A Set-Based Context Model for Program Analysis

Leandro Fachinetti[1], Zachary Palmer[2], Scott F. Smith[1], Ke Wu[1], and Ayaka Yorihiro[3]

[1] Johns Hopkins University, USA
[2] Swarthmore College, USA
[3] Cornell University, USA

Abstract. In program analysis, the design of context models is an understudied topic. This paper presents a study of context models for higher-order program analyses and develops new approaches. We develop a context model which equates control flows with the same *set* of call sites on the program stack, guaranteeing termination without the arbitrary cutoffs which cause imprecision in existing models. We then selectively polyinstantiate these contexts to avoid exponential growth. We evaluate this model and existing models across multiple higher-order program analysis families. Existing demand-driven analyses cannot support the set model, so we construct a demand-driven analysis, Plume, which can. Our experiments demonstrate that the set-based model is tractable and expressive on representative functional programs for both forward- and demand-driven functional analyses.

Keywords: program analysis, control flow, data flow, context sensitivity, higher-order, object-oriented

1 Introduction

In higher-order program analysis, there exists a fundamental tension between context sensitivity and field sensitivity (also called structure-transmitted data dependence [41]). Context sensitivity relates to how the analysis accounts for the calling context of a function while analyzing the function's body. Field sensitivity relates to how the analysis aligns constructions and destructions as it explores structured data: for instance, whether it can accurately project a field from a constructed record or, equivalently, look up a non-local variable captured in closure. Context and field sensitivity inform each other: an analysis lacking in context sensitivity may lead to spurious data flows despite perfect field sensitivity. Any analysis which is perfectly context- and field-sensitive has been shown to be undecidable [29] so, for an analysis tool to guarantee termination, some concessions must be made.

A common approach is to preserve field sensitivity by approximating context sensitivity using an abstract model. When introducing one of the first higher-order program analyses, kCFA, Shivers wrote about context models: "Choosing a good abstraction that is well-tuned to typical program usage is not a topic that I have explored in depth, although it certainly merits study." [33, p.34] The choice of context models is a critical factor in analysis precision and running time, but explorations of this question have been largely confined to truncated call strings à la kCFA [38,19,18,12,4,23,39]. Recent work has explored selective approaches to polyinstantiation [16,37] and using different context models for parts of the same program [22,21,17], but these approaches must still contend with a crucial weakness: in kCFA-like models, polyinstantiation of a saturated context will lose the oldest call site. This conflates that call site's control flows with those of other call sites and weakens the analysis.

Alternative models of control flow exist in the space of object-oriented alias analyses. The context and field sensitivity problems can be reduced to matched parenthesis problems, so they can be modeled as a linear conjunctive language (LCL) [26] reachability problem. While that problem is undecidable, performant and relatively precise approximations have been recently developed [41]. Unfortunately, it is not clear what information is lost in these approximations or which programs would be affected by using LCL reachability in an analysis.

Another recent technique, synchronized pushdown systems (SPDS) [34], involves making *no* concessions on either context sensitivity or field sensitivity but treating them as separate problems. The resulting analysis performs well on traditional object-oriented programs. But functional programs rely heavily upon the interplay of data and interprocedural control flow and we show that this approach is problematic for those programs (see Section 4.3).

In contrast with the kCFA-like models, we propose not to discard old call site information at all. Instead, we represent calling contexts as the *set* of call sites on the program stack. This identifies calls appearing at the same site but retains information about the entire sequence of calls, preventing the conflation of control flows in the k-limited models described above. This precision introduces a problem: because the set stores *call sites* rather than *called functions*, a recursive function calling itself at n sites may create 2^n different contexts, all of which analyze the same recursive function. We address this by selectively polyinstantiating contexts in a fashion similar to context tunneling [16].

We evaluate these techniques both in terms of precision and performance. Evaluating the precision of a *component* of a program analysis is a challenge: it is difficult to separate the effects of the component from how it interacts with the surrounding analysis. Our evaluation is a *reproducability* experiment: we test a Cartesian product of program analyses and context models, demonstrating that the k-cutoff and set-based context models exhibit the same difference in behavior across those analyses. Given that these differences are reproducible in different analyses, we ascribe them to the context model.

For the reproducability experiment's result to apply broadly, the analyses must be significantly different. We perform the experiment on three analyses. The first two are ADI, a state-of-the-art functional analysis [4], and an analysis similar to mCFA [24] in the style of object-oriented CFA analyses.

For the third analysis, we desired to use a higher-order analysis in a *demand-driven* style. Demand-driven analyses differ from forward-running analyses in that they only look up values on demand rather than propagating abstract heaps throughout the program. Demand-driven analyses were originally developed for first-order programs [30,15,28,31,32,5,13] where they were shown to achieve good performance/expressiveness trade-offs. Unfortunately, previous higher-order demand-driven analyses [9,27,6,7,35,34] do not support set-based context models. We develop a new demand-driven higher-order program analysis, Plume, to support set-based contexts and selective polyinstantiation. We prove that Plume is sound, decidable, and strictly more expressive than DDPA [6], a previous analysis in this class.

We describe Plume, set-based context models, and selective polyinstantiation in Section 2. We formalize Plume in Section 3. Precision and performance testing are discussed in Sections 4 and Section 5. (The full performance evaluation as well as the proofs of Plume's soundness and decidability appear in a supplemental report [8].) Section 6 discusses related and future work; we conclude in Section 7.

2 Overview

This section gives an overview of Plume, set-based context models and selective polyinstantiation. Although our examples focus on the Plume analysis, set-based context models and selective polyinstantiation are applicable to other analyses as well. We discuss their use in other analyses in later sections.

2.1 Shallow A-Normalized Lambda Calculus

Throughout this paper, we will focus on a shallow A-normalized lambda calculus. The grammar for this language appears in Figure 1. An expression is a list of clauses to be executed in sequence; the result is the last assigned variable in that sequence.

Call site annotations Θ are used for selective polyinstantiation; we discuss them in Section 2.4 below.

$e ::= [c, \ldots]$	*expressions*	$v ::= f$	*values*
$c ::= x = b$	*clauses*	$f ::=$ `fun` x `->` (e)	*functions*
$b ::= f \mid x \mid x\ x\ \Theta$	*clause bodies*	$\Theta ::= [\theta, \ldots]$	*call site annotation lists*
$x ::=$ *(identifiers)*	*variables*	$\theta ::=$ `@`x	*call site annotations*
$E ::= [x = v, \ldots]$	*environments*		

Fig. 1. Grammar of Analyzed Language

We require that all programs are *alphatized*: all clauses define a unique variable. This creates a bijection between variable names and program points, simplifying the theory and presentation. We include more language features in the implementation evaluated in Sections 4 and 5.

2.2 Plume By Example

Plume is a *demand-driven* program analysis inspired by DDPA [6]. Plume proceeds by incrementally constructing a *contextual control flow graph* (CCFG). This structure tracks control flow in a context-sensitive manner by associating a calling context with each graph node. DDPA does not include context information in CFG nodes. The CCFG is the only data structure in Plume; there are no stores or program states. Plume iteratively expands call sites, effectively inlining function bodies into the CCFG.

Consider the example program in Figure 2. `f` is simply an η-converted identity function. The functions defined in `g` and `h` are never called; they are simply used

```
1  f = fun x -> ( # λx.(λy.y)x
2    i = fun y -> ( # λy.y
3      ri = y;
4    );
5    rf = i x;
6  );
7  g = fun p -> ( rg = p ); # λp.p
8  h = fun q -> ( rh = q q ); # λq.q q
9  c1 = f g; # evaluates to λp.p
10 c2 = f h; # evaluates to λq.q q
```

Fig. 2. Identity Example: ANF

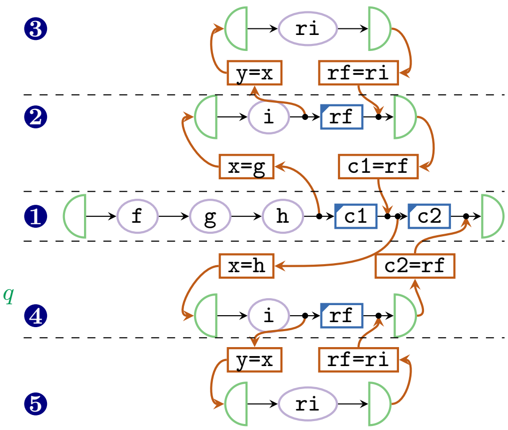

Fig. 3. Identity Example: CCFG Result

as distinct values for discussion. In the execution of the program, the call assigned to variable `c1` will return `g`; the call assigned to variable `c2` will return `h`.

Constructing the CCFG Plume's CCFG initially consists only of the middle row of nodes (marked ❶) representing top-level program points. Because the analysis is demand-driven, we are not concerned with `f`, `g`, and `h`: they are value assignments and, if those values are required, we will look them up on demand.

The first function call appears at `c1`. We start by tracing *backward* from `c1` to find the called function. We pass two clauses — `h = ...` and `g = ...` — which do not define `f` and so are skipped. We then discover variable `f` and its function.

We next add the second row of nodes (marked ❷). The top line is the body of the `f` function; the two nodes below are *wiring* nodes that represent the call's parameter and return data flows.

This is why the analysis does not require a store: values may be established on demand by retracing these control flow edges.

The call site `rf` is now reachable via non-call nodes. Expanding `rf` yields the top row of nodes (marked ❸).

The call site `c2` becomes reachable so, like before, we identify `f` as the called function. We do not reuse the previous `f` subgraph: because this call occurs at a distinct site, we create a new subgraph. This subgraph appears in the second-to-last row in the diagram (marked ❹). Finally, we expand the call site `rf`, adding the nodes in the last row (marked ❺).

The completed CCFG is Plume's result and can be used to perform lookups. To look up `c2` from the end of the program, for instance, we move backward through the graph and encounter `c2=rf`; our lookup therefore reduces to finding the value of `rf` from that node. Moving backward from `c2=rf`, we discover `rf=ri`, changing our goal to finding the value of `ri`. This process proceeds through `ri=y`, `y=x`, and `x=h`, eventually leading us to the function defined on line 8.

This example does not show the lookup of a non-local variable. This is a delicate process in demand-driven analyses and is solved in Plume with a *stack* of lookup variables, a technique originally developed for DDPA [6]. We discuss this approach in Appendix A in the supplemental material [8] for reasons of space.

2.3 Models of Context Sensitivity

Multiple passes over a program point allow different calls of a function to be distinguished. These passes manifest in Plume as copies of the function in the CCFG; in other analyses, they may manifest as additional program states, edges in an automaton, or similar structures. A decidable program analysis must limit how many times it analyzes each program point to keep these structures finite.

One typical finitization approach is to associate each function call with a calling context derived from the circumstances under which the function is called. In kCFA [33], for instance, calling contexts are a list of the k most recent call sites visited by the program. In polyvariant P4F [12], calling contexts are represented by the call site from which we have most recently returned. DDPA [6] and Plume, like many program analyses, are parametric in the model of calling context used. We use Σ to denote a context model and use Σ_k to denote the model in which contexts are the top k call sites of the stack.

```
o = fun x -> ( r = x x; )
z = o o; # (λx.x x)(λx.x x)
```

Fig. 4. Ω-combinator: ANF

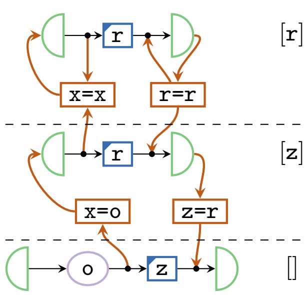

Fig. 5. 1Plume CCFG

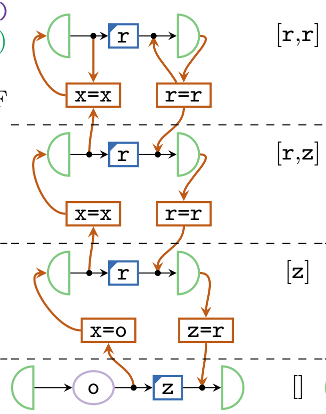

Fig. 6. 2Plume CCFG

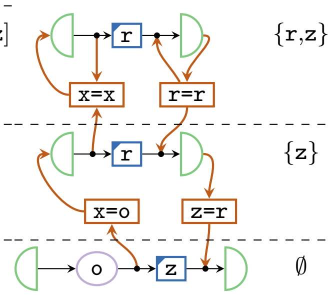

Fig. 7. SetPlume CCFG

One contribution of this paper is the development of a tractable analysis using a *set-based* context model denoted $\Sigma_{\texttt{Set}}$, which represents contexts as the set of *all* call sites on the call stack. $\Sigma_{\texttt{Set}}$ addresses a weakness of traditional k-cutoff models: recursion. Consider the non-terminating Ω-combinator program in Figure 4 analyzed by Plume using Σ_1 (which we call 1Plume). The generated CCFG appears in Figure 5. Initially, the calling context is empty: the top level of the program is not generated from any calls. When the first `r` call site is expanded, it introduces nodes associated with the context [`r`]. (The context for groups of nodes appears to the right.) The `z` is dropped from this context because the list is limited to size 1. When the second `r` call site is expanded, we also associate that call with [`r`], reusing the subgraph associated with this context.

By the time a program analysis begins to reuse resources in recognition of the recursive call, we have lost all information about where the recursive call started. In the final context of the CCFG, [`r`], the call site `z` is no longer present. If the same recursive function were called in multiple locations, all such calls would eventually converge on the [`r`] context and their control flows would be conflated. As illustrated in Figure 6, increasing k does nothing to prevent this: the context [`r`,`r`] has similarly lost all information before the recursive call.

Recent developments in object-oriented k-cutoff models mitigate this problem in a variety of ways. Context tunneling [16] is the most relevant to this case: at each call, we can decide whether to polyinstantiate the context as above or to proceed with the context we already have. This technique is almost identical in expressiveness to selective polyinstantiation, which we discuss below. Instead of applying this technique to prevent information loss, however, we use the set-based context model (which loses none of this information) and apply this technique to support performance.

The CCFG in Figure 7 is generated by Plume using $\Sigma_{\texttt{Set}}$ (which we call SetPlume); SetPlume does not conflate recursive calls in this way. While this CCFG initially appears to be the same as the one generated by 1Plume, the contexts associated with each node retain every call site encountered since the top level of the program. As a consequence, calls to the same function at different sites will not be conflated. This is critical to allow recursive polymorphic functions such as `List.map` to be analyzed correctly.

SetPlume is not the first program analysis to retain the context of recursive calls by eshewing k-limited context. LCL reachability-based analyses [41] have a similar approximation which tracks the call stack even more precisely at the expense of some data flow information. However, most state-of-the-art analyses use a k-cutoff model [18,4] or rely upon an externally generated CFG [35,34].

2.4 Selective Polyinstantiation

$\Sigma_{\texttt{Set}}$ distinguishes calls to recursive functions at different call sites by retaining information about where the recursive function was called. Unlike Σ_k, there is no point at which polyinstantiation loses information. As a result, $\Sigma_{\texttt{Set}}$ is vulnerable to an exponential expansion of contexts. We address this issue using a selective polyinstantiation technique similar to the context tunneling work mentioned above.

```
fact0 = fun self -> (
  factfn = fun n -> (
    factret =
      ifzero n then (
        factret1 = 1;
      ) else (
        n' = n - 1;
        selfself = self self @self;
        factn' = selfself n' @n;
        fact = factn' * n;
); ); );
fact = fact0 fact0 @self;
x = 5;
fact5 = fact x;
```

Fig. 8. Factorial Example: Extended ANF

Consider a recursive function whose body contains n recursive call sites (e.g. an expression interpreter). This recursive function may be called through any combination of the n recursive sites, leading to 2^n possible contexts. This is clearly intractable. Further, it is a waste of effort: the analysis is only more precise if different recursive calls yield different (abstract) values, and the inference of polymorphic recursion is known to be undecidable [14].

Our strategy is to be selective: when a function calls itself, we choose not to polyinstantiate it. The challenge is that, while $\Sigma_{\texttt{Set}}$ correctly identifies and avoids polyinstantiation for recurring *call sites*, it does not identify recursive *functions*. To identify a recursive call, we must take into account both the position of call site and the function being called there. We explicitly mark each call site with the identities of those functions which should *not* be polyinstantiated if they are called in that location.

Consider the self-passing factorial program written in Figure 8 in an extended ANF. The only contexts generated during the analysis of this program in SetPlume will be $\emptyset$ and $\{\texttt{fact5}\}$ despite the fact that there are several other function calls in the program. Upon reaching line 8, for instance, the analysis looks up `self` and discovers that the function being called is the one assigned to `fact0`. Because the ANF is alphatized, the name of the function's parameter, `self`, uniquely identifies it in the program. The annotation `@self` indicates that, if this function is called on line 8, it should *not* be polyinstantiated. As a result, this call site is wired to the body of that function associated with the current context, $\{\texttt{fact5}\}$, rather than to a new copy. These annotations are often automatically inferrable: the performance benchmark programs evaluated in Appendix D of the supplement [8] are written in an ML-like surface language *without* annotations and are then machine translated to an annotated ANF.

Selective polyinstantiation is almost equivalent in expressiveness to context tunneling. Both systems determine whether or not to polyinstantiate based upon the pairing of call site and called function. This choice is driven here by annotations and in the context tunneling work by a global relation. (Selective polyinstantiation can differentiate between call sites within the same method while context tunneling cannot, but this distinction seems unlikely to be useful.) There are two key differences between this work and context tunneling. First: the context tunneling paper [16] uses a data-driven machine learning algorithm to generate its pairwise relation; by comparison, we use a simple lexical annotator here. Second: the motivations differ. The data-driven algorithm is used to prevent the k-limited context from losing precision; here, we apply the technique to mitigate performance concerns. Selective polyinstantiation also shares some properties with earlier work [37] which eliminate provably redundant polyinstantiations, although that work is not applicable to the set-based context model discussed here.

Note that this approach is not limited to $\Sigma_{\texttt{Set}}$ or to Plume. Selective polyinstantiation is similar to context tunneling [16], which has been applied to k-limited context models to prevent new, unimportant context information from supplanting old, important context information. Here, polyinstantiation is used to prevent a blow-up in complexity instead.

3 Formalizing Plume

We now formally define the Plume analysis. As outlined in Section 2.2, the analysis proceeds in two steps. First, the program is embedded into an initial

CCFG; second, we perform a full closure of the CCFG using information from a demand-driven value lookup algorithm. There is no store or heap; all values are looked up by following the CCFG backward from the point of interest. We define the analysis in three parts: the initial embedding and corresponding preliminary definitions (Section 3.1), the demand-driven lookup function (Section 3.2), and the CCFG closure algorithm (Section 3.3).

3.1 Preliminary Definitions

We begin by abstracting the target program. We define "hatted" analogs for each grammar term in Figure 1: $\hat{e}$ for abstract expressions, $\hat{c}$ for abstract clauses, and so on. We denote the abstraction of a concrete expression as $\alpha(e) = \hat{e}$. For convenience, we define RV as a function returning the last defined variable in an expression and use $||$ to denote list concatenation.

Recall that a CCFG is a *contextual* control flow graph; it contains context information. We begin by defining a general notion of context model, Σ.

Definition 1. *A* context model Σ *is a triple* $\langle \hat{\boldsymbol{C}}, \epsilon, \oplus \rangle$ *where*
- $\hat{\boldsymbol{C}}$ *is a set whose elements, denoted* $\hat{C}$*, are calling contexts.*
- ϵ*, the "empty context", is an element of* $\hat{\boldsymbol{C}}$*.*
- *For all* $\hat{C} \in \hat{\boldsymbol{C}}$ *and all* $\hat{c}$*,* $\hat{C} \oplus \hat{c} = \hat{C}'$ *and* $\hat{C}' \in \hat{\boldsymbol{C}}$*.*

We formalize the k-cutoff and set models of Section 2 as follows:

Definition 2.
- $\Sigma_k = \langle \hat{\boldsymbol{C}}, [], \oplus \rangle$ *where* $\hat{\boldsymbol{C}}$ *contains all lists of* $\hat{c}$ *of length up to* k *and* $[\hat{c}_n, \ldots, \hat{c}_1] \oplus \hat{c}_0 = [\hat{c}_{k-1}, \ldots, \hat{c}_0]$.
- $\Sigma_{\texttt{Set}} = \langle \hat{\boldsymbol{C}}, \emptyset, \oplus \rangle$ *where* $\hat{\boldsymbol{C}}$ *is the power set of all* $\hat{c}$ *and* $\hat{C} \oplus \hat{c} = \hat{C} \cup \{\hat{c}\}$*.*

Each context model defines a distinct Plume variant; for instance, we give Plume using $\Sigma_{\texttt{Set}}$ the name SetPlume. Throughout the remainder of this section, we assume some fixed context model meeting the conditions of Definition 1.

$\hat{a} ::= \hat{c} \mid \hat{x} \overset{(\!\!\vert\hat{c}}{=} \hat{x} \mid \hat{x} \overset{\vert\!\!)\hat{c}}{=} \hat{x} \mid \textsc{Start} \mid \textsc{End}$	*annotated clauses*	$\hat{\eta} ::= \langle \hat{a}, \hat{C} \rangle$	*CCFG nodes*
$\hat{V} ::= \{\hat{v}, \ldots\}$	*value sets*	$\hat{g} ::= \hat{\eta} \ll \hat{\eta}$	*CCFG edges*
$\hat{X} ::= [\hat{x}, \ldots]$	*variable lookup stacks*	$\hat{G} ::= \{\hat{g}, \ldots\}$	*CCFG's*

Fig. 9. Analysis Grammar

Given a context model, the remaining constructs required for the Plume analysis appear in Figure 9. A CCFG $\hat{G}$ is a set of edges between contextual control flow points $\hat{\eta}$, each of which is a pairing between a program point and the calling context in which that program point is visited. To work with these graphs, we introduce the following notation:

Definition 3. *We use the following notational sugar for CCFG graph edges:*
- $\hat{a}_1 \ll \ldots \ll \hat{a}_n$ *abbreviates* $\{\hat{a}_1 \ll \hat{a}_2, \ldots, \hat{a}_{n-1} \ll \hat{a}_n\}$*.*
- $\hat{a}' \ll \{\hat{a}_1, \ldots, \hat{a}_n\}$ *(resp.* $\{\hat{a}_1 \ldots \hat{a}_n\} \ll \hat{a}'$*) denotes* $\{\hat{a}' \ll \hat{a}_1, \ldots, \hat{a}' \ll \hat{a}_n\}$ *(resp.* $\{\hat{a}_1 \ll \hat{a}', \ldots, \hat{a}_n \ll \hat{a}'\}$*).*
- *We write* $\hat{a} \lll \hat{a}'$ *to mean that* $(\hat{a} \ll \hat{a}') \in \hat{G}$ *for* $\hat{G}$ *understood from context.*

Using the above, we define the initial state of the CCFG as just the clauses of the main program, with no function calls (yet) wired in:

Definition 4. *The initial embedding of an expression into a CCFG,* $\widehat{\textsc{Embed}}(e)$*, is the graph* $\hat{G} = \langle \textsc{Start}, \epsilon \rangle \lll \langle \hat{c}_1, \epsilon \rangle \lll \ldots \lll \langle \hat{c}_n, \epsilon \rangle \lll \langle \textsc{End}, \epsilon \rangle$ *where* $\alpha(e) = [\hat{c}_1, \ldots, \hat{c}_n]$*.*

For example, the subgraph labeled ❶ in Figure 3 is the initial embedding of the Figure 2 expression.

3.2 The Lookup Function

Plume does not require an explicit representation of the heap. Instead, we look up the value of each variable when it is needed by starting from the point where it is *used* and tracing backward through the CCFG to the point where it is *defined*.

Given a CCFG $\hat{G}$, we formalize variable lookup as a relation $\hat{G}, \langle \hat{a}, \hat{C} \rangle \vdash \hat{X} \rightarrowtail \hat{v}$ which indicates that the value $\hat{v}$ may be discovered by reducing the lookup stack $\hat{X}$ from program point $\hat{a}$ in calling context $\hat{C}$. For instance, if lookup of variable $\hat{x}$ from the end of the program produces value $\hat{v}$, we may write "$\hat{G}, \langle \textsc{End}, \epsilon \rangle \vdash [\hat{x}] \rightarrowtail \hat{v}$". (As mentioned briefly in Section 2.2 and illustrated in Appendix A in the supplemental material [8], we use a *stack* of variables to facilitate looking up non-local (i.e. closure-captured) variables.) Note that the provided program point $\hat{a}$ is assumed *not* to have executed yet; each time we step backward through the graph, we are undoing the effect of the preceding clause.

We formally define this relation as follows:

Definition 5. $\hat{G}, \hat{\eta} \vdash \hat{X} \rightarrowtail \hat{v}$ *holds iff there is a proof using the rules of Figure 10.*

Given a position $\hat{\eta}$ in the CCFG $\hat{G}$ and a lookup stack $\hat{X}$, the rules in Figure 10 describe which transitions are legal during lookup. Any valid path through the CCFG to locate a variable definition corresponds to a proof in that system.

The Alias rule indicates that, when looking for variable $\hat{x}$ and about to undo the assignment $\hat{x} = \hat{x}'$, we can reduce our lookup to finding the value of $\hat{x}'$ from that point. The Value Discovery rule indicates that, when stepping back to $\hat{x} = \hat{v}$ while looking for $\hat{x}$, our lookup is complete: $\hat{v}$ is the answer. The Function Enter Non-Local and Value Discard rules represent the beginning and end (respectively) of the lookup of a closure-captured variable, using the stack to retain the variable while finding the definition site of the closure. The other two function rules represent a value flowing into or out of a function (and update the current lookup variable appropriately); the Skip rule handles clauses which do not have an impact on the current lookup.

In Section 2.2 we informally described the lookup of the value of `c2` of Figure 2 from the end of the program; formally that lookup corresponds to a proof of $\hat{G}, \langle \textsc{End}, \epsilon \rangle \vdash [\texttt{c2}] \rightarrowtail (\texttt{fun p ->} \ldots)$ in the lookup system of Figure 10, for $\hat{G}$ being the CCFG of Figure 3.

$$\textsc{Value Discovery}\quad \frac{\langle \hat{x} = \hat{v}, \hat{C}\rangle \lll \hat{\eta}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \rightarrowtail \hat{v}} \qquad \textsc{Value Discard}\quad \frac{\hat{\eta}' = \langle \hat{x} = \hat{f}, \hat{C}\rangle \qquad \hat{\eta}' \lll \hat{\eta} \qquad \hat{G}, \hat{\eta}' \vdash \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

$$\textsc{Alias}\quad \frac{\hat{\eta}' = \langle \hat{x} = \hat{x}', \hat{C}\rangle \qquad \hat{\eta}' \lll \hat{\eta} \qquad \hat{G}, \hat{\eta}' \vdash [\hat{x}'] \,||\, \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

$$\textsc{Function Enter Parameter}\quad \frac{\hat{\eta}' = \langle \hat{x} \overset{(\!|\hat{c}}{=} \hat{x}', \hat{C}\rangle \qquad \hat{\eta}' \lll \hat{\eta} \qquad \hat{G}, \hat{\eta}' \vdash [\hat{x}'] \,||\, \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

$$\textsc{Function Enter Non-Local}\quad \frac{\hat{\eta}' \lll \hat{\eta} \qquad \hat{x}'' \neq \hat{x} \qquad \begin{array}{c}\hat{\eta}' = \langle \hat{x}'' \overset{(\!|\hat{c}}{=} \hat{x}', \hat{C}\rangle \\ \hat{c} = (\hat{x}_f\ \hat{x}_v\ \hat{\Theta})\end{array} \qquad \hat{G}, \hat{\eta}' \vdash [\hat{x}_f, \hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

$$\textsc{Function Exit}\quad \frac{\hat{\eta}' = \langle \hat{x} \overset{|\!)\hat{c}}{=} \hat{x}', \hat{C}\rangle \qquad \hat{\eta}' \lll \hat{\eta} \qquad \hat{G}, \hat{\eta}' \vdash [\hat{x}'] \,||\, \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

$$\textsc{Skip}\quad \frac{\hat{\eta}' = \langle \hat{x}'' = \hat{b}, \hat{C}\rangle \qquad \hat{\eta}' \lll \hat{\eta} \qquad \hat{x}'' \neq \hat{x} \qquad \hat{G}, \hat{\eta}' \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}{\hat{G}, \hat{\eta} \vdash [\hat{x}] \,||\, \hat{X} \rightarrowtail \hat{v}}$$

Fig. 10. Abstract Value Lookup

3.3 CCFG Closure Construction

Given a CCFG, the lookup function allows us to determine the values that variables may have. We can use this to in turn deductively close over the CCFG: we add to the CCFG when we discover new control flows based upon looking up values of variables. In this way, CCFG closure and value lookup work in tandem: closure grows the CCFG based upon lookup, that growth increases the set of values that lookup provides, closure grows the CCFG further, and so on.

When a function application is reached with a novel function-argument pair, we add its body to the graph and add edges wiring that body around the call site, effectively inlining that function as described in Section 2.2. We pair each of the function's clauses with the calling context $\hat{C}$ in which they will be executed. Below, we formalize this process as a function: it creates an edge from each predecessor of the call site ($\textsc{Preds}(\hat{\eta})$) to a parameter wiring node ($\langle \hat{x}_0 \overset{(\!|\hat{c}}{=} \hat{x}_1, \hat{C}'\rangle$), connects that wiring node to the body of the function via a sequence of edges, adds an edge from the body to a return wiring node ($\langle \hat{x}_2 \overset{|\!)\hat{c}}{=} \mathrm{RV}(\hat{c}_n), \hat{C}'\rangle$), and then draws edges from that return wiring node to the call site's successors ($\textsc{Succs}(\hat{\eta})$). We delegate the choice of calling context $\hat{C}'$ to the caller of the wiring function.

Definition 6. *Let* $\widehat{\text{W}\textsc{irefun}}(\hat{\eta}, \texttt{fun}\ \hat{x}_0\ \texttt{->}\ \texttt{(}[\hat{c}_1, \ldots, \hat{c}_n]\texttt{)}, \hat{x}_1, \hat{x}_2, \hat{C}') =$

$$\textsc{Preds}(\hat{\eta}) \ll \langle \hat{x}_0 \overset{(\hat{c}}{=} \hat{x}_1, \hat{C}' \rangle \ll \langle \hat{c}_1, \hat{C}' \rangle \ll \ldots \ll \langle \hat{c}_n, \hat{C}' \rangle \ll \langle \hat{x}_2 \overset{)\hat{c}}{=} \text{RV}(\hat{c}_n), \hat{C}' \rangle \ll \textsc{Succs}(\hat{\eta})$$

where $\hat{\eta} = \langle \hat{c}, \hat{C} \rangle$, $\textsc{Preds}(\hat{\eta}) = \{\hat{\eta}' \mid \hat{\eta}' \lll \hat{\eta}\}$, *and* $\textsc{Succs}(\hat{\eta}) = \{\hat{\eta}' \mid \hat{\eta} \lll \hat{\eta}'\}$.

We describe a call site which can be reached via a control flow from the beginning of the program (and therefore must be analyzed) as *active*:

Definition 7. *The predicate* $\widehat{\text{A}\textsc{ctive}?}(\hat{\eta}', \hat{G})$ *holds iff path* $\textsc{Start} \ll \hat{\eta}_1 \ll \ldots \ll \hat{\eta}_n \ll \hat{\eta}'$ *appears in* $\hat{G}$ *such that no* $\hat{\eta}_i$ *is of the form* $\langle \hat{x} = \hat{x}'\ \hat{x}''\ \hat{\Theta}, \hat{C} \rangle$.

We are now ready to define the closure construction.

Contextual Application

$$\frac{\hat{\eta} = \langle \hat{c}, \hat{C} \rangle \qquad \hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3\ \hat{\Theta}) \qquad \widehat{\text{A}\textsc{ctive}?}(\hat{\eta}, \hat{G}) \qquad \hat{G}, \hat{\eta} \vdash [\hat{x}_2] \rightarrowtail \hat{f} \qquad \hat{G}, \hat{\eta} \vdash [\hat{x}_3] \rightarrowtail \hat{v} \qquad \hat{f} = \texttt{fun}\ \hat{x}_4\ \texttt{->}\ \texttt{(}\hat{e}\texttt{)} \qquad \texttt{@}\hat{x}_4 \notin \hat{\Theta} \qquad \hat{C}' = \hat{C} \oplus \hat{c}}{\hat{G} \mathrel{\widehat{\longrightarrow}^1} \hat{G} \cup \widehat{\text{W}\textsc{irefun}}(\hat{\eta}, \hat{f}, \hat{x}_3, \hat{x}_1, \hat{C}')}$$

Acontextual Application

$$\frac{\hat{\eta} = \langle \hat{c}, \hat{C} \rangle \qquad \hat{c} = (\hat{x}_1 = \hat{x}_2\ \hat{x}_3\ \hat{\Theta}) \qquad \widehat{\text{A}\textsc{ctive}?}(\hat{\eta}, \hat{G}) \qquad \hat{G}, \hat{\eta} \vdash [\hat{x}_2] \rightarrowtail \hat{f} \qquad \hat{G}, \hat{\eta} \vdash [\hat{x}_3] \rightarrowtail \hat{v} \qquad \hat{f} = \texttt{fun}\ \hat{x}_4\ \texttt{->}\ \texttt{(}\hat{e}\texttt{)} \qquad \texttt{@}\hat{x}_4 \in \hat{\Theta}}{\hat{G} \mathrel{\widehat{\longrightarrow}^1} \hat{G} \cup \widehat{\text{W}\textsc{irefun}}(\hat{\eta}, \hat{f}, \hat{x}_3, \hat{x}_1, \hat{C})}$$

Fig. 11. Control Flow Graph Closure Construction

Definition 8. *We define* $\hat{G} \mathrel{\widehat{\longrightarrow}^1} \hat{G}'$ *to be the least relation satisfying the rules in Figure 11. We write* $\hat{G}_0 \mathrel{\widehat{\longrightarrow}^*} \hat{G}_n$ *to denote* $\hat{G}_0 \mathrel{\widehat{\longrightarrow}^1} \ldots \mathrel{\widehat{\longrightarrow}^1} \hat{G}_n$. *We write* $\widehat{\longrightarrow}^!$ *to denote the transitive closure of* $\widehat{\longrightarrow}^1$.

To understand the rules in this definition, consider a function-argument pair at a call site. We must select a calling context $\hat{C}$ to ascribe to the call. The rules are otherwise similar: given an active call site for which values can be found for both the function ($\hat{x}_2$) and argument ($\hat{x}_3$), we wire the body of the called function around the call site ($\hat{x}_1$) using the wiring function defined above. The only difference regards $\hat{\Theta}$ and $\hat{C}$. Since the program is alphatized, all function parameters are unique, so we identify each function by its parameter ($\hat{x}_4$). If the parameter appears in a call site annotation in $\hat{\Theta}$, we do *not* polyinstantiate the call site (the Acontextual Application rule); if the parameter *does not* appear in the annotations, then we *do* (the Contextual Application rule).

3.4 Soundness and Decidability

The Plume analysis defined above is both sound and decidable. Here, soundness means that the lookup relation $\hat{G}, \hat{\eta} \vdash \hat{X} \rightarrowtail \hat{v}$ is always an over-approximation: if a value can exist at runtime, then the lookup relation holds for its abstract counterpart. Soundness is demonstrated in Appendix C.1 in the supplemental material [8] in two parts: first by showing the operational semantics in Appendix B in the supplemental material [8] equivalent to a *graph-based* operational semantics and then by showing the Plume analysis to be an abstraction of the latter.

Decidability proceeds by upper bounding the size of the CCFG and then by a counting argument. This proof appears in Appendix C.2 in the supplemental material [8].

4 Evaluation of Precision

In this section, we evaluate the precision of the analysis techniques presented in this paper. We perform this evaluation in three parts:

1. We directly compare Plume to DDPA, a closely-related functional analysis.
2. We compare the context models Σ_k and $\Sigma_{\texttt{Set}}$ and evaluate the precision impact of selective polyinstantiation. We do so via a reproducibility experiment involving multiple functional analyses.
3. We consider another state-of-the-art analysis technique — synchronized pushdown systems [34] — and discuss how it may apply to functional programs.

All higher-order program analyses evaluated in this section are available as supplementary material associated with this submission.

4.1 *k*Plume $\geq$ *k*DDPA

DDPA [6], like Plume, is a demand-driven higher-order functional program analysis. Both analyses iteratively construct a CFG and use on-demand lookups rather than explicit value stores. Unlike Plume, DDPA uses an *acontextual* control flow graph (ACFG); calling contexts are represented as an extra parameter in lookup. The ACFG in DDPA is much smaller than the CCFG of Plume, but (1) the graph closure rules of DDPA perform all lookups irrespective of context and (2) the caching structures necessary to make DDPA efficient are of the same size complexity as Plume's CCFG.

Like Plume, DDPA is parametric in its context model, but DDPA is more restrictive and cannot support $\Sigma_{\texttt{Set}}$. With list-based models, the analyses are directly comparable and kPlume is *more precise* than kDDPA. Formally,

Theorem 1. *For any program $\hat{e}$ and any natural number k, let $\dot{G}$ be the ACFG produced for $\hat{e}$ by kDDPA and let $\hat{G}$ be the CCFG produced for $\hat{e}$ by kPlume. Then, for any variable $\hat{x}$ and program point $\hat{c}$ in $\hat{e}$, every value produced by lookup on $\hat{G}$ in kPlume is also produced by lookup on $\dot{G}$ in kDDPA.*

For space, the proof of this Theorem appears in Appendix C.3 in the supplemental material [8]. As kPlume subsumes kDDPA, we elide kDDPA from the remainder of this discussion.

4.2 Comparing Context Models

We now focus not on Plume or any one analysis but instead upon the effect that context models and selective polyinstantiation have on functional program analyses in general. We cannot simply compare two analyses: it would be unclear how the choice of analysis affected the result. We cannot even do so while holding the rest of the analysis theory constant (e.g. comparing kPlume vs. SetPlume) as the results may only pertain to the theory in question (e.g. Plume).

To draw conclusions about context sensitivity models independent of the program analysis, we examine the *reproducibility* of changes as the program analysis itself is varied. We compare pairs of program analysis from a variety of analysis families; each analysis in a pair differs from its counterpart only by context sensitivity model, while each pair differs from the other pairs significantly. We contend that, if changing the context sensitivity model of an analysis produces an effect which is consistent across all pairs, it is reasonable to ascribe this effect to the context model rather than to the program analyses. This conclusion is more reliable the larger the differences are between the analysis families. We therefore conduct our experiments on the following families of program analyses:

- Plume, the demand-driven functional program analysis family in this paper.
- ADI, a state-of-the-art forward functional program analysis family [4].
- mADI, a modification of [4] using techniques from mCFA [24] to more closely match object-oriented program analysis behavior.

We chose ADI to represent a series of higher-order program analyses that include P4F [12], AAC [19], PDCFA [18], CFA2 [39], and others. ADI is the most recent of the series and its precision is the state-of-the-art. ADI's reference implementation does not include a notion of context sensitivity so, for these experiments, we use a purpose-built implementation of ADI over the same ANF language used by Plume. This artifact yields two analyses, kADI and SetADI, with context sensitivity models identical to kPlume and SetPlume.

We also modified ADI to produce an analysis family called mADI that models the precision of object-oriented CFA-based analyses [24]. The main distinction is in how non-local variables are handled when constructing a closure: ADI stores a reference to the non-local while mADI stores a fresh copy of its value. As a result, mADI is *less precise* than ADI but *more performant*. mADI is to ADI what mCFA [24] is to kCFA. Just as with ADI, we define two variants of mADI with different context models: kmADI uses Σ_k and SetmADI uses $\Sigma_{\mathtt{Set}}$. (Note that the ADI paper [4] only used a list model).

Functional Test Cases Presently, no standard suite of functional precision benchmarks exist. For this experiment, we developed a series of small programs which are representative of common functional programming patterns:

- **rec-ident**, two calls to a recursive identity function. This function recurses, decrementing a counter to zero, and then returns its argument. It is called once with an integer and again with a boolean.
- **list-2map**, which generates an integer list in a loop and then maps over that list twice. The first mapper is `(+1)`; the second mapper is `(==0)`.
- **nest-pairmap**, which uses a homogeneous pair mapping function to increment the elements of a pair (as in: `pairmap (pairmap inc)((0,1),(1,0))`) or to convert them to boolean values.
- **foldl-2L2F**, which performs two left folds on two distinct lists. The first list (of integers) is summed; the second list (of booleans) is "and-ed".
- **foldl-2L1F**, which generates the same lists as foldl-2L2F using a single mapping function with case analysis.
- **foldl-1L2F**, which folds over a single list of integers twice. The first fold sums the list; the second fold produces `true` iff the list contains no zeroes.

Each of the tests above calls a function on two types of primitive data: integers and booleans. For each of the above programs, we ran each analysis both with selective polyinstantiation annotations and without them. (k-limited analyses without selective polyinstantiation are presented here for completeness but are not representative of the state of the art.) A test passes if the analysis can distinguish integers from booleans in every case.

For analyses not using k-cutoff models, we indicate whether the test passed (denoted ✓) or failed (denoted ✗) by the above criteria. For analyses using k-cutoff models, we give the minimum value of k necessary for the test to pass (or ✗ if no such k exists). In real programs, the two function calls to be distinguished do not necessarily appear side by side. To simulate this, we η-converted the two call sites some number of times d; thus, d appears in the results in places where the number of η conversions affects the choice of k.

Analysis	kPlume		SetPlume		kADI		SetADI		kmADI		SetmADI		Boomerang	Bmg. SPDS
Annot.?	yes	no	yes	no	yes	no	yes	no	yes	no	yes	no	n/a	n/a
rec-ident	1+d	✗	✓	✓	1+d	✗	✓	✓	1+d	✗	✓	✓	✓	✓
list-2map	2+d	✗	✓	✓	2+d	✗	✓	✓	2+d	✗	✓	✓	✓	✗
nest-pairmap	3+d	3+d	✗	✗	3+d	3+d	✗	✗	3+d	3+d	✗	✗	✗	✗
foldl-2L2F	2+d	✗	✓	✓	2+d	✗	✓	✓	2+d	✗	✓	✓	✓	✗
foldl-2L1F	2+d	✗	✓	✓	2+d	✗	✓	✓	2+d	✗	✓	✓	✓	✗
foldl-1L2F	2+d	✗	✓	✓	2+d	✗	✓	✓	2+d	✗	✓	✓	✓	✗

Fig. 12. Precision of Analyses on Functional Test Cases

Functional Test Results The results of our experiments appear in Figure 12. (Note that Boomerang and Boomerang SPDS analyses are not list-vs-set and are discussed below.) Some clear patterns emerge from these results.

First and foremost: the differences between the Σ_k and $\Sigma_{\texttt{Set}}$ context models are reproducible across all three analysis families. Each family's four-column group is identical. This degree of similarity suggests that the change in behavior is, in fact, due to the context model.

Second: selective polyinstantiation had no impact on the precision of $\Sigma_{\texttt{Set}}$. This is intuitive as these functions do not exhibit polymorphic recursion. In agreement with previous work [16], selective polyinstantiation *improved* the Σ_k analyses. This is because Σ_k may lose information on polyinstantiation; $\Sigma_{\texttt{Set}}$ does not.

Third: $\Sigma_{\texttt{Set}}$ fails on `nest-pairmap`. In this example, Σ_k requried three call sites worth of context: one for the outer `pairmap` call, one for the inner `pairmap` call (which served as the outer call's mapping function), and one to call the element mapping function itself. Because this test was not recursive, no annotations were present. $\Sigma_{\texttt{Set}}$ failed on this example because it conflated the calls to the two mappers (the inner `pairmap` function and the element mapper), as they occurred at the same call site (within `pairmap` itself). Σ_k succeeded here because it admits duplicate call sites in its contexts.

In conclusion, the precisions of $\Sigma_{\texttt{Set}}$ and Σ_k are incomparable: each has advantages over the other. $\Sigma_{\texttt{Set}}$ succeeds unconditionally in most cases; selective polyinstantiation merely improves performance. Σ_k without selective polyinstantiation unsurprisingly fails in all recursive cases; with selective polyinstantiation, it suceeds on every case (including `pairmap`). But annotated Σ_k is still fragile because k must be large enough to accommodate d, the number of polyinstantiations between the two calls' nearest ancestor, which cannot be determined at analysis time.

4.3 Synchronized Pushdown Systems

Two types of precision are key to higher-order program analyses: context sensitivity (specifically with respect to interprocedural control flow) and so-called "field sensitivity" or "structure-transmitted data dependence" (such as which values were stored in a particular record or object field). Any analysis with perfect precision in both of these forms is known to be undecidable [29], so program analyses must decide which concessions to make. In SetPlume, for instance, context sensitivity is approximated with a set while field sensitivity is handled by the variable lookup stack $\hat{X}$, which is represented by the stack of a pushdown automaton in our implementation and not approximated.

Boomerang SPDS [34] uses a *synchronized pushdown system*: both context and field sensitivity are represented without approximation but in separate pushdown automata. Boomerang SPDS's separation of these concerns showed promise but functional programs rely upon the interplay between control and data flow, so we chose to run the examples from the previous section on these two analyses to investigate their precision on common functional-style code.

The Boomerang analysis family artifacts perform analysis of at-scale Java programs and not our ANF grammar, so we translated each of our examples by hand. These translations attempt to preserve the control flow of the original program while minimizing the number of program points introduced.

Our results from running these experiments appears in the rightmost two columns of Figure 12 above. The original Boomerang analysis bears a striking resemblance to the behavior of set-based context models on these examples. Boomerang SPDS, on the other hand, failed on every example except for rec-ident. This is unsurprising in retrospect: Boomerang SPDS intentionally disregards interactions between interprocedural calls and structured data flow.

This interaction does not appear in rec-ident (as there is no structured data) but is critical in every other example; indeed, that type of interaction is common in functional programs and in related higher-order object-oriented design patterns such as the Visitor Pattern. Contrary to suppositions in the SPDS paper [34], these results suggest that the SPDS technique is *not* appropriate for higher-order programming patterns in functional languages.

4.4 Threats to Validity

Test cases. There does not presently exist a standard functional test suite for analysis precision. The test cases presented here represent common functional programming patterns but are not numerous or complete.

Translations. The conclusions regarding the Boomerang family of analyses rely upon translations of functional programming idioms to Java. We only make claims regarding the Boomerang analysis technique with respect to existing functional programming languages and not with respect to the object-oriented languages for which those analyses were designed.

5 Summary of Performance

We subjected the analysis techniques in this paper to two forms of preliminary performance experiments: one which used typical functional microbenchmarks from previous work [12] and another which used pumped versions of pathological patterns to simulate use at scale. We leave experiments on programs from the wild to future work. The details of these experiments appear in Appendix D in the supplemental material [8] for reasons of space; we summarize them here.

We applied each of SetPlume, kPlume, P4F [12], and Boomerang SPDS to each microbenchmark; kPlume is most similar to SetPlume and so most directly demonstrates the impact of $\Sigma_{\texttt{Set}}$. P4F and Boomerang SPDS are recent state-of-the-art analyses. We used P4F in lieu of 1ADI as they are theoretically similar and the P4F artifact has been used in previously published benchmarks.

SetPlume performs comparably or favorably to the other analyses in the microbenchmarks and pumped examples with one significant exception: a regular expression matching program. This program makes use of continuation passing, effectively hiding self-reference from our annotator and thus preventing selective polyinstantiation from occurring. In the remaining cases, SetPlume performs well; indeed, in the analysis of a brute-force SAT solving program, SetPlume completes the analysis while both P4F and Boomerang SPDS trigger thirty-minute timeouts. While more thorough and realistic benchmarks remain to be conducted, we conclude that set-based context models with selective polyinstantiation show promise as a practical tradeoff between precision and performance.

6 Related Work

6.1 Context Models

The higher-order program analysis community has long known that, in practice, the widely-used kCFA context model [33,38,19,18,12,4,23,39] is imprecise and slow [39, p.25], issues that have been the biggest impediments in the adoption of higher-order analyses. The closest to a systematic study of context models in the higher-order analysis literature is *Allocation Characterizes Polyvariance* [11], but the main intent of that paper is to identify a layer of abstraction between context models (what they call *polyvariance techniques*) and the AAM [38] underlying analysis technique; the paper is not concerned with *evaluating* the context models empirically to determine how tractable they are in practice.

Object-oriented analysis research has explored the choice of context model further. Recent efforts have explored how to avoid polyinstantiation [16,37] and how to vary polyvariance models within a singe analysis run [22,21,17]. These analyses are still brittle in a way, as polyinstantiating a saturated context still loses information. However, they preserve the ordered property of k-cutoff models and so can often correctly handle the `pairmap` example in Section 4.2.

Other context models have been explored for object-oriented analyses, both in theory [2] and in practice [20,25,3]. The experiments in these papers confirm the weaknesses of the k-limited context models and point at better alternatives, including a context model based on the arguments of a method call (the *Cartesian Product Algorithm* [1]), and a context model based on the object whose method is called (termed *object sensitivity* [25]).

mCFA [24] simulates running kCFA in an object-oriented program. mCFA inspired the mADI analysis we used in our evaluation (Section 4).

To the best of our knowledge the set-based context model introduced in this paper is novel in the literature of both higher-order and object-oriented analysis.

6.2 Selective Polyinstantiation

As mentioned in Section 2.4, selective polyinstantiation is most similar to context tunneling [16]. It also bears some resemblance to to *Polymorphic Splitting* [40]. Both selective polyinstantiation and polymorphic splitting involve annotating the analyzed program to direct decisions on polymorphism. In selective polyinstantiation, the annotations occur at *call sites* and indicate functions for which the analysis *should not* be polymorphic. In polymorphic splitting, by contrast, the annotations occur at function *definitions* and indicate where the analysis *should* be polymorphic. The selective polyinstantiation technique prevents building spurious contexts and can be adapted to other underlying analysis techniques. Polymorphic splitting is an analysis technique in and of itself.

6.3 Analysis Techniques

DDPA. DDPA [6] is an ancestor of Plume. The difference between Plume and DDPA is in how they handle context: Plumes' context is stored in the CCFG while DDPA's context is reconstructed during lookup. This has two consequences. First, all Plume lookups include context, making Plume more precise than DDPA (Section 4.1). Second, because Plume does not reconstruct contexts, it is more permissive than DDPA and allows set-based models to be defined.

Demand CFA. Beyond DDPA, the technique closest to Plume is Demand CFA [10]. Plume has the advantage of context sensitivity while Demand CFA does not. However, Plume builds a full CCFG to answer localized lookups; Demand CFA may need to construct only a small part of the CFG for some lookups.

Other Higher-Order Analysis Techniques. Unlike most other higher-order analysis techniques [33,24,38,19,18,12,4,23,39], Plume does not maintain an abstraction of the heap (sometimes also called a *store* elsewhere in the literature); Plume reconstructs only the relevant parts of the heap on demand with a lookup function over the CFG. Some other higher-order analysis techniques feature something called a *pushdown abstraction*, which yields perfect call–return alignment [39,18,19,12] (though not perfect context sensitivity), but Plume only aligns calls and returns up to the precision of its context model.

Boomerang. The Boomerang family of analyses consists of two object-oriented alias analyses for Java: the original Boomerang [35] and the recently-defined "synchronized pushdown system" variant [34] called Boomerang SPDS. These analyses do not model context sensitivity using a model of the form Σ. The Boomerang analysis computes control flow in tandem with IFDS [30] and uses additional iterations to address non-distributive flow problems; Boomerang SPDS instead models control flow using a pushdown system which is intentionally separated from the modeling of field-sensitive data flow. The SPDS technique is not specific to Boomerang; it has been applied to the IDEal taint analysis [36] and has shown promise as a performance improvement there. All evaluations of these theories prior to this paper have been on traditional object-oriented code.

Other Object-Oriented Analysis Techniques. The idea of reconstructing the heap on demand was inspired by first-order demand-driven CFL-reachability analyses [30], and DDPA was the first analysis to bring this technique to a higher-order setting. The primary challenge of that setting is the interdependence between control-flow and data-flow: no CFG is available a priori and so one must be built as the analysis proceeds. Another challenge is lookup of closure-captured variables: previous attempts to bring the technique to a higher-order setting [9] lost precision in those cases, but Plume and DDPA are both able to preserve precision by performing a series of subordinate lookups.

Recent analyses based on linear conjunctive language (LCL) reachability [41] bear some resemblance to Plume in that they reduce lookup to an automaton reachability question. While Plume is related to CFL reachability analyses [30], this recent work reduces to the undecidable problem of LCL reachability and then uses a computable approximation algorithm. Both classes of analysis approach context- and field-sensitivity as an approximation of reachability on a two-stack pushdown automaton; one avenue of future work is to determine if LCL reachability can be applied to Plume-style analyses.

7 Conclusions

This paper introduced set-based context sensitivity. This addresses the weakness of k-limiting models – that polyinstantiation can cause information loss – without compromising field sensitivity or separating it into a distinct problem. To make set-based models practical, we applied selective polyinstantiation, an adaptation of techniques used in k-limiting model research. This technique prevents recursive functions from triggering the worst case performance of the set-based model.

To demonstrate the viability of these techniques, we have formally defined Plume, a demand-driven higher-order program analysis which supports them, and implemented several analysis artifacts. Our experiments show that, for representative functional examples, several set-based, selectively polyinstantiated analyses are superior in precision to their k-cutoff counterparts. We have also demonstrated that analyses using these techniques yield performance comparable with state-of-the-art analyses on typical functional benchmarks.

References

1. Agesen, O.: The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. ECOOP (1995)

2. Besson, F.: CPA beats ∞-CFA. In: Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs (2009)
3. Bravenboer, M., Smaragdakis, Y.: Strictly declarative specification of sophisticated points-to analyses. OOPSLA (2009)
4. Darais, D., Labich, N., Nguyen, P.C., Horn, D.V.: Abstracting definitional interpreters. CoRR (2017)
5. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. TOPLAS (6) (1997)
6. Facchinetti, L., Palmer, Z., Smith, S.: Higher-order demand-driven program analysis. TOPLAS (2019)
7. Facchinetti, L., Palmer, Z., Smith, S.F.: Relative store fragments for singleton abstraction. In: Static Analysis (2017)
8. Fachinetti, L., Palmer, Z., Smith, S.F., Wu, K., Yorihiro, A.: Appendices to a set-based context model for program analysis. `https://www.cs.swarthmore.edu/~zpalmer/publications/supplemental/aplas2020-supplement.pdf` (2020)
9. Fähndrich, M., Rehof, J., Das, M.: Scalable context-sensitive flow analysis using instantiation constraints. PLDI (2000)
10. Germane, K., McCarthy, J., Adams, M.D., Might, M.: Demand control-flow analysis. In: VMCAI. Lecture Notes in Computer Science (2019)
11. Gilray, T., Adams, M.D., Might, M.: Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. ICFP (2016)
12. Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. POPL (2016)
13. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. PLDI (2001)
14. Henglein, F.: Type inference with polymorphic recursion. TOPLAS (2) (1993)
15. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. SIGSOFT (1995)
16. Jeon, M., Jeong, S., Oh, H.: Precise and scalable points-to analysis via data-driven context tunneling. Proc. ACM Program. Lang. (OOPSLA) (2018)
17. Jeong, S., Jeon, M., Cha, S., Oh, H.: Data-driven context-sensitivity for points-to analysis (OOPSLA) (2017)
18. Johnson, J.I., Sergey, I., Earl, C., Might, M., Van Horn, D.: Pushdown flow analysis with abstract garbage collection. JFP (2-3) (2014)
19. Johnson, J.I., Van Horn, D.: Abstracting abstract control. In: DLS (2014)
20. Lhoták, O., Hendren, L.: Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. TOSEM (1) (2008)
21. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Precision-guided context sensitivity for pointer analysis. Proc. ACM Program. Lang. (OOPSLA) (2018)
22. Li, Y., Tan, T., Møller, A., Smaragdakis, Y.: Scalability-first pointer analysis with self-tuning context-sensitivity. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/FSE 2018 (2018)
23. Might, M.: Environment Analysis of Higher-order Languages. Ph.D. thesis (2007), aAI3271560
24. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis. PLDI (2010)
25. Milanova, A., Rountev, A., Ryder, B.G.: Parameterized object sensitivity for points-to analysis for java. TOSEM (1) (2005)
26. Okhotin, A.: Conjunctive grammars. Journal of Automata, Languages and Combinatorics (2001)

27. Rehof, J., Fähndrich, M.: Type-base flow analysis: From polymorphic subtyping to CFL-reachability. POPL (2001)
28. Reps, T.: Shape analysis as a generalized path problem. PEPM (1995)
29. Reps, T.: Undecidability of context-sensitive data-dependence analysis. TOPLAS (1) (2000)
30. Reps, T., Horwitz, S., Sagiv, M.: Precise interprocedural dataflow analysis via graph reachability. POPL (1995)
31. Reps, T.W.: Demand Interprocedural Program Analysis Using Logic Databases (1995)
32. Saha, D., Ramakrishnan, C.R.: Incremental and demand-driven points-to analysis using logic programming. PPDP (2005)
33. Shivers, O.G.: Control-flow Analysis of Higher-order Languages. Ph.D. thesis (1991), uMI Order No. GAX91-26964
34. Späth, J., Ali, K., Bodden, E.: Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. Proc. ACM Program. Lang. (POPL) (2019)
35. Späth, J., Do, L.N.Q., Ali, K., Bodden, E.: Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. ECOOP (2016)
36. Späth, J., Ali, K., Bodden, E.: Ideal: Efficient and precise alias-aware data-flow analysis. PACMPL (OOPSLA) (2017)
37. Tan, T., Li, Y., Xue, J.: Making k-object-sensitive pointer analysis more precise with still k-limiting. In: Static Analysis (2016)
38. Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP (2010)
39. Vardoulakis, D., Shivers, O.: CFA2: A context-free approach to control-flow analysis. ESOP (2010)
40. Wright, A.K., Jagannathan, S.: Polymorphic splitting: An effective polyvariant flow analysis. TOPLAS (1) (1998)
41. Zhang, Q., Su, Z.: Context-sensitive data-dependence analysis via linear conjunctive language reachability. POPL (2017)