Higher-Order Demand-Driven Program Analysis

LEANDRO FACCHINETTI, The Johns Hopkins University
ZACHARY PALMER, Swarthmore College
SCOTT SMITH, The Johns Hopkins University

Developing accurate and efficient program analyses for languages with higher-order functions is known to be difficult. Here we define a new higher-order program analysis, Demand-Driven Program Analysis (DDPA), which extends well-known *demand-driven* lookup techniques found in first-order program analyses to higher-order programs.

This task presents several unique challenges to obtain good accuracy, including the need for a new method for demand-driven lookup of non-local variable values. DDPA is flow- and context-sensitive and provably polynomial-time. To efficiently implement DDPA we develop a novel pushdown automaton metaprogramming framework, the Pushdown Reachability automaton (PDR). The analysis is formalized and proved sound, and an implementation is described.

CCS Concepts: • **Software and its engineering** → **Automated static analysis**; • **Theory of computation** → *Formal languages and automata theory*;

Additional Key Words and Phrases: functional programming, program analysis, polynomial-time, demand-driven, pushdown system, flow-sensitive, context-sensitive

ACM Reference format:
Leandro Facchinetti, Zachary Palmer, and Scott Smith. 2019. Higher-Order Demand-Driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 1, 1, Article 1 (August 2019), 53 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Developing an accurate but efficient higher-order program analysis is hard. Building on older first-order abstract interpretations [8], a large array of higher-order analyses have been developed over the last twenty years [11, 22, 24, 32–35, 44, 51, 52]. Unfortunately, in spite of all the technical advances these analyses are not commonly employed in compilers for higher-order functional languages today: MLton [54] and Stalin [45] are among the few whole-program optimizing compilers that use these tools. Even in those cases, the analyses are often used in their less expressive versions. The primary reason is the difficulty in getting both expressiveness and efficiency in the presence of higher-order functions due to their significantly greater complexity than the first-order case.

To address this issue, we have developed Demand-Driven Program Analysis (DDPA), a novel analysis for higher-order programs. We do not (yet) claim that it solves this longstanding problem, but it takes a fundamentally different approach with fundamentally different trade-offs between

Leandro Facchinetti is supported by a CAPES Fellowship, process number 13477/13-7.

0164-0925/2019/8-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

expressiveness and performance. Demand-driven program analyses are different than forward-running analyses in that they only look up values when needed ("on demand"). Demand-driven analyses were initially developed for first-order programs [10, 20, 21, 38, 40, 42, 43] where they were shown to achieve good performance/expressiveness trade-offs: if only some of the variables of the programs need analyzing, the work of analyzing un-needed variables is avoided.

The primary goal of this paper is to extend first-order demand-driven analyses to higher-order programs. It is well-known that program analysis for languages with higher-order functions is more difficult since the control flow graph or CFG ("which functions are called where?") is dependent on the data flow ("which functions could be in which variables?") [29, 35], and so data- and control-flow must be simultaneously computed. A similar issue, so-called *structure-transmitted data dependence* [39], also arises in first-order programs: precision on data structure accesses requires combined control- and data-flow precision. These first-order techniques transfer poorly to higher-order programs due to a lack of *non-local variable alignment*, which affects the lookup precision of variables captured in a higher-order function's closure.

Here we develop a novel demand-driven higher-order analysis which precisely aligns non-local variables; that is, the precision of the analysis on local and non-local variables is the same. Ours is not the first higher-order demand-driven analysis, but previous analyses in this class [14, 37, 47] do not align non-local variables and so overapproximate their possible values. A lack of non-local alignment also appears in first-order demand-driven analyses as well as some higher-order forward analyses [24, 51]; these analyses use a stack to align calls and returns but lose precision on non-local variables. The non-local alignment technique in this paper is in spirit based on the use of *access links* in a compiler runtime. Context-sensitivity, aka polyvariance/polymorphism, can be achieved solely by call-return alignment since both local and non-local variables are aligned; there is no need for explicit copying as in e.g. kCFA [44].

In comparing DDPA with a state-of-the-art forward analysis for higher-order programs, we find that, when the analyses are configured to achieve the same precision on particular questions, their comparative performance varies. In some cases, demand-driven analysis is faster than the forward analysis; in others, the relationship is reversed. Similar results were discovered in the space of first-order analyses [21].

Synopsis of the paper

In Section 2, we give a detailed overview of DDPA by demonstrating how it analyzes several small examples. In Section 3, we show how we can approximate the values of variables using pushdown reachability. Although this reachability technique is common among stack-aligning analyses, we generalize DDPA's pushdown automaton (PDA) to a novel *pushdown reachability automaton* (PDR) which vastly reduces the state-space search size: schemas of PDA states or transitions can be represented by single PDR states or transitions.

After this overview, we present the theory of DDPA. We formally define the kDDPA analysis in Section 4; the parameter k here represents the constant number of stack frames the analysis preserves, in analogy with kCFA. We prove soundness by first showing in Section 5 that a small-step operational semantics for the λ-calculus is isomorphic to ωDDPAc, a *graph-based* operational semantics derived from DDPA which retains an unbounded number of stack frames (ω) and has perfect contextual precision (c). We believe ωDDPAc is of interest in its own right: it is a novel presentation of operational semantics for the λ-calculus which never duplicates any expressions. The isomorphism between these operational semantics relations easily leads to the soundness of the analysis in Section 6. The decidability of the analysis is presented in Section 7. In Section 8, we present the theory of PDR systems as an efficient technique for implementing the analysis.

$$\begin{array}{lcll@{\qquad}lcll} e & ::= & [c, \ldots] & \textit{expressions} & v & ::= & f \mid r \mid \texttt{ref}\ x & \textit{values} \\ c & ::= & x = b & \textit{clauses} & f & ::= & \texttt{fun}\ x\ \texttt{->}\ (\ e\ ) & \textit{functions} \\ b & ::= & v \mid x \mid x\ x \mid x \sim p\ ?\ f : f \mid & \textit{clause bodies} & r & ::= & \{\ell = x, \ldots\} & \textit{records} \\ & & x\,.\,\ell \mid x \texttt{<-} x \mid\ !\ x & & p & ::= & \{\ell, \ldots\} & \textit{patterns} \\ x & & & \textit{variables} & & & & \end{array}$$

Fig. 1. Expression Grammar

In the theoretical presentation we focus on the pure call-by-value λ-calculus to not get overwhelmed by details; in Section 9 we outline how additional language features may be incorporated. In Section 10 we evaluate our implementation of this more featureful language in terms of running time and precision.

We discuss related work in Section 11 and conclude in Section 12.

This paper is extends the original conference paper on DDPA [36] with a PDR-based reference implementation. Sections 3, 8 and 10 are new. Extended explanations and a more complete theoretical analysis, including a proof that the unbounded-stack ωDDPAc analysis is a full and faithful λ-calculus interpreter, is also included.

2 ANALYSIS OVERVIEW

This section informally presents DDPA by example. We begin with a high-level description of the analysis and roughly sketch how it works on a small program. This description is incomplete: we will show how the analysis as described is too imprecise, and we will then describe another feature of the analysis which in fact addresses this imprecision. This process will be repeated several times until the entire DDPA algorithm has been described. This section does not touch on implementation; the following Section 3 will describe at a high level how DDPA may be efficiently implemented.

2.1 A Simple Language

We use a simple functional language defined in Figure 1. It is a call-by-value λ-calculus extended with conditionals (via pattern matching), records, and state (via ML-like reference cells). Conditionals are written $x \sim p\ ?\ f_1 : f_2$: either f_1 or f_2 is called with x as argument, depending on whether x's value matches the pattern p. We require that programs are closed, and that variable identifiers are unique ("alphatized"). We use an A-normal form (ANF) [17] intermediate representation, so a clause c denotes a program point. The operational semantics for this language is straightforward and deferred to Section 5.1.

2.2 The Basic Analysis

Consider the program in Figure 2 and the steps to construct its Control-Flow Graph (CFG) in Figure 3.

CFG construction

In higher-order languages, control flow and data flow are intertwined, so the CFG construction informs the analysis and vice-versa. The initial CFG on Step 1 of Figure 3 is created by inspecting the program source to determine the blocks determined by the main program and function bodies. Program points are abbreviated to the identifier they introduce; for example, `m=i a` becomes $\boxed{\texttt{m}}$. Then, to preserve the order of evaluation, we look for call sites we can reach from the beginning of the program by traversing only immediate program points (those that are not function calls

```
i = fun x -> ( y = x; );
e = {};
a = {a = e};
b = {b = e};
m = i a;
n = i b;
```

Fig. 2. Basic analysis example program

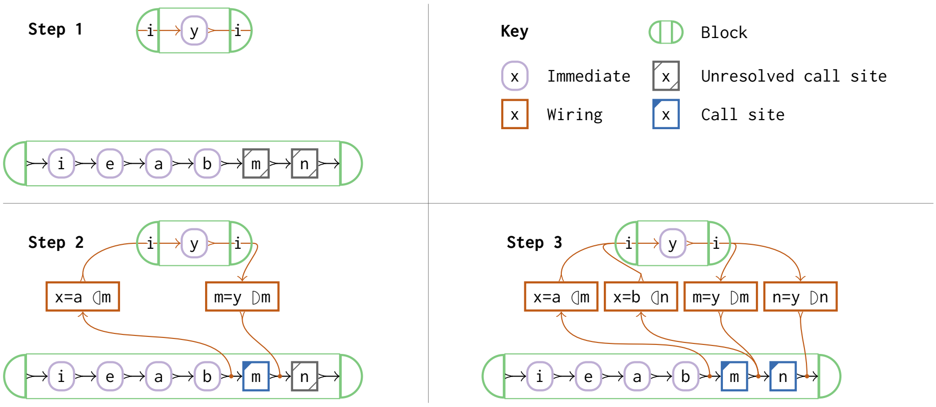


Fig. 3. Basic analysis example CFG

or conditionals). In this example, $\boxed{\mathtt{m}}$ satisfies this condition, but $\boxed{\mathtt{n}}$ does not: the only path to it from the start of the program includes $\boxed{\mathtt{m}}$, which is a call site and not an immediate program point. We say $\boxed{\mathtt{m}}$ is *active*, and we resolve it next. *Resolving* a call site means adding wiring edges and nodes, which connect the body of the called function *around* the call site. The wiring nodes contain information about the call site, the argument and the return value, for reasons which will become evident later. The resolution of call site $\boxed{\mathtt{m}}$ leads to Step 2; in that step, we write the call site as $\boxed{\mathtt{m}}$ to indicate it is resolved. As $\boxed{\mathtt{m}}$ is resolved, the call site $\boxed{\mathtt{n}}$ becomes active. So, we resolve $\boxed{\mathtt{n}}$ next, adding more wiring edges and nodes and arriving at Step 3: a complete CFG with no unresolved call sites.

In general, the call site resolution process is constructing a graph of all potential control flows in the program as it moves forward through the partial CFGs, which implies there might be multiple active call sites at any given point, and the analysis is free to choose which to resolve next, as the final CFG will be the same when all the call sites are resolved regardless of the order in which steps are taken. (We prove this confluence result as Lemma 7.6.) Because we never remove edges or nodes from the partial CFGs, they are monotonically growing during the construction, so the process can happen incrementally. Also, when there are multiple calls to the same function, there might be multiple wiring edges connecting to it, but the function body itself is not copied, and there can only be one pair of wiring edges and nodes between a particular call site and function body. This guarantees a polynomial upper bound on CFG size. It is possible that the analysis has to revisit a previously resolved call site and resolve it again when more functions and arguments become available to it from wirings added elsewhere in the CFG – the construction is complete when this process reaches a least fixed point. All examples in the remainder of this section start with a complete control-flow graph built with this process.

Lookup

To resolve a call site, DDPA has to know which functions were called at that point; how does this lookup work? For example, when we resolved the call site `m` between Steps 1 and 2, how did DDPA know that `i` was the appropriate function to wire? In a typical forward-running program analysis, abstract stores would be propagated forward along the CFG and the values of function variables would be established from those stores. Here, we take a different approach.

The lookup began with the question "what are the possible values of `i` at `m`?" To find the answer, DDPA starts traversing the partial CFG *backward* with respect to control flow, by following the arrows opposite to where they are pointing in the diagram. It immediately finds `b`, which contained no information about our variable of interest, so DDPA reformulated our lookup question into another that must have the same answer: "what are the possible values of `i` at `b`?" We then follow another arrow in reverse and found `a`, which also contains no information about `i` and causes DDPA to reformulate the lookup question again. The same happens on the next step, when we reach `e`. Finally, on the next iteration, DDPA finds the definition of `i`, at which point it concludes the obvious: "the possible values of `i` at `i` include the function `fun x ->...`" This answer percolates back to our original question, because we carefully constructed the intermediary questions so that their answers would be the same: "the possible values of `i` at `m` include the function `fun x ->...`" This lookup result is why we wire that function around `m`.

A similar process occurs when resolving call site `n` between Steps 2 and 3. Again, we have to lookup the possible values of variable `i`, but this time we start from a different perspective, `n`. We look at the arrows pointing at `n` backward, and find two nodes: `m` and `m=y ◗m`. We could non-deterministically explore both paths, but it would be redundant: the function body does not affect the definition of the (immutable) variable `i`, so information discovered from exploring it is equivalent to that discovered by *skipping over* it and moving to `m`. Thus, we proceed by doing the latter and find ourselves asking the same question as before: "what are the possible values of `i` at `m`?" We already know the answer from the previous step, so can reuse it to conclude that "the possible values of `i` at `n` include the function `fun x ->...`" Thus, we may add the corresponding wiring edges and nodes.

The CFG construction and the lookup process are carefully ordered by the *active* nodes to preserve evaluation order, so the answer to a lookup question is always complete and never changes. This allows us to visit nodes with a particular lookup question only once and reuse that answer later. Also, the analysis is naturally flow-sensitive, because lookup questions are relative to particular program points.

Context Sensitivity

Lookups are not restricted to finding which functions could be called at a call site – it is possible to query the value of any variable in scope. For example, let us consider one more lookup: `n` from the end of the program in the complete CFG (Step 3). On the first step we look at the arrows in reverse from the end of the program and find two nodes, `n` and `n=y ◗n`. In our previous example we had skipped over a function call because it had not affect the result, but this time it does, so, instead of skipping to `n`, we explore the function call by taking the path to `n=y ◗n`. At this point, we follow our lookup process, changing the lookup question to another one with the same answer, and this new question is not only a change in perspective, but also a change in subject. The wiring node contains the information that `n` is `y`, so we now ask "what are the possible values of `y` at `n=y ◗n`?" The next step is similar: we find that `y=x` and change our lookup subject to `x` at `y`.

However, now we encounter a problem: there are two nodes we can follow in reverse from `y`, `x=a ◖m` and `x=b ◖n`, and if we follow `x=b ◖n`, then we eventually find the right answer: "`{b=e}` is a possible value for `n` by the end of the program;" but if we follow `x=a ◖m`, then we discover that "`{a=e}` is another possible value for `n` by the end of the program!" Programmers looking at the program know that actual execution would never output the latter, because the execution path leading to it does not make sense: it includes `x=a ◖m` (the entrance node for `m`) and `n=y ◗n` (the exit node for `n`, which means a function is called at `m` and returns to `n`. The lookup described thus far does not rule out this misalignment and non-deterministically explores all possible paths, over-approximating the answer and losing precision.

To resolve this, we introduce another component to the lookup process: the *context stack*. The lookup question changes to the form "what are the possible values for variable `x` at `p` with context stack `[a, b, c]`." We push the call site name to the context stack when we visit an exit node for a function call (in reverse), and we only visit the entrance node for which we can pop the corresponding call site. For example, when we visit `n=y ◗n`, we push `n` to the context stack, and, when we are at `y`, we only follow the path to `x=b ◖n` and discard the path to `x=a ◖m`, because `n` is on the top of the stack. Then the final result will be exact: "only `{b=e}` is a possible value for `n` by the end of the program."

Numerous higher-order analyses have been developed which show how higher-order functions can be analyzed, and some of those analyses also include call-return alignment mechanisms [11, 19, 24, 37, 51]. But, the call-return alignment in these analyses is not fully solving the higher-order call-return alignment problem: they are more or less porting the first-order notion of call return alignment over and, since first-order programs have no non-locals, these higher-order analyses only align *local* variables. For example, PDCFA [24] and CFA2 [51] only align the so-called stack references (locals), not the so-called heap references (non-locals). As a result, forward higher-order analyses cannot rely solely upon stack alignment to achieve the effect of polymorphism; an additional polymorphism mechanism must be employed. This additional polymorphism machinery often generates redundant work (e.g. in cases where no non-local variables were used and call-return alignment would be sufficient) but must be conservatively applied to maintain precision. A key contribution of our previous work [36] is a method for alignment of non-locals in call-return alignment via the context stack discipline, which also makes DDPA a context-sensitive (polyvariant) analysis. This feature is commonly achieved by copying function bodies – *k*CFA being the canonical example [35] – but, in DDPA, context-sensitivity relies solely on call-return alignment. The subtleties of how this works are covered next.

2.3 Non-Local Variable Lookup

The lookup process described so far only included local variables (stack references). We now address non-local variables (heap references). Consider the program in Figure 4. We build its CFG as described in the previous section (Figure 5) and start looking up for the values of `r` from the end of the program. There is only one call to each function in the program, so we ignore the context stack and the call-return alignment issue for the rest of this example. We first move to `r=b ◗r`, changing the lookup subject to `b`; then we go to `b`, changing the lookup subject to `x`. At this point, the lookup process described so far would fail to abstract the notion of lexical scoping and instead look the for non-local variable in the context of the call site, as opposed to the scope in which the function was defined: it would skip `y=e ◖r`, `p`, `e` and `k`, because they do not immediately affect the value of our subject `x`, reach the beginning of the program without finding a value, and fail. The search process needs to be modified to take lexical scoping rules of non-local variables into account.

The solution to this issue is inspired by access links used in compilers for higher-order functions. When visiting `y=e ◖r` in the lookup above, we notice that `x` is not the function argument, so it

```
k = fun x -> (
    a = fun y -> (
        b = x;
    );
);
e = {};
p = k e;
r = p e;
```

Fig. 4. Non-local variable lookup example program

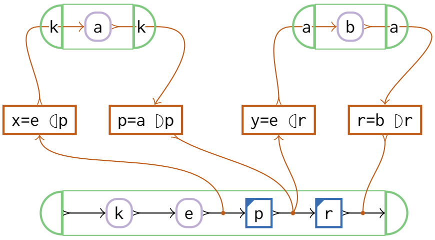


Fig. 5. Non-local variable lookup example CFG

must be a non-local variable which was in scope at the function definition, not its use. To properly abstract the notion of lexical scoping we must defer the current lookup, find the program point defining the function we are exiting (in reverse) and then resume from there. In our example DDPA executes this by suspending the lookup for x and starting a subordinate lookup for the function we are exiting, p.

The subordinate lookup for p follows the rules we discussed thus far, entering p=a ▷p. The subject of the lookup changes to a, which we find on the next step at a. This is the definition of the function we were looking for, so we can resume the deferred lookup of x from there: we move to x=e ◁p, change the subject to e, move to e and finally find the answer.

The function definition in the subordinate lookup could be a non-local itself, so, in general, it is necessary to chain on the above idea, akin to access links. This requires the introduction of a second stack, besides the context stack discussed above, which we call the *lookup stack*. The lookup stack contains the deferred lookups, and the lookup process is complete only when the lookup stack is empty. There are other uses for the lookup stack in DDPA; for example, to lookup a record projection, the analysis has to lookup the record and then the value under the projected key. DDPA keeps the projected key in the lookup stack from the program point containing the projection until the record is found. Because of this generality, we also refer to the lookup stack as a *continuation stack*.

2.4 Function Call Lookup

The lookup process described thus far still loses precision on function calls. Consider the program in Figure 6, the corresponding CFG in Figure 7, and a lookup for q from the end of the program with an empty context stack. On the first step, DDPA visits q=b ▷q, changes the lookup subject to b, and finds three possible paths to continue. The first is b, which it discards because it would mean skipping over a call site that contains information about the lookup subject. On the second path, it visits b=d ▷b, d, and finds the expected answer: {h=e}. But, on the third path, it visits b=c ▷b, c, and finds an imprecise answer: {g=e}.

This third path does not make sense during run-time: it requires the q call site to pass h as the value of f but then expects f to be g at b. The call-return alignment discipline we described thus far is not enough to eliminate this path because we reached the answer at c by visiting only immediate and exit nodes, so we pushed call sites to the context stack but never visited entrance nodes to try (and fail) pop them.

The solution here is to start a subordinate lookup for operand at the call site before visiting an exit node and only proceed if the result includes the pertinent function. This subprocess shows that

```
e = {};
a = fun f -> ( b = f e; );
g = fun m -> ( c = {g = e}; );
h = fun n -> ( d = {h = e}; );
p = a g;
q = a h;
```

Fig. 6. Function call lookup example program

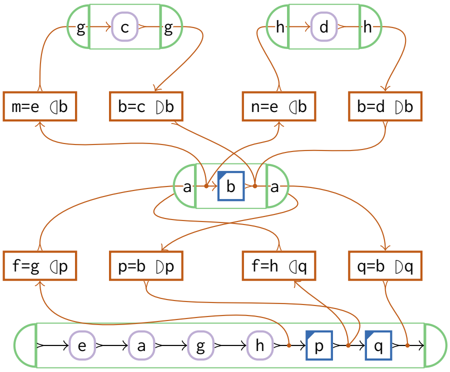


Fig. 7. Function call lookup example CFG

our call-return alignment discipline aligns not only the entrance and exit nodes, but also arguments, even those that are higher-order functions. In our example, before we visit `b=c ◗b` we have to verify whether g can flow into f. This is not the case, because q is on the top of the context-stack, and the call-return alignment discipline fails when trying to pop p at `f=g ◖p`, so we discard this entire path. But the condition is satisfied for `b=d ◗b`, part of the path leading to the more exact answer.

2.5 Recursion and Decidability

We now address how the analysis handles recursive functions, which induces cycles in the CFG. A lookup based on graph traversal as we have thus far described would never end, following the cycle indefinitely. Consider the program in Figure 8, the corresponding CFG in Figure 9, and a lookup for the values of r by the end of the program with an empty context stack. DDPA starts by visiting `r=a ◗r`, and then `a=a ◗a`. From there, it can repeatedly follow the cycle back around `a=a ◗a` and never reach an answer, which is unsurprising as the run-time for this program also does not terminate. Moreover, the lookup process described thus far includes two stacks, the context stack and the lookup stack, and with them we could simulate the tape of a Turing Machine, so it looks very unlikely to be undecidable with two un-finitized stacks. We concretely prove an undecidability result in Section 5: we define ωDDPAc, a two-stack-unbounded analysis, and prove it is a complete interpreter for the call-by-value λ-calculus.

```
o = fun x -> ( a = x x; );
r = o o;
```

Fig. 8. Recursion example program

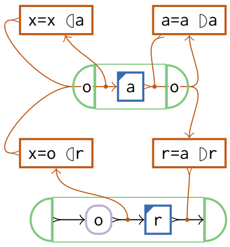


Fig. 9. Recursion example CFG

To make lookup decidable, we have to start by artificially bounding one of these stacks: if we do it for the lookup stack, then we impact the precision on non-local variables, record projection, and

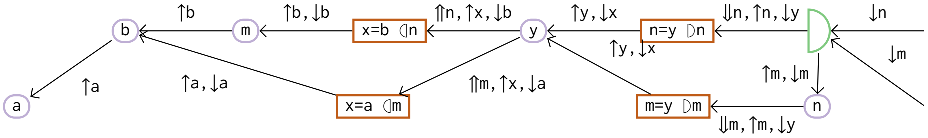


Fig. 10. A *two-stack* PDS encoding traversals for lookups of m and n from the end of the program in Figure 3. Thick arrows represent pushes (⇓) and pops (⇑) to the context stack, and thin arrows represent pushes (↓) and pops (↑) to the lookup stack.

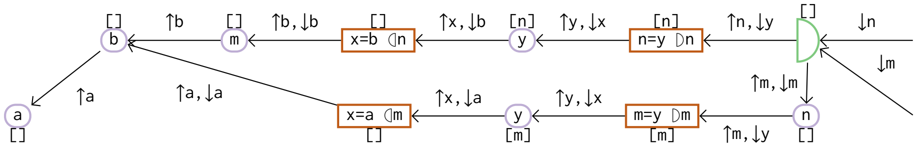


Fig. 11. A one-stack PDS which approximates the two-stack PDS by finitely abstracting the context stack and embedding it in the nodes.

so forth; and a bounded context stack would impact context sensitivity. We choose the latter. Any systematic finitization technique of the context stack would work, and in DDPA we use one of the simplest possible: we truncate the stack to a fixed maximum size of k, in the spirit of kCFA [44]. This characterizes kDDPA as a family of program analyses, parameterized over the choice of k. Other finitization techniques lead to different trade-offs with respect to precision and performance, but they are orthogonal to the demand-driven aspects that we are exploring in this work.

In our example, kDDPA would follow the loop around a=a ◗a a maximum of k times, at which point it would exhaust the context stack. When this happens, a subordinate lookup question will be identical to a lookup question already underway: "what are the possible values for a at a=a ◗a with context stack [a, a, ...]?" In this case, the subordinate lookup immediately returns the empty set, because there are no other paths contributing to the answer.

Reachability

We described the lookup process in terms of a traversal of the CFG, but this is not how the algorithm is actually implemented. To realize the analysis in a way that promotes the reuse of previously computed answers, lookup is encoded in terms of Pushdown System (PDS) reachability questions [6]. The answer to these questions, in turn, are lazily calculated with a novel construction called Pushdown Reachability automaton (PDR). The next section describes these automata and gives an overview of how DDPA can be efficiently implemented.

3 IMPLEMENTATION OVERVIEW

We now give a high-level overview of how lookup in the previous section is efficiently implemented.

3.1 The Basic Analysis – Revisited

In the previous section we introduced lookups as (reverse) traversals on the CFG, but the cycles induced by recursive functions could be traversed indefinitely, rendering this intuition unrealizable. DDPA's solution to this issue comes in three parts: (1) encode the CFG traversals in terms of a *two-stack pushdown automaton*; (2) derive a one-stack pushdown automaton which approximates it; and (3) interpret lookup queries as *reachability* questions on this automaton, a task known to be decidable [6].

To begin with, we construct a *two-stack pushdown automaton* (two-stack PDA) corresponding to the CFG traversals from Section 2: nodes in this PDA correspond to nodes in the CFG, edges correspond to transitions allowed by following control flow in reverse, and edge annotations correspond to stack manipulations – thick arrows (⇓/⇑) are related to the context stack, and thin arrows (↓/↑) to the lookup stack. Since we are searching backwards through the program, we encounter function *returns* before the *calls* and so the context stack stack operations are *pushing* returns and *popping* calls. (This can take some getting used to as it is the exact opposite of the program run-time stack operations.) The edge to the start state in the PDA represents the start of the traversal in the CFG by pushing the query subject to the lookup stack, and immediate nodes are accepting states. The first outstanding characteristic of this PDA, which is also true of all other automata we will encounter in this paper, is that its input alphabet is empty, because our purpose is not to recognize a string but to reason about the stack discipline. Automata of this kind are called *pushdown systems* (PDS) [6].

As an example, consider the PDS in Figure 10, which illustrates lookups of n and m from the end of the given program in the CFG from Figure 3. The edge labeled ↓n on the upper-right corner starts the traversal looking for n from the end of the program, the node n=y ◗n represents visiting that node in the CFG traversal, and the node b is an accepting node. The main row on the top of the PDS in the figure corresponds to the complete traversal caused by looking n up from the end of the program, and the row below corresponds to the traversal caused by looking m up from the end of the program. The automaton permits sharing the nodes that are identical in both traversals, for example, y. The paths from ↓n to b and from ↓m to a are realizable, but from ↓n to a and from ↓m to b are not, because they make incompatible choices when popping from the context stack, preserving the call-return alignment described in Section 2.2.

It is well-known that a two-stack PDA is equivalent in power to a Turing Machine, so we are clearly flirting with danger here, and concretely we show in Section 5 that ωDDPAc, the two-stack-unbounded analysis, is Turing-complete and thus undecidable. So, we must convert the two-stack PDS into something strictly weaker to make the analysis computable. DDPA's approach is to finitely abstract the context stack configurations and embed them in the PDS nodes, and, if multiple context stack configurations can occur at the same CFG node, create multiple versions of that node in the PDS to distinguish them. Any abstraction that bounds the context stack into a finite space suffices, and different choices have different effects on the analysis' capacity to align calls and returns. For simplicity DDPA uses a very basic abstraction technique, similar in spirit to the contour treatment in kCFA [44]: truncate the context stack beyond length k to create a family of increasingly precise kDDPA analyses. See Figure 11 for the 1-stack PDS for our running example, in which the node y is duplicated and differentiated with two context stacks because we choose $k = 1$. As illustrated by b, for example, node sharing between paths can still occur in this PDS, but it is more rare because the traversals have to visit the common node with the same context stack configuration.

Both the CFG and PDSs constructions can be incremental, responding to the demands of the query, because the information discovered is never invalidated at later steps. Moreover, cycles in the CFG induce cycles in the PDSs, but their construction is finite and decidable because both CFG and PDSs grow monotonically and in tandem, and nodes and edges cannot be repeated.

Finally, after constructing the PDS, we can use it to answer lookup questions by encoding them in terms of *reachability* questions of the form "which accepting states (immediate nodes) can we reach from this given start state with an empty lookup stack (the context stack embedded in the node might be non-empty)?" In our example, the accepting state b is reachable when starting from ↓n, but a is not.

In the literature, several different approaches have been used to refine program analyses via aligning calls and returns: context-free language (CFL) reachability [40], set constraint solving [25], PDA reachability [3], and logic programming specifications [42] can all be used. The DDPA implementation is not aiming for perfect alignment of calls and returns like these analyses, because it instead focuses on perfect access of non-local variables, and uses the PDS stack for that purpose; the call stack is always a finite approximation in the nodes of the PDS and so can be viewed more as having a stack-free finite automaton representation in DDPA. This follows the dominant stream of higher-order analyses inclluding kCFA and its many refinements – the k represents the fixed finite length of the call stack. As mentioned above there are many other possibilities for abstraction other than the simple one used here, for example in code with no recursive non-locals we could in principle swap and make the call stack arbitrary and the non-locals stack of bounded size. Some higher-order forward analyses opt for perfect precision on the call stack but then lose precision on non-locals [19, 24, 51]. The notion of finding good trade-offs between data- and control-precision is studied in [55].

The above analysis aligns both local and non-local variables, and it is achieving the full effect of kCFA-style polymorphism; this is apparent in the precision of the above example. So, while the analysis algorithm is closely related to the CFL-reachability analyses, the effect in the higher-order space is more closely related to polyvariance than stack alignment.

Finitization of the call stack is not an optimal solution: there are exponentially many finitized call stacks of length k, so the algorithm is exponential as k grows.[1] Also, this approach will never be able to keep perfect call-return alignment for recursive programs on any k. Future work is to improve on this approximation method.

3.2 Pushdown Reachability Automata

kDDPA depends on solving pushdown reachability, for which the standard algorithm is to perform an edge closure [6]: continually replace adjacent matching push and pop edges with a single transitive no-op edge. For any valid path in the original automaton, after closure there will be a path consisting solely of no-op edges. Pushdown reachability is decidable in polynomial time [6], which is a pleasant property, and the proof-of-concept implementation of DDPA [36] used this algorithm to perform variable lookup. But, while straightforward, this approach can be slow in practice when the number of states and edges is large. Moreover, most of them do not affect the lookup result, so we define a novel "pushdown reachability" (PDR) automaton, partly inspired by PDCFA [24], to accelerate reachability queries.

Observe that, over the course of kDDPA, the CFG grows monotonically. As a result, the variable lookup PDS (and its closure) exhibits similar monotonic growth. As long as we grow the PDS in tandem with the CFG throughout the course of the analysis, we can use the same data structure for all variable lookups and avoid duplicating PDA closure work. Moreover, we can identify structural similarities between edges and *compress* them to reduce the size of the automaton representation. These monotonic properties and this compression technique will be crucial in our implementation of a tractable analysis.

We demonstrate this different form of PDS using the code in Figure 12. Consider the task of looking up the values of x and y from the end of the program, using the same PDS each time. The PDS for this program appears in Figure 13, and PDR appears in Figure 14. The only difference between these automata is that the PDR (Figure 14) contains a new edge annotation. Consider, for instance, the edge labeled "⥮¬y". The PDR treats this edge as a no-op as long as the top of the

[1] This is different from kCFA, which is exponential in the *program size* whenever $k \geq 1$ [50].

```
x = {};
y = {};
z = {};
```

Fig. 12. PDR example

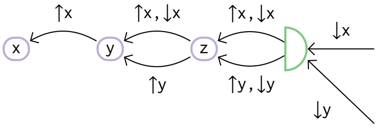


Fig. 13. PDR example: PDS

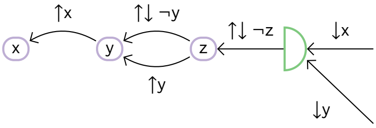


Fig. 14. PDR example: PDR

$$\begin{array}{rcll}
\hat{e} & ::= & [\hat{c}, \ldots] & \textit{expressions} \\
\hat{c} & ::= & \hat{x} = \hat{b} & \textit{clauses} \\
\hat{b} & ::= & \hat{f} \mid \hat{x} \mid \hat{x}\ \hat{x} & \textit{bodies} \\
\hat{x} & ::= & \textit{(identifiers)} & \textit{variables} \\
\hat{v} & ::= & \hat{f} & \textit{values} \\
\hat{f} & ::= & \texttt{fun}\ \hat{x}\ \texttt{->}\ (\ \hat{e}\ ) & \textit{functions}
\end{array}
\qquad
\begin{array}{rcll}
\hat{V} & ::= & \{\hat{v}, \ldots\} & \textit{value sets} \\
\hat{a} & ::= & \hat{c} \mid & \textit{annotated clauses} \\
 & & \hat{x} \overset{\langle\!|\hat{c}}{=} \hat{x} \mid \hat{x} \overset{|\!\rangle\hat{c}}{=} \hat{x} \mid \textsc{Start} \mid \textsc{End} & \\
\hat{g} & ::= & \hat{a} \ll \hat{a} & \textit{control flow graph edges} \\
\hat{G} & ::= & \{\hat{g}, \ldots\} & \textit{control flow graphs} \\
\hat{X} & ::= & [\hat{x}, \ldots] & \textit{variable lookup stacks}
\end{array}$$

Fig. 15. Analysis Grammar Abstract Elements

stack is any variable *other than* y. This means, for example, that the path $\xrightarrow{\downarrow \mathtt{y}}$ ◖ $\xrightarrow{⇅\neg \mathtt{z}} \boxed{\mathtt{z}} \xrightarrow{\uparrow \mathtt{y}} \boxed{\mathtt{y}}$ is valid in this PDR but the path $\xrightarrow{\downarrow \mathtt{y}}$ ◖ $\xrightarrow{⇅\neg \mathtt{z}} \boxed{\mathtt{z}} \xrightarrow{⇅\neg \mathtt{y}} \boxed{\mathtt{y}}$ is not. These so-called *dynamic pop* edges come in a variety of forms and represent PDS transition schemas; the single ⇅¬z edge, for instance, replaces an appropriate pop/push edge pair for *every variable in the program* (excepting z) – the pop immediately followed by a push of the same non-z subject amounts to a no-op.

In general, the PDS which embodies the above lookup algorithm contains patterns of states and edges which are dictated by the execution semantics of the language. We use this and other forms of dynamic pop edges to represent these patterns directly rather than encode them, leading to a much smaller automaton and therefore faster lookup process. In Section 8, we generalize this notion of dynamic pop edge and use them to efficiently implement a form of primitive computation for the more complex rules of the analysis.

4 THE ANALYSIS

In this section, we formalize the DDPA analysis. To simplify presentation, we restrict ourselves to an A-normalized [17] lambda calculus; we outline how additional language features can be introduced in Section 9. The operational semantics of the language is eager and standard, and we postpone it and the soundness proof to Section 6.

The grammar constructs needed for the analysis appear in Figure 15. Items on the left are just the hatted versions of the corresponding program syntax. Functions are the only data type in the simplified language. In the analysis, closure environments are subsumed by our treatment of non-local variables; function values are then represented in the abstract by their bodies alone. Recall from above that we require variables to be bound uniquely (so-called “alphatised” or “uniqued” variables). We also have the common requirement that analyzed expressions are closed: a variable is not used until after a clause in which it is bound.

Edges $\hat{g}$ in a control flow graph $\hat{G}$, are written $\hat{a} \ll \hat{a}'$ and mean clause $\hat{a}$ happens right before clause $\hat{a}'$. New clause annotations $\langle\!|\hat{c}$ / $|\!\rangle\hat{c}$ are used to mark the entry and exit points for functions. The Start node is a special node placed at the very start of the program, and similarly for End. These nodes are needed if any wirings are placed around the first or last program clause.

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

- $\hat{a}_1 \lll \hat{a}_2 \lll \ldots \lll \hat{a}_n$ abbreviates $\{\hat{a}_1 \lll \hat{a}_2, \ldots, \hat{a}_{n-1} \lll \hat{a}_n\}$.
- $\hat{a}' \lll \{\hat{a}_1, \ldots, \hat{a}_n\}$ (resp. $\{\hat{a}_1, \ldots, \hat{a}_n\} \lll \hat{a}'$) denotes $\{\hat{a}' \lll \hat{a}_1, \ldots, \hat{a}' \lll \hat{a}_n\}$ (resp. $\{\hat{a}_1 \lll \hat{a}', \ldots, \hat{a}_n \lll \hat{a}'\}$).
- We overload $\hat{a} \lll \hat{a}'$ to mean $\hat{a} \lll \hat{a}' \in \hat{G}$ for some graph $\hat{G}$ understood from context.

Definition 4.2. Let $\text{RV}(\hat{e}) = \hat{x}$ if $\hat{e} = [\hat{c}, \ldots, \hat{x} = \hat{b}]$. That is, $\hat{x}$ is the return variable of $\hat{e}$.

Definition 4.3. The initial embedding of an expression into a graph, $\widehat{\text{Embed}}([c_1, \ldots, c_n])$, is the graph $\hat{G} = \text{Start} \lll \hat{c}_1 \lll \ldots \lll \hat{c}_n \lll \text{End}$, where each $\hat{c}_i = c_i$.

This initial graph is just the linear sequence of clauses of the main program.

We are using standard notation $[\hat{x}_1, \ldots, \hat{x}_n]$ for lists and $||$ for list append.

4.1 Lookup

As was described in Section 2, the analysis will search back along $\lll$ edges in the graph $\hat{G}$ to find the definitions of variables it needs. We now define this lookup function.

4.1.1 Context Stacks. The definition of lookup proceeds with respect to a current *context stack* $\hat{C}$. 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, and was described in Section 2.2.

The proof of decidability relies upon bounding the depth of the call stack. We first define a general call stack model for DDPA, and in Section 7 below we instantiate the general model with a fixed k-depth call stack version notated kDDPA; this is a simple bounding strategy and our model can in principle work with other strategies.

Definition 4.4. A *context stack model* $\Sigma = \langle \hat{\mathbf{C}}, \epsilon, \text{Push}, \text{Pop}, \text{MaybeTop} \rangle$ obeys the following:

(1) $\hat{\mathbf{C}}$ is a set. We use $\hat{C}$ to range over elements of $\hat{\mathbf{C}}$ and refer to such $\hat{C}$ as *context stacks*.
(2) $\epsilon \in \hat{\mathbf{C}}$.
(3) $\text{Push}(\hat{c}, \hat{C})$ and $\text{Pop}(\hat{C})$ are total functions returning context stacks. $\text{Pop}(\epsilon) = \epsilon$.
(4) $\text{MaybeTop}(\hat{c}, \hat{C})$ is a predicate. $\text{MaybeTop}(\hat{c}, \text{Push}(\hat{c}, \hat{C}))$ and $\text{MaybeTop}(\hat{c}, \epsilon)$ hold.
(5) If $\text{MaybeTop}(\hat{c}, \hat{C})$ then $\text{MaybeTop}(\hat{c}, \text{Pop}(\text{Push}(\hat{c}', \hat{C})))$.

Generally, the context stack is an approximation of the program's runtime call stack. The Push and Pop function derive new context stacks upon calls and returns and the MaybeTop predicate determines whether the top of the runtime call stack *may* be a call from site $\hat{c}$. Models err on the side of overapproximating MaybeTop for soundness. The distinguished context stack ϵ signifies a lack of any context information (and not an empty call stack): popping from ϵ yields ϵ and $\text{MaybeTop}(\hat{c}, \hat{C})$ is always true for any call site $\hat{c}$.

A natural family of context stacks is one which retains up to k top stack frames; to also admit the unbounded case we let k range over $\mathbf{Nat} \cup \omega$ for ω the first limit ordinal. We let $[\hat{c}_1, \ldots, \hat{c}_n]\lceil k$ denote $[\hat{c}_1, \ldots, \hat{c}_m]$ for $m = \min(k, n)$.

Definition 4.5. For every $k \in \mathbf{Nat} \cup \omega$, we define context stack model Σ_k to have $\hat{\mathbf{C}}$ contain the set of all lists of up to length k of clauses $\hat{c}$ occurring in the program. We define the remainder of Σ_k as follows:

- $\epsilon = []$
- $\text{Push}(\hat{c}', [\hat{c}_1, \ldots, \hat{c}_n]) = [\hat{c}', \hat{c}_1, \ldots, \hat{c}_n]\lceil k$
- $\text{Pop}([\hat{c}_1, \ldots, \hat{c}_n]) = [\hat{c}_2, \ldots, \hat{c}_n]$ if $n > 0$; $\text{Pop}([]) = []$.
- $\text{MaybeTop}(\hat{c}', [\hat{c}_1, \ldots, \hat{c}_n])$ is true if $\hat{c}' = \hat{c}_1$ or if $n = 0$; it is false otherwise.

We use the term "kDDPA" to refer to DDPA with context stack model Σ_k.

As above, note that $\epsilon = []$ reflects a lack of *knowledge* of the call stack and not necessarily a lack of *stack frames.*

4.1.2 Lookup stacks. Lookup also proceeds with respect to a *lookup stack* $\hat{X}$. 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.

Unlike the context stack above, the lookup stack is unbounded: the process of looking up a non-local could trigger another non-local lookup of a non-lexically-enclosing function, so there is no lexical upper bound on the depth of this stack in the general case. Though no finite *bound* exists on this stack's depth, every lookup stack is still finite in size.

Also unlike the context stack, there is no graceful way to approximate when lookup stack information is lost. So, we must preserve the whole stack in the analysis. Section 2.3 gave motivation and examples for non-local variable lookup.

4.1.3 Defining the lookup function. Lookup finds the value of a variable starting from a given graph node. Given a control flow graph $\hat{G}$, we write $\hat{G}(\hat{X}, \hat{a}_0, \hat{C})$ to denote a lookup using stack $\hat{X}$ in $\hat{G}$ relative to graph node $\hat{a}_0$ with context $\hat{C}$. For instance, a lookup of variable $\hat{x}$ from program point $\hat{a}$ with unknown context would be written $\hat{G}([\hat{x}], \hat{a}, \epsilon)$. Note that this refers to looking for values of $\hat{x}$ *upon reaching* program point $\hat{a}$ but *before* that point is executed (much like the convention of interactive debuggers); we are looking for a definition of $\hat{x}$ in the *predecessors* of $\hat{a}$ but not within $\hat{a}$ itself.

Definition 4.6. Given control flow graph $\hat{G}$, let $\hat{G}(\hat{X}, \hat{a}_0, \hat{C})$ be the function returning the least set of values $\hat{V}$ satisfying the following conditions given some $\hat{a}_1 \lll \hat{a}_0$:

(1) $\boxed{\textsc{Value Discovery}}$
If $\hat{a}_1 = (\hat{x} = \hat{v})$ and $\hat{X} = [\hat{x}]$, then $\hat{v} \in \hat{V}$.
(2) $\boxed{\textsc{Value Discard}}$
If $\hat{a}_1 = (\hat{x}_1 = \hat{f})$ and $\hat{X} = [\hat{x}_1, \ldots, \hat{x}_n]$ for $n > 0$, then $\hat{G}([\hat{x}_2, \ldots, \hat{x}_n], \hat{a}_1, \hat{C}) \subseteq \hat{V}$.
(3) $\boxed{\textsc{Alias}}$
If $\hat{a}_1 = (\hat{x} = \hat{x}')$ and $\hat{X} = [\hat{x}] \,||\, \hat{X}'$ then $\hat{G}([\hat{x}'] \,||\, \hat{X}', \hat{a}_1, \hat{C}) \subseteq \hat{V}$.
(4) $\boxed{\textsc{Function Enter Parameter}}$
If $\hat{a}_1 = (\hat{x} \overset{(\!|\hat{c}}{=} \hat{x}')$, $\hat{X} = [\hat{x}] \,||\, \hat{X}'$, and $\textsc{MaybeTop}(\hat{c}, \hat{C})$, then $\hat{G}([\hat{x}'] \,||\, \hat{X}', \hat{a}_1, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.
(5) $\boxed{\textsc{Function Enter Non-Local}}$
If $\hat{a}_1 = (\hat{x}'' \overset{(\!|\hat{c}}{=} \hat{x}')$, $\hat{X} = [\hat{x}] \,||\, \hat{X}'$, $\hat{x}'' \neq \hat{x}$, $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, and $\textsc{MaybeTop}(\hat{c}, \hat{C})$, then $\hat{G}([\hat{x}_f, \hat{x}] \,||\, \hat{X}', \hat{a}_1, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.
(6) $\boxed{\textsc{Function Exit}}$
If $\hat{a}_1 = (\hat{x} \overset{|\!)\hat{c}}{=} \hat{x}')$, $\hat{X} = [\hat{x}] \,||\, \hat{X}'$, and $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, then $\hat{G}([\hat{x}'] \,||\, \hat{X}', \hat{a}_1, \textsc{Push}(\hat{c}, \hat{C})) \subseteq \hat{V}$, provided `fun` $\hat{x}''$ `->` `(` $\hat{e}$ `)` $\in \hat{G}([\hat{x}_f], \hat{c}, \hat{C})$ and $\hat{x}' = \textsc{RV}(\hat{e})$.
(7) $\boxed{\textsc{Skip}}$
If $\hat{a}_1 = (\hat{x}'' = b)$, $\hat{X} = [\hat{x}] \,||\, \hat{X}'$, and $\hat{x}'' \neq \hat{x}$, then $\hat{G}(\hat{X}, \hat{a}_1, \hat{C}) \subseteq \hat{V}$.

Note this is a well-formed inductive definition by inspection. Each of the clauses above represents a different case in the reverse search for a variable. We now give clause-by-clause intuitions.

(1) We finally arrived at a definition of the variable $\hat{x}$ and so it must be in the result set.

(2) The variable $\hat{x}_1$ we are searching for has a function value and, unlike clause (1), there are more variables on the stack. This occurs because clause (5), described below, needed to look up the next variable, $\hat{x}_2$, in the place where $\hat{x}_1$ was defined (as in non-local lookup). Now that we have found $\hat{x}_1$, we remove it from the lookup stack and resume the lookup of $\hat{x}_2$.
(3) We have found a definition of $\hat{x}$ but it is defined to be another variable $\hat{x}'$. We transitively switch to looking for $\hat{x}'$.
(4) We have reached the start of the function body and the variable $\hat{x}$ we are searching for was the formal argument $\hat{x}'$. So, continue by searching for $\hat{x}'$ from the call site. The MaybeTop clause constrains this stack frame exit to align with the frame we had last entered (in reverse).
(5) We have reached the beginning of a function body and did not find a definition for the variable $\hat{x}$. In this case, we switch to searching for the clause that defined this function body, which leads us to push $\hat{x}_f$ onto the lookup stack. Once the defining point of $\hat{x}_f$ is found, we will pop it and resume looking for $\hat{x}$ (see clause (2)). The MaybeTop clause constrains the stack frame being exited to align with the frame we had last entered (in reverse).
(6) We have reached a return copy which is assigning our variable x, so to look for x we need to continue by looking for x' inside this function. Push $\hat{c}$ on the stack since we are now entering the body (in reverse) via that call site. For a more accurate analysis, the "provided" line additionally requires that we *only* "walk back" into function(s) that could have reached this call site; so, we launch a subordinate lookup of $\hat{x}_f$ and constrain $\hat{a}_1$ accordingly.
(7) Here the examined clause is not a match so the search continues at any predecessor node. Note this will chain past function call sites which did not return the variable $\hat{x}$ we are looking for. This is sound in a pure functional language; when we address state in Section 9.4, we will enter such a function to verify an alias to our variable was not assigned.

4.2 Abstract Evaluation

We are now ready to present the single-step abstract evaluation relation which incrementally adds edges to the control flow graph. This system has some parallels with a graph-based notion of evaluation [27, 53], but in our system function bodies are never copied – a single body is shared.

4.2.1 Active nodes. While evaluation is abstract and graph-based, it shares some features with standard evaluation: there is an evaluation context [16] of the already-evaluated "expression" (here a graph) and we need to next evaluate the current "redex", which here we call the *active node*. In particular, only nodes with all previous nodes wired-in can fire.

Definition 4.7. $\widehat{\text{Active?}}(\hat{a}', \hat{G})$ iff path $\textsc{Start} \lll \hat{a}_1 \lll \ldots \lll \hat{a}_n \lll \hat{a}'$ appears in $\hat{G}$ such that no $\hat{a}_i$ is of the form $\hat{x} = \hat{x}'\ \hat{x}''$. We write $\widehat{\text{Active?}}(\hat{a}')$ when $\hat{G}$ is understood from context.

4.2.2 Wiring. Recall from Section 2 how function application required the concrete function body to be "wired" directly in to the call site node, and how additional nodes were added to copy in the argument and out the result. The following definition accomplishes this.

Definition 4.8. Let $\widehat{\text{Wire}}(\hat{c}', \texttt{fun}\ \hat{x}_0 \texttt{ -> } (\,[\hat{c}_1, \ldots, \hat{c}_n]\,), \hat{x}_1, \hat{x}_2) =$
$$\textsc{Preds}(\hat{c}') \lll (\hat{x}_0 \overset{\Lsh\hat{c}'}{=} \hat{x}_1) \lll \hat{c}_1 \lll \ldots \lll \hat{c}_n \lll (\hat{x}_2 \overset{\Rsh\hat{c}'}{=} \text{RV}([\hat{c}_n])) \lll \textsc{Succs}(\hat{c}')$$
where $\textsc{Preds}(\hat{a}) = \{\hat{a}' \mid \hat{a}' \lessdot \hat{a}\}$ and $\textsc{Succs}(\hat{a}) = \{\hat{a}' \mid \hat{a} \lessdot \hat{a}'\}$.

$\hat{c}'$ here is the call site, and $\hat{c}_1 \lll \ldots \lll \hat{c}_n$ is the wiring of the function body. The Preds/Succs functions reflect how we simply wire to the existing predecessor(s) and successor(s).

Next, we define the abstract small-step relation $\widehat{\longrightarrow}^1$ on graphs. With the above preliminaries, this is easy to define: for each reachable function application with a particular function and argument,

$$\textsc{Application}\quad \frac{\hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3) \qquad \widehat{\textsc{Active?}}(\hat{c}, \hat{G}) \qquad \hat{f} \in \hat{G}([\hat{x}_2], \hat{c}, \epsilon) \qquad \hat{v} \in \hat{G}([\hat{x}_3], \hat{c}, \epsilon)}{\hat{G} \mathrel{\widehat{\longrightarrow}^1} \hat{G} \cup \widehat{\textsc{Wire}}(\hat{c}, \hat{f}, \hat{x}_3, \hat{x}_1)}$$

Fig. 16. Abstract Evaluation Rule

we add wiring nodes to copy the argument into the function and copy its return value to the call site.

Definition 4.9. We define the small step relation $\widehat{\longrightarrow}^1$ to hold if a proof exists in the system in Figure 16. We write $\hat{G}_0 \mathrel{\widehat{\longrightarrow}^*} \hat{G}_n$ to denote $\hat{G}_0 \mathrel{\widehat{\longrightarrow}^1} \hat{G}_1 \mathrel{\widehat{\longrightarrow}^1} \ldots \mathrel{\widehat{\longrightarrow}^1} \hat{G}_n$.

The A-normalized lambda calculus only requires an application rule, and languages with additional control flow constructions (such as the language of Section 9) will need additional rules.

The next sections show the formal properties of the above analysis. Section 5 proves undecidability with a non-finite context stack model. Section 6 demonstrates the soundness of DDPA with respect to a standard small-step operational semantics, while Section 7 proves DDPA to be decidable by reducing the lookup procedure to a PDS reachability problem.

5 A GRAPH-BASED OPERATIONAL SEMANTICS

In this and the following sections we show the soundness of DDPA. We do so in a fashion common to higher-order program analyses [30, 44, 51]: we prove a standard operational semantics equivalent to a non-standard one and then show the analysis is a sound abstract interpretation [8] of the non-standard semantics. For clarity of presentation, our translation of the operational semantics moves through two intermediate systems as illustrated in Figure 17.

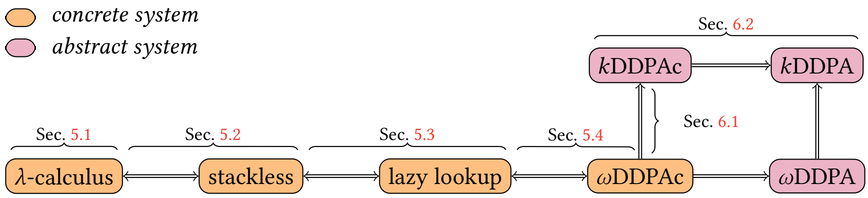


Fig. 17. Soundness Proof Systems

The operational semantics we start with is closure-based; we take this as ground truth as it is well-known to be equivalent to other operational semantics for the call-by-value λ-calculus. The target non-standard operational semantics, which we term ωDDPAc, is a *graph-based* operational semantics which is nearly identical in form to DDPA as presented in Section 4 but is still a full and faithful interpreter for the CBV λ-calculus.

We believe ωDDPAc is a unique and interesting presentation of operational semantics in its own right which may independently have other applications. It lacks closures, substitution, fresh variables, or an environment, and reductions are all polynomially bounded in length, a very surprising feature for a Turing-complete interpreter. (This bound may initially seem paradoxical, but the reduction steps themselves are undecidable. We discuss this further in Section 5.5.)

We dedicate this section of the paper to demonstrating the equivalence of the ground truth semantics and ωDDPAc; soundness of DDPA with respect to ωDDPAc is then shown in Section 6.

$$\begin{array}{rcll} e & ::= & [c, \ldots] & \textit{expressions} \\ c & ::= & x = b & \textit{clauses} \\ b & ::= & f \mid x \mid x\ x & \textit{bodies} \\ x & ::= & \textit{(identifiers)} & \textit{variables} \\ v & ::= & f & \textit{values} \\ f & ::= & \texttt{fun}\ x\ \texttt{->}\ \texttt{(}\ e\ \texttt{)} & \textit{functions} \end{array}$$

Fig. 18. Concrete Language Grammar

$$\begin{array}{rcll} \kappa & ::= & \langle f, E \rangle & \textit{closures} \\ E & ::= & [x \mapsto \kappa, \ldots] & \textit{environments} \\ \phi & ::= & [\langle E, e \rangle, \ldots] & \textit{evaluation states} \end{array}$$

Fig. 19. Closure-Based Evaluation Grammar

$$\textsc{Definition}\qquad \frac{}{[\langle E, [x = f] \mathbin{||} e \rangle] \mathbin{||} \phi \longrightarrow^1 [\langle E \mathbin{||} [x \mapsto \langle f, E \rangle], e \rangle] \mathbin{||} \phi}$$

$$\textsc{Alias}\qquad \frac{(x_2 \mapsto \kappa) \in E}{[\langle E, [x_1 = x_2] \mathbin{||} e \rangle] \mathbin{||} \phi \longrightarrow^1 [\langle E \mathbin{||} [x_1 \mapsto \kappa], e \rangle] \mathbin{||} \phi}$$

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

$$\textsc{Return}\qquad \frac{}{[\langle E \mathbin{||} [x \mapsto \kappa], [] \rangle, \langle E', [x_1 = x_2\ x_3] \mathbin{||} e \rangle] \mathbin{||} \phi \longrightarrow^1 [\langle E' \mathbin{||} [x_1 \mapsto \kappa], e \rangle] \mathbin{||} \phi}$$

Fig. 20. Closure-Based Operational Semantics

5.1 Closure-Based Operational Semantics

We begin by defining an environment/stack/closure-based operational semantics for the λ-calculus. This is not far in spirit from a CEK machine [15]. The grammar of our language appears in Figure 18; this is simply the grammar from Figure 15 with hats removed (or, the grammar from Figure 1 without records, pattern matching, or state). We additionally define the grammar of Figure 19 for use in our operational semantics: environments E are mappings from variables to closures, closures κ are pairs of functions and environments, and evaluation states ϕ are a stack of pairings between environment and instructions to be executed.

As in Section 2, we restrict all variable bindings throughout a particular program to be unique for convenience. We define $\textsc{rv}(e)$ similarly to Definition 4.2 to return the last variable defined by an expression.

We define the closure-based small step operational semantics as a relation $\phi \longrightarrow^1 \phi$ as follows:

Definition 5.1. $\phi \longrightarrow^1 \phi$ holds if a proof exists in the system of Figure 20. We write $\phi_0 \longrightarrow^* \phi_n$ iff $\phi_0 \longrightarrow^1 \ldots \longrightarrow^1 \phi_n$.

The rules in Figure 20 are largely straightforward. Each rule acts upon the topmost element of the ϕ stack. The Definition rule, upon encountering an assignment of function f to variable x, will add the binding $x \mapsto \langle f, E \rangle$ to the environment, where E is a copy of the previous environment

$$
\begin{array}{rcll}
l & ::= & x \mapsto \kappa \mid \text{◖} \mid \text{◗} & \textit{stackless evaluation terms} \\
L & ::= & [l, \ldots] & \textit{stackless evaluation logs} \\
\iota & ::= & c \mid x \overset{\text{◗}c}{=} x & \textit{stackless evaluation instruction} \\
I & ::= & [\iota, \ldots] & \textit{stackless evaluation expressions}
\end{array}
$$

Fig. 21. Stackless Evaluation Grammar

representing the non-local variables of f. The Alias rule is similar, copying the current value of x_2 from the environment into a new binding for x_1. The Call rule pushes a new frame onto the ϕ stack in which to execute the function's body; the Return rule pops a completed frame and updates the caller's environment with the functions return value which, as in Definition 4.2, we take to be the last assignment in the function's body.

Note that, throughout evaluation, the only variables added to an environment E on the stack are those from the function's closure and from the function body. Due to the unique variable requirement of expressions, no variable can be defined in both; as a result, each $x \mapsto \kappa$ within an environment maps a distinct variable.

5.2 A Stackless Operational Semantics

We now begin taking the steps toward ωDDPAc as outlined in Figure 17. Each step toward ωDDPAc makes some aspect of the system more demand-driven rather (while still maintaining call-by-value evaluation semantics). In this first step, we define an operational semantics which stores binding information in a flat historical log rather than in a stack. In addition to bindings, this log stores events in which stack frames are pushed and popped, so it can fully replace the environments E of the previous system.

The grammar for the stackless system appears in Figure 21. An evaluation state is a pairing between a log L and a stackless expression I; note that every e is of form I. In addition to clauses, stackless expressions may include annotated assignments indicating when functions return. When functions are called, ◖ is added to the log to record the event; when a function returns, this is recorded with ◗. As a result, the log L stores all bindings throughout the execution of program (even those which are no longer in scope); we then define a function to extract an environment from a provided log which will skip over any variables in functions that have already returned.

Definition 5.2. We define the environment splitting function $\textsc{SplitEnv}(L, n, E)$ for non-negative integer n as follows:

$$
\begin{array}{rcll}
\textsc{SplitEnv}([], n, E) & = & \langle [], E \rangle & \\
\textsc{SplitEnv}(L' \,||\, [\text{◗}], n, E) & = & \textsc{SplitEnv}(L', n+1, E) & \\
\textsc{SplitEnv}(L' \,||\, [\text{◖}], 0, E) & = & \langle L', E \rangle & \\
\textsc{SplitEnv}(L' \,||\, [\text{◖}], n, E) & = & \textsc{SplitEnv}(L', n-1, E) & \text{when } n > 0 \\
\textsc{SplitEnv}(L' \,||\, [x \mapsto \kappa], 0, E) & = & \textsc{SplitEnv}(L', 0, [x \mapsto \kappa] \,||\, E) & \\
\textsc{SplitEnv}(L' \,||\, [x \mapsto \kappa], n, E) & = & \textsc{SplitEnv}(L', n, E) & \text{when } n > 0
\end{array}
$$

We write $\textsc{SplitEnv}(L)$ to abbreviate $\textsc{SplitEnv}(L, 0, [])$. We define $\textsc{ExtractEnv}(L) = E$ when $\textsc{SplitEnv}(L) = \langle L', E \rangle$.

This function traces the log backwards, building the environment E by finding each binding which occurred during the call to this function or one of its calling ancestors. The second argument is a number counting intermediate function calls to ensure that non-closure-captured bindings from within previously-called functions are not included in the resulting environment.

Definition

$$\frac{}{L; [x = f] \,||\, I \longrightarrow^1 L \,||\, [x \mapsto \langle f, \textsc{ExtractEnv}(L)\rangle]; I}$$

Alias

$$\frac{x_2 \mapsto \kappa \in \textsc{ExtractEnv}(L)}{L; [x_1 = x_2] \,||\, I \longrightarrow^1 L \,||\, [x_1 \mapsto \kappa]; I}$$

Call

$$\frac{c = (x_1 = x_2\ x_3) \qquad (x_2 \mapsto \langle \texttt{fun}\, x_4 \texttt{ -> } (\ e'\ )\ , E\rangle) \in \textsc{ExtractEnv}(L) \qquad (x_3 \mapsto \kappa) \in \textsc{ExtractEnv}(L)}{L; [c] \,||\, I \longrightarrow^1 L \,||\, [\text{◖}] \,||\, E \,||\, [x_4 \mapsto \kappa]; e' \,||\, [x_1 \overset{\text{◗}c}{=} \text{RV}(e')] \,||\, I}$$

Return

$$\frac{(x_2 \mapsto \kappa) \in \textsc{ExtractEnv}(L)}{L; [x_1 \overset{\text{◗}c}{=} x_2] \,||\, I \longrightarrow^1 L \,||\, [\text{◗}, x_1 \mapsto \kappa]; I}$$

Fig. 22. Stackless Operational Semantics

With this extraction function, we can define the stackless operational semantics. We overload the $\longrightarrow^1$ operator as follows:

Definition 5.3. $L; I \longrightarrow^1 L'; I'$ holds if a proof exists in the system of Figure 22. We write $L_0, I_0 \longrightarrow^* L_n, I_n$ iff $L_0, I_0 \longrightarrow^1 \ldots \longrightarrow^1 L_n, I_n$.

5.2.1 Proof of Equivalence. We now demonstrate the equivalence of the above operational semantics to the one in the previous section. This is accomplished by establishing a bisimulation $\cong$ between the original program states ϕ and the stackless program states $L; I$. We formalize this bisimulation as follows:

Definition 5.4. We write $\phi \cong L; I$ to mean either $\phi = L = I = []$ or all of the following:

- $\phi = [\langle E, e\rangle] \,||\, \phi'$. (The stack is non-empty)
- $\textsc{SplitEnv}(L) = \langle L', E\rangle$. (The extracted environment matches the topmost stack frame.)
- $I = e \,||\, [\text{◗}] \,||\, I'$. (The remaining instructions are the same in both systems.)
- If $\phi' \neq []$ then $\phi' \cong L'; I'$. (This property holds inductively for each stack frame.)

This bisimulation trivially holds for the start of evaluation: that is, for any e, $[\langle [], e\rangle] \cong []; e$. Proving the remainder of equivalence relies upon a key lemma to preserve the bisimulation at each evaluation step:

Lemma 5.5. *If $\phi \cong L; I$ then*

- *if $\phi \longrightarrow^1 \phi'$ then $L; I \longrightarrow^1 L'; I'$ such that $\phi' \cong L'; I'$, and*
- *if $L; I \longrightarrow^1 L'; I'$ then $\phi \longrightarrow^1 \phi'$ such that $\phi' \cong L'; I'$.*

Proof. By case analysis on the rule used. In particular, each rule in Figure 20 aligns with each rule in Figure 22 such that the premises of a rule can be proven by the premises of its counterpart and the properties of the bisimulation. □

In the above proof, the only non-trivial steps exist between the Call and Return rules. First, the systems differ in how they represent a call in progress (an existing call site on the stack in the original system and an annotation in the stackless system). Second, we must be able to demonstrate that ExtractEnv correctly describes the environment E both when a new function is called and when a running function returns. The latter relies upon the ◖ and ◗ symbols appearing in the binding log;

$$
\begin{array}{rcll}
z & ::= & x = v \mid x = x \mid x \overset{⦇c}{=} x \mid x \overset{⦈c}{=} x & \textit{environment terms} \\
Z & ::= & [z, \ldots] & \textit{environments} \\
w & ::= & c \mid x \overset{⦇c}{=} x \mid x \overset{⦈c}{=} x & \textit{clauses} \\
W & ::= & [w, \ldots] & \textit{expressions} \\
X & ::= & [x, \ldots] & \textit{variable stacks}
\end{array}
$$

Fig. 23. Lazy Lookup Evaluation Grammar

that these annotations are present is a consequence of the inductive property of the bisimulation described above.

5.3 An Operational Semantics with Lazy Lookup

Our next step toward ωDDPAc is to define an operational semantics which looks up the value of variables on demand rather than eagerly constructing bindings. This lazy lookup operation is starting to get close to DDPA's lookup operation (Definition 4.6): like DDPA, it traces backward through the program to reconstruct bindings as needed, including reconstruction of the context of a function's closure when needed.

We define the grammar we require for this system in Figure 23. Unlike the previous systems, our program state for this operational semantics will simply be an expression W with no explicit environment. Although we provide a grammar of environments Z, this is primarily used to describe the first unevaluated call site of the expression W.

Lazy lookup is defined in function $Z(X, n)$ as follows. We use X as a stack of variables in a fashion similar to DDPA's Definition 4.6. The integer n serves a purpose similar to Definition 5.2: to skip over bindings no longer in scope.

Definition 5.6. For a given environment Z, we define the lookup function $Z(X, n)$ as follows:

(1) If $Z = Z' \,||\, [x = v]$ then $Z([x], 0) = v$.
(2) If $Z = Z' \,||\, [x = f]$ and $X = [x] \,||\, X'$ for $X' \neq []$ then $Z(X, 0) = Z'(X', 0)$.
(3) If $Z = Z' \,||\, [x = x']$ then $Z([x] \,||\, X', 0) = Z'([x'] \,||\, X', 0)$.
(4) If $Z = Z' \,||\, [x \overset{⦇c}{=} x']$ then $Z([x] \,||\, X', 0) = Z'([x'] \,||\, X', 0)$.
(5) If $Z = Z' \,||\, [x'' \overset{⦇c}{=} x']$, $x'' \neq x$, and $c = (x_r = x_f\ x_v)$, then $Z([x] \,||\, X', 0) = Z'([x_f, x] \,||\, X', 0)$.
(6) If $Z = Z' \,||\, [x \overset{⦈c}{=} x']$ then $Z([x] \,||\, X', 0) = Z'([x'] \,||\, X', 0)$.
(7) If $Z = Z' \,||\, [x' = b]$ and $x \neq x'$ then $Z([x] \,||\, X', n) = Z'([x] \,||\, X', n)$.
(8) If $Z = Z' \,||\, [x = b]$ and $n > 0$ then $Z(X, n) = Z'(X, n)$.
(9) If $Z = Z' \,||\, [x \overset{⦇c}{=} x']$ and $n > 0$ then $Z(X, n) = Z'(X, n - 1)$.
(10) If $Z = Z' \,||\, [x \overset{⦈c}{=} x']$ and $n > 0$ then $Z(X, n) = Z'(X, n + 1)$.

We write $Z(X)$ to mean $Z(X, 0)$.

As with the environment extraction function in Definition 5.2, the integer here is used to disregard variables which were bound in calls that have since completed. One key difference in these definitions is that Definition 5.2 stops its work upon reaching a ⦇ symbol with $n = 0$ (which indicates that we have left local scope) whereas Definition 5.6 continues past the ⦇ with $n = 0$. In the eager stackless system of Section 5.2, we copied the closure of each function into place immediately after the start-of-call symbol ⦇. In this lazy system, we will not; instead, for non-local

$$\textsc{Application}\quad \frac{c = (x_1 = x_2\ x_3) \qquad Z([x_2], 0) = \texttt{fun}\, x_4 \texttt{ -> } (\, e'\,) \qquad Z([x_3], 0) = v}{Z \,||\, [c] \,||\, W \longrightarrow^1 Z \,||\, [x_4 \overset{◖c}{=} x_3] \,||\, e' \,||\, [x_1 \overset{◗c}{=} \mathrm{RV}(e')] \,||\, W}$$

Fig. 24. Lazy Lookup Operational Semantics

variables captured in closure, clause 5 will first identify the point in time at which the closure was captured and then continue lookup from that point.

This lookup definition is similar to Definition 4.6 of DDPA. The most notable differences are the absence of a context parameter and the presence of the n parameter. The former is not necessary here as context can be established from a traversal of Z. The latter is required here but not in the analysis because the program point immediately following a function call in DDPA has at least two predecessors – the call's wiring nodes and the call node itself – while the list Z may only have one. The next section defines an operational semantics to bridge this gap.

Given the above lazy lookup function, we can define an operational semantics as follows:

Definition 5.7. $W \longrightarrow^1 W'$ holds if a proof exists in the system of Figure 24. We write $W_0 \longrightarrow^* W_n$ iff $W_0 \longrightarrow^1 \ldots \longrightarrow^1 W_n$.

Note that Figure 24 contains only one evaluation rule, lining up closely with DDPA Figure 16 and considerably simplifying the four rules from the previous systems. The previous Definition and Alias rules are obsolete here due to the lazy manner in which lookup occurs and the fact that we no longer construct explicit closures. The Call and Return rules have been grouped into a single Application rule; this is also possible due to lazy lookup, as we no longer need to process the exit annotation when the function returns.

5.3.1 *Proof of Equivalence.*

We now demonstrate that the operational semantics just defined is equivalent to the stackless semantics defined in Section 5.2. As in Section 5.2.1, we demonstrate this via a bisimulation between states of the two systems. This bisimulation is somewhat more subtle, however, as we must first align the eager and lazy environments and then align evaluation states.

To describe the relationship between the systems' environments, we overload the $\cong$ notation to describe an alignment between eager environments E (which are generated in the stackless system by ExtractEnv) and *pairs* of lazy lookup environment Z and variable stack X. It is necessary but not sufficient to require each binding in E to match the results of lookup on Z and vice versa; to correctly handle higher-order functions, we must also ensure that closures are correctly represented. The variable stack X in this bisimulation describes the sequence of lookups necessary to reach the point where a particular closure is defined. We thus write this bisimulation as follows:

Definition 5.8. We write $E \cong Z; X$ to mean:

- For all $x \mapsto \langle f, E' \rangle$ in E, $Z(X \,||\, [x]) = f$ and $E' \cong Z; (X \,||\, [x])$.
- For all $Z(X) = f$, $x \mapsto \langle f, E' \rangle$ appears in E such that $E' \cong Z; (X \,||\, [x])$.

That is, an eager function lookup aligns with a lazy function lookup if the alignment property applies recursively to the eager function's closure. The additional x is used to continue to describe a path through Z to the point at which the function's closure is defined. This definition is well-founded because the first defined function will always have an empty closure, making the bisimulation property for that function trivial.

Given a means by which environments can be aligned, we can then define the bisimulation between the stackless and lazy lookup systems:

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

Fig. 25. ωDDPAc Evaluation Grammar

Definition 5.9. We write $L; I \cong W$ to mean:

- $W = Z \,||\, W'$ for the largest possible Z; that is, the first element of W' is the first application in W (or W' is empty).
- $I = W'$; that is, the list of unperformed work is the same.
- $\textsc{ExtractEnv}(L) \cong Z; []$; that is, every binding is correctly represented by lazy lookup.

Again the main lemma is bisimulation preservation:

Lemma 5.10. *If $L; I \cong W$ then*

- *if $W \longrightarrow^1 W'$ then $L; I \longrightarrow^* L'; I'$ such that $L'; I' \cong W'$, and*
- *if $L; I \longrightarrow^1 L'; I'$ then $L'; I' \longrightarrow^* L''; I''$ and $W \longrightarrow^1 W''$ such that $L''; I'' \cong W''$.*

This lemma displays an asymmetry which hints at a difference between the two systems: intuitively, the stackless system takes smaller steps than the lazy lookup system. The only step in the lazy system is application; the definition clauses which appear as a result, for instance, are implicitly processed by lazy lookup later and upon demand. In essence, the eager system may need to take many steps to "catch up" to the lazy system's state. Likewise, a single step in the eager system may not align directly with the lazy system, but the eager system will eventually catch up to the single step taken by the lazy system by processing any definitions, aliases, etc. which the lazy system deferred.

The initial bisimulation is not immediate but is relatively easy to prove using the above reasoning: the starting expression may have to take a few steps to catch up to the initial state of the lazy system, but a bisimulation is provable upon reaching the first application (or the end of the program).

Lemma 5.11. *For all e, $[]; e \longrightarrow^* L; I$ such that $L; I \cong W$ where $W = e$.*

In this lemma note that all e are of form W. The equivalence of the stackless and lazy systems then follows directly by induction on computation length using the above two lemmas.

5.4 ωDDPAc: A Graph-Based Operational Semantics

We now present ωDDPAc, our final operational semantics, and prove it equivalent to the lazy lookup system just defined. ωDDPAc is a *graph-based operational semantics* which represents expressions as *concrete (runtime) control flow graphs* rather than lists with a fixed point of execution. It only differs from the analysis of Section 4 in two ways: the calling contexts are fixed to be the full call stack without approximation (the ω in the name), and the wiring rule is refined compared to DDPA to take the current context into account (the additional "c" at the end of ωDDPAc).

We define the grammar of ωDDPAc in Figure 25. This grammar is structurally very similar to the analysis. The only difference is that we are using a list of contexts C as opposed to a general context stack model Σ. We use notation similar to Definition **??** for these graphs; for instance, we sometimes write $a \lessdot a'$ to mean $(a \ll a') \in G$ for some G understood from context.

In comparison to the lazy lookup system of Section 5.3, the key difference is that, rather than representing execution as a *list* W, this system uses a *graph* G. The nodes of the graph are individual program clauses and the special nodes Start and End, representing the start and end of the overall program. Edges in the graph will represent control flow which occurs *at least once* during execution. Intuitively, if we evaluate the same program in each system in lock-step and pick a moment during evaluation, the lazy lookup evaluation state W will correspond to a *path* in the ωDDPAc state G.

A consequence of this model is that, for any program, the set of possible control flow decisions (here, graph edges) is finite and monotonically increasing. (We discuss this further in Section 5.5.) However, the number of program states (here, paths in the graph) may not be. A loop, for instance, manifests as a repeated pattern in W but as a cycle in G. As a result, a particular node in the graph is insufficient to describe the current state of execution: we must also be able to identify how we reached that point. This is achieved by the use of a stack of clauses C which for the simple language in this section, C corresponds to the runtime call stack of the program.

To define ωDDPAc, we require a lazy lookup function similar to that which characterized the system of Section 5.3. In this case, the graph affords us the ability to skip over out-of-scope bindings without an integer counter by relying on a wiring process similar to the $\widehat{\textsc{Wire}}$ function in Section 4. This leads us to a definition in near-perfect alignment with the lookup function of Section 4.1:

Definition 5.12. Given control flow graph G, $G(X, a_0, C)$ is the function returning the least set of values V satisfying the following conditions given some $a_1 \lessdot a_0$:

(1) $\boxed{\textsc{Value Discovery}}$
If $a_1 = (x = v)$ and $X = [x]$, then $v \in V$.
(2) $\boxed{\textsc{Value Discard}}$
If $a_1 = (x_1 = \hat{f})$ and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $G([x_2, \ldots, x_n], a_1, C) \subseteq V$.
(3) $\boxed{\textsc{Alias}}$
If $a_1 = (x = x')$ and $X = [x] \mathbin{||} X'$ then $G([x'] \mathbin{||} X', a_1, C) \subseteq V$.
(4) $\boxed{\textsc{Function Enter Parameter}}$
If $a_1 = (x \overset{(\!|c}{=} x')$, $X = [x] \mathbin{||} X'$, and $C = [c] \mathbin{||} C'$, then $G([x'] \mathbin{||} X', a_1, C') \subseteq V$.
(5) $\boxed{\textsc{Function Enter Non-Local}}$
If $a_1 = (x'' \overset{(\!|c}{=} x')$, $X = [x] \mathbin{||} X'$, $x'' \neq x$, $c = (x_r = x_f\ x_v)$, and $C = [c] \mathbin{||} C'$, then $G([x_f, x] \mathbin{||} X', a_1, C') \subseteq V$.
(6) $\boxed{\textsc{Function Exit}}$
If $a_1 = (x \overset{|\!)c}{=} x')$, $X = [x] \mathbin{||} X'$, and $c = (x_r = x_f\ x_v)$, then $G([x'] \mathbin{||} X', a_1, [c] \mathbin{||} C) \subseteq V$, provided `fun` x'' `->` `(` e `)` $\in G([x_f], c, C)$ and $x' = \textsc{rv}(e)$.
(7) $\boxed{\textsc{Skip}}$
If $a_1 = (x'' = b)$, $X = [x] \mathbin{||} X'$, and $x'' \neq x$, then $G(X, a_1, C) \subseteq V$.

Given the above lookup function, we define ωDDPAc in the same fashion as CFG construction in the analysis: an incremental construction of the graph structure based upon the values discovered by demand-driven lookup. See Figure 26 for the (sole) rule, in analogy with Figure 16 of the analysis. Comparing these two figures there is one non-trivial change which makes ωDDPAc slightly more refined (and is the source of the appended "c" in the name): variables f and v are looked up relative to the current context C, whereas DDPA's wiring rule drops this context information for simplicity. The definition of $\textsc{Active}$ is also different in ωDDPAc since this context is needed: it returns the set of possible contexts that could be active at this point in the graph. In Section 4's definition of DDPA, the context is not needed and the corresponding $\widehat{\textsc{Active?}}$ function is a predicate rather than a function onto sets of contexts. The precise definition of $\textsc{Active}$ is as follows.

Definition 5.13. Let $\textsc{Active}(G, a)$ be least set $\boldsymbol{C}$ conforming to the following conditions:

- If $c \lessdot a$ then $\textsc{Active}(G, c) \subseteq \boldsymbol{C}$.

$$\textsc{Application}\quad \frac{c = (x_1 = x_2\ x_3) \qquad C \in \textsc{Active}(c, G) \qquad f \in G([x_2], c, C) \qquad v \in G([x_3], c, C)}{G \longrightarrow^1 G \cup \textsc{Wire}(c, f, x_3, x_1)}$$

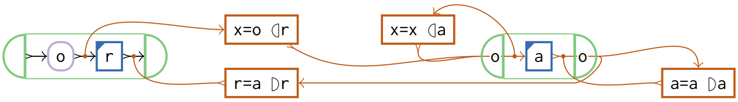
Fig. 26. The ωDDPAc Operational Semantics

```
o = fun x -> ( a = x x );
r = o o;
```

ω-combinator code

$$W = [\mathtt{o} = \ldots, \mathtt{x} \overset{\llparenthesis \mathtt{r}}{=} \mathtt{o}, \mathtt{x} \overset{\llparenthesis \mathtt{a}}{=} \mathtt{x}, \ldots, \mathtt{x} \overset{\llparenthesis \mathtt{a}}{=} \mathtt{x}, \underline{\mathtt{a} = \mathtt{x}\ \mathtt{x}}, \mathtt{a} \overset{\rrparenthesis \mathtt{a}}{=} \mathtt{a} \ldots, \mathtt{a} \overset{\rrparenthesis \mathtt{a}}{=} \mathtt{a}, \mathtt{r} \overset{\rrparenthesis \mathtt{r}}{=} \mathtt{a}]$$

Example of lazy lookup evaluation in progress



Example of ωDDPAc evaluation (completed graph)

Fig. 27. ω-combinator Execution

- If $a' \lessdot a$ for $a' = (x \overset{\llparenthesis c}{=} x')$ and $C \in \textsc{Active}(G, a')$, then $(C \,||\, [c]) \in \boldsymbol{C}$.
- If $\textsc{Start} \lessdot a$ then $[] \in \boldsymbol{C}$.

Note that $\boldsymbol{C}$ may not be finite.

We define $\textsc{Wire}$ in analogy with $\widehat{\textsc{Wire}}$ (Definition 4.8) simply by removing the hats from each term. We then define ωDDPAc itself as follows:

Definition 5.14. $G \longrightarrow^1 G'$ holds if a proof exists in the system of Figure 26. We write $G_0 \longrightarrow^* G_n$ iff $G_0 \longrightarrow^1 \ldots \longrightarrow^1 G_n$.

5.4.1 *Proof of Equivalence.*

We now show that ωDDPAc as defined above is equivalent to the lazy lookup system of Section 5.3. This is the final operational semantics equivalence proof: it allows us to connect the standard operational semantics to this graph-based form.

Considering Alignment. Alignment between the lazy lookup system and ωDDPAc is somewhat more involved than the previous alignments. Non-recursive programs always evaluate in lock-step between the two systems – for each list substitution in the lazy lookup system, the same nodes and edges are added in ωDDPAc – but, in recursive programs, the ωDDPAc semantics may "get ahead" of the lazy lookup semantics.

For instance, recall the ω-combinator example from Section 2.5 which we reproduce here in Figure 27. The lazy lookup system will continuously grow a list W by substituting the call site `a = x x` (underlined in the figure) with itself surrounded by wiring annotations. In ωDDPAc, however, the graph stops growing after the Application rule acts once on each call site. The cycles formed by that rule effectively describe *all* such lists W, since those lists will grow according to a pattern in this divergent case.

This does not mean that ωDDPAc is sub-Turing or that we claim to predict halting (via whether the $\textsc{End}$ node is $\textsc{Active}$). While the size of the concrete control flow graph for any program is finite, computing the next graph edge to be added is a recursively enumerable but not recursive property: the Application rule must consider unboundedly many contexts, any one of which may lead to additional control flow edges.

Non-divergent recursive programs exhibit behavior similar to the above. In a program which recurses a fixed number of times (e.g. via a counter), the count-down step would be processed

$$\textsc{Step}\quad \frac{W_n \longrightarrow^1 W_{n+1}}{[W_1, \ldots, W_n] \longrightarrow^1 [W_1, \ldots, W_n, W_{n+1}]}$$

Fig. 28. Historical Lazy Lookup Operational Semantics

exactly once by the graph-based semantics. In such cases, a finite number of steps in the lazy lookup system is sufficient to catch up to the graph-based system, at which point the systems continue operating in step.

Historical Evaluation. Defining the bisimulation between these two systems is not straightforward because ωDDPAc leaves application nodes in the graph. This means there are nodes in G which do not appear in W. At the same time, the bisimulation must be two-directional: we can't allow arbitrary extra content in G. That is, we must allow "good junk" (old call sites) without allowing arbitrary "bad junk".

We address this by defining a form of *historical evaluation* for the lazy lookup system. This system mimics the lazy lookup system but retains every step of evaluation and so implicitly retains the original application nodes. We use $\boldsymbol{W}$ to represent a list of W. We formally define that system as follows:

Definition 5.15. $\boldsymbol{W} \longrightarrow^1 \boldsymbol{W}'$ holds if a proof exists in the system of Figure 22. We write $\boldsymbol{W}_0 \longrightarrow^* \boldsymbol{W}_n$ iff $\boldsymbol{W}_0 \longrightarrow^1 \ldots \longrightarrow^1 \boldsymbol{W}_n$. We say a list $\boldsymbol{W} = [W_1, \ldots W_n]$ is *historical* iff, for all $1 \leq i < n$, $W_i \longrightarrow^1 W_{i+1}$.

The lazy lookup system and the historical lazy lookup system are trivially equivalent:

Lemma 5.16. *For any* W_0, $W_0 \longrightarrow^1 \ldots \longrightarrow^1 W_n$ *iff* $[W_0] \longrightarrow^1 \ldots \longrightarrow^1 [W_0, \ldots, W_n]$.

We can use the historical information to drive our bisimulation.

Bisimulation and Equivalence. Define bisimulation between the historical system and ωDDPAc as follows:

Definition 5.17. We write $\boldsymbol{W} \cong G$ to mean the following:

- $\boldsymbol{W}$ is historical.
- For each $[w_1, \ldots, w_n]$ in $\boldsymbol{W}$, $\{\textsc{Start} \lll w_1 \lll \ldots w_n \lll \textsc{End}\} \subseteq G$.
- For each $(w \lll w') \in G$, there is some $W \in \boldsymbol{W}$ such that $W = W' \,||\, [w, w'] \,||\, W''$.
- For each $(\textsc{Start} \lll w) \in G$, there is some $W \in \boldsymbol{W}$ such that $W = [w] \,||\, W'$.
- For each $(w \lll \textsc{End}) \in G$, there is some $W \in \boldsymbol{W}$ such that $W = W' \,||\, [w]$.

As in the previous proofs, this proof relies upon a bisimulation preservation lemma:

Lemma 5.18. *If* $\boldsymbol{W} \cong G$ *then*

- *If* $\boldsymbol{W} \longrightarrow^1 \boldsymbol{W}'$, *then either* $\boldsymbol{W}' \cong G$ *or* $G \longrightarrow^1 G'$ *such that* $\boldsymbol{W}' \cong G'$.
- *If* $G \longrightarrow^1 G'$ *then* $\boldsymbol{W} \longrightarrow^* \boldsymbol{W}'$ *such that* $\boldsymbol{W}' \cong G'$.

Much like Lemma 5.10, the inductive case between the stackless and lazy systems, this lemma is asymmetric because ωDDPAc takes larger steps than the historical system. In the case of a terminating recursive function call, for instance, the historical system may need to take many steps to catch up to a single wiring in ωDDPAc. Nonetheless, this number of steps is always finite. In cases of divergence, the historical system never catches up but the graph is unable to grow any further.

5.5 Overall Equivalence

The previous equivalences can be combined to produce the desired overall equivalence: the closure-based λ-calculus of Section 5.1 is equivalent to ωDDPAc. For notational convenience, we write $\downarrow$ to indicate a sequence of $\longrightarrow^1$ (in all overloadings) which cannot make further progress; for instance, $G \downarrow G'$ iff $G \longrightarrow^* G'$ and no $G'' \neq G'$ exists such that $G' \longrightarrow^1 G''$. We then phrase equivalence as follows:

Theorem 5.19 (Equivalence of Operational Semantics). *For any e, $[\langle [], e\rangle] \downarrow \phi$ if and only if $\{\textsc{Start} \ll e \ll \textsc{End}\} \downarrow G$ such that $\textsc{Active}(G, \textsc{End}) \neq \emptyset$.*

Proof. By composition of the equivalence arguments in Sections 5.2.1, 5.3.1, and 5.4.1. □

This equivalence is key to our soundness proof, which we present in the next section. Before proceeding, however, we take a moment to consider some unusual features of ωDDPAc.

Reflecting on ωDDPAc. Every operational semantics for the λ-calculus we are aware of relies upon one or more of the following mechanisms:

- Substitution (as in classic presentations of λ-calculus)
- An environment and closures (as in the CEK machine [15])
- Variable freshening (as in our previous work [36])

None of these three mechanisms appear in the definition of ωDDPAc, yet Theorem 5.19 shows it is a full and faithful implementation of CBV λ-calculus. This shows that the DDPA analysis, which is clearly very close in spirit to ωDDPAc, emerges from a fundamentally different operational basis. Optimal λ-reduction [27] is perhaps the closest to ωDDPAc of the existing λ-semantics; it is a substitution-based model but maximally shares syntax subtrees in a graph structure. The sharing graphs in that semantics grow unboundedly unlike the ωDDPAc graph.

Section 5.4 pointed out that the graph constructed by ωDDPAc is finite and monotonically increasing. From this, we can demonstrate a polynomial bound on ωDDPAc execution steps:

Lemma 5.20. *For a program of size n, there are $O(n^2)$ non-trivial evaluation steps in ωDDPAc. That is, for any $G = \{\textsc{Start} \ll e \ll \textsc{End}\}$, $G \downarrow G'$ in $O(n^2)$ steps for some G' (making no assertions regarding $\textsc{Active}(G, \textsc{End})$).*

Proof. All edges in G' are between either clauses (of which there are $O(n)$) or wiring nodes. Wiring nodes are bounded by $O(n^2)$: there are three elements of variation in each (call site, left variable, and right variable) and, in both cases, one variable is fixed by the call site. As there are $O(n)$ call sites and $O(n)$ variables in the program, there are at most $O(n^2)$ wiring nodes in G'. □

This lemma may at first appear to be a blatant contradiction: we have a full and faithful interpreter for the λ-calculus, but it is tightly bounded in the number of steps it can take! On further inspection, we have only shifted the work – the global state space is small, but searching through all the possible contexts is a potentially unbounded search.

To understand this it may also be helpful to informally consider an alternative presentation of ωDDPAc where the global state is not just G but also includes the current stack context and so is $\langle G, C\rangle$: here there is no need to search for a viable stack C since it has been explicitly recorded, but the stack can grow arbitrarily and so the overall state space is now unbounded, unlike the case of ωDDPAc.

In summary, any program evaluated using the ωDDPAc operational semantics requires at most $O(n^2)$ steps, but the operational semantics relation must be undecidable by Turing-completeness of the ωDDPAc interpreter, and even the single step relation of ωDDPAc has to be undecidable.

Lemma 5.21. (1) $\{\textsc{Start} \ll e \ll \textsc{End}\} \downarrow G$ *is undecidable.*

(2) *the* ω*DDPAc lookup relation* $G(X, a_0, C)$ *of Definition* 5.12 *is undecidable, and thus* $G \longrightarrow^1 G'$ *is undecidable.*

Proof. For (1), this follows directly from Theorem 5.19 and the Turing completeness and thus undecidability of the call-by-value λ-calculus computation relation that computes to a value. For (2), suppose lookup was decidable. Since there are only finitely many states G possible for any program by Lemma 5.20 there are also finitely many non-repeating sequences $G_1, \ldots, G_n$. If lookup were decidable it would be possible to enumerate all these sequences and for each one verify if each of the steps in a given sequence were legal using lookup and the single-step rule of Figure 26 (which would itself trivially be decidable if lookup is decidable). But, then it would also be possible to decide (1) as we could consider all sequences with $G_1 = \{\textsc{Start} \ll e \ll \textsc{End}\}$ and $G_n = G$ and check if any of them constituted a valid n-step computation, a contradiction. □

The following section demonstrates how DDPA soundly approximates this operational semantics.

6 SOUNDNESS

We now show the soundness of the analysis with respect to ωDDPAc and therefore, by Theorem 5.19, with respect to the lambda calculus in Section 5.1. Comparing DDPA with ωDDPAc, there are two key differences as outlined above:

- Call stacks C in ωDDPAc correspond to context stacks $\hat{C}$ in DDPA.
- The $\textsc{Active}$ function returns a set of possible call stacks while the $\widehat{\textsc{Active?}}$ function is a predicate.

In brief, the soundness proof proceeds by showing DDPA is abstract interpreter for ωDDPAc. The abstraction function α is lossless on all components of the language grammar, mapping each term to its hatted counterpart (e.g. x maps to $\hat{x}$, f maps to $\hat{f}$, etc.). Call stacks $[c_1, \ldots, c_n]$ are mapped to context stacks $\textsc{Push}(\alpha(c_n), \ldots \textsc{Push}(\alpha(c_1), \epsilon))$ and $\widehat{\textsc{Active?}}$ holds when $\textsc{Active}$ returns a non-empty set. No other information is lost. This outline is now expanded.

6.1 DDPAc

Our first step is to define an analysis DDPAc and then to weaken that analysis to DDPA (in accordance with Figure 17). The DDPAc analysis is a midpoint between ωDDPAc and DDPA; in fact, DDPAc can be seen as a generalization of the ωDDPAc defined in Section 5.4. DDPAc uses the context stack models and lookup function from Section 4.1 but relies upon a function $\widehat{\textsc{Active}}$ (the abstract version of Definition 5.13) to produce a set of valid contexts for each lookup. For each Σ, we define that abstract function as follows:

Definition 6.1. Let $\widehat{\textsc{Active}}(\hat{G}, \hat{a})$ be least set $\hat{C}$ conforming to the following conditions:

- If $\hat{c} \mathrel{<\!\!\lhd} \hat{a}$ then $\widehat{\textsc{Active}}(\hat{G}, \hat{c}) \subseteq \hat{C}$.
- If $\hat{a}' \mathrel{<\!\!\lhd} \hat{a}$ for $\hat{a}' = (\hat{x} \overset{(\!|\hat{c}}{=} \hat{x}')$ and $\hat{C} \in \widehat{\textsc{Active}}(\hat{G}, \hat{a}')$, then $\textsc{Push}(\hat{c}, \hat{C}) \in \hat{C}$.
- If $\textsc{Start} \mathrel{<\!\!\lhd} a$ then $\epsilon \in \hat{C}$.

Note that $\hat{C}$ may not be finite.

In the interest of showing DDPAc to be an analysis in its own right, we observe that $\widehat{\textsc{Active}}$ can be computed for finite context stack models Σ. One simple algorithm is to store a set of context stacks for each node in $\hat{G}$. Initially, $\textsc{Start}$ has the only non-empty set (containing ϵ); the result of $\widehat{\textsc{Active}}$ is calculated by propagating the context stacks throughout the graph to saturation.

DDPAc can re-use most other definitions from DDPA including lookup. Only the ωDDPAc-inspired evaluation rule needs to be incorporated to complete the definition.

$$\textsc{Application}\quad \frac{\hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3) \qquad \hat{C} \in \widehat{\textsc{Active}}(\hat{c}, \hat{G}) \qquad \hat{f} \in \hat{G}([\hat{x}_2], \hat{c}, \hat{C}) \qquad \hat{v} \in \hat{G}([\hat{x}_3], \hat{c}, \hat{C})}{\hat{G} \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \hat{G} \cup \widehat{\textsc{Wire}}(\hat{c}, \hat{f}, \hat{x}_3, \hat{x}_1)}$$

Fig. 29. DDPAc Abstract Evaluation Rule

Definition 6.2. For a particular Σ, we define $\hat{G} \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \hat{G}'$ to hold if a proof exists in the system of Figure 29. We write $\hat{G}_0 \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^* \hat{G}_n$ iff $\hat{G}_0 \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \ldots \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \hat{G}_n$. We write kDDPAc to refer to DDPAc using context stack model Σ_k.

If we choose Σ to be Σ_ω, then this system is identical to ωDDPAc. In that case, ϵ corresponds to the concrete call stack []; this is not a guarantee of call stacks (since $\textsc{Pop}(\epsilon) = \epsilon$ but this is not true for []), but it holds in DDPAc due to the definition of $\widehat{\textsc{Active}}$.

Soundness of DDPAc can be demonstrated by abstract interpretation. The abstraction function α here is identity except in the case of call stacks and context stacks: each call stack $[c_1, \ldots, c_n]$ is mapped to the context stack $\textsc{Push}(\alpha(c_n)), \ldots \textsc{Push}(\alpha(c_1), \epsilon))$. We first establish soundness of lookup:

Lemma 6.3. *Let $\hat{G} \supseteq \alpha(G)$ for any G. Let $V = G([x] \,||\, X, a, C)$ and let $\hat{V} = \hat{G}([\hat{x}] \,||\, \hat{X}, \hat{a}, \hat{C})$; then $\alpha(V) \subseteq \hat{V}$.*

Proof. By induction on the size of the proof of $G([x] \,||\, X, a, C)$ and case analysis on the rule used at each proof step. □

We then state soundness of DDPAc as follows:

Lemma 6.4. *Let $\hat{G} \supseteq \alpha(G)$ for any G. Then $G \longrightarrow^1 G'$ implies $\hat{G} \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \hat{G}'$ such that $\alpha(G) \subseteq \hat{G}$.*

Proof. By matching each premise of the concrete Application rule to its abstract counterpart. In particular, lookup is monotonic in the size of the graph. □

6.2 DDPA

Finally, we demonstrate that DDPAc is conservatively approximated by DDPA from Section 4. The only difference between these two analyses is in the handling of active nodes: DDPAc calculates possible contexts using $\widehat{\textsc{Active}}$ whereas DDPA simply determines whether any contexts exist using $\widehat{\textsc{Active?}}$. This weakening is motivated by performance, as computing all possible contexts for a given program point is generally expensive. While DDPAc is more precise than DDPA in this respect, this increased precision is rarely necessary.

We show DDPA approximates DDPAc by observing the differences in the lookups of Figures 16 and 29. In DDPA, the context provided to lookup is *always* ϵ; no other context stacks are provided to lookup. However, Definition 4.4 requires that $\textsc{Pop}(\epsilon) = \epsilon$. With respect to lookup, this imposes a form of subsumption on finite call stack models with ϵ at the top of the lattice. Formally, we state this property as follows:

Lemma 6.5. *For all $\hat{C}$, $\hat{G}([\hat{x}] \,||\, \hat{X}, \hat{a}, \hat{C}) \subseteq \hat{G}([\hat{x}] \,||\, \hat{X}, \hat{a}, \epsilon)$.*

Proof. By induction on the proof of $\hat{G}([\hat{x}] \,||\, \hat{X}, \hat{a}, \hat{C})$ and Definition 4.4. □

As a consequence, each lookup and CFG construction step in DDPAc is approximated by DDPA. We formalize soundness between these systems as follows:

Lemma 6.6. *For any $\hat{G}_1 \subseteq \hat{G}_1'$, if $\hat{G}_1 \mathrel{\widehat{\xrightarrow[\mathrm{c}]{}}}^1 \hat{G}_2$ then $\hat{G}_1' \mathrel{\widehat{\longrightarrow}}^1 \hat{G}_2'$ such that $\hat{G}_1' \subseteq \hat{G}_2'$.*

$$\hat{K} \quad ::= \quad [\hat{k}, \ldots] \quad \textit{continuation stacks} \qquad \hat{k} \quad ::= \quad \hat{x} \mid \hat{v} \mid \textsc{Jump}(\hat{a}, \hat{C}) \mid \textsc{Capture}(2) \quad \textit{continuations}$$

Fig. 30. Alternate Lookup Grammar

Proof. By Lemma 6.5 and matching each premise of the rule in Figure 29 to its counterpart in Figure 16. □

We now can assert the soundness of DDPA as follows:

Theorem 6.7 (Soundness). *For any e, x, and a, if* $\{\textsc{Start} \ll e \ll \textsc{End}\} \longrightarrow^* G, C \in \textsc{Active}(G, a)$, *and* $G([x], a, C) = V$, *then* $\alpha(\{\textsc{Start} \ll e \ll \textsc{End}\}) \longrightarrow^* \hat{G}, \widehat{\textsc{Active?}}(\hat{G}, \alpha(a))$, *and* $\alpha(V) \subseteq \hat{G}([\alpha(x)], \alpha(a), \epsilon)$.

Proof. By Lemmas 6.1 and 6.6 and the monotonicity of Definition 4.6. □

7 DECIDABILITY

We now show the analysis defined in Section 4 is decidable. We in fact show a stronger result: for a kCFA-like stack model (See Definition 4.4) which retains the fixed k most recent call sites, we show that the control flow graph can be constructed in polynomial time.

Much of the decidability argument is immediately evident: functions like $\widehat{\textsc{Active?}}$ are computable by inspection and, for any particular program and finite Σ, CFG size can be bounded by simple counting arguments. The core of the proof is showing the decidability of variable lookup, Definition 4.6. So, first we will first show lookup is decidable. We then formally prove the above decidability property for DDPA.

7.1 Decidable Lookup

In Section 3 we informally outlined how we can implement lookup in terms of a pushdown reachability question. Recall from that discussion that a state is created in the PDS for each (program point, variable lookup) question, and additionally including the (approximate) context stack in the state. PDS transitions then represent recursive calls to lookup. Inspecting Definition 4.6, as written it is in fact not directly implementable as a pushdown reachability problem. The particular catch is clause 6, which invokes lookup *twice*; if this was modeled as two edges in the PDS it would mean *either* path can succeed, but clause 6 requires a conjunction of two lookups and that is not directly encodable.

The solution for two invocations of lookup is not all that difficult: the PDS edge transitions for the first lookup action, and a frame is pushed on the PDS stack to trigger the second invocation. In order to formalize this idea, we first develop an equivalent version of lookup which replaces clause 6 with only one call to lookup and pushes the remaining one on the stack; various other clauses are also needed to implement continuations. We then demonstrate the two lookup functions are equivalent. Since the second definition of lookup is directly implementable on a PDS this shows lookup is computable, since pushdown reachability is computable in time polynomial in the size of the PDS [6].

Our alternate definition of lookup requires generalizing the lookup stack to a *continuation stack* to keep track of other lookups that must eventually be performed, and for how to combine the results of multiple lookups; the stack grammar appears in Figure 30. This continuation stack will directly map to the stack of the pushdown system. We present the definition of the lookup function and discuss the role of these continuations below.

Definition 7.1. Given control flow graph $\hat{G}$, $\hat{G}\llparenthesis \hat{K}, \hat{a}_0, \hat{C} \rrparenthesis$ is the function returning the least set of values $\hat{V}$ satisfying the following conditions given some $\hat{a}_1 \lessdot \hat{a}_0$:

(1) If $\hat{a}_1 = (\hat{x} = \hat{v})$ and $\hat{K} = [\hat{x}]$ then $\hat{v} \in \hat{V}$.
(2) If $\hat{a}_1 = (\hat{x} = \hat{f})$ and $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$ for $\hat{K}' \neq []$ then $\hat{G}\llparenthesis \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
(3) If $\hat{a}_1 = (\hat{x} = \hat{x}')$ and $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$ then $\hat{G}\llparenthesis [\hat{x}'] \mathbin{||} \hat{X}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
(4) If $\hat{a}_1 = (\hat{x} \overset{◖\hat{c}}{=} \hat{x}')$, $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$, and $\textsc{MaybeTop}(\hat{c}, \hat{C})$, then $\hat{G}\llparenthesis [\hat{x}'] \mathbin{||} \hat{X}', \hat{a}_1, \textsc{Pop}(\hat{C}) \rrparenthesis \subseteq \hat{V}$.
(5) If $\hat{a}_1 = (\hat{x}'' \overset{◖\hat{c}}{=} \hat{x}')$, $\hat{x}'' \neq \hat{x}$, $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$, $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, and $\textsc{MaybeTop}(\hat{c}, \hat{C})$, then $\hat{G}\llparenthesis [\hat{x}_f] \mathbin{||} \hat{K}, \hat{a}_1, \textsc{Pop}(\hat{C}) \rrparenthesis \subseteq \hat{V}$.
(6a) If $\hat{a}_1 = (\hat{x} \overset{◗\hat{c}}{=} \hat{x}')$, $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, and $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$, then $\hat{G}\llparenthesis [\hat{x}_f, \textsc{Capture}(2), \textsc{Jump}(\hat{a}_0, \hat{C}), \textsc{RealFlow?}] \mathbin{||} \hat{K}, \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
(6b) If $\hat{a}_1 = (\hat{x} \overset{◗\hat{c}}{=} \hat{x}')$ and $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$, $\hat{K} = [\textsc{RealFlow?}, \texttt{fun}\ \hat{x}'' \texttt{ -> (} \hat{e} \texttt{) }, \hat{x}] \mathbin{||} \hat{K}'$, $\hat{x}' = \mathrm{RV}(\hat{e})$, then $\hat{G}\llparenthesis [\hat{x}'] \mathbin{||} \hat{K}', \hat{a}_1, \textsc{Push}(\hat{c}, \hat{C}) \rrparenthesis \subseteq \hat{V}$.
(7) If $\hat{a}_1 = (\hat{x}'' = b)$, $\hat{x}'' \neq \hat{x}$, and $\hat{K} = [\hat{x}] \mathbin{||} \hat{K}'$, then $\hat{G}\llparenthesis \hat{K}, \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
(8) If $\hat{K} = [\textsc{Jump}(\hat{a}', \hat{C}')] \mathbin{||} \hat{K}'$ then $\hat{G}\llparenthesis \hat{K}', \hat{a}', \hat{C}' \rrparenthesis \subseteq \hat{V}$.
(9) If $\hat{K} = [\hat{v}, \textsc{Capture}(2), \hat{k}_1, \hat{k}_2] \mathbin{||} \hat{K}'$, then $\hat{G}\llparenthesis [\hat{k}_1, \hat{k}_2, \hat{v}] \mathbin{||} \hat{K}', \hat{a}_0, \hat{C} \rrparenthesis \subseteq \hat{V}$.

This definition is very similar to the original lookup of Definition 4.6, but one key difference is critical for decidability: clause 6 of the original Definition 4.6 is here divided into clauses 6a and 6b. This is to solve the problem alluded to above of the two invocations of lookup in that clause: each clause must invoke lookup only once to be encodable as a PDS, so the two lookups in that clause are now expressed as two different clauses and with continuation frames added on the stack to connect them. In this alternative formulation, lookup of a value that was returned from a function call proceeds as follows:

(1) Clause 6a reacts to a function exit wiring node by triggering the addition of $[\hat{x}_f, \textsc{Capture}(2), \textsc{Jump}(\hat{a}_0, \hat{C}), \textsc{RealFlow?}]$ to the stack. (While this might look like jibberish now, let us proceed with how the lookup function works and the purpose of these new "continuation frames" will be made clear.)
(2) Lookup proceeds to find a value for $\hat{x}_f$ since that is now the top frame on the stack. The top of the stack is then $[\hat{v}, \textsc{Capture}(2), \textsc{Jump}(\hat{a}_0, \hat{C}), \textsc{RealFlow?}]$ at the $\hat{a}$ and $\hat{C}$ where the value was found.
(3) Clause 9 activates to push $\hat{v}$ deeper into the stack, turning the top of the stack into $[\textsc{Jump}(\hat{a}_0, \hat{C}), \textsc{RealFlow?}, \hat{v}]$.
(4) Clause 8 activates, "jumping" to the same program point and calling context that started this lookup. The stack after the jump is performed is $[\textsc{RealFlow?}, \hat{v}]$.
(5) Now that we are back at CFG node $\hat{a}_0$ where we started, Clause 6b applies and continues lookup, entering the discovered wiring node only when the discovered function's return variable matches that wiring node.

Lemma 7.2. *Given fixed CFG $\hat{G}$, for all $\hat{X}$, $\hat{a}$, and $\hat{C}$, $\hat{G}(\hat{X}, \hat{a}, \hat{C}) = \hat{G}\llparenthesis \hat{X}, \hat{a}, \hat{C} \rrparenthesis$.*

Proof. By induction on the size of the proof of $\hat{G}(\hat{X}, \hat{a}, \hat{C})$ and then by case analysis on the clause used from Definition 4.6. □

7.1.1 Proving Lookup is Decidable via PDS Reachability. Now that we have removed multiple invocations of lookup from the original clause 6, each lookup clause examines a finite prefix of $\hat{K}$, applies some number of computable restrictions (such as limiting the form of $\hat{a}_1$ or checking the predicate $\textsc{MaybeTop}$), and then specifies a lower bound on $\hat{V}$. Given this normal form, it is possible to directly encode lookup as a PDS reachability problem. We let each pair of program point $\hat{a}$ in $\hat{G}$ and context $\hat{C}$ in $\hat{\mathbb{C}}$ define a state in the PDS. For every $\hat{a}_1 \lll \hat{a}_0$, each clause of Definition 7.1

dictates a set of transitions in the PDS. For example: for any $\hat{C}$, clause 3 transitions from $\langle \hat{a}_0, \hat{C} \rangle$ to $\langle \hat{a}_1, \hat{C} \rangle$, popping $\hat{x}$ and pushing $\hat{x}'$. Accepting states are dictated by clause 1. Given this encoding, each value $\hat{v}$ in $\hat{G}(\!|\hat{K}, \hat{a}, \hat{C}|\!)$ corresponds to a node $\hat{a}' = (\hat{x} = \hat{v})$ in the set of nodes reachable in the PDS from initial state $\langle \hat{a}, \hat{C} \rangle$ with initial stack $\hat{K}$. This leads us to the following lemma:

Lemma 7.3. *For any context stack model Σ with a finite $\hat{\mathbf{C}}$ and computable* Push/Pop/MaybeTop *operations, $\hat{G}([\hat{x}_0], \hat{a}, \epsilon)$ is computable in time polynomial in the product of the number of nodes in $\hat{G}$ and the size of $\hat{\mathbf{C}}$.*

Proof. By reduction to the problem of reachability in a pushdown system accepting by empty stack. Pushdown reachability is computable in time polynomial in the size of the automaton [6, 11], so it suffices to bound the number of states and transitions by the product of the sizes of $\hat{G}$ and $\hat{\mathbf{C}}$. States are bounded by this product by definition. Transitions are bounded by this product because the grammar of stack elements $\hat{k}$ is bounded by this product and each clause of Definition 7.1 pushes and pops a constant number of stack elements. □

The proof strategy in the rest of this section can generally be applied to any finite context stack model, but we primarily concern ourselves with the kDDPA analyses described in Section 4.1.1. For that reason, it is helpful to simplify the statement of the lookup decidability lemma in those cases:

Lemma 7.4. *Fixing Σ to some Σ_k for fixed constant k, $\hat{G}([\hat{x}], \hat{a}_0, \epsilon)$ is computable in polynomial time in the number of nodes in graph $\hat{G}$.*

Proof. Let n be the number of nodes in graph $\hat{G}$. For Σ_k, the number of stacks is of order $O(n^k)$ where k is constant. By Lemma 7.3, $\hat{G}([\hat{x}], \hat{a}_0, \epsilon)$ is computable in time polynomial in $O(n^{k+1})$, which is polynomial in n. □

Note that if k was not fixed and was in fact increasing with the size of the program, lookup would become exponential.

7.2 Proof of Decidability

Having shown lookup to be polynomial, it is now possible to show a similar result for the overall analysis.

Lemma 7.5. *Variable lookup is monotonic; that is, for any $\hat{x}$ and $\hat{a}$, if $\hat{G}_1 \subseteq \hat{G}_2$ then $\hat{G}_1([\hat{x}], \hat{a}, \epsilon) \subseteq \hat{G}_2([\hat{x}], \hat{a}, \epsilon)$.*

Proof. Variable lookup is encodable as a PDS reachability problem (see Lemma 7.3) and the PDS grows monotonically with the graph $\hat{G}$. PDS reachability grows monotonically with the PDS. Therefore, the set of results from variable lookup grows monotonically with the graph $\hat{G}$. □

Lemma 7.6. *The evaluation relation $\widehat{\longrightarrow}^*$ is confluent.*

Proof. By inspection of Figure 16, single-step evaluation only adds to graph $\hat{G}$. The $\widehat{\text{Active?}}$ relation is also clearly monotone: any enabled redex is never disabled. Confluence is trivial from these two facts. □

Lemma 7.7. *The evaluation relation $\widehat{\longrightarrow}^*$ is terminating,* i.e. *for any $\hat{G}_0$ there exists a $\hat{G}_n$ such that $\hat{G}_0 \;\widehat{\longrightarrow}^*\; \hat{G}_n$ and if $\hat{G}_n \;\widehat{\longrightarrow}^*\; \hat{G}_{n+1}$, $\hat{G}_n = \hat{G}_{n+1}$. Furthermore, n is polynomial in the size of the initial program.*

Proof. By inspection of Figure 16, we have for any step $\hat{G}' \overset{\frown}{\longrightarrow}^1 \hat{G}''$ that $\hat{G}' \subseteq \hat{G}''$. The only new nodes that can be added to $\hat{G}$ in the course of evaluation are the entry/exit nodes $\hat{x}' \overset{(\hat{c}}{=} \hat{x}$ / $\hat{x} \overset{)\hat{c}}{=} \hat{x}'$, and only one of each of those nodes can exist for each call site / function body pair in the source program: $\hat{c}$ is the call site, and $\hat{x}$ / $\hat{x}'$ are variables in that call site and function body source, respectively. So, the number of nodes that can be added is always less than two times the square of the size of the original program. A similar argument holds for added edges. □

We let $\hat{G} \downarrow \hat{G}'$ abbreviate $\hat{G} \overset{\frown}{\longrightarrow}^* \hat{G}'$ such that $\hat{G}' \not\overset{\frown}{\longrightarrow}^1 \hat{G}'$. We write $e \downarrow \hat{G}$ to abbreviate $\widehat{\textsc{Embed}}(e) \downarrow \hat{G}$; this means the analysis of e returns graph $\hat{G}$. Given the pieces assembled above, it is now easy to prove that the analysis is polynomial-time.

Theorem 7.8. *Fixing Σ to be some Σ_k and fixing some expression e, the analysis result $\hat{G}$, where $e \downarrow \hat{G}$, is computable in time polynomial in the size of e.*

Proof. By Lemma 7.4, each lookup operation takes poly-time. The evaluation rules are trivial computations besides the required lookups and, by Lemma 7.7, there are polynomially many evaluation steps before termination. Thus $e \downarrow \hat{G}$ is computable in poly-time. □

8 IMPLEMENTING LOOKUP

We showed in the previous section how lookup may be encoded as a PDS reachability question. Although lookup on the PDS described in Section 7.1.1 is polynomial time (Lemma 7.4), the PDS is quite large and its naive construction is slow in practice.

In this section, we formally describe *pushdown reachability automata* (PDR's), which can be viewed as syntax for schematically defining collections of PDS transitions with a single piece of syntax, collapsing the state-space blowup alluded to above. We do not prove theoretical bounds of the PDR – the worst case time complexity of reachability remains the same – but the evaluation of this approach in Section 10 demonstrates that a DDPA implementation using this approach is comparable to other recent higher-order program analyses.

8.1 Pushdown Reachability

The standard algorithm for solving pushdown reachability is to perform an edge closure [6]: for each pair of adjacent matching push and pop edges, add a single transitive no-op edge. For any path between two states in the original automaton, closure will ensure a path between those same states which consists solely of no-op edges. Pushdown reachability is decidable in polynomial time [6] but, if the number of states and edges is large, it still may be slow in practice.

To illustrate the size of the PDS described in Section 7.1.1, let us consider rule 7 of the alternative lookup function (Definition 7.1). This rule dictates that the PDS should contain an edge for each pairing between a variable ($\hat{x}$) and an alias clause not defining that variable ($\hat{x}'' = \hat{b}$ such that $\hat{x}'' \neq \hat{x}$); that is, this one rule increases the size of the PDS at least *quadratically* in the size of the program and reachability is a polynomial of that size. In practice, most of those edges will not be used: not every variable is looked up from every position in the CFG, if only for reasons of scope.

Previous work [24] describes an efficient algorithm for reachability on pushdown systems which have a large number of states. This algorithm is also applied in the domain of program analysis. The algorithm expands the automaton lazily, adding states (with their outgoing edges) as necessary to solve the reachability query; this avoids the addition of states which are not involved in computing reachability.

The PDS of Section 7.1.1 not only has a large number of states – one for each program point $\hat{a}$ in each context $\hat{C}$ – but, as illustrated above, a large number of edges at each state. Our algorithm extends the work of PDCFA [24] by representing collections of edges *schematically* and adding them to the automaton on demand.

8.1.1 Formalizing Pushdown Systems. Our objective is to construct an automaton to facilitate a reachability algorithm which produces results equivalent to that of naive pushdown reachability. For this reason, it is useful to formalize our notion of a pushdown system. We begin by defining common notation:

Definition 8.1. Given a finite set of stack symbols Γ, we use $\Gamma^{\updownarrow}$ to denote the set $\{\epsilon\} \cup \{\gamma^{\downarrow} \mid \gamma \in \Gamma\} \cup \{\gamma^{\uparrow} \mid \gamma \in \Gamma\}$ of *actions* (no-ops, pushes, and pops) over Γ. We use $\gamma^{\updownarrow}$ to denote individual elements of $\Gamma^{\updownarrow}$. We denote the set of lists of these actions as $\Gamma^{\updownarrow}_{*}$.

Given the above, we define a pushdown system as follows:

Definition 8.2. For finite sets of states Q and stack symbols Γ, a *pushdown system* (PDS) is a triple $\langle Q, \Gamma, \delta \rangle$ where transition relation δ is a finite subset of $Q \times \Gamma^{\updownarrow} \times Q$.

Note that this definition of a PDS has no marked start or finish states. We select a start state based upon the reachability question. As mentioned in the proof of Lemma 7.3, the finish states are those at which the PDS stack is empty.

It is also important to note that Definition 8.2 describes a "single action" pushdown system: each transition may *either* push, pop, or do nothing. Other PDS definitions, such as the canonical "single pop, multi push" formulation (in which each transition must pop one stack element and may then push any fixed string of stack elements), may be encoded in "single action" form by adding intermediate states.

8.2 Pushdown Reachability Automata

We now define PDRs. We will give a general PDR algebraic signature, and will also point out how that signature is instantiated for our analysis. We begin with the domain, which is fixed throughout closure.

Definition 8.3. A *pushdown reachability domain* is a finite 4-tuple $\langle S, Q, \Gamma, \Psi \rangle$ where S is a set of pre-states, Γ are stack elements, Ψ are atomic dynamic pop actions, and Q is a fixed subset of $S \times \Gamma^{\updownarrow}_{*}$. We denote elements of Q as exponents, as in $s^{[\gamma^{\updownarrow},\ldots]}$.

This defines the states and stack elements of a PDR. Pre-states S are just the core state information. In DDPA, they are defined as contextualized states: pairs of $\hat{a}$ and $\hat{C}$ which represent the small round nodes in Figure 11. Full PDR states Q include a pre-state element in S as well as an *action stack* from $\Gamma^{\updownarrow}_{*}$. Intuitively, the PDR state $s^{[\gamma^{\updownarrow},\ldots]}$ expresses "I will be in state s once I complete the actions $[\gamma^{\updownarrow},\ldots]$." These are the small red intermediate states in Figure 14. Lastly, the PDR domain includes a finite set of dynamic pop actions Ψ to label actions having schematic target states; the purpose of dynamic pop actions will be clarified below.

Next, we define the general transition function signature for a PDR.

Definition 8.4. Given a PDR domain $\langle S, Q, \Gamma, \Psi \rangle$, a *pushdown reachability transition specification* is a 3-tuple $\langle t, u, p \rangle$ of computable functions:

- $t : Q \to \mathcal{P}(\Gamma^{\updownarrow} \times Q)$, the *targeted* transitions
- $u : Q \to \mathcal{P}(\Psi)$, the *untargeted* transitions
- $p : \Gamma \times \Psi \to \mathcal{P}(Q \cup \Psi)$, the states and/or additional dynamic pops reached by a dynamic pop

Targeted transitions t have a concrete state target whereas untargeted transitions u have a schematic target which is made concrete by p. We show how Definition 7.1 is given a PDR transition specification in Section 8.4 below. Together, the PDR domain and PDR transition specification define the structure of a PDR separate from a particular reachability question.

8.3 Pushdown Reachability Closure

We now give the algorithm for the PDR reachability closure process. The objective of this process is to yield the same results as the closure of a corresponding pushdown system. Unlike PDS closure, which starts with all nodes and edges present, in PDR closure the states and transitions are lazily added based on the transition specification. This lazily constructed automaton we call the *PDR graph*:

Definition 8.5. Given a PDR domain $\langle S, Q, \Gamma, \Psi \rangle$, a *pushdown reachability graph* is a 3-tuple $\langle Q^{\mathrm{C}}, \delta, \eta \rangle$ where

- $Q^{\mathrm{C}} \subseteq Q$
- δ is a set of static transitions, a finite subset of $Q \times \Gamma^{\updownarrow} \times Q$
- η is a set of dynamic pop transitions, a finite subset of $Q \times \Psi$

In this definition, the set of lazily constructed current states Q^{C} are the states in Q that are reachable from the starting point of a query and should be explored. The set of stack action transitions δ have the same meaning as in a PDS. Additionally, a PDR graph has a set of dynamic pop transitions η; the only transitions in this set are those attached to states that have already appeared in Q^{C}. Recall from Section 3 that our goal is to determine reachability on schematic pushdown automata; the dynamic pop transitions η act as transition *generators* defined by the schema and we use them to add to the graph only those transitions which may affect the result of a particular lookup question.

With this data structure we now begin defining the closure process for computing reachability. For motivation, consider a PDR graph construction for our analysis: the lookup of a variable x from a program point p in the empty (i.e. unknown) context $[]$. To compute this closure, it is sufficient to close over a PDR based upon the initial PDR graph $\langle \{ s^{[\mathsf{x}^{\downarrow}]}) \}, \varnothing, \varnothing \rangle$ for $s = \langle \mathsf{p}, [] \rangle$. This graph contains a single state $s^{[\mathsf{x}^{\downarrow}]}$ (read: "I will be in state s once I push x onto the lookup stack") and no transitions. The closure process will perform this push and then apply the PDR transition spec to the resulting states, gradually expanding the graph to discover all states reachable from this starting location.

We formally define PDR closure as follows:

Definition 8.6. For fixed PDR domain $\langle S, Q, \Gamma, \Psi \rangle$ and PDR transition spec $\langle t, u, p \rangle$, we define $\Longrightarrow$ as the least relation between PDR graphs which obeys the rules in Figure 31.

The first three rules in Figure 31 are formal presentations of the basic PDS closure algorithm informally described in Section 8.2: when pushes can reach pops, both are canceled and a no-op results. The Push+Dynamic Pop to State rule performs closure over the dynamic pops as described in Section 3.2: as new push elements reach dynamic pops, the appropriate specification function is invoked to determine the resulting destination. Note that this destination is an element of Q with a list of pending stack actions; the Pending Action rule ensures that such states are expanded to eventually reach their destinations. The Push+Dynamic Pop to Dynamic Pop rule performs a similar dynamic closure: the result is another dynamic pop, which is attached to the source of the push in a form of continuation processing discussed in Section 8.4 below. Finally, the Transition Expansion rule ensures that we add appropriate transitions for each state; although our graph initially contains no transitions at all, we draw transitions from functions t and u to add to the graph as closure proceeds. Note that the other rules add states reachable via pushes and no-ops to the Q^{C} set, ensuring that transitions from those states will be explored.

Push+Pop

$$\frac{(q_1, \gamma^{\downarrow}, q_2) \in \delta \qquad (q_2, \gamma^{\uparrow}, q_3) \in \delta}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}} \cup \{q_3\}, \delta \cup \{(q_1, \epsilon, q_3)\}, \eta \rangle}$$

Push+Nop

$$\frac{(q_1, \gamma^{\downarrow}, q_2) \in \delta \qquad (q_2, \epsilon, q_3) \in \delta}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}} \cup \{q_3\}, \delta \cup \{(q_1, \gamma^{\downarrow}, q_3)\}, \eta \rangle}$$

Nop+Nop

$$\frac{(q_1, \epsilon, q_2) \in \delta \qquad (q_2, \epsilon, q_3) \in \delta}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}} \cup \{q_3\}, \delta \cup \{(q_1, \epsilon, q_3)\}, \eta \rangle}$$

Push+Dynamic Pop to State

$$\frac{(q_1, \gamma^{\downarrow}, q_2) \in \delta \qquad (q_2, \psi) \in \eta \qquad q_3 \in p(\gamma, \psi)}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}} \cup \{q_3\}, \delta \cup \{(q_1, \epsilon, q_3)\}, \eta \rangle}$$

Push+Dynamic Pop to Dynamic Pop

$$\frac{(q_1, \gamma^{\downarrow}, q_2) \in \delta \qquad (q_2, \psi_1) \in \eta \qquad \psi_2 \in p(\gamma, \psi_1)}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}}, \delta \}, \eta \cup \{(q_1, \psi_2)\} \rangle}$$

Pending Action

$$\frac{q_1 = s^{[\gamma_1^{\updownarrow}, \gamma_2^{\updownarrow}, \ldots, \gamma_n^{\updownarrow}]} \qquad q_1 \in Q^{\mathrm{C}} \qquad q_2 = s^{[\gamma_2^{\updownarrow}, \ldots, \gamma_n^{\updownarrow}]}}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}} \cup \{q_2\}, \delta \cup \{(q_1, \gamma_1^{\updownarrow}, q_2)\}, \eta \rangle}$$

Transition Expansion

$$\frac{q \in Q^{\mathrm{C}}}{\langle Q^{\mathrm{C}}, \delta, \eta \rangle \Longrightarrow \langle Q^{\mathrm{C}}, \delta \cup t(q), \eta \cup u(q) \rangle}$$

Fig. 31. PDR Closure Rules

In our analysis implementation, a variety of common optimizations are applied to prevent duplicate work and realize these rules efficiently, including an algorithm similar to the "work-list" algorithm of PDCFA [24] (which prevents two edges from being closed with each other more than once). Throughout closure, this algorithm ensures that only states reachable from an initial query state via a series of push and no-op transitions are "alive", meaning that they are the source of transitions in the δ and η sets.

An important feature of DDPA is that the same PDR structure can be used throughout the analysis of a particular program. This is a consequence of the monotonicity property proven in Lemma 7.5: when performing a second variable lookup, one need only add an appropriate lookup state to Q^{C} and perform only the incremental new closure steps needed. This means that, if two lookup operations' proofs of Definition 7.1 share a derivation subtree, the closure work to establish that portion of the proof will only be performed once. In practice, this work sharing is utilized, for instance, whenever the analysis looks up two non-local variables from the same function; this has a significant effect, contributing to a $\tilde{1}$000x speedup relative to the proof-of-concept implementation that does not feature work sharing (see Section 10).

8.4 From Inductive Definition to PDR Specification

Finally, we show how Definition 7.1 is expressed as a PDR. For brevity, we focus on a select few cases of that definition. We will incrementally define the transition functions $\langle t, u, p \rangle$ by adding mappings to these transition relations as the discussion proceeds; we will similarly incrementally define the dynamic pop actions Ψ.

8.4.1 Using the Transition Functions. We begin by considering rule 1. This rule is invoked when we discover the definition of the variable we are currently seeking, which is based upon the edges contained within our CFG. This could be encoded in a PDS as follows: for each CFG edge with a source clause of the form $\hat{a}_1 = \hat{x} = \hat{b}$ and for each calling context $\hat{C}$, add an edge to pop $\hat{x}$ and then push $\hat{b}$, with the state $\langle \hat{a}_1, \hat{C} \rangle$ as the ultimate destination. Since there may be many possible stack contexts $\hat{C}$ there could be many possible edges added, illustrating the inefficiency of using a PDS directly.

Rather than enumerating these edges, we can define in our PDR a function $t_{\hat{a}_1 \ll \hat{a}_0}$ for each CFG edge $\hat{a}_1 \ll \hat{a}_0$ of this form:

$$t_{\hat{a}_1 \ll \hat{a}_0}(\langle \hat{a}_0, \hat{C} \rangle^{[]}) = \{\hat{x}^{\uparrow} \mapsto \langle \hat{a}_1, \hat{C} \rangle^{[\hat{b}^{\downarrow}]}\}$$

This is to say that, whenever we can reach state $\langle \hat{a}_0, \hat{C} \rangle$, we should add a pop of $\hat{x}$ from it to an intermediate state, which will then push $\hat{b}$ to reach state $\langle \hat{a}_1, \hat{C} \rangle$. This function will be called during PDR closure with each currently-reachable state, ensuring both that $\langle \hat{a}_1, \hat{C} \rangle$ is reachable for the lookup of $\hat{x}$ (correctness) and also that these edges are only added to the PDR if a lookup for $\hat{x}$ occurs with context stack $\hat{C}$ (efficiency). Rule 1 does not require dynamic transitions, so no additions to u or p are needed here.

8.4.2 Dynamic Operations. Some clauses base their behavior upon the contents of the stack. For instance, rule 7 addresses clauses which do not define the variable for which we are looking. To represent a clause $\hat{x}' = \hat{x}''$ in the PDS, we must be able to pop and push *any* variable *other* than $\hat{x}'$.

As stated above, encoding this rule in a simple PDS would require a pop-then-push transition for each variable in the program, but most of these transitions would never be used. Many unused transitions can be statically eliminated with simple reasoning; we can, for instance, omit transitions for out-of-scope variables. Static elimiation is limited, however, as some transitions remain unused during analysis simply because they are not necessary to solve the question at hand; that is, the transition may not be necessary to look up one variable but may be necessary for another. Static elimination is also insufficient when modeling the stack-based continuations we discuss below.

We instead represent groups of transitions succinctly using a dynamic pop action. We include elements of the form $\textsc{ClauseSkip}(s, \hat{x})$ in our set of dynamic pop actions Ψ. We then define a function $u_{\hat{a}_1 \ll \hat{a}_0}$ for CFG edges of the form $\hat{a}_1 = (\hat{x}' = \hat{b})$ as follows:

$$u_{\hat{a}_1 \ll \hat{a}_0}(q) = \begin{cases} \{\textsc{ClauseSkip}(\langle \hat{a}_1, \hat{C} \rangle, \hat{x}')\} & \text{when } q = \langle \hat{a}_0, \hat{C} \rangle^{[]} \\ \varnothing & \text{otherwise} \end{cases}$$

This ensures that a $\textsc{ClauseSkip}$ is added to each PDR node representing this clause (in any calling context). We then write a function $p_{\textsc{Skip}}$ to ensure that, when an appropriate lookup variable arrives, it is handled correctly.

$$p_{\textsc{Skip}}(\hat{k}, \textsc{ClauseSkip}(s, \hat{x}')) = \begin{cases} \varnothing & \text{if } \hat{k} \text{ not of form } \hat{x} \text{ or if } \hat{k} = \hat{x}' \\ \{s^{[\hat{x}^{\downarrow}]}\} & \text{otherwise} \end{cases}$$

Recall that this function is invoked by the $\textsc{Push+Dynamic Pop to State}$ rule when a push for variable $\hat{x}$ arrives at the state where the $\textsc{ClauseSkip}$ was added by $u_{\hat{a}_1 \ll \hat{a}_0}$ above. This function will then examine $\hat{x}$. If it matches the $\hat{x}'$ of the clause, we add no transitions to the PDR: we've discovered the variable we want and the $t_{\hat{a}_1 \ll \hat{a}_0}$ function above will handle this case. Otherwise, we add edges that allow the lookup to proceed by ignoring this clause and moving to the one before it.

Again, PDR closure will ensure that this only occurs for lookup variables that actually arrive at this program point. We can safely avoid adding to our PDR the vast majority of PDS edges that this clause would naively imply.

8.4.3 Continuations. In addition to the above-described dynamism, PDR closure can model a rudimentary form of computation over multiple stack elements. This is best illustrated by implementing rule 9 (the "capture" rule) of Definition 7.1. This rule is instrumental in the restructuring of the lookup function to be amenable to embedding as a pushdown reachability problem: it allows the first subordinate lookup in Definition 4.6's rule 6 to be written as Definition 7.1's rule 6a and allows the result of that subordinate lookup to be stored in the stack for use by rule 6b.

The PDR is further motivated by this use case: statically constructing a pushdown system to support capture is prohibitively expensive, as the rule must apply to *any* state in the PDS.

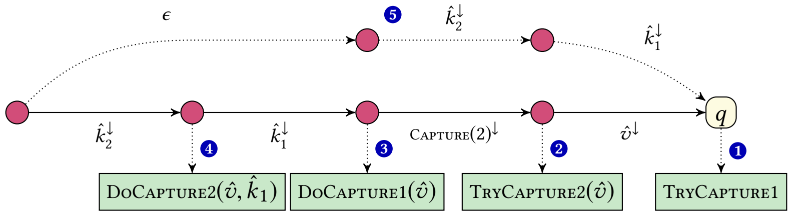


Fig. 32. Example PDR Closure of "Capture"

Specializing capture to the aforementioned rules 6a and 6b would require quadratically many PDS transitions in the number of states, while a more general form of capture applicable to e.g. binary operators would require exponentially many transitions. As most of these transitions would be unused and as it is not clear how to preemtively identify those transitions, we rely upon the lazy nature of the PDR to solve this problem.

In a walk of the pushdown system, the capture behavior of rule 6a would require four pops: the value, the capture symbol, and two arbitrary elements. As our closure algorithm only admits one pop per transition, we encode this process by introducing *four* dynamic pop forms to the Ψ set. Each dynamic pop form represents a continuation in the process of capturing a value. Figure 32 illustrates this process with a sample PDR, showing the closure process operating on the transition functions we now describe.

We initially suppose that the PDR fragment contains only the nodes and solid edges in the middle of the diagram. Consider a naive walk of the automaton: the sequence of push operations in the chain of solid edges suggests that the q node can be reached with $[\hat{v}, \textsc{Capture}(2), \hat{k}_1, \hat{k}_2]$ as the top four elements of the stack (with $\hat{v}$ topmost). Our goal is to reorganize those stack elements in accordance with rule 9.

We begin by adding our first new dynamic pop form, $\textsc{TryCapture1}$, to every new state. This is accomplished by the following function:

$$u_{\textsc{TryCapture1}} = \{\textsc{TryCapture1}\}$$

After step ❶ is complete, $\textsc{TryCapture1}$ is added to the set of dynamic pop transitions by the Transition Expansion rule (Figure 31). This is denoted in the diagram by the unlabeled edge from q to the rectangular node below it. This dynamic pop is used to begin the capture process whenever a value arrives. Next, we define a function which reacts to an element on top of the stack by storing it in a dynamic pop of the form $\textsc{DoCapture2}(\hat{v},)$ for safe keeping:

$$p_{\textsc{TryCapture2}}(\hat{v}, \textsc{TryCapture1}) = \{\textsc{TryCapture2}(\hat{v})\}$$

After step ❷ is complete, $\textsc{TryCapture2}(\hat{v})$ is introduced to the graph. Note that this occurs regardless of whether a $\textsc{Capture}(2)$ appears on the stack. This is a consequence of the single action model of PDR closure and is an intentional tradeoff. The single action model may lead to the addition of spurious $\textsc{TryCapture2}$ elements to the PDR graph, but it ensures that the work-list mentioned in Section 8.3 is bounded quadratically by the size of the graph.

Next, we wish to pop a $\textsc{Capture}(2)$ element, continuing to keep the $\hat{v}$ value so we may insert it lower in the stack. We define a function to introduce a dynamic pop $\textsc{DoCapture1}(\hat{v})$ to reflect this:

$$p_{\textsc{DoCapture1}}(\textsc{Capture}(2), \textsc{TryCapture2}(\hat{v})) = \{\textsc{DoCapture1}(\hat{v})\}$$

We likewise must pop $\hat{k}_1$ and, like $\hat{v}$, store it until we have enough elements to reorganize the stack. A function introducing a dynamic pop $\textsc{DoCapture2}(\hat{v}, \hat{k}_1)$ addresses this:

$$p_{\textsc{DoCapture2}}(\hat{k}_1, \textsc{DoCapture1}(\hat{v})) = \{\textsc{DoCapture2}(\hat{v}, \hat{k}_1)\}$$

This function is step ❹ in the process above. By this point, we have popped three elements from the stack; once another element is popped, we will be ready to introduce a series of push operations which effectively reorders the stack. We complete the process by introducing a function to do so:

$$p_{\textsc{Capture}(2)}(\hat{k}_2, \textsc{DoCapture2}(\hat{v}, \hat{k}_1)) = \{q^{[\hat{v}^{\downarrow}, \hat{k}_2^{\downarrow}, \hat{k}_1^{\downarrow}]}\}$$

The above function together with the Push+Dynamic Pop to State and Pending Action rules creates the dotted path along the top of the diagram which is labeled as step ❺. Note that the above functions assume a fixed notion of q which was elided for simplicity. In reality, q would be included as a parameter of each of the above dynamic pop forms.

The above demonstrates how a finite sequence of arbitrary stack operations may be encoded in a continuation passing-like form in PDR closure. The full formal specification of DDPA uses this technique extensively to implement features such as binary operators and state. As with other closure operations, our use of a monotone, compact automaton yields significant work sharing benefits

8.4.4 Decidability. Reachability is clearly decidable in a PDR due to the finiteness of the PDR domain: it is baked into Definition 8.3.

Lemma 8.7. *For fixed PDR domain $\langle S, Q, \Gamma, \Psi \rangle$ and PDR transition spec $\langle t, u, p \rangle$, the transitive closure of $\Longrightarrow$ is computable.*

In order to show our analysis is decidable we only need to show it uses a (finite) PDR domain. The pre-states S are pairs of program points and finite call stacks, which are finite; the stack grammar Γ is the continuation stack $\hat{K}$ from Section 7, which is also finite. The set of states Q is finite because the length of the action lists from $\Gamma_*^{\updownarrow}$ appearing in Q is bounded by a constant; this argument is subtle as some dynamic pop closures can lead to more elements being pushed onto the stack. With a finite Q, the set Ψ is finite. PDR closure thus operates on finitely many possible states $\langle Q^{\mathrm{C}}, \delta, \eta \rangle$ and so will eventually run out of new states to add.

We show in Section 10 how this algorithm is fast enough to compete with modern higher-order demand-driven analyses.

9 EXTENSIONS

In this section, we outline four extensions: records, conditional branching, path sensitivity, and mutable state. Our goal here is to show there is no fundamental limitation to the model given in the previous sections: DDPA can in principle be extended to the full feature set of a realistic programming language. For the first three extensions, we use the Overview grammar in Figure 1 and we incrementally build a theory with all three extensions since they are overlapping. For mutable state, we show how just the core theory is extended for simplicity.

9.1 Records

Here we outline an extension to a standard notion of records and projection as per the grammar of Figure 1. Consider a lookup of variable $\hat{x}$: if $\hat{x}$ is defined as $\hat{x} = \hat{x}'\,.\,\ell$, then we must first look up $\hat{x}'$ and then project ℓ from its record value. $\hat{x}'$ may also be defined as a record projection and so on: in the general case, there could be a stack of record projections to be performed. This is similar to the non-local lookup stack of our analysis, and not coincidentally: non-locals may be encoded in terms of records via closure conversion.

Fortunately, the continuation stack we defined in Section 7 lends itself to solving this problem. We add to the grammar of continuation actions $\hat{k}$ a projection form $.\,\ell$. We then augment Definition 7.1 with the following new clauses:

Definition 9.1. We extend Definition 7.1 to records by adding the following clauses; assume $\hat{a}_1 \lll \hat{a}_0$.

$\boxed{\textsc{Record Projection Start}}$
If $\hat{a}_1 = (\hat{x} = \hat{x}' \,.\, \ell)$ and $\hat{K} = [\hat{x}] \,||\, \hat{K}'$ then $\hat{G}\llparenthesis [\hat{x}', \,.\ell] \,||\, \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
$\boxed{\textsc{Record Projection Stop}}$
If $\hat{a}_1 = (\hat{x} = \{\ell_1 = \hat{x}', \ldots\})$ and $\hat{K} = [\hat{x}, \,.\ell_1] \,||\, \hat{K}'$ then $\hat{G}\llparenthesis [\hat{x}'] \,||\, \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.

The two clauses above are symmetric: clause $\boxed{\textsc{Record Projection Start}}$ introduces the projection action $.\ell$ when we discover that we will need to project from the variable we find while clause $\boxed{\textsc{Record Projection Stop}}$ eliminates this projection action when the corresponding record value is found.

9.2 Conditional Branching

In Section 2, we give conditionals the syntax $x \sim p \,?\, f : f$. The analysis of conditions is straightforward: the bodies are wired in just like function calls. The following clauses may be added to the records extension above to obtain an analysis for conditionals.

Definition 9.2. We extend Definition 4.6 to conditionals by adding the following clauses. Assume $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c}$ below is always a conditional clause.

$\boxed{\textsc{Conditional Top}}$
If $\hat{a}_1 = (\hat{x} \overset{(\!|\hat{c}}{=} \hat{x}')$ and $\hat{K} = [\hat{x}] \,||\, \hat{K}'$ then $\hat{G}\llparenthesis [\hat{x}'] \,||\, \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
$\boxed{\textsc{Conditional Bottom}}$
If $\hat{a}_1 = (\hat{x} \overset{|\!)\hat{c}}{=} \hat{x}')$ and $\hat{K} = [\hat{x}] \,||\, \hat{K}'$ then $\hat{G}\llparenthesis [\hat{x}'] \,||\, \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.
$\boxed{\textsc{Conditional Top: Non-Subject Variable}}$
If $\hat{a}_1 = (\hat{x}'' \overset{(\!|\hat{c}}{=} \hat{x}')$ and $\hat{K} = [\hat{k}] \,||\, \hat{K}'$ such that $\hat{k} \neq \hat{x}''$, then $\hat{G}\llparenthesis \hat{K}, \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.

In this version we do not refine the analysis based on whether the conditional pattern (abstractly) matched or not, so the analysis is not particularly accurate. The filtering extension below gives a much more precise analysis for conditionals.

9.3 Filtering for path sensitivity

We can formalize path sensitivity in DDPA by keeping track of sets of accumulated patterns in our lookup function, and disallowing matches not respecting the patterns they passed through. We use Π^+ and Π^- to range over sets of patterns which a discovered value *must* or *must not* match, respectively. Formally, we will define path-sensitive DDPA as an extension of both the core, records, and conditionals theories.

Definition 9.3. We extend Definition 7.1 first by adding all clauses from Definitions 9.1 and 9.2. We then modify the grammar of $\hat{k}$ to replace elements of the form $\hat{x}$ with elements of the form $\hat{x}^{\Pi^+}_{\Pi^-}$. Each existing clause conveys these pattern sets unchanged. We then replace clause $\boxed{\textsc{Value Discovery}}$ of Definition 7.1 and clause $\boxed{\textsc{Conditional Top}}$ of Definition 9.2 with the following clauses:

$\boxed{\textsc{Matching Value Discovery}}$
If $\hat{a}_1 = (\hat{x} = \hat{v})$, $\hat{K} = [\hat{x}^{\Pi^+}_{\Pi^-}]$, and $\hat{v}$ matches all patterns in Π^+ and none in Π^-, then $\hat{v} \in \hat{V}$.
$\boxed{\textsc{Conditional Top Positive}}$
If $\hat{a}_1 = (\hat{x} \overset{(\!|\hat{c}}{=} \hat{x}')$, $\hat{K} = [\hat{x}^{\Pi^+}_{\Pi^-}] \,||\, \hat{K}'$, $\hat{c} = (\hat{x}_2 = \hat{x}_1 \sim \hat{p} \,?\, \hat{f}_1 : \hat{f}_2)$, $\hat{f}_1 = \texttt{fun}\ \hat{x}' \texttt{ -> (} \hat{e} \texttt{)}$, and $\hat{x} \in \{\hat{x}_1, \hat{x}'\}$, then $\hat{G}\llparenthesis [\hat{x}'^{\Pi^+ \cup \{\hat{p}\}}_{\Pi^-}] \,||\, \hat{K}', \hat{a}_1, \hat{C} \rrparenthesis \subseteq \hat{V}$.

$\boxed{\textsc{Conditional Top Negative}}$

If $\hat{a}_1 = (\hat{x} \overset{(\!|\hat{c}}{=} \hat{x}')$, $\hat{K} = [\hat{x}^{\Pi^+}_{\Pi^-}] \,||\, \hat{K}'$, $\hat{c} = (\hat{x}_2 = \hat{x}_1 \sim \hat{p}\,?\,\hat{f}_1 : \hat{f}_2)$, $\hat{f}_2 = \texttt{fun}\ \hat{x}' \texttt{ -> } (\,\hat{e}\,)$, and $\hat{x} \in \{\hat{x}_1, \hat{x}'\}$, then $\hat{G}(\!|[\hat{x}'^{\Pi^+}_{\Pi^- \cup \{\hat{p}\}}] \,||\, \hat{K}', \hat{a}_1, \hat{C}|\!) \subseteq \hat{V}$.

The $\boxed{\textsc{Matching Value Discovery}}$ clause shows how the filters are used: any value not matching the positive filters is discarded, and oppositely for the negative filters.

The original $\boxed{\textsc{Conditional Top}}$ clause was the case where we reached the start of a case clause and search variable $\hat{x}$ was passed as the parameter; in that case, the clause continued by searching for the argument at the call site. Here, we have separated that clause into two cases. In $\boxed{\textsc{Conditional Top Positive}}$, the function was the first branch of a conditional, so we know that any discovered value is only relevant if it matches the conditional's pattern. Thus, we add the pattern to the filter set to constrain it so. Clause $\boxed{\textsc{Conditional Top Negative}}$ is the opposite case.

9.4 State

Lookup in the presence of state may also be performed using only a call graph, but there are several subtle issues that must be addressed. We consider here a variation of the presentation language which includes OCaml-style references with `ref` x / `!` x / x `:=` x syntax.

First, a simple search back to find the most recent mutation or creation site may not always give the correct answer as references may be changed through aliases, as illustrated in the following pseudocode:

```
inner = ref true;
outer = ref inner;
inner := false;
(!outer) := 0;
```

If we only consider updates to the `inner` variable, a search for its contents by the end of the program would incorrectly yield `false`. The correct answer is `0` due to the last line, which updates the same reference cell using the contents of `outer`. This is the classic problem of *alias analysis*: when searching for the contents of mutable variables, we must consider the possibility that statements not directly involving the cell we are examining may update it nonetheless. So, explicit alias testing is needed to verify potential aliases are not being passed by.

Second, we must be careful not to confuse *allocation* with an *allocation site*. Recursive functions which allocate cells illustrate this issue. In the following code, two cells are allocated:

```
f = fun x ->
  { cell = ref false;
    if x then (f false, cell) else cell };
(a,b) = f true;
b := 0;
```

Here, the call `f true` returns two cells, both allocated at the site `ref false`. Although both `ref` values are allocated at the same program point, we must recognize that they are *not* aliases; thus, by the end of the program, `a` contains `false` and not `0`. Recognizing this distinction requires us to bear calling context in mind when performing alias analysis.

Not only may DDPA be adapted to address state, but the alias analysis itself may be accomplished by the lookup function. The implementation machinery necessary to accomplish this alias analysis is verbose. For legibility and brevity, we present the necessary steps at a high level.

We begin by adding a single bit of information to contexts in the form of a function: $\textsc{IsDirty}(\hat{C})$, which determines if a context is "dirty". We require that the initial context is clean and that contexts are marked dirty when precision is lost; for instance, $\textsc{Pop}(\epsilon)$ is dirty since the empty context ϵ indicates that we do not know anything about the stack above the current point. Dirty contexts will allow us to recognize the recursive case above - if the context has been pruned we cannot be certain on an alias question.

With these functions, we present revised clauses for lookup based upon the original Definition 4.6.

Definition 9.4. Definition 4.6 is extended to a stateful language as follows. First, the codomain of the function is modified to a set of pairs $\langle \hat{v}, \hat{C} \rangle$ by replacing the $\boxed{\textsc{Value Discovery}}$ clause with the following:

$\boxed{\textsc{Contextual Value Discovery}}$
If $\hat{a}_1 = (\hat{x} = \hat{v})$ and $\hat{X} = []$, then $\langle \hat{v}, \hat{C} \rangle \in \hat{V}$.

We write $\hat{v} \in \hat{V}$ to indicate $\langle \hat{v}, \hat{C} \rangle \in \hat{V}$ for any $\hat{C}$. Next, we add the following clauses:

$\boxed{\textsc{Dereference}}$
If $\hat{a}_1 = (\hat{x} = \,!\, \hat{x}')$ and $\hat{a}_1 \lll \hat{a}_0$ then letting $\hat{V}' = \hat{G}([\hat{x}'], \hat{a}_1, \hat{C})$, for each $\texttt{ref}\ \hat{x}'' \in \hat{V}'$, $\hat{G}([\hat{x}''] \,||\, \hat{X}, \hat{a}_1, \hat{C}) \subseteq \hat{V}$.

$\boxed{\textsc{May Alias}}$
If $\hat{a}_1 = (\hat{x}_1' = \hat{x}_2' := \hat{x}_3')$, $\hat{a}_1 \lll \hat{a}_0$, and $\textsc{MayAlias}(\hat{x}, \hat{x}_2', \hat{a}_1, \hat{C})$, then $\langle \texttt{ref}\ \hat{x}_3', \hat{C} \rangle \in \hat{V}$.

$\boxed{\textsc{May Not Alias}}$
If $\hat{a}_1 = (\hat{x}_1' = \hat{x}_2' := \hat{x}_3')$, $\hat{a}_1 \lll \hat{a}_0$, and $\textsc{MayNotAlias}(\hat{x}, \hat{x}_2', \hat{a}_1, \hat{C})$, then $\hat{G}([\hat{x}] \,||\, \hat{X}, \hat{a}_1, \hat{C}) \subseteq \hat{V}$.

In these clauses, the terms $\textsc{MayAlias}$ and $\textsc{MayNotAlias}$ refer to the following predicates:

- $\textsc{MayAlias}(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{G}([\hat{x}_1], \hat{a}, \hat{C})$, $\hat{V}'' = \hat{G}([\hat{x}_2], \hat{a}, \hat{C})$, and $\exists \texttt{ref}\ \hat{x}'' \in (\hat{V}' \cap \hat{V}'')$
- $\textsc{MayNotAlias}(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{G}([\hat{x}_1], \hat{a}, \hat{C})$, $\hat{V}'' = \hat{G}([\hat{x}_2], \hat{a}, \hat{C})$, and $\hat{V}' \neq \hat{V}''$ or $\hat{V}' \neq \{\langle \texttt{ref}\ \hat{x}, \hat{C}' \rangle\}$ or $\hat{V}' = \{\langle \texttt{ref}\ \hat{x}, \hat{C}'' \rangle\}$ where $\textsc{IsDirty}(\hat{C}'')$

In the above definition, clause $\boxed{\textsc{Dereference}}$ handles dereferencing. It finds the `ref` values which may be in $\hat{x}'$ from the current point in the program; it then returns to that point to find all of the values that those variables may contain. This return is necessary since we want the value at the point the ! happened.

Clauses $\boxed{\textsc{May Alias}}$ and $\boxed{\textsc{May Not Alias}}$ address cell updates. Clause $\boxed{\textsc{May Alias}}$ determines if the updated cell in $\hat{x}_2'$ *may* alias the cell we are looking up; if this is the case, the value assigned by the cell update may be our answer. Clause $\boxed{\textsc{May Not Alias}}$ addresses the case in which the updated cell *may* be different from the target of our lookup. This happens when the lookups of each variable yield different results or when they result in multiple cells – even if the sets of cells are equal, the orders in which the program modifies the cells might differ, so we take the conservative approach and call them different. Here, we use the $\textsc{IsDirty}$ function to address the recursive allocation case described above. $\textsc{MayAlias}$ and $\textsc{MayNotAlias}$ can be simultaneously satisfiable; when that happens, the analysis explores both clauses $\boxed{\textsc{May Alias}}$ and $\boxed{\textsc{May Not Alias}}$.

Along with the above modifications, the existing clauses $\boxed{\textsc{Skip}}$ and $\boxed{\textsc{Function Exit}}$ need to be extended to support state. As written, clause $\boxed{\textsc{Skip}}$ allows us to skip by call sites and pattern matches whose output do not match the variable for which we are searching. This is sound in a pure system, but in the presence of side-effects we must explore these clauses to ensure that they did not affect the cell we are attempting to dereference. We thus modify clause $\boxed{\textsc{Skip}}$ by prohibiting $\hat{b}$ from being a call site or pattern match. We require a new clause similar to clause $\boxed{\textsc{Function Exit}}$ but for the case in which the search variable does not match the output variable. In that case, we proceed into the body of the function but in a "side-effect only" mode: we skip by every clause which is not a cell assignment or does not lead to one. We leave side-effect only mode once we leave the beginning of the function which initiated it.

10 EVALUATION

We implemented DDPA and conducted a series of experiments to determine whether it is viable. The implementation supports the extensions described in Section 9 and is available along with the test cases, experiment runners, and raw results.[2]

10.1 Goals

We evaluated our implementation of DDPA in comparison to the proof-of-concept implementation[3] [36] and to a state-of-the-art higher-order forward analysis, P4F [19]. We had three goals: first, our implementation must be correct and produce the same outputs as the proof-of-concept; second, it should out-perform the proof-of-concept; and third, it should perform similar to P4F or better.

Our implementation succeeded at the first two goals. It produces the same results as the proof-of-concept implementation throughout a test suite designed for the latter implementation's language. Further, our implementation (which includes the PDR automata from Section 8) delivers a speedup of at least 1000x over the proof-of-concept implementation. Evaluation of our third goal – performance relative to P4F – is less straightforward and is the subject of the rest of this section.

We chose to compare to P4F because it is a recent analysis similar to ours in expressiveness: it is flow-sensitive and polyvariant. P4F's reference implementation[4] aligns with our DDPA implementation in several ways: for example, they both lose precision on numbers and arithmetic operations. More interestingly, we chose P4F because it allows us to shed some light into a broader open question: the trade-offs between context-sensitivity and data-dependence [39] in analyses for higher-order languages. DDPA and P4F represent the opposite sides of the trade-off: DDPA approximates the call-stack and captures data dependencies exactly, while P4F captures the call-stack exactly and approximates data dependencies. In analyses for first-order and object-oriented languages, it is clear that context-sensitivity dominates data-dependence [55]. But the results in the rest of this section suggest that this may not hold in higher-order languages, because DDPA out-performs P4F in a some cases.

10.2 Test Cases

To compare P4F and DDPA, we selected a series of test cases which ran on both analyses' implementations. Our choice of P4F limited our selection of test cases, as we were unable to run the P4F reference implementation on bigger test cases closer to real-world programs (see Section 10.6). Instead, we selected test cases from P4F's reference implementation (including tests which did not appear in the corresponding P4F paper). We also included a test case, `flatten`, from OOAM's [23] reference implementation,[5] which is the only test case in OAAM's suite that is not included in P4F as well and that is supported by both implementations.

Unfortunately, as of this writing no standard benchmark for higher-order program analyses exists; however, as other evaluations in the literature use these benchmarks, they appear to be a workable approximation for such a benchmark suite. We describe these benchmarks below:

eta Tests spurious function calls that do not affect the lookup subject.

[2] https://github.com/JHU-PL-Lab/odefa/tree/toplas

[3] https://github.com/JHU-PL-Lab/odefa-proof-of-concept

[4] https://bitbucket.org/ucombinator/p4f-prototype

[5] https://github.com/dvanhorn/oaam

mj09 Tests the alignments of calls and returns.

kcfa-2 and kcfa-3 The worst-case for k-CFA. Test non-local variables in increasingly nested functions.

blur and loop2-1 Test functions with non-local variables created in a loop.

facehugger Tests recursive functions with control-flow paths that may only cross if precision is lost.

tak and ack Test recursive functions.

cpstak Continuation passing style version of *tak*. Stresses the call-return alignment in our analysis.

sat-1, sat-2 and sat-3 Brute-force SAT solver, an exponential problem. *sat-1* solves a formula with four variables, and *sat-2* and *sat-3* solve the same formula with seven variables, which is defined as a curried function in *sat-2* and as an uncurried function in *sat-3*.

flatten Flatten deeply nested lists.

map Map a function over the elements of a list.

rsa Encryption and decryption algorithms from the RSA public-key cryptosystem.

primtest Fermat primality test.

deriv Symbolic derivation.

regex Regular expression matching with derivatives.

The last four test cases are closer to real-world programs: *rsa* and *primtest* are numerical programs, and *deriv* and *regex* manipulate lists and symbols. The other test cases are micro-benchmarks based on common functional programming idioms. Figure 33 contains statistics on these test cases including number of program points, number of function definitions, and so forth.

The test cases are written in Scheme and not in the language presented throughout Sections 4 through 9. In our experiments, we run DDPA on the output of a translator from Scheme to our presented language. This translator preserves the semantics of the abstract interpretation, but it may not preserve the concrete semantics. This compromise simplifies the analysis implementation by reducing the number of features it must support. For example, all arithmetic operations are translated into additions because DDPA abstracts all numbers to the same value and looses precision on all arithmetic operations the same way. We also encode Scheme features that our implementation does not support: for example, lists are encoded as records that represent cons cells, and functions with multiple arguments are encoded as functions with a record argument.

10.3 Experiments

We conducted two experiments to compare our implementation to P4F's: *Monovariant*, in which we used the less expressive (and often, but not always, more performant) settings; and *Polyvariant*, in which we used the more expressive settings. In Monovariant, we chose $k = 0$ for both analyses, disabling context-sensitivity. In Polyvariant, we chose $k = 1$ for P4F, because that is the only polyvariance setting supported by the reference implementation. But choosing k for the DDPA implementation was less straightforward.

We could not simply choose the same k for both analyses. The variables k in DDPA and k in P4F may share a name and serve a similar purpose (to determine the amount of context to preserve), but they have different implications. In DDPA, k applies both to the top-level queries *and* to the sub-queries necessary to fulfill it, causing the context stack to be exhausted more rapidly and the analysis to converge sooner. When compared to P4F, the k of DDPA may have a lesser impact on running time but may also achieve less precision.

So, when possible, we were conservative and chose k such that our analysis is at least as precise as P4F, modeling the control-flow accurately. In other cases we defaulted to $k = 1$, which happened for one of three reasons. First, our analysis models the control-flow accurately with $k = 0$, so we chose $k = 1$ in the Polyvariant experiment to distinguish it from the Monovariant experiment. Second, no choice of k suffices, because the source of inaccuracy is a factor other than the context-stack finitization, for example, an arithmetic operation. Third, an accurate control-flow is hard to determine. This occurs in the bigger test cases, but $k = 1$ is a reasonable default for them because they tend not to use convoluted higher-order functions. Our choice of k for each test case appears in Figure 33.

When running the experiments, we measured running times and memory use, but could not measure expressiveness (see Section 10.6). We ran experiments on a machine with an Intel Xeon (3.10GHz) processor and 8GB of RAM, running Debian 9.5. The machine was dedicated to running the experiments and the load average remained at approximately 1.

10.4 Results

We ran ten trials per test case per experiment. The running times appear in Figure 33. The memory use remained low, never exceeding approximately 230MB, and correlates with running times: when an analysis runs for longer, it also consumes more memory. Our analysis used 0.3x as much memory as P4F on average, a difference we believe to be immaterial and attribute to the implementation languages: our implementation is written in OCaml while P4F's is written in Scala (see Section 10.6). The dispersion in all measurements was negligible: the coefficient of variation was lower than approximately 3%.

10.5 Analysis

In most cases, DDPA's and P4F's running times were within the same order of magnitude, supporting the goal that our analysis should perform similar to P4F. But in two test cases DDPA is much slower than P4F: *deriv* and *regex*. These test cases consist of data structure (list) manipulation and demonstrate the effect of our analysis' perfect continuation-stack precision. We conjecture DDPA is slower on those cases because it is also more precise: DDPA does not lose the connection when data flows into and out of a list, but P4F may. We leave for future work measuring how much precision is recovered (see Section 10.6). When structure-transmitted data dependencies are not fundamental to a program, our analysis performs similar to P4F, as illustrated by the other two test cases closer to real-world programs, *rsa* and *primtest*, which consist of numeric operations.

Beyond our analysis and P4F, our results illuminate the trade-offs between context-sensitivity and data-dependence [39] in analyses for higher-order languages. Our results suggest context-sensitivity may not dominate data-dependence the same way it does in first-order and object-oriented languages [55]. In test cases *kcfa-3* and *rsa*, for example, our analysis out-performs P4F. We conjecture this may be a consequence of how these languages are used: closures with non-local variable references in functional languages are created more often and affect the analyses more than the structure-transmitted data dependencies in object-oriented languages.

10.6 Threats to Validity

Test Cases. Our test cases represent common functional programming idioms, but they are not at the scale of real-world programs, they do not stress the state extension, and some of them consist of numeric operations, on which both our analysis and P4F are imprecise. Unfortunately, no standard benchmark for higher-order analyses exists; these are the test cases used by many other publications and so serve as an approximation for such a benchmark. This shortcoming a broader issue on the evaluation of higher-order program analyses.

Case	PP	FD	FC	VR	NL	LD	Monovariant	O÷P	k	Polyvariant	O÷P
eta	16	4	5	13	1	1	419 429	1.02	1	424 421	0.99
mj09	18	4	6	16	3	3	426 450	1.06	M	449 466	1.04
kcfa-2	22	6	8	21	2	3	427 434	1.02	5	471 468	0.99
blur	27	3	8	30	9	2	466 498	1.07	I	506 531	1.05
facehugger	27	3	6	29	6	2	454 514	1.13	M	505 509	1.01
tak	27	1	5	39	14	2	431 622	1.44	M	550 1573	2.86
ack	28	1	4	32	9	3	427 510	1.19	M	496 713	1.44
kcfa-3	30	8	11	29	3	4	444 473	1.07	7	642 519	0.81
loop2-1	33	3	5	39	12	4	447 537	1.20	M	507 642	1.27
cpstak	40	6	7	56	19	6	466 1381	2.96	M	566 3219	5.69
sat-1	43	7	8	43	20	5	456 619	1.36	4	655 1213	1.85
flatten	50	2	5	49	9	4	411 1153	2.81	1	427 1139	2.67
map	54	6	10	53	7	3	481 626	1.30	1	541 665	1.23
sat-2	74	16	17	68	36	13	571 1363	2.39	14	33866 82720	2.44
sat-3	78	11	12	73	36	8	563 1250	2.22	14	5694 53936	9.47
rsa	126	9	15	155	39	3	576 919	1.60	H	4580 1661	0.36
primtest	128	5	9	141	45	9	553 898	1.62	M	1267 1798	1.42
deriv	203	5	13	200	27	7	463 3828	8.27	I	469 9352	19.94
regex	314	24	51	326	113	8	714 28173	39.46	H	3003 84312	28.08

1 10 100 1000 10000 100000

🟪 P4F
🟦 Our Analysis

Running Time (ms)

1 10 100 1000 10000 100000

Fig. 33. The first section characterizes the input programs, which are sorted by number of program points. PP: Program Points. FD: Function Definitions. FC: Function Calls. VR: Variable References. NL: Non-Local Variable References. LD: Lexical Depth. The remaining two sections are the results of the two experiments: *Monovariant* and *Polyvariant*. In Monovariant, $k = 0$. In Polyvariant, $k = 1$ for P4F and the k column lists the choices for DDPA. Numbers represent DDPA builds an *accurate* control-flow model, in all other cases we default to $k = 1$. M: A *monovariant* DDPA also models control-flow accurately. I: The control-flow model is *inaccurate* for all k for a reason other than the context-stack finitization, for example, an arithmetic operation. H: An accurate control-flow model is *hard* to determine. The bar graphs are the arithmetic mean of the running times of ten trials; we omit the error bars because the dispersion was negligible (in all cases the coefficient of variation was lower than approximately 3%). The columns labeled $O{\div}P$ are the running times of DDPA normalized with respect to P4F.

We settled for these test cases because, to compare the implementations of DDPA to P4F, we required cases which run in both implementations. This limited our choices, because P4F's reference implementation does not support features necessary to analyze many bigger programs and, even when it does, it may fail due to what appears to be an implementation bug. P4F terminates with an exception and does not produce a result when a non-function appears to flow into the operator position of a function call. This occurs regardless of whether the test case really contains an ill-formed function call or the analysis overapproximated. The minimal non-trivial Scheme program that triggers this behavior is the following:

```
(define f (if (< 1 2) (lambda (x) x) #t))
(f 3)
```

P4F's reference implementation loses precision on the condition and allows `#t` to flow into `f`. It then throws an exception when it reaches the function call because `#t` is not a function. Our implementation loses precision on the condition in a similar fashion, but it succeeds to analyze the function call and even detects that it *may* be ill-formed. Our implementation also ran on bigger test cases that are closer to real-world programs, but they were excluded from the experiments for the lack of a P4F baseline.

We worked around this problem by selecting test cases available in P4F's reference implementation. We plan to address the broader issue of evaluating higher-order program analyses in future work.

Expressiveness Measurements. A general metric for expressiveness cannot exist, because different analyses may capture different program properties and their outputs may be incomparable. Other evaluations in the literature work around this issue by measuring expressiveness via a proxy, either an intrinsic analysis property or a client. For example, Earl et al. [12] measures an intrinsic property, the number of *singletons* (abstract value sets containing a single function); and Might et al. [34] uses a client, the number of function inlinings justified by the analyses.

Unfortunately, neither approach is practical for comparing DDPA to P4F because the analyses do not share technical foundations. Intrinsic properties of the analyses are too far apart; for example, our analysis would be at an unfair advantage if we compared the number of singletons, because P4F's abstract functions include an abstract environment while DDPA's abstract functions do not. (Instead, DDPA relies on non-local variable lookups.) These differences also account for part of the reason why designing and implementing a client compatible with both analyses is an engineering problem of its own, which we leave for future work.

When possible, we worked around this issue by conservatively choosing k for our analysis such that it is as precise as, or more precise than, P4F. We will be able to fine tune this choice when we have a client compatible with both analyses.

Experimental Setup. We implemented DDPA in OCaml. P4F's reference implementation is written in Scala. The general performance difference between these languages is negligible for our purposes.

11 RELATED WORK

DDPA uses many concepts of first-order demand-driven CFL-reachability analyses [21] to give precise analysis of higher-order functions: like demand-driven CFL-reachability analyses, DDPA is centered around using a CFG to look up variable values in a demand-driven fashion, calls and returns are aligned, and lookup is computed lazily. Two issues make a higher-order analysis more challenging: the CFG needs to be computed on-the-fly due to the presence of higher-order functions, and non-local variable lookup is subtle. The demand-driven analyses cited in this paper delve further into the trade-off between active propagation and demand-driven lookup, and this is something we plan to explore in future work. There are many other first-order program analyses with a demand-driven component; several use Datalog-style specification formats [42, 46, 56].

The challenges we face in precise non-local variable lookup are related to data propagation challenges in first-order languages. Intuitively, one might attempt to address non-local variables via a *closure conversion* pass [5]: we can explicitly add closure structures to the language syntax and function values become pairs between the function's code and a list or record containing the non-local values. While this translates the challenge into the first-order analysis space, however, it is not any easier to solve: the problem of finding the correct binding for a non-local variable is now the problem of accessing the correct field in a list or record. This problem is known to be difficult even in first-order program analyses: Reps [39] proved that these *structure-transmitted data dependences* are impossible to track perfectly. The tight relationship between the analyses of non-local lookup and of structured data lookup is clear in DDPA: both non-locals and record accesses use the same continuation stack as per Section 9, and the requirement for a second fully-precise (call) stack leads to the undecidability of ωDDPAc. There does not appear to be any mention of this connection between first-order data dependencies and higher-order non-local variable accesses in the literature.

Some first-order analyses also track structure-transmitted data dependencies (e.g. LCL-reachability analysis [55]) are in a similar design space as DDPA but have a different optimization function. In higher-order analyses, the loss of data dependencies causes imprecision in closure lookup, which rapidly pollutes the CFG and degrades analysis precision. For this reason, DDPA preserves a perfect stack for data dependencies and approximates the call stack only. First-order analyses sometimes sacrifice data dependence precision to improve call stack precision as there is a less drastic fall-off in overall precision when data dependencies are imperfect.

DDPA Compared with higher-order forward analyses

Comparing DDPA with the extensive literature of higher-order forward analyses is a difficult task: while there are a great many overlapping concepts, they don't precisely align and so it is hard to make accurate comparisons. Here it will have to suffice to point out some commonalities and differences.

Higher-order program analyses are generally based on abstract interpretations [8]; such analyses define a finite-state abstraction of the operational semantics transition relation to soundly approximate the program's runtime behavior. The resulting analysis has the same general structure as the operational semantics it was based on: program points, environments, stacks, stores, and addresses are replaced with abstract counterparts which have finite cardinality, "hobbling" the full operational semantics of the language to guarantee termination of the analysis [31]. A sound analysis will visit the (finitely many) abstract counterparts of all reachable concrete program states, producing a finite automaton representing all potential program runs. Previous abstract interpretation based higher-order program analyses are forward analyses [11, 24, 29, 31, 34, 35, 44, 52].

Non-local variables. Dealing properly with non-local variables is a longstanding concern in higher-order program analyses; the classic *environment problem* [18, 30, 44] centers around obtaining precision in analyzing non-locals.

Perhaps the biggest contribution of DDPA is how our notion of call-return alignment also aligns non-local variables; this is the purpose of the additional non-locals stack which is not found in any previous work. Other works incorporating call-return alignment lack this non-locals stack and so do not obtain the degree of expressiveness we do with only call-return alignment. The particular advantage of aligning both locals and non-locals is that a full polymorphism model is obtained comparable to kCFA [44], without any explicit machinery for polymorphism.

In previous higher-order demand-driven analysis work non-local variables are not aligned and so call-return alignment cannot fully replace polymorphism – explicit let-polymorphism is also included [14, 37]. Another analysis in this space, Boomerang [47], targets Java and also does not address call-return alignment for non-local variables. None of this previous work is flow-sensitive.

Some higher-order forward analyses incorporate call-return alignment [19, 24, 51], but they also do not align non-local variables. One sign of how DDPA is more powerfully aligning calls and returns than these works is that ωDDPAc, DDPA without a pruned call stack, is undecidable, whereas these analyses are decidable with a full call stack. So, we are losing a full call stack, but gaining accuracy on non-local variables. These analyses incorporate other elements to achieve a fuller effect of polymorphism: CFA2 [51] additionally uses a polymorphism model similar to CPA [1], with a different contour allocated for each different function argument; PDCFA [24] includes an (abstracted) call stack in the program state; and, P4F [19] includes an orthogonal polymorphism layer of the kCFA variety. DDPA's improved precision on non-local variables is still insufficient for more sophisticated clients including environment analysis [32], because it does not preserve information about allocations of concrete vs. abstract bindings, but a variation on DDPA called DRSF addresses this shortcoming [13]. Our PDR automata build on ideas in PDCFA [24] to achieve more efficient reachability results.

Polymorphism and runtime complexity. Another important dimension of expressiveness is polymorphism aka context-sensitivity: whether functions can take on different forms in different contexts of use. The classic higher-order analysis polymorphism model is kCFA [44], which copies contours in analogy to forall-elimination in a polymorphic type system. But there are many routes to behavior that appears as polymorphism, and, as mentioned in the previous paragraph, call-return alignment can provide different contexts for different function calls and achieves the same effect as polymorphism in DDPA. The example we gave in Figure 3, for instance, needs only call-return alignment to give polyvariant behavior in DDPA. Another example of polymorphism as an emergent phenomenon of other program analysis features is the abstract garbage collection in ΓCFA, which can align calls and returns in tail position, for example, in the example program in Section 2.2 [32, § 6.6].

Even though we are using a call-stack approximation k levels deep in a similar fashion as kCFA, keeping the most-recent k frames, kDDPA polymorphism is not equivalent to kCFA. One sign that it must be different is that 1DDPA is provably polynomial (Theorem 7.8) whereas 1CFA is EXPTIME-complete [49]. The difference is that DDPA must also “spend” stack frames searching for the functions where non-local variables were defined, and so for non-local variables requires more stack frames to get the same approximation. We conjecture that, for a program with a maximal lexical nesting depth of d, the analysis $(k + d)$DDPA will be at least as expressive as kCFA. The additional d levels are needed because each lexical level gives the potential for one more level needed to search through to find the original definition of a non-local variable, in analogy with d-stages access links in a compiler implementation of non-locals. Each lexical level will entail its defining function being added to the call stack, and overall one extra function will appear per lexical level.

Insights into the run-time complexity of kDDPA are also gained when considering the complexity of non-local variable polymorphism. Non-locals are the (only) source of exponential behavior in kCFA [34, 50]; in particular, if lexical nesting were assumed to be of some constant depth not tied to the size of the program, kCFA would not be exponential. The complexity of kDDPA comes from the other direction: for any fixed k, the algorithm kDDPA is polynomial; but k needs to be increased by one for each level of stack alignment we wish to achieve in non-local lookup. Related to this are provably polynomial context-sensitive analyses which, like kDDPA, restrict context-sensitivity in the case of high degree of lexical nesting [22, 34]. mCFA [34] is a polyvariant analysis hierarchy for functional languages that is provably polynomial in complexity. This is achieved by an analysis that “in spirit” is working over closure-converted source programs: by factoring out all non-local variable references, their worst-case behavior has also been removed. But, this also affects the precision of the analysis: non-locals that are distinguished in kCFA are merged in mCFA. In kDDPA, the level of non-local precision is built into the constant k of how deep the run-time stack approximation is, so more precision is achieved as k increases.

The need for call stack approximation. DDPA requires the call stack to be finitely approximated to at most k frames in kDDPA. The call stack in fact *has* to be approximated by Theorem 5.19: the unbounded-stack ωDDPAc is a full and faithful λ-calculus interpreter. Still, kDDPA is currently wasteful on recursion, often unrolling a recursive function k levels only to see them all merge. We plan to address this shortcoming in future work. Note that higher-order analyses run into a similar problem; for example, kCFA keeps a k-depth stack, ΔCFA’s [32] Δ-frame strings are finite approximations, and PDCFA [24] must regularize the call stack to incorporate abstract garbage collection in a decidable fashion.

Path sensitivity and must analysis. DDPA only has a weak notion of path-sensitivity via the filters of Section 9.3. It also has only a primitive must-alias analysis in the mutable state extension in Section 9.4. So in these dimensions it is currently short of the state-of-the-art of forward analyses and represents an avenue for future work.

In the end, forward- and reverse- higher order program analyses are in parallel universes: many things appear the same but are subtly different, and some things that appear to be very different are in fact achieving a very similar effect. This shows up in the performance evaluation of Section 10: DDPA does much better on some examples compared to forward analyses, and much worse on others. The fact that the performance varies so widely implies their theoretical basis is also far apart, and points out that demand-driven analyses have the potential to bring new expressiveness and performance advances to higher-order program analyses.

11.1 Implementation Techniques

The technique of looking up variables on-demand and of aligning calls and returns was first developed in so-called CFL-reachability analyses for first-order languages [10, 21, 38, 40]. To solve the call-return alignment problem some reduction to grammars or automata or other formalism is needed, and several different approaches have been used. CFL-reachability reduces to a context-free language question [40], and reduction to pushdown automata formalisms were used for other first-order analyses [41]. In DDPA, we utilize the pushdown stack differently. The unbounded-stack case, ωDDPAc, is undecidable, so we needn't align calls and returns with the pushdown stack; however, we still need a continuation stack for non-locals lookup and other actions. This leads us to the use of pushdown systems in the implementation of our analysis and, ultimately, the PDR that was described in Section 8.

Considerable study ([4, 7, 26] among numerous others) has been made of pushdown-like automata and their reachability properties. Notably, although reachability on unrestricted two-stack pushdown automata is undecidable, restricted multi-stack pushdown automata have proven to be useful approximations in program analyses. One avenue of future work is the exploration of such automata as a basis for a DDPA-like analysis.

Our PDR automata were partly inspired by PDCFA [24], which computes pushdown reachability by maintaining a "compact" structure: only states stemming from the start state are analyzed. Like that work, our PDR incrementally introduces transitions according to a general schema as they become relevant to reachability. Unlike PDCFA [24], however, our PDR closure algorithm retains a schematic form of transitions as they are introduced; this leads to smaller automata and less redundant effort, something that is particularly applicable to our domain. We then utilize this mechanism to develop a form of "continuation programming" on the automaton, which can handle the more complex clauses of variable lookup (including deep pattern matching). We are unaware of any work which performs this sort of non-trivial "pushdown reachability programming," and it may be a technique applicable to other domains.

Other researchers have taken a related approach of "compiling an analysis" to logic programming or Datalog DSLs [42, 46, 56]. We expect the implementation of this paper could also be mapped onto Datalog, but we have chosen to take the PDR route as it encapsulates a simpler (polynomial vs not) complexity class and hope to thereby achieve better performance in the long run. Reduction to set constraints is also polynomial [25, 28] and so could be an alternative compilation strategy worthy of study.

11.2 Other DDPA Precursors

DDPA in fact was not derived from first-order demand-driven analyses; it emerged as a flow-sensitive extension of subtype constraint theory [2]. Each abstract program clause corresponds 1-1 to a subtype constraint; DDPA's addition is the happens-before relation which temporally relates the subtype flows.

The sharing of the CFG and the labels added to refine lookup was inspired by the sharing graphs of optimal λ-reduction [27]. In both DDPA and sharing graphs, the non-locals are not copied in but rather looked up via a careful trace back to their originating definition, with information added to paths to refine lookup accuracy – fan-in and fan-out nodes in optimal reduction, and z◖ / z◗ in DDPA.

We are not the first to be inspired by sharing graphs for program analyses; but the existing work is closer to a subtype constraint inference system than to DDPA because, while sharing graphs are used, non-local arguments are wired in directly [49].

We are not the first to propose program analyses for higher-order languages that have a "demand driven" component [9, 48]. Dubé and Feeley [9] propose an analysis which iterates between a standard forward pass and a refinement pass which re-tunes the (forward) analysis based on the results of the previous pass, demanding more precision only where it is needed, and then making another pass. This analysis is thus demand-driven in a different sense than DDPA – at the root it is a forward analysis. DDP [48] is more demand-driven in our sense in that it has a value lookup process which is explicitly goal-directed. Like Dubé it incorporates an alternating demand-driven and forward data flow algorithm, at a smaller scale. DDP is focused on object-oriented programs so is not addressing the non-locals issue of functional programs, and it is flow-insensitive. Also, DDP queries cannot share intermediary results the same way DDPA lookups do, because the DDP query that runs first may prune a subgoal that is distant from it, giving it an overapproximate solution that is trivially true, and precision on that subgoal may be essential to a later query.

12 CONCLUSION

In this paper, we have developed a demand-driven program analysis (DDPA) for higher-order programs, extending ideas of first-order demand-driven analyses to higher-order functions. The primary novelty is the use of a separate *non-locals stack* which allows call-return alignment to strictly subsume polymorphism; previous higher-order demand-driven analyses did not align non-locals. DDPA has flow-, context-, and (limited) path-sensitivity as naturally emergent properties.

We believe DDPA shows promise primarily because it represents a significantly different approach compared with the existing large literature of higher-order program analyses. A high-level analogy can be made with eager and lazy programming languages: it is a fundamental decision in language design which approach to take and there are significant trade-offs. We believe the demand-driven side of higher-order program analyses deserves further exploration.

We have established soundness of DDPA and proved a polynomial-time bound on kDDPA. The abstract call stack *must* be restricted to at most k frames, unlike in first-order demand-driven analyses: we show that ωDDPAc, DDPA with an unbounded call stack, fully and faithfully implements the λ-calculus and so is Turing-complete.

We described our implementation of DDPA which uses a novel Pushdown Reachability (PDR) automaton, a higher-order abstraction of a PDA which significantly improves the efficiency of variable value lookup. We gave a high-level specification of the implementation and presented benchmark results that shows demand-driven analyses have very different trade-offs compared to forward analyses, meaning they show promise for improving analysis expressiveness and performance. The implementation includes a broader feature set than the theoretical treatment, showing the methodology can scale.

REFERENCES

[1] Ole Agesen. 1995. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95)*. Springer-Verlag, Berlin, Heidelberg, 2–26. http://dl.acm.org/citation.cfm?id=646153.679533

[2] Alexander Aiken and Edward L. Wimmers. 1993. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*. ACM, New York, NY, USA, 31–41. https://doi.org/10.1145/165180.165188

[3] Rajeev Alur, Ahmed Bouajjani, and Javier Esparza. 2012. *Handbook of Model Checking*. Springer, Chapter Model Checking Procedural Programs, 541–572.

[4] Rajeev Alur and P. Madhusudan. 2004. Visibly Pushdown Languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*.

[5] Andrew W. Appel. 2007. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA.

[6] Ahmed Bouajjani, Javier Esparza, and Oded Maler. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of the 8th International Conference on Concurrency Theory (CONCUR '97)*. Springer-Verlag, Berlin, Heidelberg, 135–150. http://dl.acm.org/citation.cfm?id=646732.701281

[7] José Castano. 2004. *Global Index Languages*. Ph.D. Dissertation.

[8] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[9] Danny Dubé and Marc Feeley. 2002. A Demand-driven Adaptive Type Analysis. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 84–97. https://doi.org/10.1145/581478.581487

[10] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1997. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (Nov. 1997), 992–1030. https://doi.org/10.1145/267959.269970

[11] Christopher Earl, Matthew Might, and David Van Horn. 2010. Pushdown Control-Flow Analysis of Higher-Order Programs. In *Workshop on Scheme and Functional Programming*.

[12] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective Pushdown Analysis of Higher-order Programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2364527.2364576

[13] Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. 2017. Relative Store Fragments for Singleton Abstraction. In *Static Analysis*, Francesco Ranzato (Ed.). Springer International Publishing, Cham, 106–127.

[14] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. 2000. Scalable Context-Sensitive Flow Analysis Using Instantiation Constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*. ACM, New York, NY, USA, 253–263. https://doi.org/10.1145/349299.349332

[15] Matthias Felleisen and Daniel P. Friedman. 1986. Control Operators, the SECD-Machine, and the Lambda-Calculus. In *3rd Working Conference on the Formal Description of Programming Concepts*.

[16] Matthias Felleisen and Robert Hieb. 1992. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science* 103, 2 (Sept. 1992), 235–271. https://doi.org/10.1016/0304-3975(92)90014-7

[17] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*. ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[18] Kimball Germane and Matthew Might. 2017. A Posteriori Environment Analysis with Pushdown Delta CFA. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 19–31. https://doi.org/10.1145/3009837.3009899

[19] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown Control-flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '16)*. ACM, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

[20] Nevin Heintze and Olivier Tardieu. 2001. Demand-Driven Pointer Analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI '01)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

[21] 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

[22] Suresh Jagannathan and Stephen Weeks. 1995. A Unified Treatment of Flow Analysis in Higher-order Languages. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 393–407. https://doi.org/10.1145/199448.199536

[23] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. 2013. Optimizing Abstract Abstract Machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, New York, NY, USA, 443–454. https://doi.org/10.1145/2500365.2500604

[24] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. 2014. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming* 24, 2-3 (2014), 218–283. https://doi.org/10.1017/S0956796814000100

[25] John Kodumal and Alex Aiken. 2004. The Set Constraint/CFL Reachability Connection in Practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 207–218. https://doi.org/10.1145/996841.996867

[26] Salvatore La Torre and Margherita Napoli. 2011. Reachability of Multistack Pushdown Systems with Scope-Bounded Matching Relations. In *CONCUR 2011 – Concurrency Theory*.

[27] John Lamping. 1990. An Algorithm for Optimal Lambda Calculus Reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '90)*. ACM, New York, NY, USA, 16–30. https://doi.org/10.1145/96709.96711

[28] David Melski and Thomas Reps. 1997. Interconvertbility of Set Constraints and Context-Free Language Reachability. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '97)*. ACM, New York, NY, USA, 74–89. https://doi.org/10.1145/258993.259006

[29] Jan Midtgaard. 2012. Control-flow Analysis of Functional Programs. *ACM Computing Surveys (CSUR)* 44, 3, Article 10 (June 2012), 33 pages. https://doi.org/10.1145/2187671.2187672

[30] Matthew Might. 2007. *Environment Analysis of Higher-order Languages*. Ph.D. Dissertation. Atlanta, GA, USA. Advisor(s) Shivers, Olin G. AAI3271560.

[31] Matthew Might. 2010. Abstract Interpreters for Free. In *Proceedings of the 17th International Conference on Static Analysis (SAS'10)*. Springer-Verlag, Berlin, Heidelberg, 407–421. http://dl.acm.org/citation.cfm?id=1882094.1882119

[32] Matthew Might and Olin Shivers. 2006. Environment Analysis via ΔCFA. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 127–140. https://doi.org/10.1145/1111037.1111049

[33] Matthew Might and Olin Shivers. 2006. Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/1159803.1159807

[34] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

[35] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

[36] 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

[37] Jakob Rehof and Manuel Fähndrich. 2001. Type-base Flow Analysis: From Polymorphic Subtyping to CFL-reachability. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '01)*. ACM, New York, NY, USA, 54–66. https://doi.org/10.1145/360204.360208

[38] Thomas Reps. 1995. Shape Analysis As a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM '95)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/215465.215466

[39] Thomas Reps. 2000. Undecidability of Context-sensitive Data-dependence Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 22, 1 (Jan. 2000), 162–186. https://doi.org/10.1145/345099.345137

[40] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*. ACM, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[41] Thomas Reps, Akash Lal, and Nick Kidd. 2007. Program Analysis Using Weighted Pushdown Systems. In *Proceedings of the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'07)*.

Springer-Verlag, Berlin, Heidelberg, 23–51. http://dl.acm.org/citation.cfm?id=1781794.1781799

[42] Thomas W. Reps. 1995. *Demand Interprocedural Program Analysis Using Logic Databases.* Springer US, Boston, MA, 163–196. https://doi.org/10.1007/978-1-4615-2207-2_8

[43] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-Driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP '05).* ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

[44] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages.* Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[45] Jeffrey Mark Siskind. 1999. *Flow-Directed Lightweight Closure Conversion.* Technical Report.

[46] Yannis Smaragdakis and Martin Bravenboer. 2011. Using Datalog for Fast and Easy Program Analysis. In *Proceedings of the First International Conference on Datalog Reloaded (Datalog'10).* Springer-Verlag, Berlin, Heidelberg, 245–251. https://doi.org/10.1007/978-3-642-24206-9_14

[47] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. 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, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

[48] S. Alexander Spoon and Olin Shivers. 2004. Demand-Driven Type Inference with Subgoal Pruning: Trading Precision for Scalability. In *European Conference on Object-Oriented Programming (ECOOP)*, Martin Odersky (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 51–74.

[49] David Van Horn and Harry G. Mairson. 2007. Relating Complexity and Precision in Control Flow Analysis. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07).* ACM, New York, NY, USA, 85–96. https://doi.org/10.1145/1291151.1291166

[50] David Van Horn and Harry G. Mairson. 2008. Deciding kCFA is Complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08).* ACM, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243

[51] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *Proceedings of the 19th European Conference on Programming Languages and Systems (ESOP'10).* Springer-Verlag, Berlin, Heidelberg, 570–589. https://doi.org/10.1007/978-3-642-11957-6_30

[52] Dimitrios Vardoulakis and Olin Shivers. 2011. Pushdown Flow Analysis of First-class Control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11).* ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/2034773.2034785

[53] Christopher P. Wadsworth. 1971. *Semantics and Pragmatics of the Lambda-calculus.* Ph.D. Dissertation. University of Oxford.

[54] Stephen Weeks. 2006. Whole-program Compilation in MLton. In *Proceedings of the 2006 Workshop on ML (ML '06).* ACM, New York, NY, USA, 1–1. https://doi.org/10.1145/1159876.1159877

[55] Qirun Zhang and Zhendong Su. 2017. Context-sensitive Data-dependence Analysis via Linear Conjunctive Language Reachability. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017).* ACM, New York, NY, USA, 344–358. https://doi.org/10.1145/3009837.3009848

[56] Xin Zhang, Mayur Naik, and Hongseok Yang. 2013. Finding Optimum Abstractions in Parametric Dataflow Analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 365–376. https://doi.org/10.1145/2491956.2462185