Control-Based Program Analysis

Now we learn our CBA's

Zachary Palmer
Swarthmore College
zachary.palmer@swarthmore.edu

Scott F. Smith
The Johns Hopkins University
scott@cs.jhu.edu

Abstract

We explore a novel approach to higher-order program analysis which combines lazy data propagation with ideas from optimal lambda-reduction to produce a different form of program analysis. This analysis produces *only* a control-flow graph; we derive all other information (e.g. values of variables) directly from this graph. 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.

1. Introduction

Traditional data-flow analysis for imperative first-order programs is a relatively straightforward process. The program analysis monotonically accumulates information about what values program variables could take on at each program point, propagating information along the *fixed* control flow graph of the program. Program analysis becomes much more challenging for programs with full higher-order functions, because the data-flow of higher-order functions itself influences the control flow [15], making it necessary to simultaneously compute data- and control-flow information if an accurate result is desired. A large component of modern "control flow analysis" for higher-order programs [19] is the computation of accurate data-flow information; this in turn permits accurate control-flow information to be computed.

This paper

In this paper, we explore a different approach to higher-order program analysis: we focus entirely on control-flow computation. At each global step, the algorithm *only* produces additional control flow information concerning the relative ordering of program points and stores *no* specific information about the values that variables may have at different program points. Since the analysis is centered around producing only new control-flow computation for the source program at each step, we term it a *Control-Based program Analysis* (CBA). Data lookup (data flow) is then defined via a search process over a given control flow graph: starting from the "current" point in the graph, search "backwards" looking for the most recent definition of the variable. This lookup process is a lazy, reversed version of standard program analysis which eagerly pushes all data values forward. The data flow and control flow are still mutually dependent in CBA: if a variable is invoked as a function, we must look up its potential function values to know how to expand the control flow graph. Although this dependency between data flow and control flow is not broken, CBA changes the basis for the algorithm's calculations.

The particular form of data structure we produce is a "happens-before" relation on source program points, $A \ll B$. This relation indicates the program point A may happen immediately before program point B. The happens-before relation is isomorphic to what is commonly called the control-flow graph of higher-order programs [23]: *which call sites invoke which functions?* In standard higher-order program analysis the call graph is an *end result*, whereas it is the primary *means* of generating the analysis in CBA. During the analysis, our algorithm need only maintain a table of call sites and the function literals that they invoke. CBA can be configured to be context, flow-, and path-sensitive, but has a clear $O(n^2)$ space bound because that is the maximal size of a control flow graph.

During the course of the analysis, each variable lookup occurs relative to a particular context in the control flow graph where the variable is used; this makes flow-sensitivity a natural component of CBA. The technically deepest aspect of this paper is how nonlocal variables in function bodies may be accurately looked up solely from the control flow graph. It uses some ideas inspired by optimal lambda reduction [6, 12] to more precisely look up a variable based on the control flow context.

Related work from 50,000 feet

Higher-order program analyses are based on abstract interpretation [2]: a finite-state abstraction of the operational semantics transition relation is defined which represents a sound abstraction of program runtime behavior. A given abstract interpretation of a higher-order language can be thought of as a particular "hobbling" of the full operational semantics of the language to make the program transition relation state space finite; this intuitive view is made precise in [16], where two general classes of hobbling are described, namely *snipping* (cutting cycles in the state-space), and *trickling* (lifting abstractions up through the data structures), and the authors show how a wide class of program analyses can be built with these two transformations. Whatever abstraction is chosen, the resulting analysis has the same general structure as the operational semantics it was based on. The simplest form is 0CFA in which the memory addresses correspond precisely to source program points. Analyses may be refined by using multiple (but still finitely many) addresses for each program point to represent that program point's values in different situations, but this is still a direct refinement of *data flow*. We use the term "ABC analyses" to refer to these standard analyses, to emphasize how they propagate data forward (ABC) through the program, as opposed to moving backward (CBA) through the program to find it. The vast majority of higher-order program analyses, including [3, 10, 15, 16, 18, 19, 23, 27], are ABC analyses.

Many of these analyses literally produce a graph which represents the program's abstract execution, with an associated heap and environment mapping associated with each graph node. Each transition in the graph represents a control flow step. Since there may be multiple addresses for a given program point (e.g. if the analysis was polyvariant and produced a different address for each polyinstantiation of a function), this final graph may contain many more nodes than the program call graph. CBA also produces a graph in the form of the happens-before relation, but the nodes in the graph

are only *source* program points. So, the data structure both has fewer nodes and CBA needs no data-flow information stored with each node. Surprisingly even with so few nodes it is still possible to define a very expressive program analysis. CBA uses as much context information as it can in looking up variable values. First, lookup is parameterized by the node of the control flow graph the lookup is from, giving flow-sensitivity; second, the lookup process traces back the control flow to find a value but can rule out many paths because they are non-sensical as control flow due to mismatch of call and return; this gives a notion of context-sensitivity.

CBA is somewhat related to CFL-reachability analyses for non-higher-order programs [21, 22]: both forms of analysis use a demand-driven model to conservatively approximate data flow. They are distinct, however, in that CFL-reachability assumes a fixed control flow graph; CBA is designed for higher-order programs and so computes data flow and control flow in tandem as the analysis proceeds, a considerably more difficult task.

We will refine the relationship to related work in Section 7.

Practical Utility

How powerful is the resulting analysis? We will show that, in standard terminology, it is flow- and context-sensitive, and without much effort: these properties naturally emerge from how variable values are looked up from the contextual perspective of a given point in the computed control flow graph. One sign of how different the analysis is compared to ABC forms is we can prove a version of CBA with flow-and context-sensitivity runs in polynomial time, in contrast to kCFA which is EXPTIME complete [25]. (There are in fact other kCFA-like analyses which are polynomial, e.g. [18]; we contrast with them in the related work section).

Note that there is a hidden cost in CBA: looking up what value(s) a variable could take on is expensive in a naïve implementation. One challenge of the implementation is how variable lookup results can be efficiently cached; this means the challenge of building an expressive yet efficient CBA analysis is different than the challenges for a standard ABC analysis. Note that, even without caching, the algorithm is polynomial; without optimization, however, the degree of the polynomial is high.

When variable lookups are cached, the analysis moves a step closer to a standard ABC analysis: the analysis maintains the invocation table as well as node-specific cached lookup data. CBA can be viewed as a lazy algorithm, only looking up variable values as needed and from the context where they are needed, whereas ABC is fully eager, pushing ahead all values.

This paper is not claiming a practical advantage over ABC analyses today, only that the theoretical bound is sometimes better and CBA shows potential to have practical impact. The analysis could be weak in ways we have not yet noticed, or the complexity could be problematic in practice. It is known for example that in spite of the EXPTIME completeness of kCFA, it performs well on specific domains in practice [18].

We have completed an implementation, described in Section 6, and include it as supplementary information with this submission. This implementation is not optimized but it serves as a proof of concept.

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.

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

Figure 2.1. Expression Grammar

```
1  f = fun x -> ( # an identity function: 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 2.2. Invocation Example: A-Normal Form

2. Overview of the Analysis

This section informally presents CBA by example.

2.1 A Simple Language

For this paper we use the simple functional language defined in Figure 2.1. (Throughout this document, 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: $g_1 \,||[g_2, g_3] = [g_1, g_2, g_3]$.)

For a given program, our objective is to establish the possible execution paths that the program will take. As mentioned in the Introduction, we encode this information in the form of a happens-before relation $c \ll c'$: program point c related to c' means there may be control flow from c to c'. 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]. In shallow ANF, the arguments to each operation are trivial – for example, we write "`x = {}; r = f x`" rather than "`r = f {}`" – and expressions are a sequence of such clauses which yield the value assigned by the last clause. 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.

The operational semantics of this language are completely 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. Functions are given a standard call-by-value interpretation.

2.2 The Basic Analysis

We demonstrate CBA on the program appearing in Figure 2.2. In this program, `f` is just a fancy identity function used to help illustrate the analysis. Note that clause `n0` has no effect but helps clarify the diagrams we will present.

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

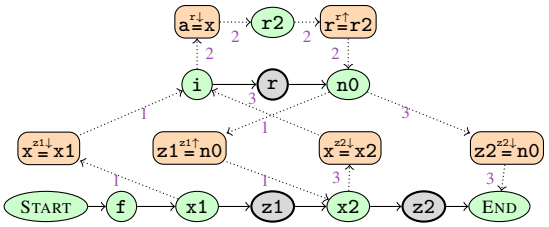

Figure 2.3. Invocation Example: Analysis

distinguished start and end nodes before `f` and after `z2`, respectively.

Figure 2.3 shows the completed analysis; up to now we have only described the bottom row. Focusing only on that row for now, the solid arrows (⟶) represent nodes with a $\ll$ 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 2.2. Green nodes are atomic clauses, and gray nodes are call sites.

Adding control flows Each gray node representing a call site must be connected to any possible function body invoked at that call site – that is how a call graph is built. Proceeding forward on the control flow from the start node, the first call site is `z1`. The dotted edges (⋯⟶) which are annotated with the number "1" are next added to the graph to represent the flow of control through the function `f` (the `i`, `r`, `n0` clauses) which is invoked at the `z1` call site. A simplistic way to "wire" the control flow would be to run an edge from `z1` to `i` for the enter-function flow and from `n0` back to `z1` for the return, but we use a particular scheme for wiring control flow so variable values can be easily looked up using only the graph data structure. (Both ⟶ and ⋯⟶ edges define $\ll$ relationships between program points; the two edge sorts are for helpful illustration in this section only.)

Observe that the call to `f` adds two *new* 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 the parameter passing for the `z1` call. All of the information in these new nodes can in fact be inferred from context: for example, for `x=x1` the `x` is the formal parameter and `x1` is the argument variable of the call site, and the label `z1` in this new node is just the return variable of the call, i.e. the name of the call site. A corollary of this inferance is there can be only one connection between any pair of call site and function body and thus there is a bound on the number of nodes in the graph as the product of the size of those two sets. Put another way, there is always an isomorphism between a happens-before graph and a call graph.

Notice also that the "1"-marked edges are not connected directly to `z1`, but are wired into the predecessor and successor nodes `x1` and `x2`, respectively. This reflects how a function call is really just an "inlining" of its body into the current context, after which the call site itself is dead code.

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 control flows labeled "2" to the graph. The "3" flows are similarly added when we elaborate the call site `z2`. At this point, there are no more call site wirings possible and the analysis is complete.

Variable lookup The graph of Figure 2.3 is the only information the analysis needs to construct; there is no environment or store needed, only caller-callee relationships. A key point of the analysis is how accurate variable lookups are possible from the graph only.

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 the function `f`, and all possible concrete function values of `f` need to be wired to the call site. This particular case is simple: the only way that control flow can reach `z1` after assigning a value to `f` is through the `x1` node. So, from the perspective of the `z1` node, the most recent value of `f` is only the function body we wired in. This case does illustrate how all variable lookup is *contextual*: lookup happens relative to a node in the graph, in this case relative to the call site `z1`. Intuitively, the lookup starts at the clause where the variable is being used, so the result of lookup should be the value of that variable most recently assigned from that point. In this initial example which has no recursion or other complex structure, this is trivial, but this perspective on lookup is important for more complex programs.

For another example, consider the task of finding the value of `x` when the program is at line 5. This line defines the `r` clause, so we trace back through the control flow from that point in the graph. There are two paths to the `r` clause: one from `x=x1` and one from `x=x2`. Both of these nodes define `x` in terms of another variable rather than an immediate value. So, lookup proceeds by in turn looking up that variable (`x1` in the former case and `x2` in the latter). Ultimately, the values of `x` at `i` are ascribed all the values which can be reached in this manner, transitively looking up variables as appropriate; in this case `x` has final result set {`{y}`, `{n}`}, the set of arguments the function was invoked on.

Observe that 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 is not complete; it is possible to rule out some search paths for variables because they provably correspond to no program execution. Consider how lookup as described above would proceed on finding 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: we may proceed to node `x=x1` or we may proceed to `x=x2`. By the description given above, we do both, union the result, and obtain the answer {`{y}`, `{n}`}. Given the code in Figure 2.2, the only value of `z2` at runtime is clearly `{n}`, so we did not obtain as accurate a result as the actual program run; why did this happen?

The spurious `{y}` value appears in our lookup as a result of traversing the edge from `x=x1` to `i`. This edge observes that control may flow from the passing of the argument `x1` to the first clause in the function's body. But, looking at the overall path we took from the end node, 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: calls and returns always align during program execution.

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. Repeating our example above, we would start by looking

```
1  k = fun v -> ( # the K combinator: 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 2.4. Non-Local Variable Example

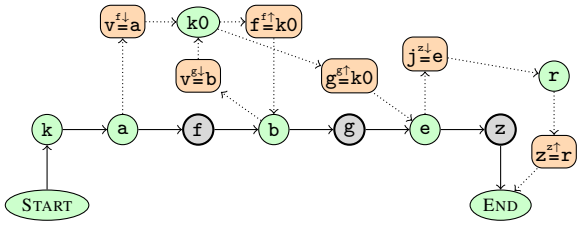

Figure 2.5. Non-Local Variable: Graph-Based Analysis

up `z2` from the end of the program with an empty stack. The next step is to look up `n0` from `z2=n0` with the call stack $[\texttt{z2}]$: we have entered the function (in reverse) via call site `z2`. Upon reaching `r=r2`, we are looking up `r2` with stack $[\texttt{r}, \texttt{z2}]$; this `r` stack element is consumed at the corresponding entry site `a=x`. Finally, we reach the same point as above: looking up `x` at `i`, this time with the call stack $[\texttt{z2}]$. We cannot proceed to `x=x1` as this would require us to consume a `z1` call site from the top of the stack; thus, our lookup results in just the value `{n}`. The idea of call-return alignment is found in some recent ABC analyses [3, 10, 26]; in CBA analysis, it is a simple refinement.

Note that the above analysis exhibits polyvariance: different invocations of the same function are not analyzed uniformly. Polyvariance is usually achieved by making multiple copies of function body data structures, kCFA being a canonical example [19]. In CBA, the previous paragraph shows how a similar result can be achieved by refining the lookup procedure to discard spurious control flow paths; no copying is needed, which has potential performance advantage because often only a small portion of a function is polyvariant and the standard method of copying the whole body is inefficient.

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 the same way, the lookup operation would interpret nonlocals as being dynamically scoped. To see why, consider the code example in Figure 2.4 and the corresponding analysis graph in Figure 2.5. 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. 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`$\downarrow$ on this node, we can deduce that we are leaving the current function which was called at call site `z`. Since `v` was not found we have deduced that it is a non-local. To proceed with lookup, we delay our search for `v`, examine the call site `z` (that appeared in the wiring annotation), and perform a lookup of the *function* invoked at that point; here this means next step is to search for `f` starting from node `e`. This search leads us through `f=k0` to the node `k0`, the point at which the function was originally defined, and from here we resume our search for `v`. Using this approach, the lookup of `z` from the end of the program yields `{a}` (and only that value, due to the call stack filtering described in the previous section).

```
1  f = fun s -> ( # a function to unwrap any number of "l"s
2          f0 = fun a -> (
3              n0 = {};
4              r = a ~ {l}
5                    ? fun a1 -> ( n2 = {};
6                                  ss = s s;  # self-apply
7                                  v = a1.l;  # project l
8                                  r1 = ss v; # recurse
9                                  n3 = r1; )
10                   : fun a2 -> ( r2 = a2; );# base case
11             n1 = r;
12         );
13 );
14 ff = f f;      # initial self-application
15 x1 = {};
16 x2 = {l=x1};
17 x3 = {l=x2};  # the record {l={l={}}}
18 z = ff x3;    # evaluates to {} (the innermost value)
```

Figure 2.6. Recursion Example

The above process might seem more convoluted than is necessary: a non-local variable in a function is lexically itself contained in only one function. So, it would be sound to simply keep a mapping from nonlocals to the sole function they were defined in, and to simply "jump" to that point in the graph to search for the variable. However, such a "jump" would lose context information and so the search would be less precise; in particular the above example would also return `{b}` as a potential value for `z` in such an implementation.

The variable lookup process description is now complete. The fully general case is a bit more complex; for example, the defining function itself could be non-locally defined, and so there could be a chain to follow. Fortunately, this chain is up the lexical scoping structure so the size is bounded by the program source.

2.5 Recursion and Decidability

The analysis described above naturally analyzes recursive programs: consider the program in Figure 2.6 which, for illustrative purposes, we code in an extension of the presentation language including proper records (e.g. `{a:x,b:y}`, consistent with shallow A-normalization) and record projection (e.g. `r.a`). 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 make the diagrams easier to follow but have no impact on the meaning of the program or the analysis.)

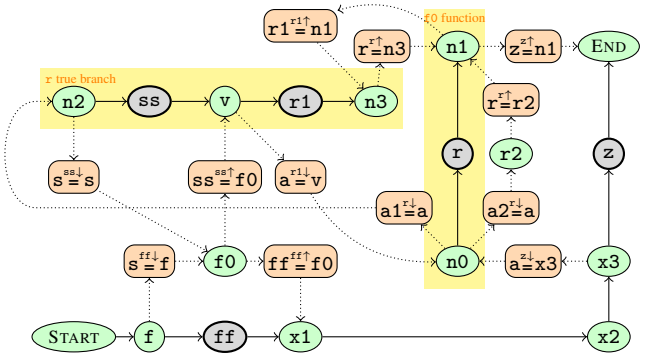

Figure 2.7. Recursion: Graph-Based Analysis

We give the analysis of this program in Figure 2.7. Note that this graph is the final result of the analysis, and was generated by a method similar to the previous examples.

In spite of the recursion, lookup generally proceeds by the same high-level process as for the previous examples. The one additional complication is that some control flow paths through the graph could be cyclic. For instance, the path `n0` $\rightsquigarrow$ `n2` $\rightsquigarrow$ `v` $\rightsquigarrow$ `n0` of Figure 2.7 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 we leave open the question whether lookup over an arbitrary stack is computable. So, in Section 3, we define a stack finitization which retains the k most recent contexts in the spirit of k-CFA [23]. Although we take a simple stack finitization approach here, we believe it is possible to symbolically handle all cases as in [3, 10, 26], a topic we leave for future work.

CBA as presented here is fundamentally flow-sensitive and context-sensitive. It is not, however, 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 2.6. According to the rules above, the path END $\rightsquigarrow$ `n1` $\rightsquigarrow$ `r2` $\rightsquigarrow$ `n0` $\rightsquigarrow$ `x3` in Figure 2.7 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; CBA 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: 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 2.1; we describe the extension to full records in Section 5.1 and they are also included in our current implementation.

2.6 Implementation

In this paper, we are primarily focusing on the principles of the algorithm. In principle and as implemented, the algorithm is polynomial, but our current implementation is naïve with respect to performance and so the polynomial degree is high. The primary performance issue is duplication of work between different variable lookups, especially as the CBA graph is augmented throughout the analysis. This problem is not theoretically challenging; the solution involves the (non-trivial) caching of internal data structures used during lookup. Fortunately, due to the control-centric nature of the analysis, caching has been factored into an orthogonal issue and the reader may safely disregard it while conceptualizing the execution model. We discuss our implementation in Section 6.

$\hat{e}$	::=	$[\hat{c}, \ldots]$	*abstract expressions*
$\hat{c}$	::=	$\hat{x}$ `=` $\hat{b}$	*abstract clauses*
$\hat{b}$	::=	$\hat{v} \mid \hat{x} \mid \hat{x}\ \hat{x} \mid \hat{x}$ `~` $\hat{p}$ `?` $\hat{f}$ `:` $\hat{f}$	*abstract clause bodies*
$\hat{x}$	::=	x	*abstract variables*
$\hat{v}$	::=	$\hat{r} \mid \hat{f}$	*abstract values*
$\hat{p}$	::=	$\hat{r}$	*abstract patterns*
$\hat{f}$	::=	`fun` $\hat{x}$ `->(` $\hat{e}$ `)`	*abstract functions*
$\hat{r}$	::=	r	*abstract records*
$\hat{V}$	::=	$\{\hat{v}, \ldots\}$	*abstract value sets*
$\hat{a}$	::=	$\hat{c} \mid \hat{x} \overset{\hat{c}\downarrow}{=} \hat{x} \mid \hat{x} \overset{\hat{c}\uparrow}{=} \hat{x} \mid$ START $\mid$ END	*abstract annotated clauses*
$\hat{d}$	::=	$\hat{a} \ll \hat{a}$	*abstract dependencies*
$\hat{D}$	::=	$\{\hat{d}, \ldots\}$	*abstract dependency graphs*
$\hat{X}$	::=	$[\hat{x}, \ldots]$	*abstract variable lookup stacks*

Figure 3.1. Analysis Grammar

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 3.1. The items above the line 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.

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

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

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* START $\ll \hat{c}_1 \ll \ldots \ll \hat{c}_n \ll$ 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 $\ll$ 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.

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.

With an unbounded-depth stack and no other limits placed on the analysis we currently have no proof of decidability, so we define a general call stack model which admits both unbounded as well as bounded (a la kCFA) call stacks. Our current implementation allows the user to fix a stack depth; we believe it is possible to implement an analysis with unbounded stack, but that topic is beyond the scope of this paper.

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 elements 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. $\textsc{IsTop}(\hat{c}, \textsc{Push}(\hat{c}, \hat{C}))$ *and* $\textsc{IsTop}(\hat{c}, \epsilon)$ *hold.*
5. *If* $\textsc{IsTop}(\hat{c}, \hat{C})$ *then* $\textsc{IsTop}(\hat{c}, \textsc{Pop}(\textsc{Push}(\hat{c}', \hat{C})))$*.*

For now we assume some fixed but unspecified context stack model Σ. In Section 3.3 we define some concrete finite- and unbounded-depth context stack models. A reasonable finite-depth stack model only records the most recent k frames; a standard stack can model unbounded-depth stack frames. 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.

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. It is generally a stack since the function itself could prove to be a nonlocal, etc. Section 2.4 provides motivation and examples for non-local variable lookup.

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* $\hat{a}_1 = (\hat{x} = \hat{v})$*,* $\hat{a}_1 \lll \hat{a}_0$*, and* $\hat{X} = []$*, then*
   $\hat{v} \in \hat{V}$*.*
2. *If* $\hat{a}_1 = (\hat{x} = \hat{f})$*,* $\hat{a}_1 \lll \hat{a}_0$*, 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* $\hat{a}_1 = (\hat{x} = \hat{x}')$ *and* $\hat{a}_1 \lll \hat{a}_0$*, then*
   $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$*.*
4. *If* $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{=} \hat{x}')$*,* $\hat{a}_1 \lll \hat{a}_0$*, and* $\textsc{IsTop}(\hat{c}, \hat{C})$ *then*
   $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$*.*
5. *If* $\hat{a}_1 = (\hat{x} \overset{\hat{c}\uparrow}{=} \hat{x}')$ *and* $\hat{a}_1 \lll \hat{a}_0$*, then*
   $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Push}(\hat{c}, \hat{C})) \subseteq \hat{V}$*.*
6. *If* $\hat{a}_1 = (\hat{x}'' = b)$*,* $\hat{a}_1 \lll \hat{a}_0$*, and* $\hat{x}'' \neq \hat{x}$*, then*
   $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$*.*
7. *If* $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{=} \hat{x}')$*,* $\hat{a}_1 \lll \hat{a}_0$*,* $\hat{x}'' \neq \hat{x}$*,*
   $\hat{c} = (\hat{x}_r = \hat{x}_f\ \hat{x}_v)$*, and* $\textsc{IsTop}(\hat{c}, \hat{C})$*, then*
   $\hat{D}(\hat{x}_f, \hat{a}_1, \hat{x} \mid\mid \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$*.*
8. *If* $\hat{a}_1 = (\hat{x}'' \overset{\hat{c}\downarrow}{=} \hat{x}')$*,* $\hat{a}_1 \lll \hat{a}_0$*,* $\hat{x}'' \neq \hat{x}$*,*
   $\hat{c} = (\hat{x}_2 = \hat{x}_1 \sim \hat{p}\ ?\ \hat{f}_1 : \hat{f}_2)$*, and* $\textsc{IsTop}(\hat{c}, \hat{C})$ *then*
   $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \textsc{Pop}(\hat{C})) \subseteq \hat{V}$*.*

(Note this is a well-formed inductive definition because all recursive invocations are strictly positive.) 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 $\hat{x}'$.
4. We have reached the start of the function body and the variable $\hat{x}$ we are searching for was the formal argument $\hat{x}'$. So, continue by searching for $\hat{x}'$ from the call site. The IsTop clause constraints this stack frame exit to align with the frame we had last entered (in reverse).
5. 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 (by the nature of copy-out edges, $\hat{c}$ must be a call site).
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.
7. 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 constraints the stack frame being exited to align with the frame we had last entered (in reverse).
8. This is the case clause variation of the previous.

APPLICATION

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

RECORD CONDITIONAL TRUE

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

RECORD CONDITIONAL FALSE

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

Figure 3.2. Abstract Evaluation Rules

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 [6, 12, 14, 28] etc, it is a graph-based notion of evaluation, but where function bodies are never copied – a single body is shared.

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.

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

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

$\hat{c}'$ 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 3.2.

Definition 3.7. *We define the small step relation* $\longrightarrow^1$ *to hold if a proof exists in the system in Figure 3.2. 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 [1, 3].

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 $\hat{a}_1 = (\hat{x}' \texttt{ = } \hat{v}')$ 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 $\hat{v}$. □

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 **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.*

Lemma 3.10. *Fixing* Σ *to be* Σ_1*,* $\hat{D}(\hat{x}, \hat{a}_0)$ *is computable in polynomial time in the number of nodes in graph* $\hat{D}$*.*

Proof. 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 the number of stacks and the number of nodes in the graph; for Σ_1 the number of stacks is of the order of the number of nodes in the graph. Since PDS reachability is computable polynomially in the number of nodes in the PDS, the result immediately follows. □

Lemma 3.11. *The evaluation relation* $\longrightarrow^*$ *is confluent.*

Proof. Since the single-step rules only add to graph $\hat{D}$ and the ACTIVE relation is also clearly monotone, any enabled redex is never disabled and confluence is trivial. □

Lemma 3.12. *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. 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. □

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 it is now easy to prove that the analysis is polynomial-time.

Theorem 3.13. *Fixing Σ to be Σ_1 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 lookups required, and by Lemma 3.12, 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 ABC 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 much 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 ABC analyses.

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 of the analysis.

4.1 Standard Operational Semantics

An operational semantics for the language appears in Figure 2.1. 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

$$\textsc{Lookup}\quad \frac{x_2 = v \in E}{E\,||[x_1 = x_2]\,||\,e \longrightarrow^1 E\,||[x_1 = v]\,||\,e}$$

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

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

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

Figure 4.1. Small-Step Evaluation

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. We also write $\textsc{rv}(e)$ to refer to the return variable of e (that is, the variable assigned in the last clause).

The rules for the small step relation are given in Figure 4.1. 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 4.1. 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.

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

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

Figure 4.2. Graph-Based Evaluation Grammar

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 4.2.

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.

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

The evaluation relation The definition of an active node, $\textsc{Active}$, 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 4.3. (Note we overload symbol $\longrightarrow^1$ for the list and graph operational semantics, it is clear from context which relation is intended.)

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

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

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

Figure 4.3. Graph Evaluation Rules

Definition 4.5. *We define the small step relation $\longrightarrow^1$ to hold if a proof exists in the system in Figure 4.3. We write $D_0 \longrightarrow^* D_n$ to denote $D_0 \longrightarrow^1 D_1 \longrightarrow^1 \ldots \longrightarrow^1 D_n$.*

In parallel with the standard operational semantics, we also define a notion of "stuckness" for graphs.

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 4.3 is defining a "reduction" relation which is in fact monotonically *increasing*. This illustrates how the graph-based operational semantics is non-standard in some dimensions; we were unable to find any presentation of operational semantics in the literature where the states monotonically grow.

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 Appendix A 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 rather 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* $\text{FR}(-,-)$. *Otherwise, let* $\text{FR}(x'', x') = x$ *(for unique x'' and x', as* $\text{FR}(-,-)$ *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 by simulation)**.** *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 left out of the formal presentation of the previous section: full records, path sensitivity, and mutable state. Our implementation includes currently includes full records but not the other two extensions. The primary purpose for including these partially worked out features here is to argue that there is no fundamental limitation to the model in terms of features left out of the initial formalization. For each of these extensions there is *no change needed* to the global analysis state or the single-step relation; only lookup needs to be modified.

5.1 Records

The "records" presented thus far – sets of labels with no values – 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. Looking up the value of a variable $\hat{x}$ may necessitate a record projection $\hat{x}'.\ell$ if $\hat{x} = \hat{x}'.\ell$ was the definition of $\hat{x}$, and $\hat{x}'$ itself may sit under a 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 though 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 2.6. 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 may 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 erroneous errors; with filters, CBA correctly identifies the recursion example as safe.

We can formalize path sensitivity in CBA 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 CBA 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 4 of Definition 3.4 with:*

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

4a. *If* $\hat{a}_1 = (\hat{x} \overset{\hat{c}\downarrow}{=} \hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, $\textsc{IsTop}(\hat{c}, \hat{C})$, *and* $\hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3)$ *then* $\hat{D}(\hat{x}', \hat{a}_1, \hat{X}, \textsc{Pop}(\hat{C}), \Pi^+, \Pi^-) \subseteq \hat{V}$.

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

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

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. Here, we have separated that rule into three cases. In clause 4a, the function appeared at a simple call site and so

the behavior of the old rule 4 is preserved. In 4b, 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 4c is the opposite case.

The above definition adds path-sensitivity to the formalism. The decision procedure for modified lookup is delicate and not yet fully specified/implemented, and is left for future work.

5.3 State

Lookup may also be performed using only a call graph in the presence of state. 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 CBA 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 $\hat{a}_1 = (\hat{x}$ `=` `!` $\hat{x}')$, $\hat{a}_1 \lll \hat{a}_0$, *then*
letting $\hat{V}' = \hat{D}(\hat{x}', \hat{a}_1, [], \hat{C})$, *for each* `ref` $\hat{x}'' \in \hat{V}'$, $\hat{D}(\hat{x}'', \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

10a. If $\hat{a}_1 = (\hat{x}'_1$ `=` $\hat{x}'_2$ `<-` $\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, *and* MAY-ALIAS$(\hat{x}, \hat{x}'_2, \hat{a}_1, [], \hat{C})$, *then* `ref` $\hat{x}'_3 \in \hat{V}$.

10b. If $\hat{a}_1 = (\hat{x}'_1$ `=` $\hat{x}'_2$ `<-` $\hat{x}'_3)$, $\hat{a}_1 \lll \hat{a}_0$, *and* not MUST-ALIAS$(\hat{x}, \hat{x}'_2, \hat{a}_1, [], \hat{C})$, *then* $\hat{D}(\hat{x}, \hat{a}_1, \hat{X}, \hat{C}) \subseteq \hat{V}$.

In these clauses, the terms MAY-ALIAS *and* MUST-ALIAS *refer to the following predicates:*

- MAY-ALIAS$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{X}, \hat{C})$ *holds iff* $\hat{V}' = \hat{D}(\hat{x}_1, \hat{a}, \hat{X}, \hat{C})$, $\hat{V}'' = \hat{D}(\hat{x}_2, \hat{a}, \hat{X}, \hat{C})$, *and* $\exists$`ref` $\hat{x}'' \in (\hat{V}' \cap \hat{V}'')$
- MUST-ALIAS$(\hat{x}_1, \hat{x}_2, \hat{a}, \hat{X}, \hat{C})$ *holds iff* $\hat{D}(\hat{x}_1, \hat{a}, \hat{X}, \hat{C}) = \hat{D}(\hat{x}_2, \hat{a}, \hat{X}, \hat{C}) = \{$`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 (with an empty $\hat{X}$) 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 is the case when lookups of $\hat{x}$ and $\hat{x}'_2$ from this point might refer to the same cell. If this is the case, we conclude that the value assigned by the cell update may be our answer. Clause 11b addresses the case in which the updated cell *might not* alias the target of our lookup. Conservatively, this is the case unless the lookup of each variable refers to exactly the same cell.

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 because this output is the only way in which evaluation may be affected. In the presence of side-effects, we must explore these clauses to ensure that they did not affect e.g. the cell we are attempting to dereference.

The necessary changes are formally verbose but easily summarized. We 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.

As with the path filtering, this specification has not yet been refined to a decision procedure or implemented.

6. Implementation

We have developed an implementation of the CBA analysis which is hosted as an open source project on GitHub[20]. This implementation analyzes the presentation language appearing in Section 2 extended with the proper record semantics described in Section 5; it permits the use of several different context sensitivity models. For each example in the overview, the implementation generates the same CBA graphs and value reachability as discussed in that section.

The implementation follows the lookup algorithm outlined in the proof of Lemma 3.8: we construct a push-down system modeling a non-deterministic backwards walk of the CBA graph and then analyze this PDA for states reachable with an empty stack. We can utilize the comprehensive literature on PDA reachability algorithms (e.g. [1, 3]) to gain efficiency, in particular [3] includes an algorithm for eliminating spurious nodes and edges which considerably accelerates reachability analysis and it has been incorporated into our implementation.

7. Related Work

CFL-reachability The core idea of our analysis can be viewed as the higher-order analogue to classic CFL-reachability analyses [21, 22] for first-order programs: there is an underying control flow graph over which values can be looked up, and flow paths can be eliminated by a grammar enforcing call/return and data structure alignments. The goal-directed nature of lookup in our analysis is also a property observed in CFL-reachability, where it is termed a *demand-driven analysis* [7, 13].

First-order CFL assumes a fixed control-flow graph since the programs are first-order; our work can be viewed as extending the philosophy of demand-driven CFL-reachability lookup to higher-order functional programs. Higher-order demand-driven lookup is much more challenging than the first-order case: nonlocal variables require a search up the "access links" for the function that captured it in closure. This is not naturally expressed via CFL labels and is why we use a push-down system (PDS), a string-less PDA, as the root machine of our algorithm.

Another issue arising in functional programs is that control flow itself depends on the data flow; in the CFL-reachability literature the analyses begins with a fixed control flow graph, whereas we calculate the control flow graph in parallel based on how PDS-reachable data propagates. In other words, our CFG is synergystially constructed based on what values can reasonably flow, and over time the CFG precision mutually informs the value precision to lead to significantly better accuracy in both. This latter property

is a key feature of higher-order program analysis in general, not just CBA.

Higher-order program analyses have been built which also incorporate the call-return alignment of CFL reachability [10, 26], but these higher-order analyses are not demand-driven and have a different algorithmic basis. We now contrast CBA with these and other higher-order program analyses.

Higher-order program analysis As mentioned in the Introduction, program analyses today are generally constructed by a finitization process on the operational semantics; it is still a state- and transition-based representation of the program, primarily focused on constructing mappings of variables to data [15]. We informally call these ABC analyses, in contrast to the Control-Based Analysis (CBA) presented here.

In the most basic abstract interpretation, a new state is created for each new store/environment, and the number of states grows very rapidly. Simplifications such as store widening must then be employed to collapse the state space. In our analysis, the global state information is fundamentally polynomial in size. The demand-driven nature of nonlocal variable lookup, derived from optimal lambda reduction lookup, is not found in existing abstract interpretations.

The end product of an abstract interpretation is a state graph which represents an abstraction of the operational semantics. Each state represents control being at a particular program point, with particular (abstracted) values for variables in the environment or heap. One of the goals of a good abstract interpretation is to minimize the number of states while not losing expressiveness of the analysis; spurious states tend to multiply, forming yet more spurious states and causing poor performance. [10], for example, shows how a more fine-grained analysis can lead to fewer states. But even a highly tuned ABC analysis will need to produce multiple states per program point. CBA produces a state graph in the form of the happens-before relation which is clearly the *least* graph in this space in the sense that it only contains nodes corresponding to non-dead program points and edges corresponding to control flow transitions that could have been taken. Any state graph produced by ABC should be soundly embeddable in the corresponding CBA machine.

Naïve abstract interpretation is flow-sensitive in that the analysis is sensitive to the order of assignment operations, but the naïve algorithms are space- and time-inefficient and furthermore stores need to be merged via a standard store widening transformation which then loses flow-sensitivity. It is still possible to recover this information using abstract garbage collection techniques [10, 17]. CBA, on the other hand, is flow-sensitive and garbage-free by construction: the context-based variable lookup methodology is fundamentally flow-sensitive. In a practical implementation of CBA, 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.

The closest previous works to CBA are the higher-order pushdown flow analyses [3, 9, 10, 26]. They also align calls and returns using ideas from first-order CFL-reachability analyses. Unlike CBA, these systems model a call stack in the natural forward direction of computation – the program execution can directly be abstracted to a PDA where the PDA stack is the call stack. CBA is different in that it needs two stacks – one for the nonlocal variable lookup continuation and one for the call stack. Since two-stack PDA's are Turing complete, we must finitize one of the two stacks; in this work, we choose to finitize the call stack, although other options may be worth considering. This is a fundamental consequence of the nature of reverse lookup in CBA and shows how CBA has a significantly different algorithmic basis compared to the forward analyses.

Complexity-wise, CBA reflects this different algorithmic basis. Fixing the finitized call stack depth at some constant k, kCBA is polynomial-time; this is surprising and unintuitive given that even 1CFA (which seems to have comparable expressiveness to 1CBA) is EXPTIME-complete [25]. A deeper analysis of kCFA has shown the exponential behavior requires the number of different nonlocal contexts in which variables occur to be the order of the size of the program itself. Although CBA's nonlocal lookup operations are stored on a separate (non-finitized) stack, the precision of these operations depends upon the accuracy of the (finitized) call stack. We conjecture that, for a program with a maximal lexical nesting depth of c, a $k+c$-level call stack CBA will be at least as expressive as kCFA (and in fact should be more expressive in many cases due to alignment of calls and returns). If c is the order of the size of the program, CBA will become exponential as well; this is in consonance with the worst-case exponential bound on kCFA.

CFA2 was originally expressed as a monomorphic analysis, and was extended to kPDCFA in [10] by adding the standard notion of polymorphic contour: making multiple copies of function bindings to achieve polyvariance. kPDCFA and kCBA cannot be directly compared, but kPDCFA abstracts an unbounded call stack in the PDA whereas kCBA needs to put a fixed bound k on call stack depth and so kPDCFA is more expressive in that dimension. On the other hand, CBA is provably polynomial for fixed k as opposed to exponential.

The issue of worst-case complexity also depends on how widely used non-local variables are; it is only in the case where there are many levels of non-local that the worst-case of exponential time arises. There already do exist several provably polynomial context-sensitive analysis [8, 18]. In particular, [18] defines mCFA, 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 as non-locals that are distinguished in kCFA are merged in mCFA. In kCBA, 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 [10, 11, 13] 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 CBA is.

Optimal λ-reduction There is a long history of graph-based reduction systems for higher-order programs [14, 28]. Our graph-based notation of variable lookup was partly inspired by the graph representation of programs in optimal lambda graph reduction a la Lamping *et al* [6, 12]; the relation is more qualitative than quantitative. They share a philosophy concerning non-local variable lookup: in both CBA and optimal reduction, the non-locals are not copied in but rather looked up via a careful trace back to their originating definition. In optimal reduction, the graphs are mutated to express reduction, whereas we only *add* edges to the graphs in our execution (both in the operational semantics and analysis). Note that our graphs significantly differ from sharing graphs in other fundamental ways, for example sharing graphs directly wire nonlocal arguments whereas we use an access link style notion to look up non-locals.

The fan-in and fan-out nodes of optimal reduction are similar to the enter and exit markings $\mathtt{z}\downarrow$ / $\mathtt{z}\uparrow$ we place on call sites, and

optimal reduction also shares the problem of how to pair the aligned in and out nodes at call site entry and exit with each other without invalidating nonlocal variable lookup. In optimal reduction, the alignment problem is much more severe than for us. Their reduction system allows cycles, which creates an ambiguity which must be solved with special bracket nodes; while CBA also has cycles in the graphs, they are lossy and represent a point of abstraction since they lack bracket node disambiguators. We are not the first to be inspired by sharing graphs for program analyses: [24] uses sharing graphs to aid in proving properties of progam analyses. The aforecited work is using the linearity of sharing nodes whereas we are not using any linearity; we are using fan-in and fan-out which the aforecited does not use. The sharing graphs of [24] also directly wire in non-local arguments and this makes the call/return alignment very difficult to do in the general case and so it appears the result of this paper cannot be directly adapted to that framework. Overall, different aspects of sharing graphs are used, but the broader conclusion is that sharing graphs have interesting relationships with program analysis which may deserve further investigation.

8. Conclusions

In this paper we developed CBA, a new form of higher-order program analysis centered around production of a call graph. CBA needs *only* the call graph to look up variable values; the specification does not maintain any other structures. It can be viewed as a lazy version of standard program analysis, and shares the trait of laziness that variables need not be propagated that are not used in the future.

CBA shows promise from a theoretical perspective. We have established a polynomial-time bound on a higher-order program analysis which is both flow- and context-sensitive, a result we are not aware of in the literature. 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.

The practical utility of CBA remains to be seen. We present preliminary results here from a simple implementation to serve as a confirmation of correctness, but our implementation needs significant fine tuning and would benefit from a head-to-head comparison with some state-of-the-art analyses. Although CBA'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 and inspiring efficient CBA implementations.

We 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 schooling us in the recent literature of higher-order program analysis. The original seed for this project was planted in the second author's collaboration with Joe Wells to develop sharing graph types more than ten years ago.

References

[1] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata: Application to model-checking. In *CONCUR'97*, 1997.

[2] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[3] C. Earl, M. Might, and D. V. 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 Comput. Sci.*, 1992.

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, 1993.

[6] G. Gonthier, M. Abadi, and J.-J. Lévy. The geometry of optimal lambda reduction. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1992.

[7] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Foundations of Software Engineering*, 1995.

[8] S. Jagannathan and S. Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95*, 1995.

[9] J. I. Johnson and D. V. Horn. Abstracting abstract control. In *DLS*, 2014.

[10] J. I. Johnson, I. Sergey, C. Earl, M. Might, and D. V. Horn. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming*, 2014.

[11] J. Kodumal and A. Aiken. The set constraint/CFL reachability connection in practice. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 2004.

[12] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1990.

[13] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with cfl-reachability. In *Compiler Construction*, 2013.

[14] I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[15] J. Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 2012.

[16] M. Might. Abstract interpreters for free. In *Proceedings of the 17th International Conference on Static Analysis*, 2010.

[17] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, Portland, Oregon, 2006.

[18] M. Might, Y. Smaragdakis, and D. Van Horn. Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[19] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[20] Z. Palmer and L. Facchinetti. Odefa proof-of-concept implementation. `https://github.com/JHU-PL-Lab/odefa-proof-of-concept`, 2015. Accessed: 2015-10-11.

[21] T. Reps. Shape analysis as a generalized path problem. In *PEPM*, 1995.

[22] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, 1995.

[23] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991. TR CMU-CS-91-145.

[24] D. Van Horn and H. G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP*, 2007.

[25] D. Van Horn and H. G. Mairson. Deciding kCFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, 2008.

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

[27] D. Vardoulakis and O. Shivers. Pushdown flow analysis of first-class control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, 2011.

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

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 A.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 A.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 A.3. *For any e, $\textsc{Embed}(e)$ is well-formed.*

Proof. Conditions 1 and 2 follow immediately from Definition A.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 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 A.4. *If D is well-formed and $D \longrightarrow^1 D'$ then D' is well-formed.*

Proof. Each rule of Figure 4.3 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 4.3. 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 A.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. □