Higher-Order Demand-Driven Program Analysis
(draft)

Zachary Palmer[1] and Scott F. Smith[2]

1 **Swarthmore College**
Swarthmore, PA, USA
zachary.palmer@swarthmore.edu
2 **The Johns Hopkins University**
Baltimore, MD, USA
scott@cs.jhu.edu

Abstract

We explore a novel approach to higher-order program analysis that brings ideas of on-demand lookup from first-order CFL-reachability program analyses to higher-order programs. The analysis needs to produce *only* a control-flow graph; it can derive all other information including values of variables directly from the graph. Several challenges had to be overcome, including how to build the control-flow graph on-the-fly and how to deal with nonlocal variables in functions. The resulting analysis is flow- and context-sensitive with a provable polynomial-time bound. The analysis is formalized and proved correct and terminating, and an initial implementation is described.

1998 ACM Subject Classification F.3.2. Semantics of Programming Languages

Keywords and phrases program analysis, polynomial, demand-driven, flow-sensitive, context-sensitive

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Traditional flow analysis for imperative first-order programs is a relatively straightforward process. Before the analysis starts, it is known which functions are being invoked at each call site in the source program. This is possible because, in absence of higher-order functions, potential control flow is determined immediately from the structure of the program. For first-order programs, it is possible to build a fixed control flow graph (CFG) where each edge in the CFG points to a potential next program point. The program analysis then can monotonically accumulate information about what values program variables could take on at each program point, propagating information along the fixed CFG.

Forward higher-order program analysis

In languages with higher-order functions, program analysis is much more challenging: it is no longer obvious which functions may be invoked at each call site. The program's data flow determines which functions appear at the call site, in turn influencing the program's control flow. Accurate analyses of programs with higher-order functions must therefore compute data- and control-flow information simultaneously [12, 16].

Higher-order program analyses are generally based on abstract interpretations [2]; 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: concrete program points,

Conference title on which this volume is based on.
Editors: Billy Editor and Bill Editors; pp. 1–29

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

environments, stacks, stores, and addresses are replaced with abstract counterparts which have finite cardinality, essentially "hobbling" the full operational semantics of the language to guarantee termination of the analysis [13]. A sound analysis will eventually visit the (finitely many) abstract counterparts of the reachable concrete program states, producing a finite automaton representing all potential program runs.

We use the term "forward analyses" to refer to standard higher-order program analyses, to emphasize how they propagate data forward through the program in the same manner as an operational semantics does. All higher-order program analyses we are aware of, including [12, 16, 21, 13, 27, 8, 15, 3], are forward analyses.

Since each node is an abstract state and there can be a great many combinations of data for environment or store, even the finitized abstract state space can be very large. For this reason, any practical analysis must compress the total number of states. Typically, the program counter / program point is always preserved. One standard compression is to replace the store in each abstract program state with a single global store; this is called *store widening* since the global store is the union of all the individual stores.

Store widening is effective in compressing the abstract state set but tends to give too little precision. For example, function polymorphism which is present before store widening may be lost as the parameters to the calls of a given function in each program state are unioned together. Store widening also generally loses any so-called flow sensitivity, where a given (often stateful) variable can take on different values at different points.

So, other methods are employed to recover expressiveness without a full store in each abstract state. One method is polyvariance, such as in kCFA [21] for $k > 0$, where each function parameter gets unique addresses relative to the k most recent stack frames. Another method is call-return alignment: the analysis uses a stack internally to force returns to always return only to the call site of the particular call being analyzed [26]. Call-return alignment also gives some, but not all, of the power of polyvariance. One alternative to store widening is abstract garbage collection [14], which also limits the number of abstract states: the store is not widened, but an analysis-time analogue of run-time garbage collection is introduced to remove from the store the bindings that are no longer accessible. Abstract GC preserves the flow-sensitivity but, since store widening is not performed, this technique is still susceptible to an explosion in analysis states for programs with nontrivial amounts of persistent data as each configuration of this data represents another abstract program state to consider.

A demand-driven approach to higher-order programs

In this paper, we explore a fundamentally different approach to higher-order program analysis which does not proceed by finitizing an operational semantics. Rather than pushing data forward as in an operational semantics, the analysis looks up data on demand. This variable lookup is accomplished by walking backward along the control flow from the point at which the variable's value is needed to search for the point at which the variable was most recently defined.

Demand-driven variable lookup is found in demand-driven CFL-reachability analyses [6]; but, that work is carried out in a first-order context, where the CFG is already known and where there are no nonlocal variables in functions. Demand-driven lookup is applied to higher-order languages in [17], but that is a context of a flow-insensitive analysis with other restrictions. To generally apply this technique to higher-order programs, we must have (1) a technique for constructing the CFG in tandem with the data flow analysis and (2) a strategy for handling non-local variables.

In this paper, we solve these two problems to produce a higher-order Demand-Driven

Program Analysis (DDPA). The CFG is constructed incrementally just as in a forward higher-order program analysis. But, unlike a forward analysis, we *only* need to construct the CFG; there is *no* abstracted environment/store/etc as all of the variable lookup happens in reverse by following CFG edges backwards. To deal with nonlocal variables in functions, we develop a novel reverse lookup process that is related to how access links are used in a compiler to look up nonlocal variables: we first find where the function itself was defined and then resume looking up the variable from there. For added precision, we also incorporate the call-return alignment idea from CFL-reachability [20, 19] and pushdown higher-order flow analyses [26, 8]: we verify that the lookup paths we take through the CFG match each function return with its corresponding call.

Contrast with forward analyses

During the course of DDPA, each variable lookup occurs relative to a particular point in the (partial) control flow graph where the variable is used; this makes flow-sensitivity a natural component of DDPA. In forward higher-order analyses, such flow-sensitivity is expensive since it prohibits common performance-enhancing techniques such as store widening. Abstract garbage collection, an aforementioned alternative to store widening, prunes the stores significantly but produces many states for persistent data. Additionally, there is no need for explicit polyvariance in DDPA: the combination of call-return alignment and nonlocal lookup achieves the full effect of polyvariance without an explicit polyvariance model *a la* kCFA. DDPA appears to compare favorably to forward analyses: it has the simultaneous benefits of store widening (we have no per-node store), abstract garbage collection (demand-driven lookup prevents the generation of garbage in the first place), call-return alignment (the one feature we directly adopt) and polyvariance (which is subsumed by our call-return alignment given our novel non-local variable lookup mechanism). And, forward analyses cannot combine all these features because abstract GC is ineffective in the presence of store widening [8].

In theory, DDPA has minimal requirements on run-time data structures: the only required data structure is the CFG, which is minuscule compared to the formal treatment of forward analysis. In practice, an efficient DDPA implementation requires significant caching structures which are similar to *fragments* of an abstract store in a forward analysis. The practical difference between DDPA and forward analyses is similar to the practical difference between e.g. the parameter-passing mechanisms of Haskell and ML: we do not assert DDPA to be a strictly better form of program analysis, only a substantially *different* one with different trade-offs and worthy of study.

In terms of technical overhead, DDPA has the advantage over forward analyses of a very simple formal specification of a flow-, path-, and context-sensitive analysis. On the other hand, the analysis' reverse lookup is more distant from the original operational semantics and requires a different sort of thinking; a forward analysis can to some degree be "read off the operational semantics" [25, 13].

We have completed an implementation of DDPA, described in Section 6, and include it as non-anonymized supplementary material with this paper. The implementation builds a push-down automaton (PDA) and variable lookup questions can be cast as reachability questions in the PDA. This implementation has not been optimized but it confirms that the analysis has the expected behavior on examples.

e	::=	$[c, \ldots]$	*expressions*	v	::=	$r \mid f$	*values*
c	::=	$x = b$	*clauses*	r	::=	$\{\ell, \ldots\}$	*records*
b	::=	$v \mid x \mid x\ x \mid x \sim p\ ?\ f : f$	*clause bodies*	f	::=	`fun` x `->` `(` e `)`	*functions*
x			*variables*	p	::=	r	*patterns*
E	::=	$[x = v, \ldots]$	*environments*				

Figure 1 Expression Grammar

Paper outline

Section 2 presents an example-driven description of the main features of the analysis. Section 3 gives the full details, and Section 4 establishes its soundness relative to an operational semantics. Section 5 defines a few extensions not in the core theory, including full records, path-sensitivity, and state. Section 6 describes the implementation, Section 7 covers related work, and we conclude in Section 8.

2 Overview of the Analysis

This section informally presents DDPA by example.

2.1 A Simple Language

For a given program, our objective is to establish the possible execution paths that the program will take. We encode this information in the form of a CFG, more precisely as a *happens-before* relation $c \ll c'$: program point c related to c' means there may be control flow from c to c'.

The simple functional language we use is defined in Figure 1. To make it easier to keep track of program points and operation sequencing, we restrict our language syntax to a shallow A-normal form (ANF) [5]. We use clauses c to themselves denote program points; we require all clause variables to be unique in a program to allow this to be unambiguous. (Notation: we write lists of length n as $[g_1, \ldots, g_n]$, and use $||$ for list concatenation. We may write concatenation of a non-list as shorthand for singleton list concatenation when it is unambiguous: $g_1 \,||[g_2, g_3] = [g_1, g_2, g_3]$.)

The operational semantics of this language are straightforward and are given in Section 4.1. Note that we have a degenerate form of record with label components only; this restriction is made to simplify the formalism and we relax this restriction in the implementation. Case analysis is written $x \sim p\ ?\ f_1 : f_2$; we execute f_1 with x as argument if the value of x matches the pattern p, and f_2 with x as argument otherwise.

2.2 The Basic Analysis

Consider the program in Figure 2. Note that `f` is just a fancy identity function used to help illustrate the analysis, and that clause `n0` is a "no-op": it is present only to help clarify the diagrams.

The analysis begins by constraining the top-level clauses to happen in order. Here, clauses defining `f`, `x1`, `z1`, `x2`, `z2` are constrained to happen in that order. For technical reasons, we also add distinguished start and end nodes before `f` and after `z2`, respectively.

Figure 3 shows the completed analysis; up to now we have only described the bottom row. Focusing only on that row for now, the solid arrows ($\longrightarrow$) represent nodes with a $\ll$

```
1  f = fun x -> ( # λx.(λy.y)x
2      i = fun a -> (
3          r2 = a
4      );
5      r = i x;
6      n0 = r;
7  );
8  x1 = {y};
9  z1 = f x1;      # evaluates to {y}
10 x2 = {n};
11 z2 = f x2;      # evaluates to {n}
```

Figure 2 Invocation Example: ANF

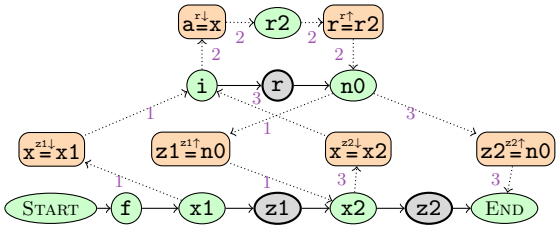

Figure 3 Invocation Example: Analysis

relationship between them based on clause ordering in the source program. As shorthand, we identify each clause by the unique variable it assigns; thus, the green node labeled `x1` refers to line 8 in Figure 2. Green nodes are atomic clauses, and gray nodes are call sites. The initial (⟶) arrows represent what we know about control flow at the beginning of the analysis: only that control may eventually move from each clause to the next.

2.2.1 Adding control flows for function calls

To develop our call graph, we must connect each call site to any function body invoked at that call site; once all such connections are made, we will have a correct CFG for our program. Proceeding forward on the control flow from the program start, the first call site (gray node) is node `z1`. The dotted edges (⇢) annotated with the number "1" are added to the graph to represent control passing from call site `z1` to the start of the body of function `f` (the `i`, `r`, `n0` clauses), and from the end of the function body back to `z1`. Note that we in fact run the wires "around" the call site itself in the figure, going out the predecessor of `z1` and back to the successor. This reflects how a function inlining would work. (Aside: both ⟶ and ⇢ edges define $\ll$ relationships between program points; the two edge sorts are used for illustration only.)

Additionally, the call adds two *new* intermediate nodes: `x=x1` represents copying the argument `x1` at the call site into the function `f`'s parameter, while `z1=n0` represents copying the result of the function into the variable defined at the call site. These nodes are marked with annotations to indicate their call site and purpose; `z1`↓, for instance, indicates a parameter passed in from the `z1` call site. All of the information in these new nodes can in fact be inferred from context since all values are unique given a (call site, function) pair. In other words, the happens-before relation is isomorphic to the call graph.

Continuing with the analysis: now that the body of `f` has been wired in, the call site `r` can execute, adding two nodes and the "2" flows. The "3" flows are similarly added when we elaborate `z2`.

2.2.2 Variable lookup as reverse walk

In the description above, we glossed over how variable lookup occurs: for each call site, we needed to look up the particular functions to invoke and this can be nontrivial in a higher-order language. The clause `z1` invokes `f`, and all potential definitions of `f` need to be wired to the call site. This particular case is simple since `f` obviously has only one value.

In general, variable lookup is *contextual*: lookup starts at the clause where the variable

is being used and proceeds backwards in time with respect to the control flow to find the most recent value. This approach to lookup is related to the demand-driven optimizations of first-order CFL-reachability analyses [6].

For a less trivial example, let us look up `x` from the perspective of `r`'s defining clause in line 5. Walking back though the graph from `r`, there are two definitions of `x` reached: `x=x1` and `x=x2`. Since `x1`/`x2` are not concrete values, lookup proceeds by looking up those variables, respectively. Ultimately, `x` has final value set {`{y}`, `{n}`}, the set of arguments the function was invoked on.

Since variable lookup search always starts at the node representing the redex where the variable is used, we obtain a flow-sensitive analysis by default. One important property of our graph is we do *not* transitively close over $\ll$ edges; while locally it makes sense, a transitively closed graph would collapse all cycles and the analysis would lose all flow-sensitivity upon recursion.

2.3 Constraining lookup to reasonable call stacks

The above description of a simple lookup leaves out an important refinement in DDPA: it is possible to rule out variable lookup search paths on which calls and returns do not align, meaning the path provably corresponds to no program execution. To show the incompleteness of lookup as described thus far, consider how lookup would find values for `z2` from the perspective of the end of the program. To find `z2`, we proceed back on the control flow to `z2=n0` (recall how the call site itself is dead code after all calls have been wired in), at which point the search is now for the value of `n0`; continuing back we reach `n0=r` and proceed by looking up `r` from node `n0`, and so on until we are looking up `x` from node `i`. Here, there are two paths, to either `x=x1` or `x=x2`. So, we take both paths and union the result, obtaining {`{y}`, `{n}`}. This is clearly a loss of precision: `{y}` cannot appear as the argument of call site `z2` at runtime. But, looking at the overall path we took above in this spurious lookup, we walked back into the function via the `z2` call site and came out of the front of the function to the `z1` call site. This is a non-sensical program execution path since the call and return site are not aligned.

The annotations `z1`↓ / `z1`↑ on the wiring nodes indicate a call into and return from call site `z1`, respectively, and are used to filter out these spurious paths. On any lookup path the wiring annotations must pair correctly; that is, one may only visit a `z1`↓ node if a corresponding `z1`↑ node was visited. We track these pairings using an abstract call stack to verify they obey a reasonable call-return discipline. The idea of call-return alignment is taken from CFL-reachability [20, 19] and higher-order pushdown analyses [26, 8].

Note that the above analysis exhibits polyvariant behavior: different invocations of the same function are not analyzed uniformly. Polyvariance is commonly achieved by copying, kCFA being a canonical example [16]. In DDPA, polyvariance can be achieved solely by call-return alignment. This is also the case in first-order analyses; in CFL-reachability, call-return alignment is termed a polyvariance method. But in higher-order programs, a forward analysis will not give perfect polyvariance with only call-return alignment due to lack of polymorphism on nonlocals which are not directly on the call-return path.

2.4 Looking up Non-Local Variables

So far, our examples of lookup have been restricted to local variables. In these cases, variable assignments can transitively be followed back to a value. However, this process does not accurately reflect how non-local variables are bound: if non-local variables were handled in

```
1  k = fun v -> ( # λx.λy.x
2      k0 = fun j -> (
3              r = v;
4      );
5  );
6  a = {a};
7  f = k a;        # λy.{a}
8  b = {b};
9  g = k b;        # λy.{b}
10 e = {};
11 z = f e;        # evaluates to {a}
```

Figure 4 Non-Local Variable: ANF

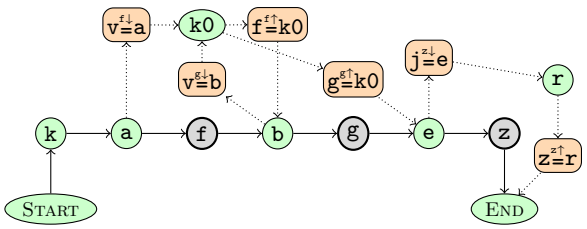

Figure 5 Non-Local Variable: Analysis

the same way, the lookup operation would interpret non-locals as being dynamically scoped. To see why, consider the code example in Figure 4 and the corresponding analysis graph in Figure 5. If we look up the value of `z` from the end of the program, we find ourselves asking for the values of `v` at the wiring of `j=e`. Proceeding as described above, `j` does not match `v` and we would attempt to find a value for `v` at the `e` clause. This leads us to the clause `v=b`, which in turn gives us the value `{b}`. This would be unsound as it is not consistent with run-time behavior.

To give standard lexical scoping semantics, the lookup function must be modified. Lexical scoping means we want the values of nonlocal variables at the point which the function containing them was *itself* declared, so we proceed by first locating the function definition and then resuming our search for the value of the variable. This is similar to the access link method of nonlocal lookup in a compiler implementation. In the example above, the pivotal decision is made when finding values of `v` when we have searched through the whole body of the function in lines 2-4 and arrived at the argument wiring `j=e`. From the annotation `z`↓ on this node, we can deduce that we are leaving the current function which was called from site `z`. Since we have not yet found `v`, it must be a non-local. To proceed with lookup, we then delay our search for `v`, examine the call site `z` (appearing in the wiring annotation), and perform a lookup of the *function* invoked at that point; here, this means to search for `f` starting from `e`. This search leads us through `f=k0` to `k0`, the point at which the function was originally defined, and from here the search for `v` can soundly resume. Using this approach, the lookup of `z` from the end of the program yields `{a}`.

The general case of nonlocal lookup chains on the above idea, since the defining function could itself be non-locally defined. The chain is up the lexical scoping structure, so its length is bound by program size.

2.5 Looking up Higher-Order Functions

The analysis described so far does not handle well the case of multiple functions showing up at a higher-order call site. It does not take in account the functions that are actually available. Consider the code example in Figure 6. It starts by defining a higher-order function `h`, which takes a function `f` as parameter and calls it with the empty record as an argument. Then, the program defines two functions `fa` and `fb` that ignore their parameters and always return `{a}` and `{b}`, respectively. Finally, there are two calls to the higher-order function `h` with `fa` and `fb` as parameters, in turn. (The `n`*i* clauses are "no-op", only present to improve readability of the analysis.) As a result, there are two functions that can show up at the `hr` call site – namely, `fa` and `fb`. The analysis in Figure 7 depicts this by showing

```
1  h = fun f -> ( # λf. f {}
2      e = {};
3      hr = f e;
4      n0 = hr;
5  );
6
7  fa = fun ia -> ( a = {a}; ); # λ_. {a}
8  fb = fun ib -> ( b = {b}; ); # λ_. {b}
9
10 ra = h fa;
11 n1 = {};
12 rb = h fb;
```

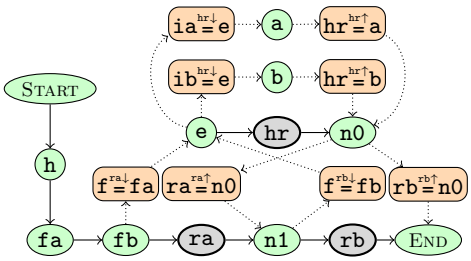

Figure 6 Higher-Order Functions: ANF **Figure 7** Higher-Order Functions: Analysis

two call edges flowing out of the `p` node.

We would expect the set of values that could appear in `rb` (from the END) to be {`{b}`}, as confirmed by manually inspecting the source code. In the process of trying to answer that question, the analysis eventually gets to node `n0`, querying for the variable `hr`. At that point, there are two paths by which to proceed, one going to $\texttt{hr}^{\texttt{hr}\uparrow}\texttt{a}$ and another going to $\texttt{hr}^{\texttt{hr}\uparrow}\texttt{b}$. If the analysis carried on, following both paths, it would answer {`{a}, {b}`}, but `{a}` is a value that can never be the result of the presented computation. This loss of precision results from taking *all* possible paths when entering a function call, regardless of the actual functions that actually show up at the site.

We address this case as follows. Before entering a function call, the analysis first filters out the paths that do not correspond to possible executions of the program by starting a new lookup for the functions that can show up at the node it is currently considering. In the given example, at the node `n0`, the analysis looks for the values of `f`. In order to align with the $\texttt{rb}^{\texttt{rb}\uparrow}\texttt{n0}$ entrance node, the only possible answer is {`fb`}, which comes from the node $\texttt{f}^{\texttt{rb}\downarrow}\texttt{fb}$. The analysis does not explore the $\texttt{hr}^{\texttt{hr}\uparrow}\texttt{a}$ node (as it does not correspond to the {`fb`} result) but follows only the $\texttt{hr}^{\texttt{hr}\uparrow}\texttt{b}$ node. This way, the analysis answers that the set of possible values for `rb` from the END is just {`{b}`}, matching our expectations.

2.6 Recursion and Decidability

The analysis described above naturally analyzes recursive programs: consider the program in Figure 8 which, for illustrative purposes, we code in an extension of the presentation language including proper records. This program defines a function `f` which is recursive (by self-passing) and which deeply projects any number of `l` fields from a record; that is, `f {l:{l:{a:{}}}}` would return `{a:{}}` as that is the outermost level without an `l` label. (Note: this example program contains several "no-op" clauses (named `n`i) which only serve to make the diagrams easier to follow.)

We give the result of DDPA analysis of this program in Figure 9. Lookup generally proceeds via the same process as for the previous examples. The only complication with recursion is that some control flow paths could be cyclic. For instance, the path `n0` $\rightsquigarrow$ `n2` $\rightsquigarrow$ `v` $\rightsquigarrow$ `n0` is a cycle representing an arbitrary number of recursive unwrappings, each of which pushes `r`$\downarrow$ and `r1`$\downarrow$ onto the stack. The specification of the analysis only requires variable lookup paths to be balanced in calls and returns, so there are arbitrarily many possible paths. It is clear in this example that the number of times around the cycle can be bounded without changing the result, but it is not known whether lookup over an arbitrary stack is

```
f = fun s -> ( # l-projecting function
    f0 = fun a -> (
        n0 = {};
        r = a ~ {l}
          ? fun a1 -> ( n2 = {};
                ss = s s;  # self-apply
                v = a1.l;  # project l
                r1 = ss v; # recurse
                n3 = r1; )
          : fun a2 -> ( r2 = a2; );
        n1 = r;
    );
);
ff = f f;     # initial self-application
x1 = {};
x2 = {l=x1};
x3 = {l=x2};  # the record {l={l={}}}
z = ff x3;    # evaluates to {}
```

Figure 8 Recursion: ANF

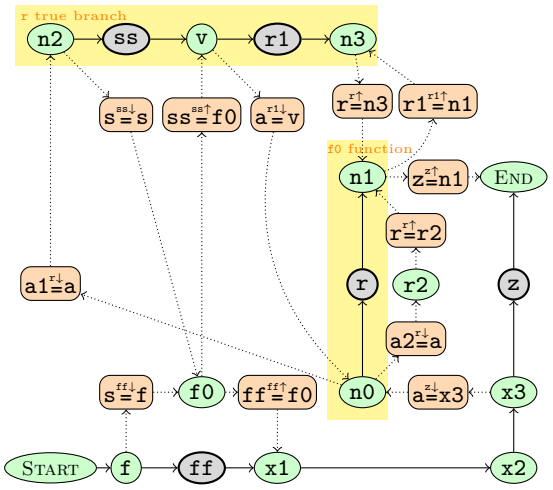

Figure 9 Recursion: Analysis

computable. We address this issue by defining a stack finitization kDDPA which retains the k most recent contexts in the spirit of kCFA [21]; see Section 3. Related restrictions on stack alignment need to be placed in a higher-order analysis with abstract garbage collection [8], a topic discussed further in the related work.

DDPA is flow-sensitive and context-sensitive. However, it is not path-sensitive; that is, a variable's values are not considered in terms of how branching decisions at conditionals led to a particular program point. We can see an example of this by considering the values of `z` which may appear at the end of the program of Figure 8. According to the rules above, the path END $\rightsquigarrow$ `n1` $\rightsquigarrow$ `r2` $\rightsquigarrow$ `n0` $\rightsquigarrow$ `x3` in Figure 9 is valid, but this would imply that the value `{l={l={}}}` could reach the end of the program. In reality, this path would never occur with the value `{l={l={}}}` because the conditional clause at `r` would not direct that value to the second branch; DDPA is ignorant of the decisions that conditional clauses make. Our current implementation has this incompleteness, but path sensitivity in principle is not difficult to add, and we present the specification for such an extension in Section 5.2. Note this example also uses "real" records and not just the label sets of our grammar of Figure 1; we describe the extension to full records in Section 5.1 and they are also included in our current implementation.

3 The Analysis Formally

In this section we formalize the analysis algorithm described in the previous section. The operational semantics for our language is standard and so we postpone it and the analysis soundness proof to Section 4.

The grammar constructs needed for the analysis appear in Figure 10. The items on the left are just the hatted versions of the corresponding program syntax (and in fact are identical for this particular analysis; we use separate forms to disambiguate analysis and runtime assertions). Edges in a dependency graph $\hat{D}$ are dependencies $\hat{d}$, are written $\hat{a} \ll \hat{a}'$ and mean atomic clause $\hat{a}$ happens right before clause $\hat{a}'$. New clause annotations $\hat{c}\downarrow$ / $\hat{c}\uparrow$ are used to mark the entry and exit points for functions/cases.

$$
\begin{array}{llll}
\hat{e} & ::= & [\hat{c}, \ldots] & \textit{abstract expressions} \\
\hat{c} & ::= & \hat{x} = \hat{b} & \textit{abstract clauses} \\
\hat{b} & ::= & \hat{v} \mid \hat{x} \mid \hat{x}\ \hat{x} \mid & \textit{abs. bodies} \\
 & & \hat{x} \sim \hat{p}\ ?\ \hat{f} : \hat{f} & \\
\hat{x} & ::= & x & \textit{abstract variables} \\
\hat{v} & ::= & \hat{r} \mid \hat{f} & \textit{abstract values} \\
\hat{p} & ::= & \hat{r} & \textit{abstract patterns} \\
\hat{f} & ::= & \texttt{fun}\ \hat{x}\ \texttt{->}\ (\ \hat{e}\ ) & \textit{abs. functions} \\
\hat{r} & ::= & r & \textit{abstract records}
\end{array}
\qquad
\begin{array}{llll}
\hat{V} & ::= & \{\hat{v}, \ldots\} & \textit{abstract value sets} \\
\hat{a} & ::= & \hat{c} \mid & \textit{abs. annotated clauses} \\
 & & \hat{x} \overset{\hat{c}\downarrow}{=} \hat{x} \mid \hat{x} \overset{\hat{c}\uparrow}{=} \hat{x} \mid \textsc{Start} \mid \textsc{End} & \\
\hat{d} & ::= & \hat{a} \lll \hat{a} & \textit{abstract dependencies} \\
\hat{D} & ::= & \{\hat{d}, \ldots\} & \textit{abs. dependency graphs} \\
\hat{X} & ::= & [\hat{x}, \ldots] & \textit{abs. var lookup stacks}
\end{array}
$$

Figure 10 Analysis Grammar

▶ **Definition 3.1.** We use the following notational sugar with respect to chronological graph dependencies:

- We write $\hat{a}_1 \lll \hat{a}_2 \lll \ldots \lll \hat{a}_n$ to mean $\{\hat{a}_1 \lll \hat{a}_2, \ldots, \hat{a}_{n-1} \lll \hat{a}_n\}$.
- We write $\hat{a}' \lll \{\hat{a}_1, \ldots, \hat{a}_n\}$ (resp. $\{\hat{a}_1, \ldots, \hat{a}_n\} \lll \hat{a}'$) to denote $\{\hat{a}' \lll \hat{a}_1, \ldots, \hat{a}' \lll \hat{a}_n\}$ (resp. $\{\hat{a}_1 \lll \hat{a}', \ldots, \hat{a}_n \lll \hat{a}'\}$).
- We write $\hat{a} \lll\!\!\lll \hat{a}'$ to mean $\hat{a} \lll \hat{a}' \in \hat{D}$ for some graph $\hat{D}$ understood from context.
- We define abbreviations $\textsc{Preds}(\hat{a}) = \{\hat{a}' \mid \hat{a}' \lll\!\!\lll \hat{a}\}$ and $\textsc{Succs}(\hat{a}) = \{\hat{a}' \mid \hat{a} \lll\!\!\lll \hat{a}'\}$.

We write $\textsc{Embed}(e)$ to denote the initial embedding of an expression e into a dependency graph $\hat{D}$.

▶ **Definition 3.2.** $\textsc{Embed}([c_1, \ldots, c_n])$ is the graph given by $\textsc{Start} \lll \hat{c}_1 \lll \ldots \lll \hat{c}_n \lll \textsc{End}$, where each $\hat{c}_i = c_i$.

This initial graph is simply the linear sequence of clauses in the "main program". The graph will then grow as the program executes.

While we give no examples in this section, the examples in the overview including Section 2.2 are fully specified dependency graphs w.r.t. the notation of this section and the reader is invited to reference those examples for intuitions.

3.1 Lookup

As was described in Section 2, the analysis will look back along $\lll$ edges in the graph $\hat{D}$ to search for definitions of variables it needs, such as the particular function to invoke at a call site. Here we define the appropriate lookup function for this task.

3.1.1 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. Overview Section 2.3 gave intuitions of how this stack helps refine the analysis result.

Our 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 3.3 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 3.3.** A *context stack model* $\Sigma = \langle \hat{\boldsymbol{C}}, \epsilon, \textsc{Push}, \textsc{Pop}, \textsc{IsTop} \rangle$ obeys the following laws:

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

The context stack is somewhat unusual in that it represents the calls of which we are *certain*; thus, the empty stack represents no knowledge (rather than no stack frames). For this reason, popping from the empty stack yields the empty stack and any clause is considered to be on top of an empty stack.

3.1.2 Lookup stacks

Lookup also proceeds with respect to a *non-locals lookup stack* $\hat{X}$ which 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. In the general case it is a stack since the function itself could prove to be a nonlocal, etc. This stack can be unbounded unlike the context stack above. Section 2.4 provides motivation and examples for non-local variable lookup.

3.1.3 Defining the lookup function

We are finally ready to define contextual lookup in the dependency graph, the central operation of this analysis.

Lookup finds the value of a variable starting from a given graph node. Given a dependency graph $\hat{D}$, we write $\hat{D}(\hat{x}, \hat{a}_0, \hat{X}, \hat{C})$ to denote the lookup of a definition of the variable $\hat{x}$ in $\hat{D}$ relative to graph node $\hat{a}_0$, access stack $\hat{X}$ and context stack $\hat{C}$. The latter two parameters are empty for top-level lookups: we let $\hat{D}(\hat{x}, \hat{a}_0)$, the lookup of $\hat{x}$ from graph node $\hat{a}_0$, abbreviate $\hat{D}(\hat{x}, \hat{a}_0, [], \epsilon)$. Note that we have oriented the definition so looking from graph node $\hat{a}_0$ means we are not looking for the value in that node itself, but in (all) predecessors of it.

▶ **Definition 3.4.** Given dependency graph $\hat{D}$, $\hat{D}(\hat{x}, \hat{a}_0, \hat{X}, \hat{C})$ is the function returning the least set of values $\hat{V}$ satisfying the following conditions:

1. If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{v})$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{X} = []$, then $\hat{v} \in \hat{V}$.
2. If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{f})$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{X} = [\hat{x}_1, \ldots, \hat{x}_n]$ for $n > 0$, then $\hat{D}(\hat{x}_1, \hat{a}_1, [\hat{x}_2, \ldots, \hat{x}_n], \hat{C}) \subseteq \hat{V}$.
3. If $\hat{a}_1 = (\hat{x} \texttt{=} \hat{x}')$ and $\hat{a}_1 \lll \hat{a}_0$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

4a. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c}$ is a function call and $\textsc{IsTop}(\hat{c}, \hat{C})$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.

4b. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c}$ is a case clause, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

5a. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c} = (\hat{x}_r \texttt{=} \hat{x}_f\ \hat{x}_v)$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Push}(\hat{c}, \hat{C})) \subseteq \hat{V}$, provided $\texttt{fun}\ \hat{x}'' \texttt{->} (\hat{e}) \in \hat{D}(\hat{x}_f, \hat{c}, [], \hat{C})$ and $\hat{x}' = \textsc{RV}(\hat{e})$.

5b. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$ and $\hat{c}$ is a case clause, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

6. If $\hat{a}_1 = (\hat{x}'' \texttt{=} b)$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{x}'' \neq \hat{x}$, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

7a. If $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{x}'' \neq \hat{x}$, $\hat{c} = (\hat{x}_r \texttt{=} \hat{x}_f\ \hat{x}_v)$, and $\textsc{IsTop}(\hat{c}, \hat{C})$, then $\hat{D}(\hat{x}_f, \hat{a}_1, \hat{x} \,||\, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$.

7b. If $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{x}'' \neq \hat{x}$ and $\hat{c} = (\hat{x}_1 \texttt{=} \hat{x}' \texttt{~} \hat{p} \texttt{?} \hat{f}_1 \texttt{:} \hat{f}_2)$, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

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

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 $\hat{x}$ we are searching for has a function value, and unlike clause 1. there is a non-empty lookup stack. This means the variable on top of the lookup stack, $\hat{x}_1$, was a non-local and was pushed on to the non-local stack while searching for the definition of the function it resides in. That function definition, $\hat{x} = \hat{f}$, has now been found, and so we may continue to search for $\hat{x}_1$ from the current point in the graph.

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 $\hat{x}'$.

4a. 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 IsTop clause constrains this stack frame exit to align with the frame we had last entered (in reverse).

4b. This is the case clause version of the previous. Case clauses can be viewed as inlined functions aligned by program context so use of $\hat{C}$ is not necessary for alignment.

5a. 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.

6. Here the previous clause is not a match so the search continues at any predecessor node. Note this will chain past function/match 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 5.3, we will enter such a function to verify an alias to our variable was not assigned.

7a. The precondition means 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 the function body we are exiting, which is $\hat{x}_f$, and push $\hat{x}$ on the nonlocals stack. Once the defining point of $\hat{x}_f$ is found, $\hat{x}$ will be popped from the nonlocals stack and we will resume searching for it. The IsTop clause constrains the stack frame being exited to align with the frame we had last entered (in reverse).

7b. This is the case clause variation of the previous; as with 4b. above, the stack is not needed for conditional alignment – there is syntactically only one entry and exit.

We encourage the reader to run lookup examples described in Section 2 through the above definition to clarify how it works. In Section 3.3 below we establish the computability of lookup for finitely-bounded stacks.

3.2 Abstract Evaluation

We are now ready to present the single-step evaluation relation on graphs. Like [28, 10] etc, it is a graph-based notion of evaluation, but where function bodies are never copied – a single body is shared.

$$\textsc{Application}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2\ \hat{x}_3) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{f} \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{v} \in \hat{D}(\hat{x}_3, \hat{c})}{\hat{D} \longrightarrow^1 \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}, \hat{x}_3, \hat{x}_1)}$$

$$\textsc{Record Conditional True}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2 \texttt{ ~ } \hat{r} \texttt{ ? } \hat{f}_1 \texttt{ : } \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{r}' \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{r} \subseteq \hat{r}'}{\hat{D} \longrightarrow^1 \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}_1, \hat{x}_2, \hat{x}_1)}$$

$$\textsc{Record Conditional False}\quad \frac{\hat{c} = (\hat{x}_1 \texttt{ = } \hat{x}_2 \texttt{ ~ } \hat{r} \texttt{ ? } \hat{f}_1 \texttt{ : } \hat{f}_2) \qquad \textsc{Active}(\hat{c}, \hat{D}) \qquad \hat{v} \in \hat{D}(\hat{x}_2, \hat{c}) \qquad \hat{v} \text{ of form } \hat{r}' \text{ only if } \hat{r} \not\subseteq \hat{r}'}{\hat{D} \longrightarrow^1 \hat{D} \cup \textsc{Wire}(\hat{c}, \hat{f}_2, \hat{x}_2, \hat{x}_1)}$$

Figure 11 Abstract Evaluation Rules

3.2.1 Active nodes

In order to preserve standard evaluation order we define the notion of an active node, Active – only nodes with all previous nodes already executed can fire. This serves a purpose similar to an evaluation context in operational semantics [4].

▶ **Definition 3.5.** $\textsc{Active}(\hat{a}', \hat{D})$ iff path $\textsc{Start} \ll \hat{a}_1 \ll \ldots \ll \hat{a}_n \ll \hat{a}'$ appears in $\hat{D}$ such that no $\hat{a}_i$ is of one of the forms $\hat{x} \texttt{ = } \hat{x}'\ \hat{x}''$ or $\hat{x} \texttt{ = } \hat{x}' \texttt{ ~ } \hat{p} \texttt{ ? } \hat{f} \texttt{ : } \hat{f}'$.

We write $\textsc{Active}(\hat{a}')$ when $\hat{D}$ is understood from context.

3.2.2 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 3.6.** Let $\textsc{Wire}(\hat{c}', \texttt{fun}\ \hat{x}_0 \texttt{ -> (}\ [\hat{c}_1, \ldots, \hat{c}_n]\ \texttt{)}, \hat{x}_1, \hat{x}_2) =$

$$\textsc{Preds}(\hat{c}') \ll (\hat{x}_0 \overset{\hat{c}'\downarrow}{=} \hat{x}_1) \ll \hat{c}_1 \ll \ldots \ll \hat{c}_n \ll (\hat{x}_2 \overset{\hat{c}'\uparrow}{=} \textsc{RV}(\hat{c}_n)) \ll \textsc{Succs}(\hat{c}').$$

$\hat{c}'$ here is the call site, and $\hat{c}_1 \ll \ldots \ll \hat{c}_n$ is the wiring of the function body. The Preds/Succs functions (defined above in Notation 3.1) reflect how we do not put special nodes before and after the call but simply wire to whatever predecessors and successors were already there, as was observed in Section 2.

Next, we define the abstract small-step relation $\longrightarrow^1$ on graphs, see Figure 11.

▶ **Definition 3.7.** We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 11. We write $\hat{D}_0 \longrightarrow^* \hat{D}_n$ to denote $\hat{D}_0 \longrightarrow^1 \hat{D}_1 \longrightarrow^1 \ldots \longrightarrow^1 \hat{D}_n$.

The evaluation rules end up being surprisingly straightforward after the above preliminaries. For application, if $\hat{c}$ is a call site that is an active redex, lookup of the function variable $\hat{x}_2$ returns function body $\hat{f}$ and *some* value $\hat{v}$ can be looked up at the argument position, we may wire in $\hat{f}$'s body to this call site. Note that $\hat{v}$ is not added to the graph, it is only observed here to constrain evaluation order to be call-by-value. The case clause rules are similar.

3.3 Decidability

We begin with the computability of the variable lookup operation, the source of all of the computational complexity of the analysis.

▶ **Lemma 3.8.** *If $\hat{\boldsymbol{C}}$ is a finite set with computable push/pop/top operations, then $\hat{D}(\hat{x}_0, \hat{a})$ is a computable function.*

Proof. This proof proceeds by reduction to the problem of reachability in a push-down system (PDS) accepting by empty stack. A push-down system is a push-down automaton (PDA) with an empty input alphabet; PDA/PDS reachability is decidable in polynomial time [3, 1].

We define a PDS in which each state is a pair between a program point and context stack (of which there are finitely many); the initial state is the pair $\hat{a} \times []$ (here, $[]$ is the empty context stack). The stack of the PDS corresponds to the lookup stack and current variable of the lookup operation; thus, the stack alphabet of the PDS contains all variables in the source program and the initial PDS stack is then $[\hat{x}_0]$ for computing $\hat{D}(\hat{x}_0, \hat{a})$. The transitions in the PDS are defined immediately by the clauses in Definition 3.4. For instance, suppose $\hat{a}_1 = (\hat{x}' = \hat{v}')$ and $\hat{a}_1 \ll \hat{a}_0$; then, by clauses 1 and 2, each node $\hat{a}_0 \times \hat{C}$ in the PDS transitions to $\hat{a}_1 \times \hat{C}$ by popping $\hat{x}'$ (and pushing no symbols).

Given the above, it suffices to show that the values given by $\hat{D}(\hat{x}_0, \hat{a})$ are exactly those of the nodes reachable with an empty stack in the PDS. Let $\hat{V} = \hat{D}(\hat{x}_0, \hat{a})$ and let $\hat{V}'$ be the values in the PDS's reachable nodes. For each $\hat{v} \in \hat{V}$, $\hat{v} \in \hat{V}'$ follows directly by induction on the size of the proof of $\hat{v} \in \hat{V}$, constructing a path through the PDS at each step. For each $\hat{v} \in \hat{V}'$, $\hat{v} \in \hat{V}$ follows directly by induction on the length of any path in the PDS which reaches the node containing $\hat{v}$. ◀

3.3.1 Some simple context stack models

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

▶ **Definition 3.9.** Fixing k, we define context stack model Σ_k as having stack set $\hat{\boldsymbol{C}}$ be the set of all lists of $\hat{c}$ occurring in the program up to length k, and with the stack operations as follows:

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

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

▶ **Lemma 3.10.** *Fixing Σ to some Σ_k for fixed constant k, $\hat{D}(\hat{x}, \hat{a}_0)$ is computable in polynomial time in the number of nodes in graph $\hat{D}$.*

Proof. Let g be the number of nodes in graph $\hat{D}$. By inspection, the PDS built in the proof of Lemma 3.8 will have a number of states which is of order the product of g and the number of context stacks in Σ_k. For Σ_k, the number of stacks is of order $O(g^k)$. Since PDS reachability is computable polynomially in the number of nodes in the PDS, the result immediately follows. ◀

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

▶ **Lemma 3.11.** *Variable lookup is monotonic; that is, for any $\hat{x}$ and $\hat{a}$, if $\hat{D}_1 \subseteq \hat{D}_2$ then $\hat{D}_1(\hat{x}, \hat{a}) \subseteq \hat{D}_2(\hat{x}, \hat{a})$.*

Proof. Variable lookup is encodable as a PDS reachability problem (see Lemma 3.8) and the PDS grows monotonically with the graph $\hat{D}$. PDS reachability grows monotonically with the PDS. Therefore, the set of results from variable lookup grows monotonically with the graph $\hat{D}$. ◀

▶ **Lemma 3.12.** *The evaluation relation $\longrightarrow^*$ is confluent.*

Proof. By inspection of Figure 11, the single-step rules only add to graph $\hat{D}$. The Active relation is also clearly monotone: any enabled redex is never disabled. Confluence is trivial from these two facts. ◀

▶ **Lemma 3.13.** *The evaluation relation $\longrightarrow^*$ is terminating,* i.e. *for any $\hat{D}_0$ there exists a $\hat{D}_n$ such that $\hat{D}_0 \longrightarrow^* \hat{D}_n$ and if $\hat{D}_n \longrightarrow^* \hat{D}_{n+1}$, $\hat{D}_n = \hat{D}_{n+1}$. Furthermore, n is polynomial in the size of the initial program.*

Proof. By inspection of Figure 11, we have for any step $\hat{D}' \longrightarrow^1 \hat{D}''$ that $\hat{D}' \subseteq \hat{D}''$. The only new nodes that can be added to $\hat{D}$ in the course of evaluation are the entry/exit nodes $\hat{x}' \overset{\hat{c}\downarrow}{=} \hat{x}$ / $\hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}'$, and only one of each of those nodes can exist for each call site (or case clause) / 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 then two times the square of the size of the original program. A similar argument holds for added edges. ◀

We let $\hat{D} \downarrow \hat{D}'$ abbreviate $\hat{D} \longrightarrow^* \hat{D}'$ for $\hat{D}'$ terminal. We write $e \downarrow \hat{D}$ to abbreviate $\textsc{Embed}(e) \downarrow \hat{D}$; this means the analysis of e returns graph $\hat{D}$. Given the pieces assembled above, it is now easy to prove that the analysis is polynomial-time.

▶ **Theorem 3.14.** *Fixing Σ to be some Σ_k and fixing some expression e, the analysis result $\hat{D}$, where $e \downarrow \hat{D}$, is computable in time polynomial in the size of e.*

Proof. By Lemma 3.10, each lookup operation takes poly-time. The evaluation rules are trivial computations besides the required lookups and, by Lemma 3.13, there are polynomially many evaluation steps before termination. Thus $e \downarrow \hat{D}$ is computable in polynomial time. ◀

4 Soundness

We now establish soundness of the analysis defined in the previous section. In forward program analyses the alignment between the operational semantics and analysis is fairly close and so soundness is not particularly difficult, but here there is a larger gap. We cross this river by throwing a stone in the middle: along with defining a mostly-standard operational semantics we build a *graph-based operational semantics* which creates a *concrete* call graph of the program run that is more directly aligned with the analysis. In particular, the graph-based operational semantics contains no environment or heap per se; as in the analysis, variable values are looked up by walking back through the call graph. Soundness is then shown by proving the standard and graph-based operational semantics equivalent and by showing the analysis sound with respect to the graph-based operational semantics; this latter is accomplished by the typical simulation-style proofs used in forward analyses.

$$\textsc{Lookup}\quad \frac{x_2 = v \in E}{E\,||[x_1 = x_2]\,||\,e \longrightarrow^1 E\,||[x_1 = v]\,||\,e} \qquad \textsc{Application}\quad \frac{x_2 = f \in E \qquad x_3 = v \in E \qquad f' = \textsc{fr}(x_1, f)}{E\,||[x_1 = x_2\ x_3]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f', v, x_1)\,||\,e}$$

$$\textsc{Record Conditional True}\quad \frac{x_2 = r' \in E \qquad r \subseteq r' \qquad f_1' = \textsc{fr}(x_1, f_1)}{E\,||[x_1 = x_2 \sim r\ ?\ f_1 : f_2]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f_1', r', x_1)\,||\,e}$$

$$\textsc{Record Conditional False}\quad \frac{x_2 = v \in E \qquad v \text{ of the form } r' \text{ only if } r \not\subseteq r' \qquad f_2' = \textsc{fr}(x_1, f_2)}{E\,||[x_1 = x_2 \sim r\ ?\ f_1 : f_2]\,||\,e \longrightarrow^1 E\,||\,\textsc{Wire}(f_2', r', x_1)\,||\,e}$$

Figure 12 Small-Step Evaluation

In this section we first present the standard operational semantics, then the graph-based operational semantics and its equivalence to the standard one, and finally we prove soundness.

4.1 Standard Operational Semantics

An operational semantics for the language appears in Figure 12. We define the operational semantics as a small step relation $e \longrightarrow^1 e'$. In many ways the operational semantics is standard, but due to our use of an A-normal form it is neither precisely substitution-based nor environment-based. It is more substitution-based in spirit since function bodies are inlined. Although variable lookup is via an environment, all names in that environment are deterministically freshened and so no variable shadowing ever arises; so, although variable lookup might appear to be dynamically scoped, the absence of shadowing ensures static scoping. This model of evaluation is designed to integrate well with the graph-based semantics of the next sub-section.

We must freshen variables as they are introduced to the expression to preserve the invariant that each variable is uniquely defined and no shadowing occurs. We give a somewhat nonstandard definition of freshening which is deterministic, so as to make alignments easier. We take $\textsc{fr}(x', x)$ to yield another variable x''; we require that $\textsc{fr}(-,-)$ is injective and that its codomain does not include variables appearing in the initial program. Here, x is the variable to be freshened and x' is the point in the program at which it is freshened. For informal illustration, one concrete freshening function could be $\textsc{fr}(x', x) = x^{x'}$. We overload $\textsc{fr}(x', v)$ to indicate the freshening of all variables bound in v.

The rules for the small step relation are given in Figure 12. Here, *wiring* is the process of inlining a function body; for this we use the following auxiliary function.

▶ **Definition 4.1.** Let $\textsc{Wire}(\texttt{fun}\,x\,\texttt{->}\,(\,e\,)\,, v, x') = [x = v]\,||\,e\,||[x' = \textsc{rv}(e)]$.

▶ **Definition 4.2.** We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 12. We write $e_0 \longrightarrow^* e_n$ to denote $e_0 \longrightarrow^1 e_1 \longrightarrow^1 \ldots \longrightarrow^1 e_n$.

▶ **Definition 4.3.** If, for some expression e not of the form E, there exists no e' such that $e \longrightarrow^1 e'$, then we say that e is *stuck*. For any e'' such that $e'' \longrightarrow^* e$, we say that e'' *becomes stuck*.

$$
\begin{array}{rcll}
V & ::= & \{v, \ldots\} & \textit{value sets} \\
a & ::= & c \mid x \overset{c\downarrow}{=} x \mid x \overset{c\uparrow}{=} x \mid & \textit{annotated clauses} \\
 & & \textsc{Start} \mid \textsc{End} & \\
d & ::= & a \ll a & \textit{dependencies}
\end{array}
\qquad
\begin{array}{rcll}
D & ::= & \{d, \ldots\} & \textit{dependency graphs} \\
X & ::= & [x, \ldots] & \textit{lookup stacks} \\
C & ::= & [c, \ldots] & \textit{context stacks}
\end{array}
$$

Figure 13 Graph-Based Evaluation Grammar

4.2 Graph-Based Operational Semantics

We now define the graph-based operational semantics, and prove it to be equivalent to the standard operational semantics just defined.

The graph-based operational semantics is structurally very similar to the analysis. The primary difference is function bodies here are *copied* to make new graph structure (as opposed to the analysis, in which only one copy of each function body exists in the graph). Otherwise, the graph-based operational semantics and the analysis are nearly identical, including the use of context stacks, lookup operations, and so on. Since many of these definitions are so similar, we will be much more brief here; we assume that the reader has absorbed the analysis definitions.

The graph-based operational semantics differs from the standard operational semantics of Section 4.1 in the following dimensions:

1. the total ordering of the list becomes a partial ordering here;
2. alias clauses $x = x'$ are resolved lazily rather than eagerly, and
3. the graph is monotonically increasing; that is, in each place that the standard semantics *replaces* a call site, the graph-based semantics builds a path *around* the call site.

The new grammar constructs needed for the graph-based operational semantics appear in Figure 13.

We write $\textsc{Embed}(e)$ to denote the initial embedding of an expression e into a dependency graph D. This definition is identical to analysis Definition 3.2 with hats removed from the metavariables. Similarly, we also will use the notational sugar of Notation 3.1 in perfect analogy.

4.2.1 Variable value lookup

The definition of lookup proceeds with respect to the context stack C which is directly aligned with the corresponding stack $\hat{C}$ of the analysis. This stack is not necessary in the operational semantics (as copying of function bodies removes all of the ambiguity that was in the analysis); we retain it for alignment. Unlike the analysis, we need not bound C; we therefore fix the context stack model of the operational semantics to the equivalent of the unbounded model Σ_ω in this grammar.

Lookup finds the value of a variable starting from a given graph node; this definition is a near-exact parallel of Definition 3.4 and the reader is referred there for intuitions. We let $D(x, a_0)$ abbreviate $D(x, a_0, [], [])$.

▶ **Definition 4.4.** Given graph D, $D(x, a_0, X, C)$ is the function returning the least set of values V satisfying the following conditions:

1. If $a_1 = (x = v)$, $a_1 \lll a_0$, and $X = []$, then $v \in V$.
2. If $a_1 = (x = f)$, $a_1 \lll a_0$, and $X = [x_1, \ldots, x_n]$ for $n > 0$, then $D(x_1, a_1, [x_2, \ldots, x_n], C) \subseteq V$.
3. If $a_1 = (x = x')$ and $a_1 \lll a_0$, then $D(x', a_1, X, C) \subseteq V$.

$$\textsc{Application}\quad \frac{c = (x_1 \texttt{ = } x_2\ x_3) \qquad \textsc{Active}(c, D) \qquad f \in D(x_2, c) \qquad v \in D(x_3, c) \qquad f' = \textsc{fr}(x_1, f)}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f', x_3, x_1)}$$

$$\textsc{Record Conditional True}\quad \frac{c = (x_1 \texttt{ = } x_2 \texttt{ ~ } r \texttt{ ? } f_1 \texttt{ : } f_2) \qquad \textsc{Active}(c, D) \qquad r' \in D(x_2, c) \qquad r \subseteq r' \qquad f_1' = \textsc{fr}(x_1, f_1)}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f_1', x_2, x_1)}$$

$$\textsc{Record Conditional False}\quad \frac{\begin{array}{c} c = (x_1 \texttt{ = } x_2 \texttt{ ~ } r \texttt{ ? } f_1 \texttt{ : } f_2) \\ \textsc{Active}(c, D) \qquad v \in D(x_2, c) \qquad v \text{ of form } r' \text{ only if } r \not\subseteq r' \qquad f_2' = \textsc{fr}(x_1, f_2) \end{array}}{D \longrightarrow^1 D \cup \textsc{Wire}(c, f_2', x_2, x_1)}$$

Figure 14 Graph Evaluation Rules

4. If $a_1 = (x \overset{\text{c}\downarrow}{=} x')$, $a_1 \lll a_0$, and $\textsc{IsTop}(c, C)$, then $D(x', a_1, X, \textsc{Pop}(C)) \subseteq V$.
5. If $a_1 = (x \overset{\text{c}\uparrow}{=} x')$, $a_1 \lll a_0$, $c = (x_r \texttt{ = } x_f\ x_v)$, $a_2 \lll c$, $D(x_f, a_2, X, C) = V'$, $\texttt{fun}\ x'' \texttt{ -> (}\ e\ \texttt{)} \in V'$, and $x' = \textsc{rv}(e)$, then $D(x', a_1, X, \textsc{Push}(c, C)) \subseteq V$.
6. If $a_1 = (x'' \texttt{ = } b)$, $a_1 \lll a_0$, and $x'' \neq x$, then $D(x, a_1, X, C) \subseteq V$.
7. If $a_1 = (x'' \overset{\text{c}\downarrow}{=} x')$, $a_1 \lll a_0$, $x'' \neq x$, $c = (x_r \texttt{ = } x_f\ x_v)$, and $\textsc{IsTop}(c, C)$, then $D(x_f, a_1, x \,||\, X, \textsc{Pop}(C)) \subseteq V$.
8. If $a_1 = (x'' \overset{\text{c}\downarrow}{=} x')$, $a_1 \lll a_0$, $x'' \neq x$, $c = (x_2 \texttt{ = } x_1 \texttt{ ~ } p \texttt{ ? } f_1 \texttt{ : } f_2)$, and $\textsc{IsTop}(c, C)$, then $D(x, a_1, X, \textsc{Pop}(C)) \subseteq V$.

4.2.2 The evaluation relation

The definition of an active node, Active, exactly parallels the analysis version of Definition 3.5, simply remove the hats from the metavariables. We write $\textsc{Active}(a')$ when D is understood from context. Wiring is also defined in perfect analogy with Definition 3.6 so is omitted here. The small-step relation $\longrightarrow^1$ on graphs is defined in Figure 14. (Note we overload symbol $\longrightarrow^1$ for the list and graph operational semantics, it is clear from context which relation is intended.)

▶ **Definition 4.5.** We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 14. We write $D_0 \longrightarrow^* D_n$ to denote $D_0 \longrightarrow^1 D_1 \longrightarrow^1 \ldots \longrightarrow^1 D_n$.

We also define a notion of "stuckness" for graphs in parallel with the standard operational semantics.

These rules are very similar to the analysis transition rules in Section 3.2; we refer the reader to the descriptions there. Here we comment on a few points unique to the operational semantics not found in the analysis.

We are precise about freshening in these rules. Because the $\textsc{fr}(x, -)$ function is deterministically freshening based on the call site x, it will always generate the same result for the same call site and value. This means that e.g. each use of the Application rule is idempotent and the graph-based operational semantics is deterministic.

Observe how Figure 14 is defining a "reduction" relation which is in fact monotonically *increasing*, an unusual property compared to standard operational semantics.

4.3 Proving Equivalence of Operational Semantics

The overall proof of soundness relies upon showing that these two systems of operational semantics are equivalent. To demonstrate this, we must show several properties of the graph-based operational semantics which are less obvious than in the standard operational semantics due to the structure of the graph. As stated above, the real differences between these systems are (1) the partial ordering of clauses, (2) lazy resolution of alias clauses, and (3) that the graph grows monotonically instead of replacing applications and conditionals. Nonetheless, the graph representation "runs" in the same way that expressions do: there is a unique clause which is evaluated next, it may cause the introduction of more clauses, and *complex clauses* (applications and conditionals) are handled by wiring and evaluating the appropriate function body.

We use this reasoning to establish a bisimulation $\cong$ between an expression and its embedding and then show that this bisimulation is preserved as evaluation proceeds. The bisimulation and corresponding preservation proof are tedious and intuitive, so we include them in the supplementary material included with this paper for reasons of space. The key equivalence lemma is stated as follows.

▶ **Lemma 4.6** (Equivalence of standard and graph-based semantics). *If $e \cong D$, then*
1. *If $e \longrightarrow^1 e'$ then $D \longrightarrow^* D'$ such that $e' \cong D'$.*
2. *If $D \longrightarrow^1 D'$ then $e \longrightarrow^* e'$ such that $e' \cong D'$.*

4.4 Soundness of the Analysis

We now show that the analysis simulates the graph-based operational semantics, $D \preccurlyeq \hat{D}$. This proof is easy, as the operational semantics graph can be projected on to an analysis graph by inverting the variable freshening function. We formally define this inversion as follows:

▶ **Definition 4.7.** The *origin* of a variable x is x itself if x is not in the codomain of $\textsc{fr}(-,-)$. Otherwise, let $\textsc{fr}(x'', x') = x$ (for unique x'' and x', as $\textsc{fr}(-,-)$ is injective). Then the origin of x is the origin of x'.

We next define the simulation between evaluation graphs and analysis graphs. The graph operational semantics was designed explicitly to make this simulation direct.

▶ **Definition 4.8** (Simulation relation). Let f be the natural graph-and-clause homomorphism from an evaluation graph D to an analysis graph $\hat{D}$ which maps variables to their origins. We say that D is simulated by $\hat{D}$ (written $D \preccurlyeq \hat{D}$) iff $f(D) = \hat{D}$.

▶ **Lemma 4.9** (Soundness). *If $D \preccurlyeq \hat{D}$ and $D \longrightarrow^1 D'$, then $\hat{D} \longrightarrow^1 \hat{D}'$ with $D' \preccurlyeq \hat{D}'$.*

Proof. By case analysis on the rule used to prove $D \longrightarrow^1 D'$. In particular, the premises of a corresponding rule $\hat{D} \longrightarrow^1 \hat{D}'$ can be proven using the premises of $D \longrightarrow^1 D'$ and the simulation. ◀

5 Extensions

In this section, we outline three extensions which we left out of the formal presentation: full records, path sensitivity, and mutable state. Our goal here is to demonstrate that there is no fundamental limitation to the model given in the previous sections: DDPA can be extended to the features that appear in realistic programming languages. Here for

simplicity of presentation we take each feature one at a time; in a language where they are all simultaneously added there are additional feature interaction cases that must be addressed.

5.1 Records

The "records" presented thus far have no projection operation since there are no values to project. We now describe how the analysis of the previous section can be extended to records with projection. Consider the process of looking up the value of a variable $\hat{x}$. If $\hat{x}$ is defined as $\hat{x} = \hat{x}'.\ell$, then this necessitates a record projection; of course, $\hat{x}'$ itself may necessitate another projection, and so on. In the general case, there could be a continuation stack of record projections to be performed. This is very similar to the nonlocal lookup stack of our analysis, and not coincidentally: nonlocals may be encoded in terms of records via closure conversion.

In light of this connection, we can extend lookup to support full records using a single lookup stack $\hat{X}$ (now more appropriately thought of as a *continuation stack*) to handle both data destructors as well as nonlocal variables. We define the grammar of lookup actions to include variables $\hat{x}$ and projections of the form $.\ell$; we use $\hat{k}$ to range over these lookup actions and extend $\hat{X}$ to all lists of $\hat{k}$. We then augment Definition 3.4 with the following new clauses:

▶ **Definition 5.1.** We extend Definition 3.4 to full records by adding the following clauses.

9. If $\hat{a}_1 = (\hat{x} = \{\ell_1 = \hat{x}', \ldots\})$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{X} = [.\ell_1, \hat{k}_2, \ldots, \hat{k}_n]$,
then $\hat{D}(\hat{x}', \hat{a}_1, [\hat{k}_2, \ldots, \hat{k}_n], \hat{C}) \subseteq \hat{V}$.

10. If $\hat{a}_1 = (\hat{x} = \hat{x}'.\ell)$, $\hat{a}_1 \lll \hat{a}_0$, and $\hat{X} = [\hat{k}_1, \ldots, \hat{k}_n]$, then $\hat{D}(\hat{x}', \hat{a}_1, [.\ell, \hat{k}_1, \ldots, \hat{k}_n], \hat{C}) \subseteq \hat{V}$.

We also extend each of the clauses of Definition 3.4 to address lookup actions generally (rather than requiring lookup stacks to consist only of variables).

Clause 10 above introduces the projection action $.\ell$ when we discover that we will need to project from the variable we find; clause 9 eliminates this projection action when the corresponding record value is found. Note that record pattern matching would also be desirable syntax, and would be possible by extending clause 8 of the original lookup definition; for brevity, we skip that here.

5.2 Filtering for path sensitivity

An additional level of expressiveness we desire from records is the ability to filter out paths that cannot have been taken. Consider, for instance, the recursive example in Figure 8. This example appears to be unsafe based on the analysis result: on line 7, we project `l` from the function's argument. The condition on line 4 makes this safe – only records with `l` labels may reach line 7 – but the analysis is unaware of this and so erroneously believes that the record `{}` may reach that point.

We address this incompleteness using *filters*: sets of patterns which the value must match in order to be considered relevant. In the above example, for instance, looking up `v` from line 7 leads us to look up `a` from line 4. Because our path moved backwards through the first clause of a conditional, however, we know that we may ignore any values we discover which do not match the `{l}` pattern. This is key to enabling path sensitivity and preventing spurious errors; with filters, DDPA correctly identifies the recursion example as safe.

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, path-sensitive DDPA is possible by modifying the lookup function as follows:

▶ **Definition 5.2.** We modify Definition 3.4 by adding to the lookup function two parameters of the forms Π^+ and Π^-. Each existing clause of that definition passes these arguments unchanged. We additionally replace clauses 1 and 4b of Definition 3.4 with:

1. If $\hat{a}_1 = (\hat{x}\texttt{ = }\hat{v})$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{X} = []$, and $\hat{v}$ matches all patterns in Π^+ and no patterns in Π^-, then $\hat{v} \in \hat{V}$.

4b1. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_2\texttt{ = }\hat{x}_1\texttt{ ~ }\hat{p}\texttt{ ? }\hat{f}_1\texttt{ : }\hat{f}_2)$, and $\hat{f}_1 = \texttt{fun }\hat{x}\texttt{ -> (}\hat{e}\texttt{)}$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}, \Pi^+ \cup \{\hat{p}\}, \Pi^-) \subseteq \hat{V}$.

4b2. If $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{\texttt{=}} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\hat{c} = (\hat{x}_2\texttt{ = }\hat{x}_1\texttt{ ~ }\hat{p}\texttt{ ? }\hat{f}_1\texttt{ : }\hat{f}_2)$, and $\hat{f}_2 = \texttt{fun }\hat{x}\texttt{ -> (}\hat{e}\texttt{)}$, then $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}, \Pi^+, \Pi^- \cup \{\hat{p}\}) \subseteq \hat{V}$.

We then redefine $\hat{D}(\hat{x}, \hat{a})$ to mean $\hat{D}(\hat{x}, \hat{a}, [], [], \emptyset, \emptyset)$.

Revised clause 1 shows how the filters are used: any value not matching the positive filter is filtered out, and oppositely for the negative filter. The original clause 4 was the case where we reached the start of a function 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. The original clause 4b 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 rule into two cases. In 4b1, 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 4b2 is the opposite case.

5.3 State

Lookup in the presence of state may also be performed using only a call graph. We consider here a variation of the presentation language which includes OCaml-style references with `ref` x / `!` x / x `<-` x syntax. There are several subtle issues that must be addressed. First, a simple linear search may not always give the correct answer; a cell containing a cell could have the inner cell mutated after the outer cell, which is out of order compared to the control flow sequence. So, we must define a branching search for references. Second, aliases may arise in the form of different variable names referring to the same cell. Lookup must not overlook aliases; otherwise, the lookup will be inaccurate and incomplete. So, explicit alias testing is needed to verify proper values are not being passed by.

Not only may DDPA be adapted to address state, but the alias analysis itself is quite easy to execute using the same lookup routine. The following definition revises the lookup operation for state. Note that we are still not adding any store or heap data structure; we're still just carefully analyzing a control flow graph.

▶ **Definition 5.3.** Definition 3.4 is extended to a stateful language by adding the following clauses.

9. If $\hat{a}_1 = (\hat{x}\texttt{ = !}\,\hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, then letting $\hat{V}' = \hat{D}(\hat{x}', \hat{a}_1, [], \hat{C})$, for each $\texttt{ref}\,\hat{x}'' \in \hat{V}'$, $\hat{D}(\hat{x}'', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

10a. If $\hat{a}_1 = (\hat{x}'_1\texttt{ = }\hat{x}'_2\texttt{ <- }\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, and $\textsc{MayAlias}(\hat{x}, \hat{x}'_2, \hat{a}_1, \hat{C})$, then $\texttt{ref}\,\hat{x}'_3 \in \hat{V}$.

10b. If $\hat{a}_1 = (\hat{x}'_1\texttt{ = }\hat{x}'_2\texttt{ <- }\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, and $\textsc{MayNotAlias}(\hat{x}, \hat{x}'_2, \hat{a}_1, \hat{C})$, then $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

In these clauses, the terms MayAlias and MayNotAlias refer to the following predicates:

- MayAlias$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{D}(\hat{x}_1, \hat{a}, [], \hat{C})$, $\hat{V}'' = \hat{D}(\hat{x}_2, \hat{a}, [], \hat{C})$, and $\exists \texttt{ref}\ \hat{x}'' \in (\hat{V}' \cap \hat{V}'')$
- MayNotAlias$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{C})$ holds iff $\hat{V}' = \hat{D}(\hat{x}_1, \hat{a}, [], \hat{C})$, $\hat{V}'' = \hat{D}(\hat{x}_2, \hat{a}, [], \hat{C})$, and $\hat{V}' \neq \hat{V}''$ *or* $\hat{V}' \neq \{\texttt{ref}\ \hat{x}'\}$ *or* $\hat{V}'' \neq \{\texttt{ref}\ \hat{x}''\}$

Clause 9 handles dereferencing. It finds the `ref` values which may be in $\hat{x}'$ at this point in the program; it then returns to this point to find all of the values that those variables may contain. This new task is necessary since we want the value at the point the `!` happened.

Clauses 10a and 10b address cell updates. Clause 10a determines if the updated cell in $\hat{x}_2'$ *may* alias the cell we are looking up; this happens when lookups of $\hat{x}$ and $\hat{x}_2'$ from this point might refer to the same cell. If this is the case, the value assigned by the cell update may be our answer. Clause 10b 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.

Along with the above modifications, the existing clauses 5 and 6 need to be extended to support state. As written, clause 6 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 6 by prohibiting $\hat{b}$ from being a call site or pattern match. We require a new clause similar to clause 5 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.

6 Implementation

We have developed a proof-of-concept implementation of DDPA which is hosted as an open source project on GitHub; a copy of this implementation appears in the non-anonymized supplementary material included with this paper. This implementation analyzes the presentation language given in Section 2 extended to the proper record semantics described in Section 5. For each example in the overview, the implementation generates the same control-flow graphs and value lookup results as discussed in this paper.

Although the proof-of-concept implementation demonstrates the behavior of DDPA, it is far from efficient and we are presently developing a performant implementation based on optimizations we will describe here. First, our implementation follows the lookup algorithm outlined in the proof of Lemma 3.8: we construct a push-down automaton modeling a non-deterministic backwards walk of the DDPA graph and then analyze this PDA for states reachable with an empty stack. There is a wealth of literature on PDA reachability algorithms [3, 1] which we can utilize to gain efficiency; in particular, [3] includes an algorithm for eliminating spurious nodes and edges which we have already integrated into the proof-of-concept implementation due to its simplicity.

Another applicable optimization arises due to the monotonicity of the DDPA graph described in Lemma 3.11. There is a direct mapping from nodes and edges in the DDPA graph to sets of nodes and edges in the variable lookup PDA. Because variable lookup and PDA reachability are both monotonic, it is possible to incrementally build the variable

lookup PDA in the same fashion as the DDPA control-flow graph and thus reuse the same developing PDA for all variable lookups throughout the analysis. We anticipate that this optimization, in combination with DDPA-specific adjustments to the PDA reachability algorithm to permit lazy construction of the automaton, will result in an DDPA implementation with similar practical applicability to forward analyses.

7 Related Work

7.1 CFL-reachability

The core idea of our analysis can be viewed as extending first-order demand-driven CFL-reachability analyses [6] to the higher-order case: the analysis is centered around using a CFG, calls and returns are aligned, and lookup is computed lazily. Two issues make the higher-order analysis challenging: the CFG needs to be computed on-the-fly due to the presence of higher-order functions, and nonlocal variable lookup is subtle. The aforecited demand-driven analyses 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 [18] including [29, 22].

Higher-order program analyses have been built which also incorporate call-return alignment [26, 8], but these higher-order analyses are not demand-driven and have a different algorithmic basis. We now contrast DDPA with these and other higher-order program analyses.

7.2 Higher-order program analysis

As mentioned in the Introduction, higher-order program analyses today are generally constructed by abstract interpretation [2], a finitization process on the operational semantics. We refer to these analyses as *forward* analyses to contrast with the backward-looking, demand-driven approach of DDPA.

In the most basic forward analysis, a new state is created for each new program point/store/environment, and the number of states grows too rapidly to be practical. Modifications including store widening, abstract garbage collection, polyvariance, and call-return alignment are thus added as means to obtain a better trade-off of expressiveness vs size of state space. DDPA in some sense comes from the opposite direction: the global state information is a graph isomorphic to the CFG which is polynomial in size and will be smaller than the state space of any forward analysis. However, variable lookup is of high complexity and we need to "spend" more space to make it more efficient.

To get into more detail we will consider the different dimensions of analysis expressiveness in turn. We begin with flow-sensitivity, meaning the analysis is sensitive to the *order* of assignment operations. While forward analyses start being wonderfully flow-sensitive since each abstract state has its own store, the standard store widening transition makes a single global store and eliminates flow-sensitivity. It is still possible to recover this information using abstract garbage collection techniques [14, 8]. Abstract garbage collection requires a per-node store since if the store were global there would be no garbage, and while this will work well for local store it will not be able to deal with persistently stored data: there still will be a state explosion in any persistent program. DDPA is flow-sensitive and garbage-free by construction: it is flow-sensitive because variables are looked up with respect to a program point, and it is garbage-free vacuously: no stores are ever created. In a practical

implementation of DDPA, the caching of values will bring it closer to an abstract garbage collection approach, but abstract garbage collection is bottom-up as opposed to top-down: rather than deleting items proved unneeded, items are added only if they may be needed in the future.

Another important dimension is polyvariance: whether functions can take on different forms in different contexts of use. The classic higher-order polyvariance model is kCFA [21], which copies contours in analogy to forall-elimination in a polymorphic type system. But there are many routes to behaviour that appears as polyvariance. Without store widening, flow-sensitivity alone can distinguish calling context and provide some polyvariance. Additionally, call-return alignment can provide different contexts for different function calls. The example we gave in Figure 3, for instance, needs only call-return alignment to give polyvariant behavior in DDPA.

Call-return alignment was first explored in a higher-order context in subtype constraint theories [17]; this work also uses the demand-driven nature of CFL-reachability to optimize lookup. However, it is flow-insensitive, uses let-polymorphism only, and does not align nonlocal variables as we do, and so is not in the same category of analysis. In the context of abstract interpretation, the first such algorithm was CFA2 [26], which was subsequently extended and refined in kPDCFA [8]. We will primarily compare with the more recent kPDCFA here.

An important observation of [8] regarding kPDCFA was that, in the presence of abstract garbage collection, it is not possible to have an arbitrary stack for aligning calls and returns. That paper proposes a regular expression modeling of call stacks to make the analysis computable. We have a similar problem in that our analysis requires two stacks but a PDA to implement it only has one, and so we elect to place the call stack in the PDA nodes themselves. We define the family of analyses kDDPA (Definition 3.3) where k is the maximum stack depth for call-return alignments. Our finitization here is simpler than the regular expression approach of kPDCFA; the regular expression approach may work in our context and is a topic for future investigation.

DDPA achieves greater polyvariance solely from call-return alignment when compared with kPDCFA or CFA2. In kPDCFA, stack alignment can be used to provide a different context for function parameters, but nonlocal variables get no such advantage since they are not in the local stack context. In DDPA, the stack context is still applicable to nonlocal variable lookup as shown in Figure 5. In general, we conjecture that DDPA achieves full polyvariance a la kCFA, provided the stack is deep enough. More precisely, we conjecture that, for a program with a maximal lexical nesting depth of c, the analysis $(k+c)$DDPA will be at least as expressive as kCFA (and in fact should be more expressive due to alignment of calls and returns). In other words, we achieve with only stack alignment what forward analyses need both stack alignment and polyvariant contours for. The reason why an extra c levels of call stack are needed in the worst case is each extra lexical level requires a search to enter (in reverse) the defining function's body, once for each lexical level and thus up to c times total, and thus taking up c extra slots on the call stack.

The run-time complexity of kDDPA can also be framed in terms of the expressiveness of nonlocal variable polyvariance. It is shown in [15] how nonlocals are the (only) source of exponential behavior in kCFA [24]; 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. For pathologically nested programs, k must be on

the order of the size of the program for kDDPA to avoid imprecision due to non-local lookup. Because kDDPA's complexity is exponential in k, such a non-local-precise kDDPA would be exponentially complex in that pathological case.

Related to this are provably polynomial context-sensitive analyses which restrict context-sensitivity in the case of high degree of lexical nesting [7, 15], much as how kDDPA also restricts. In [15] mCFA is defined, 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, the worst-case behavior has also been removed. Unfortunately, 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. mCFA does not have this property: non-locals will always be monomorphised for any m.

Our current implementation is a proof-of-concept only; we need to investigate ideas in [11, 8, 9] and other papers to obtain more optimal PDA reachability algorithms. Overall the trade-offs in performance and expressiveness are subtle and implementations will be necessary to decide how practically useful DDPA is.

7.3 Optimal λ-reduction

This work has a very indirect relationship with optimal λ-reduction [10]. They share a philosophy concerning non-local variable lookup: in both DDPA and optimal reduction, 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 $\mathtt{z}\downarrow$ / $\mathtt{z}\uparrow$ in DDPA). We are not the first to be inspired by sharing graphs for program analyses; sharing graphs are used in [23] but non-local arguments are directly wired in, placing that work closer to a subtype constraint inference system than to DDPA.

8 Conclusions

In this paper, we have developed a demand-driven program analysis (DDPA) for higher-order programs which is centered around production of a call graph. DDPA needs *only* the call graph to look up variable values; the specification does not maintain any other structures. DDPA can be viewed as the adaptation of previous CFL-reachability-based demand-driven program analyses to higher-order programs. This adaptation required two key changes in comparison to the first-order analyses supported by CFL-reachability. First, it was necessary to incrementally construct the control-flow graph as the analysis proceeded; second, the lookup of non-local variables required special handling.

We believe DDPA shows promise primarily because it represents a significantly different approach compared with other higher-order 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 a polynomial-time bound on a higher-order program analysis which is both flow- and context-sensitive. The reduced global state size holds out promise for program verification tools: the fewer the states, the less overwhelming the workload will be for a model checker, theorem prover, or other verification strategy.

We present preliminary results here from a simple implementation to serve as a confirmation of correctness, but our implementation needs tuning and would benefit from a head-to-head comparison with some state-of-the-art analyses. Although DDPA's value lookup is novel, it shares the task of PDA reachability with the existing program analysis literature and so a rich body of work is available to be utilized in developing efficient DDPA implementations.

We believe the analysis should scale well to other language features, and have presented outlines for extensions of the basic analysis to deep data structures, path-sensitivity, and state; we leave efficient decision procedures for the latter two to future work. We also intend to explore the development of extensions for other language features: exceptions and other control operators, concurrency, and modularity to name a few.

Acknowledgements We thank Alex Rozenshteyn for motivating our interest in AAM [25] and related papers which then led to our formulation of DDPA, and David Van Horn for discussions helping us better understand the AAM work.

References

1 Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, 1997.
2 Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
3 Christopher Earl, Matthew Might, and David Van Horn. Pushdown control-flow analysis of higher-order programs. In *The 2010 Workshop on Scheme and Functional Programming (SFP 2010)*, 2010.
4 M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 1992.
5 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI*, 1993.
6 Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, 1995.
7 Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95*, 1995.
8 J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. *JFP*, 2014.
9 John Kodumal and Alex Aiken. The set constraint/CFL reachability connection in practice. In *PLDI*, 2004.
10 John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL*, 1990.
11 Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with CFL-reachability. In *Compiler Construction*, 2013.
12 Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.
13 Matthew Might. Abstract interpreters for free. In *Proceedings of the 17th International Conference on Static Analysis*, 2010.
14 Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.
15 Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *PLDI*, 2010.
16 Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

17 Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL*, New York, NY, USA, 2001.

18 Thomas Reps. Demand interprocedural program analysis using logic databases. In *Application of Logic Databases*, 1994.

19 Thomas Reps. Shape analysis as a generalized path problem. In *PEPM*, 1995.

20 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

21 Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

22 Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In *Datalog Reloaded: First International Workshop*, 2011.

23 David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP*, 2007.

24 David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP*, 2008.

25 David Van Horn and Matthew Might. Abstracting abstract machines. In *ICFP*, 2010.

26 Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *European Symposium on Programming*, 2010.

27 Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *ICFP*, 2011.

28 Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. PhD thesis, University of Oxford, 1971.

29 Xin Zhang, Mayur Naik, and Hongseok Yang. Finding optimum abstractions in parametric dataflow analysis. In *PLDI*, 2013.

A Proof of Equivalence of Operational Semantics

This appendix contains a proof that the two operational semantics systems defined in Sections 4.1 and 4.2 are equivalent. We begin our discussion by formally distinguishing between complex clauses which have already been wired and those which have not.

▶ **Definition 1.1.** A complex clause a in a graph D is *complete* iff, for all f, x_1, and x_2, we have that $D \longrightarrow^1 D \cup \textsc{Wire}(a, f, x_1, x_2)$ only if $D = D \cup \textsc{Wire}(a, f, x_1, x_2)$. A complex clause which is not *complete* is *incomplete*. Simple clauses are neither complete nor incomplete.

In showing equivalence, we are only concerned with graphs that were evaluated from an embedding of a real expression. These graphs exhibit certain properties – such as the uniqueness of active, incomplete clauses – that are important for showing alignment. Essentially, we wish to demonstrate that, up to the point we are currently evaluating, the graph looks like the expression from which it was embedded except that the graph has "bumps" where complete clauses were never deleted. We formalize this intuition as the following well-formedness property:

▶ **Definition 1.2.** A graph D is *well-formed* iff all of the following are true.

1. It contains at most one incomplete clause.
2. All clauses which are not complete (including simple clauses) are totally ordered.
3. For all active clauses a appearing in D and any x, $|D(x, a)| \leq 1$.

Of course, embeddings of expressions should be well-formed. Also, well-formedness should be preserved by evaluation.

▶ **Lemma 1.3.** *For any e, $\textsc{Embed}(e)$ is well-formed.*

Proof. Conditions 1 and 2 follow immediately from Definition 1.1 and the fact that $\textsc{Embed}(e)$ is totally ordered. The proof of condition 3 follows by induction on the number of nodes between a and the $\textsc{Start}$ node; we know that such a path must exist and be unique, as $\textsc{Embed}(e)$ is totally ordered and a is active. ◀

▶ **Lemma 1.4.** *If D is well-formed and $D \longrightarrow^1 D'$ then D' is well-formed.*

Proof. Each rule of Figure 14 conditions upon an active clause a. If that clause is complete, then $D = D'$ and this property is trivial. Otherwise, the conditions of well-formedness can be demonstrated to hold by showing that a' is unique (i.e. that D cannot step to any other graph but itself). We observe that a is both active and incomplete; by condition 1 of well-formedness, it is the only such clause in D. By condition 3, there is at most one way to satisfy each rule in Figure 14. By inspection, these rules have exclusive premises; thus D' is unique.

Using similar logic, a can be shown to be complete in D'. Wiring inserts a totally ordered sequence of nodes from the predecessors of a (at most one of which is not complete) to the successors of a (at most one of which is not complete). Thus, condition 2 holds. As a result, we know by Definition 3.5 that at most one incomplete clause is active (which may have been in or after the inserted wiring); thus, condition 1 holds.

Demonstrating that condition 3 of well-formedness holds is more tedious but not complex; it proceeds by case analysis on Definition 4.4 by using the well-formedness of D. ◀

With the preservation of well-formedness, we can now prove the equivalence of these operational semantics. Key to this equivalence proof is a bisimulation relation which we

establish between an expression and its embedding. We can then show that this bisimulation is preserved throughout evaluation. The bisimulation shows that, at a given point in evaluation, the contents of each variable from the current point of evaluation appear the same and all future evaluation steps are identical. We formalize this bisimulation as follows:

▶ **Definition 1.5.** Let $e' = E \,||\, e$. Let D be a well-formed graph with a node a which is not complete and has no active successors. (This is either a unique, active, incomplete node or the End node, depending on whether D has finished evaluating.) We write $e' \cong D$ when the following conditions are met:

1. For all $x = v \in E$, $D(x, a) = \{v\}$.
2. For any path $a \ll a' \ll \ldots \ll \textsc{End}$ in D, we have $\textsc{Embed}(e) = \{\textsc{Start} \ll a \ll a' \ll \ldots \ll \textsc{End}\}$.

The first condition of bisimulation ensures that each variable in the environment matches its lookup value in the graph (and vice versa); the second condition ensures that the unevaluated portions of the expression and the graph are identical. We then prove that the operational semantics are equivalent by showing that evaluation preserves our bisimulation; this proves Lemma 4.6.

Proof. Each part of the proof proceeds by case analysis on the appropriate relation. In the first part, for instance, we proceed by case analysis on the rule used to prove $e \longrightarrow^1 e'$. If it works on a complex clause, then the corresponding graph evaluation rule can be used to show that $D \longrightarrow^1 D'$ using the conditions of bisimulation and well-formedness. If the proof of $e \longrightarrow^1 e'$ uses the Variable Lookup rule, then $e' \cong D$ (since graph evaluation is lazy in alias clauses).

The second part of the proof is similar except that the latter relation may take many steps. For any $D \longrightarrow^1 D'$, we proceed by case analysis on the rule used and apply the appropriate rule of $e \longrightarrow^1 e''$, again satisfying premises from bisimulation and well-formedness. This step may introduce some number of alias clauses, which we then evaluate ($e'' \longrightarrow^* e'$) to reach our result. ◀