Appendices to 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

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

This document contains appendices for the paper "A Set-Based Context Model for Program Analysis" published in the *18th Asian Symposium on Programming Languages and Systems, APLAS 2020, Proceedings.*

A An Overview of Non-Local Variables

The example in Section 2.2 of the main paper does not illustrate the lookup of non-local variables in Plume. This is a delicate process in demand-driven program analyses. In this appendix, we describe how non-local variables are handled using techniques from DDPA, Plume's predecessor.

Consider the program appearing in Figure 1. This program defines the K-combinator and then calls it twice to capture two different values, `g` and `h`, in the closures of two different functions, `kg` and `kh` (respectively). At the end of the program, `kg` is called with an ignored argument. During the execution of this program, `c` should therefore be assigned the value `g` (and not the value `h`).

```
k = fun v -> ( # λv.(λj.v)
  z = fun j -> ( rz = v; );
);
g = fun p -> ( rg = p ); # λp.p
h = fun q -> ( rh = q q ); # λq.q q
kg = k g; # (λj.g)
kh = k h; # (λj.h)
c = kg h; # evaluates to g
```

Fig. 1. K-Combinator: ANF

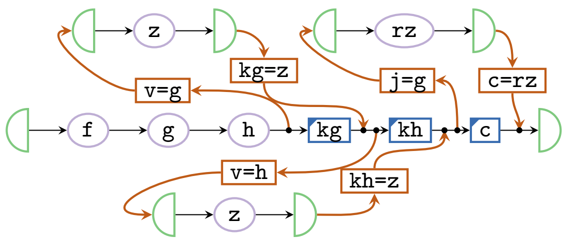

Fig. 2. K-Combinator: Analysis

The CCFG produced by analyzing this program appears in Figure 2; let us proceed to look up `c` from the end of the program. Moving backward, we first see the wiring node `c=rz`. Proceeding backward, we see `rz=v` and we reduce to looking for `v`. We then move past `j=g` (since `j` is not the variable we want) and continue looking for `v`. We are now searching for the value `v` at the top level of the program, where it is out of scope! How did we get here?

Ordinarily, skipping an unrelated variable assignment is the correct action; however, `j=g` was a parameter wiring node. If we are looking for a non-parameter variable when we discover a parameter wiring node, our search variable must have been captured in closure where the function was defined. So we should find the function's definition and resume looking for our non-local variable there. Here, the function associated with this parameter node was `kg`, so we look up `kg` and then resume looking for `v`. In general, this requires a *stack* of variables as the function we are looking up may have been captured in closure as well.

In our example, we suspend our lookup of `v` and begin looking for `kg` at the wiring node. Proceeding backward, we ignore the wiring node `kh=z` (since it doesn't define `kg`) and skip over the `kh` call site. We then discover the `kg=z` wiring node and follow it, looking for `z`. This leads us to `z` which defines a function. We have discovered the location where the function's closure was defined, so we resume looking for the variable `v`. Since `v` is the parameter of the function, we can safely follow `v=g` out to top level, where we discover `g` and find the answer to the original lookup of `c`.

B An Operational Semantics

We begin by giving a small-step operational semantics for the language shown in Figure 1 of the main paper. This semantics differs in a few respects from standard ones to make the formal correctness arguments more direct. The primary difference is that it neither substitutes values for function parameters on function application nor builds closures; instead, at function application it inlines function bodies and freshens bound variables, explicitly mapping the argument to its value in the (flat) environment. The freshening allows static scope conventions to be preserved in spite of not using closures; we will term this a *freshening* operational semantics.

These semantics require some preliminary definitions. All expressions are lists of variable assignment clauses, so we interpret the last binding of a function's body to be its returned value; it is therefore helpful to define a function which extracts the last bound variable of an expression. To facilitate freshening of inlined functions, we also define a function which obtains the set of variables bound by an expression.

Definition 1. *Preliminary definitions:*
- *We define* $\mathrm{RV}([x_1 = b_1, \ldots, x_n = b_n]) = x_n$.
- *We define* $\mathrm{BV}([x_1 = b_1, \ldots, x_n = b_n]) = \{x_1, \ldots, x_n\}$.
- *We write* $e[x_1/x_2]$ *to denote the replacement of all instances of* x_2 *with* x_1 *in* e.

Given these preliminaries, the operational semantics appears in Figure 3. As expressions are represented as lists of assignment clauses, a step of evaluation consists of finding the first assignment to a non-value and reducing it. Alias clauses $x = x'$ are reduced by replacing x' with its assigned value. Function calls $x_1 = x_2\ x_3\ \Theta$ are reduced by inlining the function's body in place of its call and adding a binding of the parameter and returned value to the environment.

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

$$\textsc{Application}\quad \frac{\begin{array}{c}(x_f = \texttt{fun}\ x_p\ \texttt{->}\ \texttt{(}e'\texttt{)}) \in E \qquad e'' = [x_p = x_a] \,||\, e' \\ \textsc{bv}(e'') = \{x_1, \ldots, x_n\} \qquad e''' = e''[x_1'/x_1] \ldots [x_n'/x_n] \qquad x_1', \ldots, x_n' \text{ fresh}\end{array}}{E \,||\, [x_s = x_f\ x_a\ \Theta] \,||\, e \longrightarrow^1 E \,||\, e''' \,||\, [x_s = \textsc{rv}(e''')] \,||\, e}$$

Fig. 3. Operational Semantics

Variable bindings in the function's body are freshened during this inlining to preserve the invariant that the expression is alphatized. Note that the annotations Θ, mentioned in Section 2 of the main paper, have no effect here; they are used only by the analysis.

C Formal Properties

This appendix establishes formal properties of the Plume analysis: soundness, decidability, and precision with respect to the closely-related DDPA analysis.

C.1 Soundness

We first prove the Plume analysis is sound with respect to a (concrete) operational semantics. The freshening operational semantics in Figure 1 of the main paper are forward-running, and include a store in the form of the prefix E. The Plume analysis, meanwhile, monotonically grows a control-flow graph and reconstructs abstract store information on demand. To bridge this significant gap, we construct a midpoint, a graph-based operational semantics which can be formally defined as a variant on the Plume analysis.

ωPlume is a Graph-Based Operational Semantics Plume as defined in the previous section is only two abstractions away from a full, concrete interpreter. First: Plume's context model Σ may be finite; second, call site annotations may cause some contexts to be re-used. Here, we define a new analysis, ωPlume, which relaxes these restrictions and serves as a full and faithful operational semantics.

Definition 2. *The ωPlume analysis is defined as a variant of Plume as follows.*

- *ωPlume will use Σ_ω as it's context model. This is the list model of Definition 2 of the main paper with $k = \omega$, i.e. the list length is unbounded.*
- *Define a function* $\textsc{Erase}(e)$ *which erases all annotations in e (replaces all Θ with $[]$). All expressions analyzed in ωPlume are first erased.*

Since ωPlume is in fact a (Turing Complete) language and not a program analysis, we will use the convention of non-hatted variables when writing ωPlume elements; for example, we may write G but view it as shorthand for $\hat{G}$-in-ωPlume. We formalize the stepping of concrete graphs as follows:

Definition 3. *We define $G \longrightarrow^1 G'$ to be the least relation satisfying the rules in Figure 11 of the main paper. We write $G_0 \longrightarrow^* G_n$ to denote $G_0 \longrightarrow^1 \ldots \longrightarrow^1 G_n$.*

Equivalence of the Operational Semantics To show the soundness of Plume, it is sufficient to prove two smaller goals: that the freshening operational semantics is bisimular to ωPlume, and that (any) Plume analysis simulates ωPlume.

The first subgoal can be proven by establishing a bisimulation relation $\cong$ between expressions under substitution-evaluation and embedded expressions under graph-evaluation; a term steps in one operational semantics if and only if its bisimulated term steps in the other.

Establishing the bisimulation is relatively straightforward; we will highlight the four notable parts of the process. First, we must align each clause in the expression with a node in the ωPlume graph. The only variation in these nodes is in the variables: the freshening system generates fresh variables while the graph system does not. In each case that fresh variables are generated, however, the call stack of the associated graph nodes is changed; therefore it suffices to be deliberate about how these fresh variables map to variable-stack pairs.

The second notable part of the process is that, by inspection, the freshening system is deterministic — it always operates on the first unevaluated clause — while the graph system operates on any Active? node. This reflects the non-determinism of the expansion of the CFG in the analysis. It is possible, however, to prove by induction that, during ωPlume evaluation, (1) at most one site is active at a time that has not yet been expanded and (2) no active site is expanded more than once. ωPlume is deterministic even though inspection is not sufficient to demonstrate it.

The third notable part of establishing the bisimulation is that, while call sites are *replaced* in the freshening semantics, ωPlume leaves old call sites in place. That these call sites do not affect future evaluations can similarly be proven by induction.

Finally, while ωPlume performs lookup on demand, the freshening operational semantics replaces alias assignments $x = x'$ with value assignments $x = v$. We resolve this by allowing the single-stepping relations not to move in lock step as long as they invariably realign. These insights allow the following result to be established:

Lemma 1. *If $e \cong G$ then*
1. *If $e \longrightarrow^* e'$ then $G \longrightarrow^* G'$ such that $e' \cong G'$.*
2. *If $G \longrightarrow^* G'$ then $e \longrightarrow^* e'$ such that $e' \cong G'$.*

Abstract Interpretation The second subgoal of soundness described above is to show that ωPlume is simulated by ($\leqslant$) Plume. This step is easier than the previous as they only differ in how Plume may lose context information, which can be shown by a similar simulation on context models.

In an un-annotated program, each ωPlume list context $[c_1, \ldots, c_n]$ can be shown by induction to be simulated by $\epsilon \oplus \hat{c}_1 \oplus \ldots \oplus \hat{c}_n$. Our simulation must be more general to support selective polyinstantiation annotations, however; an annotated function call may not grow the abstract context (i.e., when the Acontextual Application rule of Figure 11 of the main paper applies). Our map from concrete contexts to abstract contexts can generally determine if a particular ωPlume call site c_i is acontextual by using c_i to identify the call site, using c_{i+1} to identify the called function, and determining if that function-site pair is annotated as acontextual. In summary, we may establish the following.

Theorem 1 (Soundness). *For any ωPlume graph G and any Plume graph $\hat{G}$, if $G \leqslant \hat{G}$ and $G \longrightarrow^1 G'$ then $\hat{G} \widehat{\longrightarrow}^1 \hat{G}'$ such that $G' \leqslant \hat{G}'$.*

C.2 Decidability

Unlike soundness, the decidability of Plume does not hold for all context models (obviously including Σ_ω). Here, we characterize *effectively finite* models for which Plume is decidable:

Definition 4. *Let $\Sigma = \langle \hat{\boldsymbol{C}}, \epsilon, \oplus \rangle$. Using $\hat{\boldsymbol{c}}$ to denote finite sets of abstract clauses $\{\hat{c}, \ldots\}$, let $\hat{C} \curvearrowright_{\hat{\boldsymbol{c}}} \hat{C}'$ iff $\hat{C}' = \hat{C} \oplus \hat{c}$ for $\hat{c} \in \hat{\boldsymbol{c}}$. We write $\Sigma/\hat{\boldsymbol{c}}$ to denote the transitive closure of $\curvearrowright_{\hat{\boldsymbol{c}}}$ on $\{\epsilon\}$. We call Σ* effectively finite *if $\Sigma/\hat{\boldsymbol{c}}$ is finite for all finite $\hat{\boldsymbol{c}}$.*

We now show the decidability of Plume for effectively finite context models. We begin by showing the computability of the lookup function given in Definition 5 of the main paper.

Lemma 2. *For any $\hat{G}$, $\hat{\eta}$, and $\hat{X}$, $\hat{G}, \hat{\eta} \vdash \hat{X} \rightarrowtail \hat{v}$ is computable.*

Proof. By inspection, every premise in the rules of Figure 10 of the main paper is either immediately computable or a subproof of the same relation $\hat{G}, \hat{\eta} \vdash \hat{X} \rightarrowtail \hat{v}$. Throughout a proof in this system, $\hat{G}$ is constant, $\hat{\eta}$ is a position in the graph, and $\hat{X}$ is a list of variables manipulated only from the left side by a constant number of additions and removals in each rule. This problem reduces to reachability in a pushdown automaton: states are either nodes $\hat{\eta}$ or values $\hat{v}$, the stack is $\hat{X}$, and the input grammar consists solely of the empty string ϵ. In this encoding, $\hat{G}, \hat{X} \vdash \hat{\eta} \rightarrowtail \hat{v}$ holds iff $\hat{\eta}$ can reach $\hat{v}$ with initial stack $\hat{X}$ and final stack $[]$.

The pushdown reachability question described above is computable in time polynomial in the size of the graph $\hat{G}$ but can be computed more efficiently using an equivalent, specialized automaton as in DDPA.

As lookup is computable, a single step of graph closure is computable as well:

Lemma 3. *For any finite $\hat{G}$, $\hat{G} \widehat{\longrightarrow}^1 \hat{G}'$ is computable.*

Proof. All premises in Figure 11 of the main paper are either immediately computable, computable by graph traversal ($\widehat{\text{Active}}$?), or computable by Lemma 2.

To show decidability, it now suffices to show that any closure sequence converges in finitely many steps. We proceed by counting argument, showing a finite upper bound on the size of the graph and relying on the monotonicity of closure.

Lemma 4. *For any effectively finite context model Σ and any program e, let $\hat{G}_0 = \widehat{\text{Embed}}(e)$. Let $\hat{\boldsymbol{c}}$ be the set of all clauses in e. Then, for any $\hat{G}_0 \widehat{\longrightarrow}^* \hat{G}_n$, every node $\langle \hat{a}, \hat{C} \rangle$ in $\hat{G}_n$ has (1) either $\hat{a} \in \hat{\boldsymbol{c}}$ or $\hat{a}$ as a wiring node comprised of variables and clauses from $\hat{\boldsymbol{c}}$, and (2) $\hat{C} \in \Sigma/\hat{\boldsymbol{c}}$.*

Proof. By Definition 4 of the main paper, $\hat{G}_0$ contains only clauses appearing in e and only the context ϵ. By inspection of Figure 11 of the main paper, closure adds only those edges produced by $\widehat{\text{WIREFUN}}$. By Definition 6 in the main paper, the nodes of these edges contain clauses either from the graph or comprised of clauses and variables from graph. By induction on the length of the closure sequence, the clauses in the nodes of $\hat{G}_n$ are either in $\hat{\boldsymbol{c}}$ or are wiring nodes comprised of clauses and variables in $\hat{\boldsymbol{c}}$.

By Definition 6 in the main paper, each node in an edge created by $\widehat{\text{WIREFUN}}$ contains either a context already in the graph or the context provided as an argument. By inspection of Figure 11 of the main paper, the context provided to $\widehat{\text{WIREFUN}}$ is either already in the graph or is derived from an existing context and a clause from $\hat{\boldsymbol{c}}$. Because Σ is effectively finite and because $\hat{G}_0$ contains only the context ϵ, we have by induction on the length of the closure sequence that all such contexts are in $\Sigma/\hat{\boldsymbol{c}}$.

With this upper bound on the size of the graph, proof of Plume's decidability is straightforward:

Theorem 2 (Decidability). *For any effective finite context model Σ and any program e, let $\hat{G}_0 = \widehat{\text{EMBED}}(e)$. Then $\hat{G}_0 \mathrel{\widehat{\longrightarrow}}^! \hat{G}_n$ is decidable.*

Proof. By Lemma 3, each step of closure is computable. By inspection of Figure 11 of the main paper, closure is monotonic: $\hat{G}_i \subseteq \hat{G}_{i+1}$. By Lemma 4, all graphs in the sequence are upper-bounded by a finite set of edges. The maximum number of steps in any closure sequence is therefore less than or equal to the number of edges in this finite upper bound.

C.3 *k*Plume $\geq$ *k*DDPA

The proof argument for Theorem 1 in Section 4.1 of the main paper is as follows:

Proof. We proceed by constructing a simulation of the ACFG of kDDPA using the CCFG of kPlume. In particular, the initial ACFG of kDDPA may be simulated by a covering CCFG in which each the ACFG nodes are replicated into the CCFG at every possible context (and edges are inserted correspondingly); the kPlume CCFG is a subset of the covering CCFG. It may then be proven by induction on the definition of lookup that, for any simulated pair of graphs, kDDPA lookup produces at least as many values as kPlume lookup. Finally, we may prove by induction on the length of the closure sequence that this simulation is preserved throughout the analysis process.

D Evaluation of Performance

In this appendix, we conduct a preliminary performance evaluation of the analysis techniques presented in this paper. First: we wish to determine if SetPlume, an analysis using a set-based context model and selective polyinstantiation annotations has performance comparable to state-of-the-art analyses on functional programs. Second and relatedly: we wish to determine if selective polyinstantiation is effective at preventing the worst-case exponential generation of contexts in SetPlume. We do so by conducting two classes of experiments: one over a series of functional microbenchmarks and another over a pair of pumped examples.

D.1 Experiment Design

Both classes of experiments consist of several performance benchmarks. We chose to compare SetPlume, an analysis using a set-based context model, to three other analyses: kPlume, P4F, and Boomerang SPDS.

Comparison with kPlume is a natural step as it most directly illustrates the effect of the set-based context model. We included P4F and Boomerang SPDS to compare with recent state-of-the-art analyses. P4F is a forward-running functional analysis; it is theoretically quite similar to 1ADI and, while the ADI artifact is a proof of concept developed for our precision comparison above, the P4F artifact has been used for previously published benchmarks. Boomerang SPDS is a hybrid forward-backward object-oriented alias analysis; this analysis is dissimilar to SetPlume in form but, like SetPlume, uses a novel approach to context sensitivity in the presence of dynamic control flow; we believe for this reason that it warrants attention.

It should be noted that these analyses were implemented in different ways. SetPlume and kPlume are written in OCaml and analyze a toy experimental language, P4F is written in Scala and analyzes a subset of Scheme, and Boomerang SPDS is written in Java and analyzes Java programs at scale. Due to these differences, we focus only on cases in which the analyses perform dramatically differently and all charts use logarithmic scales.

Our experiments consist of repeatedly executing each of a series of test cases under each analysis. We ran each test case with each analysis ten times on a 3.4GHz Intel Xeon CPU with 32Gb of RAM running Ubuntu 18.04.3 (Linux 4.15); reported values are the mean of all ten runs. No significant variation occurred between runs for any particular analysis-test case pair. Experiments were timed out after thirty minutes. We observed that, for each analysis-test case pair, either every run completed before the timeout or every run timed out; there were no borderline cases.

D.2 Microbenchmarks

As of the time of this writing, no standard benchmark suite exists for higher-order program analyses. Instead, we selected a set of test cases used by P4F and by other evaluations in the higher-order program analysis literature. We have selected the subset of test cases which (1) work in all four analyses' implementations and (2) are not made redundant by the later experiments in Section D.3, which test scalability. The tested microbenchmarks are described below.

- **ack**, **tak**: arithmetic functions with multiple recursion sites
- **blur**, **loop2-1**: test functions creating non-local variables in a loop
- **eta**: an identity function containing a spurious call
- **facehugger**: calls to two independent recursive functions which may appear to call each other if precision is lost
- **kcfa-2**, **kcfa-3**: worst-case programs for kCFA, accessing non-locals in increasingly nested functions
- **primtest**: the Fermat primality testing function
- **regex**: regular expression matching via derivatives

– **rsa** encryption and decryption algorithms from the RSA public-key cryptosystem

The original microbenchmarks in this category were written in Scheme. Similar to our precision experiments, we translated those benchmarks by hand to Java (for Boomerang SPDS); we also translated them to a sugared surface language which we then machine translate into shallow A-normalized form with appropriate annotations for recursion (for *k*Plume and SetPlume). As a consequence, the two Plume analyses are using recursion-annotated source code; the other two analyses are not. As in Section 4 of the main paper, we attempted to translate examples in such a way as to preserve control flow but not to introduce unnecessary program points.

The results of this microbenchmark experiment appear in Figure 4. The SetPlume and Boomerang SPDS analyses are context sensitive but have no tuneable parameters. The P4F artifact may be run in either a monovariant mode or a polyvariant mode; this polyvariant mode, however, is not further configurable and is a form of 1-limited context with perfect call-return alignment. The *k*Plume analysis has a choice of k and, for each benchmark, there are three cases. If some fixed $k \geq 1$ will produce the same precision as SetPlume, we use that k. If no such k exists, we use $k = 1$ and denote this as `inacc` (for "inaccurate"). If such a k may exist but determining it is impractical, we use $k = 1$ and denote this as `imprac`.

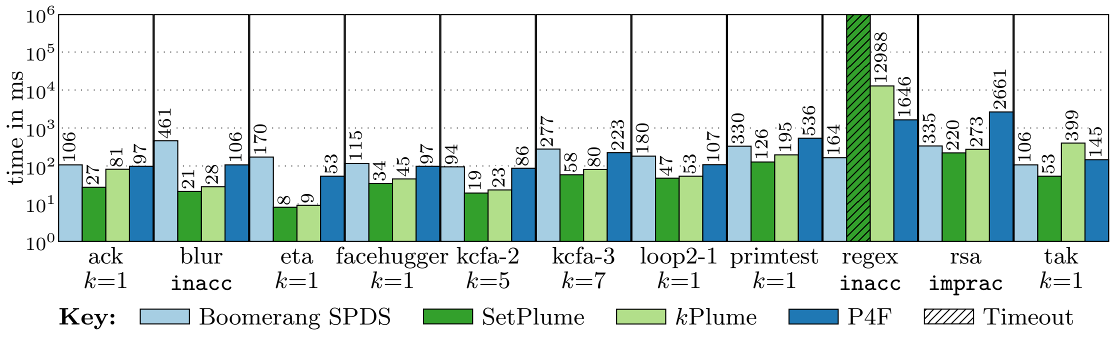

Fig. 4. Results: Microbenchmarks

With two notable exceptions, the four analyses perform comparably on all of the benchmarks. SetPlume appears to holds a slight advantage for the recursive functions `ack` and `tak`; we attribute this to the combination of a set-based context model and selective polyinstantiation. SetPlume and *k*Plume perform well in cases which involve spurious calls or non-locals; this is attributable to the demand-driven nature of the analyses.

The clearest exception to this trend is `regex`, in which SetPlume timed out. We suspect that this is because, as in the original Scheme example, the implementation indirectly generates cycles in control flow via continuations. Because this is not observed by our annotator, it triggers the worst-case exponential expansion of contexts (as if no selective polyinstantiation annotations were used). This may be mitigated in a number of ways, such as by a naive preliminary analysis, which would be able to better inform an annotator before SetPlume operates on the program; we leave such explorations to future work.

From this experiment, we conclude that SetPlume performs comparably to modern state-of-the-art analyses on small functional benchmarks, including those representative of real programming patterns.

D.3 Scalability

Our second category of test cases consist of pumped examples of two forms. The first form of test case, termed `rec`, defines a function which calls itself at several call sites (similar to the description of `ack` and `tak` above). An example of `rec` appears in Figure 5. As we scale up `rec`, the number of recursive call sites increases. As previously, Plume operates on annotated ANF. This form was specifically designed to test the efficacy of these annotations.

```
(define (pathological x)
  (if (eq? x x) (pathological x)
  (if (eq? x x) (pathological x)
  (if (eq? x x) (pathological x)
  (if (eq? x x) (pathological x) 0)))))
(pathological 5)
```

Fig. 5. Pumped Recursion Example: `rec-S4`

The results of experimentation on the `rec` series of test cases appears in Figure 6. We use 1Plume for these experiments for simplicity. *k*Plume's execution time gradually worsens the number of call sites increases; this is likely due to the ambiguity introduced during analysis by the loss of previous context. Boomerang SPDS and P4F are unaffected by this; we suspect that this is due to the primarily forward nature of the analyses, which prevents this loss of context from affecting the lookup of non-local variables so directly. SetPlume fares well because, paradoxically, it is capable of retaining all of the non-recursive context and so faces no ambiguity except in the particular values of the boolean variables.

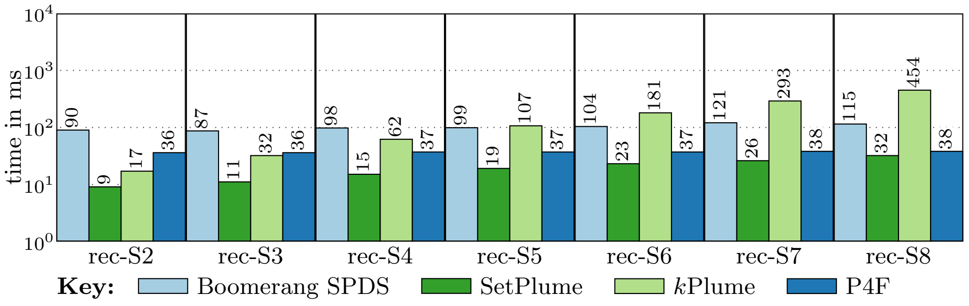

Fig. 6. Pumped `rec` Benchmark Results

The second form of test case, termed `sat`, is a program which will, if analyzed with perfect precision, induce the solution to a SAT problem; it is designed to ensure that an analysis does not attempt to maintain perfect context sensitivity in the face of indirect recursion. An example appears in Figure 7. As we scale up the number of nested calls, the number of variables in the equivalent SAT problem increases. This form was inspired by the `sat-1`, `sat-2`, and `sat-3` test cases from the test suite used by P4F; those cases were elided from the above for redundancy. Again: Plume operates on code with selective polyinstantiation annotations.

```
(define phi (lambda (x1) (lambda (x2) (lambda (x3) (lambda (x4)
    (any boolean expression using those variables))))))
(define try (lambda (f) (or (f #t) (f #f))))
(define sat-solve-4 (lambda (p)
  (try (lambda (n1) (try (lambda (n2) (try (lambda (n3) (try (lambda (n4)
    ((((p n1) n2) n3) n4)))))))))))
(sat-solve-4 phi)
```

Fig. 7. Pumped SAT Example: `sat-P4`

The results of experimentation on the `sat` series of test cases appears in Figure 8. We use $k = 1$ for kPlume in these experiments as the level of k necessary to yield perfect precision is intractable. The results in this figure demonstrate clear trend lines even on a logarithmic scale. P4F's time grows exponentially as the number of nested calls increases; this is unsurprising, as polymorphic CFA analyses (like polymorphic P4F) are known to be exponential in these cases. Boomerang SPDS's time grows less rapidly and careful examination suggests that it may not be exponential. We suspect that this performance is because Boomerang SPDS maintains a distinct pushdown system at each call site to support context sensitivity and these examples are pathologically growing all of them.

SetPlume's and 1Plume's times grow at similar rates. At twenty-two variables, the last test on which Boomerang SPDS completes before timeout, SetPlume is two orders of magnitude faster. We attribute SetPlume's good performance to the selective polyinstantiation annotations discussed Section 2.4 of the main paper. Although the `try` function is not directly recursive, each invocation of `try` which occurs *within* the `try` function itself (such as when `try` is invoked at the call site `f #t`) and so bears annotations that prevent polyinstantiation of its call sites. As a consequence, the number of contexts of `try` will be linear (rather than exponential) in the number of SAT variables in the example.

These experiments used pumped code designed to exploit the worst case of SetPlume. From these experiments, we conclude that selective polyinstantiation is successful at preventing exponential context generation. These results also suggest that $\Sigma_{\texttt{Set}}$ with selective polyinstantiation may be a practical approach to context sensitivity in a program analysis, though more experimentation at scale is required.

D.4 Threats to Validity

Test cases. The test cases used in the experiments above are representative of common functional programming idioms, but they are much smaller than real-world programs and do not include several features commonly found in

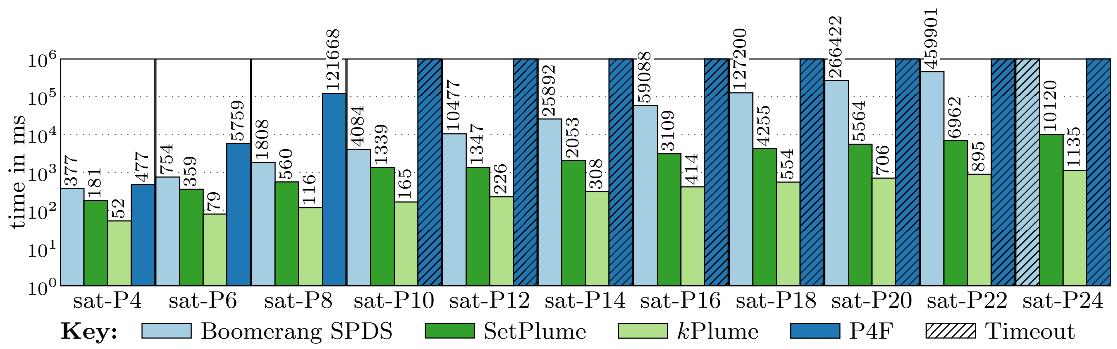

Fig. 8. Pumped `sat` Benchmark Results

practice (such as exceptions or state) due to lack of support in the implemented artifacts. This could be mitigated by the development of a test suite for functional program analyses which, as of the time of this writing, does not exist; the test cases provided here are either taken directly from or are generated based upon examples that have been used in various other program analysis publications.

Variations in expressiveness. Although Section 4 of the main paper details experiments which illustrate the precision of SetPlume in comparison to other analyses, these tests are not the subject of those experiments. This is in part because these test cases do not clearly illustrate the strengths and weaknesses of these analyses; they are instead designed to exemplify functional programming patterns which *don't* require considerable expressiveness but act as tar pits for overly ambitious analyses.

In these experiments, we attempt to mitigate this concern by making choices generally favorable to other analyses at the expense of SetPlume. In the `sat` example, for instance, we used 1Plume because the ideal value of k would induce factorial complexity in the analysis but would also produce a result *more precise* than SetPlume (in that it would actually solve the SAT problem with sufficiently high k). When translating to Java, we used static methods when possible and only instantiated objects to represent functions when such an object would meaningfully be allocated to the heap in a functional language. We do this in an attempt to err on the side of over-approximating SetPlume's cost, but this model is not perfect. In the long term, this threat may be mitigated by fixing a particular client for the three analyses performing the same task with each of them. At the time of this writing, no same client exists for all of the concerned analyses.

Language runtime performance. Three different languages are represented in the artifacts that implement these analyses. Reported times are provided by the analysis programs themselves to exclude runtime startup costs, parsing times, and so on. Mitigation of this concern would require reimplementation of the artifacts in a common language, which is impractical; instead, we simply avoid drawing conclusions without clear timing differences which cannot be explained by runtime variations (such as the trend lines in Figure 8).