = [\varsigma_2, \varsigma_1] \,||\, S_0$ where $\varsigma_2 = \langle E_2, [], \tilde{c}' \rangle$, $E_2 = E_2' \,|||[x \mapsto \dot{v}]$, $\varsigma_1 = \langle E_1', [c] \,||\, e, \tilde{c} \rangle$, and $c = x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}}$. Further, $S' = [\langle E_1, e, c \rangle] \,||\, S_0$ where $E_1 = E_1' \,|||[x_1 \mapsto \dot{v}]$. Let $C = \textsc{StackName}(S_0)$ and note by Definition C.14 that $C = \textsc{StackName}([\varsigma_1] \,||\, S_0)$. Note also that, by Lemma C.22, $\tilde{c}$ is some non-ϵ c_L and $\tilde{c}'$ is some non-ϵ c_L'. As before, we select without loss of generality some X such that $\textsc{Extract}(E_1, X) = \dot{v}'$. We have two cases: either X begins with x_1 or it does not.

Suppose $X = [x_1] \,||\, X''$. By induction on the height of the proof of $\textsc{Extract}(E_1, X = \dot{v}')$, we have $\textsc{Extract}(E_1, X) = \dot{v}' = \textsc{Extract}(E_2, [x] \,||\, X')$. Because S agrees with reverse lookup, we have $\mathbb{L}([x] \,||\, X', c_L', C) \equiv \dot{v}'$. By Lemma C.26 we have that $E_1'(x_2) = \beta$ for some β and that $\textsc{RetCl}(e_\beta) = c_L'$. Because S agrees with reverse lookup, we have $\mathbb{L}([x_2], c_L, C) \equiv \beta$. By Lemma C.23, we have $c_L = \textsc{Pred}(x_1)$ and so $\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv \beta$. These lookups are the premises of the Conditional Bottom rule of Figure 6, so we conclude $\mathbb{L}([x_1] \,||\, X'', c, C) \equiv \textsc{RawVal}(\dot{v}')$ and this case is finished.

Otherwise, X does not begin with x_1. Because $[\varsigma_1] \,||\, S_0$ agrees with reverse lookup, this case proceeds by using the Skip rule as in the Definition case above. □

Finally, we can assemble the two lemmas above to show the completeness of reverse lookup with respect to the operational semantics.

Lemma C.39 Strong Completeness. *For any well-formed e_{glob}, if $[\langle [], e_{glob}, \epsilon \rangle] \longrightarrow_\iota^* S$ then S agrees with reverse lookup.*

Proof. By induction on the size of the proof of forward evaluation. Lemma C.36 proves the base case (for a single step of evaluation); Lemma C.38 proves the inductive step. □

C.4 Soundness of Reverse Lookup

We now show the soundness of reverse lookup with respect to the operational semantics. The key difference between this proof and the completeness proof is our need to merge multiple instantiations of the inductive hypothesis. As we consider the proof tree of reverse lookup, we discover several different runs of the forward operational semantics and, in each run, we have some set of facts about the environments or expressions in the stack. We use the properties of determinism discussed above to show that this information transfers between runs, allowing us to demonstrate that the various lookups in the premises of the rules in Figure 6 apply to the same forward execution of the program.

Lemma C.40. *For some well-formed e_{glob}, suppose $\mathbb{L}(X, c, C) \equiv v$. Then $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^* S$ such that $S = [\langle E, e, c\rangle] \,||\, S'$ and $\textsc{StackName}(S') = C$ and also $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$.*

Proof. By induction on the size of the proof of $\mathbb{L}(X, c, C) \equiv v$ and then by case analysis on the rule used.

Value Discovery: If the Value Discovery rule is used, then $X = [x]$, $c = (x = v)$, and $\textsc{First}(x, c, C)$. We have two cases: either $x = \textsc{FirstV}(e_{\text{glob}})$ or not.

If $x = \textsc{FirstV}(e_{\text{glob}})$ then $\textsc{First}(x, c, C)$ gives that $C = []$. Note that e_{glob} is well-formed and so v is not a function. By the Definition rule of Figure 16, we know $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^1 [\langle E, e, c\rangle]$ where $E = [x \mapsto v]$ and $e_{\text{glob}} = [c] \,||\, e$. So, by Definition C.15, this subcase is finished.

Otherwise, $x \neq \textsc{FirstV}(e_{\text{glob}})$ and so $\textsc{First}(x, c, C)$ implies that $\mathbb{L}([\textsc{FirstV}(e_{\text{glob}})], \textsc{Pred}(c), C) \equiv v'$. By the inductive hypothesis, we have that $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(c)\rangle] \,||\, S_0'$ where $\textsc{StackName}(S_0') = C$. By Lemma C.23 we have $e_0 = [c] \,||\, e$.

We have two cases: either v is a function or it is not. If v is a function, the Closure rule of Figure 16 applies; otherwise, the Definition rule applies. In *both* cases, have that $S_0 \longrightarrow^1 [\langle E, e, c\rangle] \,||\, S_0'$ where $E = E_0 \,||[x \mapsto \dot{v}]$ such that $\textsc{RawVal}(\dot{v}) = v$. Note that $\textsc{RawVal}(\textsc{Extract}(E, [x])) = v$. We let $S' = S_0'$ and, by Definition C.15, this case is finished.

Input: If the Input rule is used, then $X = [x]$, $c = (x = \texttt{input})$, $\iota(^C x) = v$, and $\textsc{First}(x, c, C)$. This case proceeds exactly as in the Value Discovery case with a non-function v.

Value Discard: If the Value Discard rule is used, then $X = [x] \,||\, X'$, $c = (x = f)$, and $\mathbb{L}(X', \textsc{Pred}(x), C) \equiv v$. By the inductive hypothesis, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(x)\rangle] \,||\, S_0'$ and $\textsc{StackName}(S_0') = C$ and $\textsc{RawVal}(\textsc{Extract}(E_0, X')) = v$. By Lemma C.23 we have $e_0 = [c] \,||\, e$. The Closure rule of Figure 16 applies: $S_0 \longrightarrow_\iota^1 [\langle E, e, c\rangle] \,||\, S_0'$ such that $E = E_0 \,||[x \mapsto \langle f, E_0\rangle]$. We let $S' = S_0'$ so that $\textsc{StackName}(S') = C$.

It remains to show that $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$. By Definition C.35 and because $E(x) = \langle f, E_0\rangle$, we have $\textsc{Extract}(E, X) = \textsc{Extract}(E_0, X')$. As we have $\textsc{RawVal}(\textsc{Extract}(E_0, X')) = v$ from above, this case is finished.

Alias: If the Alias rule is used, then $X = [x] \,||\, X'$, $c = (x = x')$, and $\mathbb{L}([x'] \,||\, X', \textsc{Pred}(x), C) \equiv$. By the inductive hypothesis, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(x)\rangle] \,||\, S_0'$ and $\textsc{StackName}(S_0') = C$ and $\textsc{RawVal}(\textsc{Extract}(E_0, [x'] \,||\, X')) = v$. By Lemma C.23 we have $e_0 = [c] \,||\, e$. The Alias rule of Figure 16 applies: $S_0 \longrightarrow_\iota^1 [\langle E, e, c\rangle] \,||\, S_0'$ such that $E = E_0 \,||[x \mapsto \dot{v}]$ where $\dot{v} = E(x')$. We let $S' = S_0'$ so that $\textsc{StackName}(S') = C$. By induction on the height of $\textsc{Extract}(E_0, [x'] \,||\, X')$, we have that $\textsc{Extract}(E, [x] \,||\, X')$. We let $S' = S_0'$ (so that $\textsc{StackName}(S') = C$) and this case is finished.

Binop: If the Binop rule is used, then $X = [x_1]$, $c = (x_1 = x_2 \odot x_3)$, $\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv v_2$, $\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv v_3$, and $v_1 = v_2 \odot v_3$. By the inductive hypothesis, we have that $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(c')\rangle] \,||\, S_0'$ and $\textsc{StackName}(S_0') = C'$ and $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2])) = v_2$. By the inductive hypothesis and by Lemma C.30, we also have that $\textsc{RawVal}(\textsc{Extract}(E_0, [x_3])) = v_3$. By Lemma C.23 we have $e_0 = [c] \,||\, e$.

Because v_1, v_2, and v_3 are in the domain of a binary operator, none of them are functions; thus, by Definition C.34 we have $E_0(x_2) = v_2$ and $E_0(x_3) = v_3$. The Binop rule of Figure 16 applies. Let $E = E_0 \,||[x_1 \mapsto v_1]$, let $S' = S'_0$, and note that $\textsc{RawVal}(\textsc{Extract}(E_1, [x_1])) = \textsc{Extract}(E_1, [x_1]) = E_1(x_1) = v_1$. This case is therefore complete.

Function Enter Parameter: If the Function Enter Parameter rule is used, then $X = [x] \,||\, X'$, $c = (\texttt{fun}\ x \texttt{->})$, and $C = [c'] \,||\, C'$ where $c' = (x_1 = x_2\ x_3)$, $\mathbb{L}([x_3] \,||\, X', \textsc{Pred}(c'), C') \equiv v$, and $\mathbb{L}([x_2], \textsc{Pred}(c'), C') \equiv f$. Further, we have $f = [c] \,||\, e$. By the inductive hypothesis, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(c')\rangle] \,||\, S'_0$ and $\textsc{StackName}(S'_0) = C'$ and $\textsc{RawVal}(\textsc{Extract}(E_0, [x_3] \,||\, X')) = v$. By Lemma C.23 we have $e_0 = [c'] \,||\, e$. By the inductive hypothesis and Lemma C.30, we also have $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2])) = f$. So by Definition C.34, we have $E_0(x_2) = \langle f, E'\rangle$.

The Call rule of Figure 16 applies. Let $E = E' \,|||[x \mapsto E_0(x_3)]$ and let $S' = S_0$. Note that, as c' is a call site, $\textsc{StackName}(S') = [c'] \,||\, \textsc{StackName}(S'_0) = C$; note also that $\textsc{Extract}(E, X) = \textsc{Extract}(E_0, [x_3] \,||\, X')$, so $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$. This case is therefore complete.

Function Enter Non-Local: If the Function Enter Non-Local rule is used, then $X = [x] \,||\, X'$, $c = (\texttt{fun}\ x'' \texttt{->})$, and $C = [c'] \,||\, C$ for $x'' \neq x$ where $c' = (x_1 = x_2\ x_3)$ and $\mathbb{L}([x_2, x] \,||\, X', \textsc{Pred}(c'), C') \equiv v$. By the inductive hypothesis, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(c')\rangle] \,||\, S'_0$ and $\textsc{StackName}(S'_0) = C'$ and $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2, x] \,||\, X')) = v$. By Lemma C.23 we have $e_0 = [c'] \,||\, e$. By the inductive hypothesis and by Lemma C.30, we also have that $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2])) = f$ where $f = [\texttt{fun}\ x'' \texttt{->}] \,||\, e$. By Definition C.34, we have $E_0(x_2) = \langle f, E'\rangle$. By Lemma C.20, because x_3 appears free in e_0 we have that $E_0(x_3) = \dot{v}'$.

The Call rule of Figure 16 applies. Let $E = E' \,|||[x'' \mapsto E_0(x_3)]$ and let $S' = S_0$. Note that, as c' is a call site, $\textsc{StackName}(S') = [c'] \,||\, \textsc{StackName}(S'_0) = C$. It remains to show that $\textsc{RawVal}(\textsc{Extract}(E, X)) = \dot{v}$.

Recall from above that $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2, x] \,||\, X')) = v$. Because $X = [x] \,||\, X'$, we have $\textsc{RawVal}(\textsc{Extract}(E_0, [x_2] \,||\, X)) = v$. By Definition C.34 and because $E_0(x_2) = \langle f, E'\rangle$, we have $\textsc{RawVal}(\textsc{Extract}(E', X)) = v$. Because $x \neq x''$ and by induction on the height of $\textsc{Extract}(E', X)$, we have $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$ and this case is finished.

Function Exit: If the Function Exit rule is used then $X = [x] \,||\, X'$, $c = (x_1 = x_2\ x_3)$, $\mathbb{L}([x'] \,||\, X', c', [c] \,||\, C) \equiv v$, and $\mathbb{L}([x_2], \textsc{Pred}(c), C) \equiv f$ where $f = [\texttt{fun}\ x'' \texttt{->}] \,||\, e$ and $\textsc{RetCl}(e) = c' = (x' = b)$. By the inductive hypothesis (on the former lookup), we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_1$ such that $S_1 = [\langle E_1, e_1, c'\rangle] \,||\, S_2$ and $\textsc{StackName}(S_2) = [c] \,||\, C$ and $\textsc{RawVal}(\textsc{Extract}(E_1, [x'] \,||\, X')) = v$. By Lemma C.24, c' cannot be the predecessor for any clause because $c' = \textsc{RetCl}(f)$. Therefore by Lemma C.23 we have $e = []$ and by Lemma C.27 we have $S_2 = [\langle E_2, [c] \,||\, e_2, c''\rangle] \,||\, S_3$. Note that $\textsc{StackName}(S_3) = C$.

The Return rule of Figure 16 applies. Let $E = E_2 \,|||[x_1 \mapsto E_1(x')]$ and let $S' = S_3$. By Definition C.34, we have $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$ by induction on the height of the proof of $\textsc{Extract}(E_1, [x'] \,||\, X')$ and so this case is finished.

Skip: If the Skip rule is used, then $X = [x] \,||\, X'$, $c = (x'' = b)$, $x \neq x''$, $\mathbb{L}(X, \textsc{Pred}(x''), C) \equiv v$, and $\mathbb{L}([x''], c, C) \equiv v_0$. By the inductive hypothesis (on the latter lookup), we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S$ such that $S = [\langle E, e, c\rangle] \,||\, S'$ and $\textsc{StackName}(S') = C$. By the inductive hypothesis (on the former lookup), we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S''$ such that $S'' = [\langle E', e', \textsc{Pred}(c)\rangle] \,||\, S'''$, $\textsc{StackName}(S'') = C$, and $\textsc{RawVal}(\textsc{Extract}(E', X)) = v$. By Lemma C.33, we have that $E = E' \,|||[x'' \mapsto v']$, so by induction on the proof of $\textsc{Extract}(E', X)$ we have $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$. This case is therefore complete.

Conditional Top: If the Conditional Top rule is used, then $c = x_1\ !\ \beta$, $\textsc{Cl}(x_1) = (x_1 = x_2\ ?\ e_{\texttt{true}} : e_{\texttt{false}})$, $\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv \beta$, and $\mathbb{L}(X, \textsc{Pred}(x_1), C) \equiv v$. By the inductive hypothesis, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Pred}(x_1)\rangle] \,||\, S'_0$ and $\textsc{StackName}(S'_0) = C$ and

$\textsc{RawVal}(\textsc{Extract}(E_0, X)) = v$. By Lemma C.30 and the inductive hypothesis, we have $\textsc{RawVal}(\textsc{Extract}(E_0, [x_1])) = \beta$ and so, by Definitions C.35 and C.34, we have $E_0(x_1) = \beta$.

The Conditional Start rule of Figure 16 applies. We let $E = E_0$ and let $S' = S_0$. Note that by Definition C.14 we have $\textsc{StackName}(S') = \textsc{StackName}(S'_0) = C$ because $\textsc{Cl}(x_1)$ is not an application clause. Note also that $\textsc{RawVal}(\textsc{Extract}(E, X)) = \textsc{RawVal}(\textsc{Extract}(E_0, X)) = v$, so this case is finished.

Conditional Bottom: If the Conditional Bottom rule is used, then $X = [x_1] \,||\, X'$, $c = (x_1 = x_2 \;?\; e_{\texttt{true}} : e_{\texttt{false}})$, $\mathbb{L}([x_2], \textsc{Pred}(x_1), C) \equiv \beta$, $\textsc{RetCl}(e_\beta) = \textsc{Cl}(x')$ for some x', and $\mathbb{L}([x'] \,||\, X, \textsc{Cl}(x'), C) \equiv v$. By the inductive hypothesis on the latter lookup, we have $[\langle [], e_{\text{glob}}, \epsilon\rangle] \longrightarrow_\iota^* S_0$ such that $S_0 = [\langle E_0, e_0, \textsc{Cl}(x')\rangle] \,||\, S'_0$ and $\textsc{StackName}(S'_0) = C'$ and $\textsc{RawVal}(\textsc{Extract}(E_0, [x'] \,||\, X')) = v$. By Lemma C.24, there is no predecessor of $\textsc{Cl}(x')$; thus, by Lemma C.23, we have $e_0 = []$. By Lemma C.28, we have $S'_0 = [\langle E'_0, e'_0, c'_0\rangle] \,||\, S''_0$ where $e'_0 = [c] \,||\, e''_0$ and $E'_0(x_2) = \beta$. Note that $\textsc{StackName}(S'_0) = \textsc{StackName}(S''_0) = C$ and that, by Lemma C.23, we have $c'_0 = \textsc{Pred}(c)$.

The Conditional End rule of Figure 16 applies. We let $E = E'_0 \,||[x_1 = E_0(x')]$ and let $S' = S''_0$. By Definition C.34 we have by induction on the height of $\textsc{RawVal}(\textsc{Extract}(E_0, [x'] \,||\, X')) = v$ that $\textsc{RawVal}(\textsc{Extract}(E, X)) = v$ and so this case is finished. □

C.5 Correctness of Reverse Lookup

At last, we can assemble the soundness and completeness into a single result which achieves the goal of this appendix.

Theorem C.41. *For any well-formed e_{glob} and any input map ι, $[\langle [], e_{glob}, \epsilon\rangle] \longrightarrow_\iota^1 [\langle E, e', c\rangle] \,||\, S$ if and only if $\mathbb{L}([\textsc{FirstV}(e_{glob})], c, C) \equiv v$ for some v.*

Proof. By Lemmas C.39 and C.40, specializing to the lookup stack $X = [\textsc{FirstV}(e_{\text{glob}})]$. □

REFERENCES FOR THE APPENDIX

Leandro Facchinetti, Zachary Palmer, and Scott Smith. Higher-order demand-driven program analysis. *TOPLAS*, 41, July 2019.