Towards Practical Higher-Order Demand-Driven Program Analysis[1]

Leandro Facchinetti[2] *<leandro@jhu.edu>*

2017-10-27

[1] Preliminary research proposal for Graduate Board Oral (GBO) examination, a qualifying requirement for the Ph.D. program at The Johns Hopkins University.

[2] The author's work is supported by a CAPES fellowship. Process number: 13477/13-7.

We propose to research an abstract-interpretation-based static analysis for higher-order programming languages which performs demand-driven (backward) variable lookups, addressing issues of performance and expressiveness towards a foundational theory for practical analysis tools. State-of-the-art program analyses in the higher-order space generally carry variable environments forward, and only recently our research introduced an alternative backward approach, but, while preliminary investigation indicates that our technique has fundamentally different and promising trade-offs, the connection between forward and backward analysis is not well understood; we intend to examine this question, contemplating a hybrid approach which would combine the strengths of both methods. We also propose to address shortcomings of the existing backward technique, in particular regarding recursive programs, by introducing improved models for calling contexts and by adapting parameters in the course of variable lookup. Finally, we plan to probe other grounding theoretical frameworks besides the pushdown-automata reachability we currently use, including logic programming and grammar-based strategies.

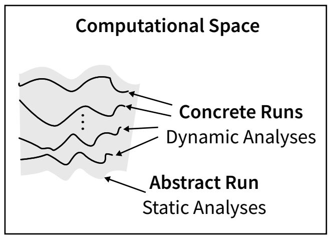
1 Program Analysis

PROGRAM ANALYSIS is the discipline of reasoning about computer programs, with the predominant goal of promoting optimizations and asserting program correctness. Tools including compilers, debuggers, Integrated Development Environments (IDEs) and program verifiers depend on program analysis to ground semantic-preserving program transformations, to assist developers, and to construct proofs. One essential task in program analysis is to determine what program points run in what order (control-flow analysis), which is useful, for example, to detect dead code that the compiler can eliminate, and to find fragments that are executed very often (hotspots) which would benefit from just-in-time (JIT) compilation. Another important task is to determine what data is required in what computation (data-flow analysis), which is useful, for example, to statically check for type errors in dynamically-typed languages, and to find accidental leaks of sensitive information in cryptographic systems.

One strategy for program analysis is to monitor the program execution by instrumenting its runtime (dynamic analysis), retrieving precise information, but limited to a particular run; another strategy, which is the subject of our research, is to perform the analysis before execution (static analysis), recovering information applicable to all possible runs. This task an undecidable in the general case,[3] but we wish for an analysis with termination guarantees, so we allow for precision loss and accept approximate answers.[4]

[3] By Rice's theorem [Rice 1953].

[4]



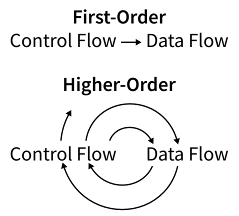
In first-order languages[5] the control flow is evident from the program structure: to determine what function is called at a call site, for example, it suffices to look at its name,[6] so precision losses arise only from data-flow analysis. But, in higher-order languages,[7] which are the subject of our research, control flow and data flow are intertwined, and the analysis of each informs the other: the paths followed during program execution influence data computation, which dictate what functions become available at call sites, and this, in turn, affects future paths; precision losses arise from both control- and data-flow analysis, so this is a more challenging task.[8] Program analysis in both first- and higher-order settings is a research area with decades of history, to the point that some techniques are part of well-established compilers and textbooks,[9] but the former has received more consideration than the latter, and higher-order features are recently growing in popularity among commercial programmers, increasing the demand for higher-order program analyses.

[5] Those in which functions are not values, for example, C.

[6] Ignoring function pointers, but they are rarely used in C programs.

[7] Those in which functions are first-class citizens that can be passed as arguments to function calls, returned from function calls, stored in data structures, and so forth, for example, Scheme, ML and JavaScript.

[8]



[9] Aho et al. 2007.

2 *Abstract Interpretation*

AMONG THE THEORIES for program analysis, we employ one of the most widespread, abstract interpretation,[10] which consists of executing a modified version of the analyzed program (abstraction). Consider the following example:

[10] Cousot and Cousot 1977.

Program	**Values**	**Abstract Program**	**Abstract Values**
a = 3	a = 3	$\widehat{a} = \widehat{\oplus}$	$\widehat{a} = \{\widehat{\oplus}\}$
b = −2	b = −2	$\widehat{b} = \widehat{\ominus}$	$\widehat{b} = \{\widehat{\ominus}\}$
c = a + b	c = 1	$\widehat{c} = \widehat{a} \mathbin{\widehat{+}} \widehat{b}$	$\widehat{c} = \{\widehat{\ominus}, \widehat{0}, \widehat{\oplus}\}$
d = a × b	d = −6	$\widehat{d} = \widehat{a} \mathbin{\widehat{\times}} \widehat{b}$	$\widehat{d} = \{\widehat{\ominus}\}$

$\widehat{\ominus}$	Negative
$\widehat{0}$	Zero
$\widehat{\oplus}$	Positive

The program in the first column is in a general-purpose programming language and ranges over the infinite domains of numbers and numeric operations, and the second column lists the results of a single run.[11] Statically anticipating all possible runs of programs of this nature is undecidable in the general case, particularly due to unbounded recursion, but abstract interpretation lets us discover approximations by executing an abstract version of the program which ranges over finite domains.[12] The choice of abstraction is open and depends on the analysis goals, both in terms of the properties it retrieves and, perhaps more importantly, in what it can approximate. In our example in the third column we are interested in the number signs and not

[11] The only possible run, in this oversimplified example.

[12] Decidability relies on a counting argument.

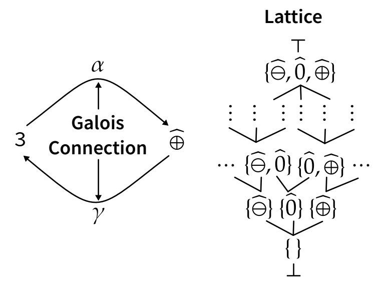
in their magnitude, so we abstract numbers into their signs, numeric operations into their counterparts abiding by the rules of signs,[13] and variables into abstract variables.[14] We then execute this abstract program, and the last column above represents the result: variables are assigned abstract values (*sets* of signs) that accommodate the potential precision loss ensuing from the addition of positive and negative numbers, for which the resulting sign is undetermined, as in $\widehat{\mathtt{c}}$. Finally, we can observe the abstract values to assert properties of all runs of the original program; for example, $\widehat{\mathtt{d}} = \{\widehat{\ominus}\}$ implies d is negative.

[13] For example, $\{\widehat{\oplus}\}\ \widehat{\times}\ \{\widehat{\ominus}\} = \{\widehat{\ominus}\}$.

[14] By convention, hats ($\widehat{\ }$) denote abstractions.

Both concrete and abstract values can be partially ordered from more precise to more general, forming lattices, and abstract interpretation establishes relationships between these latices in the form of abstraction and concretization functions (α and γ, respectively), and connections between them which, at a very high level, guarantee the round trip from concrete to abstract and then back (Galois connections).[15] Executing the abstract program reduces to finding the least fixed-points in lattices, which are guaranteed to exist regardless of unbounded loops or recursive functions in the program.[16] Beyond the simple example above in the abstract domain of signs, we can apply these techniques to collect other properties, including control and data flows of either first- or higher-order programs.[17]

[15]



[16] By Knaster–Tarski theorem [Tarski 1955].

[17] Shivers 1991.

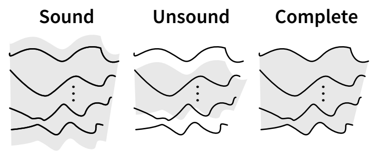
3 Design Space

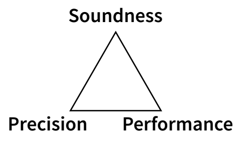
WE CHARACTERIZE program analyses in the following aspects:[18]

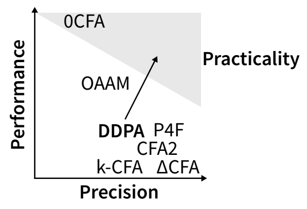
- *Soundness*: There are abstract values that correspond to any concrete value from any possible run; for example, if $\mathtt{d} = -6$ in a program run then $\widehat{\ominus} \in \widehat{\mathtt{d}}$.
- *Precision (Expressiveness) (Completeness)*: There are program runs which result in concrete values corresponding to the predicted abstract values; for example, in the program above, $\widehat{\mathtt{d}} = \{\widehat{\ominus}\}$ is more precise than $\widehat{\mathtt{c}} = \{\widehat{\ominus},\widehat{0},\widehat{\oplus}\}$, because d might assume a negative value (namely, -6), but there is no program run in which c is non-positive (it is 1 every time), so $\widehat{\ominus}$ and $\widehat{0}$ represent precision loss.
- *Performance (Decidability)*: The effort necessary to compute the analysis.

Ideally, we want an analysis to be sound, exactly precise (complete) and performant, but these are conflicting goals and cannot all three be attained simultaneously;[19] so, in practice, these characteristics define a design space in which the choice of abstractions induces analyses with varying degrees of (un)soundness, precision

[18]







k-CFA (0CFA) [Shivers 1991], OAAM [Johnson et al. 2013], P4F [Gilray et al. 2016], CFA2 [Vardoulakis 2012] and ΔCFA [Might 2007].

[19] By Gödel's incompleteness theorem [Gödel 1931].

and performance. Previous research in higher-order program analysis has explored this space, for example, 0CFA[20] is sound and reasonably fast, but imprecise; ΔCFA[21] is sound and precise, but slow; and DoctorJS[22] is unsound, but reasonably precise and fast. However, navigating the design space can be unintuitive; for example, there are cases in which more precision leads to better performance, because computation has to follow less spurious program paths. We work on the practicality of sound, decidable and reasonably precise analyses, introducing techniques inspired by first-order analysis that present different trade-offs and might unlock a new level of performance.

[20] Shivers 1991.

[21] Might 2007.

[22] Vardoulakis 2012.

4 *Demand-Driven Program Analysis (DDPA)*

THE DEFINING property of our analysis, *Demand-Driven Program Analysis* (DDPA), is the direction of abstract value computation. Generally, program analyses for higher-order languages—including all the ones in the previous section—carry variable environments forward, in close correspondence to the interpreters they abstract; in our example, the variable environment collects $\widehat{\mathsf{a}}$, $\widehat{\mathsf{b}}$, and so forth as the abstract interpretation progresses. But, in DDPA, instead of maintaining a variable environment, we look variables up on demand by following the control flow backward; for example, when we reach program point $\widehat{\mathsf{d}}$, we register the computation depends on $\widehat{\mathsf{a}}$ and $\widehat{\mathsf{b}}$, so we traverse the program upward looking for their definitions. In this simplistic example, $\widehat{\mathsf{a}}$ and $\widehat{\mathsf{b}}$ are immediately evident, but, in general, they might require further variable lookups, in a process that resembles programmers investigating stack traces and reasoning about how program execution reached a certain state.

The first distinct advantage of a backward approach is its potential to disregard unwanted information; for example, when querying $\widehat{\mathsf{d}}$, we can skip $\widehat{\mathsf{c}}$ and never compute it, because it does not affect the lookup subject. In general, backward analyses visit variable *uses* (references) before *definitions*, which provides them with more information when determining what details to retain and what precision to lose. More importantly, the opposing directions might induce fundamentally different trade-offs, similar to those in bottom-up versus top-down parsing[23] and forward versus backward chaining in inference engines.[24] While there exist first-order demand-driven program analyses,[25] as well as some limited attempts to adapt them to a higher-order setting,[26] DDPA is, to the best of our knowledge, the first fully-featured system of its kind.

[23] Aho et al. 2007.

[24] Hayes-Roth et al. 1983.

[25] Reps et al. 1995, Horwitz et al. 1995, Reps 1995, 1994, Saha and Ramakrishnan 2005, Duesterwald et al. 1997, Heintze and Tardieu 2001.

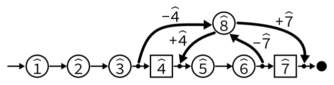
[26] The analyses in [Dubé 2003, Spoon 2005], for example, focus on adapting parameters according to the demand and in weakening answers of subgoals, rendering them less expressive.

5 *DDPA Overview*

CONSIDER THE example on the margin: a Scheme program that defines an identity function and calls it twice with different arguments. The program points (function definitions, applications, literal values and variable references) are labeled with numbered superscripts. Below the program is its abstraction, following the rules of signs from the previous sections, and, lastly, there is a complete Control-Flow Graph (CFG), in which nodes are program points, square nodes are function applications and arrows are the control flow; thin arrows, in particular, are immediately apparent from the program structure and form blocks for the top level (1–7) and function body (8). Initially, DDPA is supplied a partial CFG encompassing only these blocks of nodes and thin arrows, and its duty is to compute the thick arrows, which wire the applied functions around call sites, and are labeled after them for reasons that will become evident later (− stands for entrance and + for exit).

$$\texttt{(define (id x) x}^{8}\texttt{)}^{1}$$
$$\texttt{(id}^{2}\texttt{ 3}^{3}\texttt{)}^{4}\texttt{ ; }\Rrightarrow\texttt{ 3}$$
$$\texttt{(id}^{5}\texttt{ -6}^{6}\texttt{)}^{7}\texttt{ ; }\Rrightarrow\texttt{ -6}$$

$$\texttt{(}\widehat{\texttt{define}}\texttt{ (}\widehat{\texttt{id}}\texttt{ }\hat{\texttt{x}}\texttt{) }\hat{\texttt{x}}^{\hat{8}}\texttt{)}^{\hat{1}}$$
$$\texttt{(}\widehat{\texttt{id}}^{\hat{2}}\texttt{ }\hat{\oplus}^{\hat{3}}\texttt{)}^{\hat{4}}$$
$$\texttt{(}\widehat{\texttt{id}}^{\hat{5}}\texttt{ }\hat{\ominus}^{\hat{6}}\texttt{)}^{\hat{7}}$$



CFG construction is an iterative and incremental process that requires variable lookups in partial CFGs, due to the interlacing of control and data flows in higher-order programs; for example, to wire the appropriate function body around call site $\hat{4}$, DDPA queries which values could be bound to $\widehat{\texttt{id}}$ at that point.[27] Moreover, following the abstract interpretation framework, these call sites must be handled in an order that preserves evaluation sequence: $\hat{4}$ and $\hat{7}$. The procedure to find call sites that are ready to be resolved is to look for a path from the program start to them that does not include function calls—multiple call sites may be ready for resolution at the same time, because they represent different concrete runs, and in that case an arbitrary selection is adequate, and the final CFG would be the same.[28]

[27] In this oversimplified example there is only one candidate function for $\widehat{\texttt{id}}$, and it can be looked up by name, but this does not hold in general higher-order programs.

[28] A property called **convergence**.

An intuition for how lookups operate[29] is to begin at the program point involved in the query and traverse the CFG in reverse, following the arrows in the illustration backward; for example, consider looking for $\widehat{\texttt{id}}$ at call site $\hat{4}$: DDPA visits $\hat{4}\rightarrow\hat{3}\rightarrow\hat{2}\rightarrow\hat{1}$, and then the answer is immediate. Had the analysis found an alias—for example, $\texttt{(}\widehat{\texttt{define}}\texttt{ }\widehat{\texttt{id}}\texttt{ }\hat{\texttt{a}}\texttt{)}$—it would have changed the query subject to $\hat{\texttt{a}}$ and proceeded; and, had it found a bifurcation with multiple arrows to follow, it would have non-deterministically explored all of them and joined the results. Nodes that do not contribute to the solution are skipped, as was the case of $\hat{3}$ and $\hat{2}$ in our example; this rule applies even to nodes representing function calls, letting us skip entire irrelevant function wirings.[30]

[29] The next section covers the actual strategy.

[30] Similar to the **summarization** techniques in CFA2 [Vardoulakis 2012].

6 *Obstacles in DDPA*

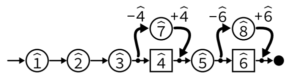
ISSUE 1: CALL–RETURN ALIGNMENT. Consider the example from the previous section, and a lookup for the return value of $\hat{4}$ starting on the program point immediately following it, $\hat{5}$. There are two conceivable paths, which result in different abstract values: $\hat{5}\to\hat{8}\to\hat{3}$ outputs $\widehat{\oplus}$, and $\hat{5}\to\hat{8}\to\hat{6}$ outputs $\widehat{\ominus}$. An unsophisticated analysis would join these results to reply the query with $\hat{4} = \{\widehat{\oplus}, \widehat{\ominus}\}$, but this represents precision loss because the only concrete value at that program point is positive (namely, 3). The inaccuracy source is the absurd second path, which misaligns functions call and return: it enters $\widehat{\mathtt{id}}$ through an edge linked to $\hat{4}$ and exits it through an edge linked to $\hat{7}$ (in reverse). To discard this path and preserve precision, DDPA employs a stack, called the *context stack*,[31] which it observes in the course of CFG traversal, for example, by pushing $\hat{4}$ when going past the $+\hat{4}$ edge and popping when going past the $-\hat{4}$ edge—the $-\hat{7}$ edge mismatches the stack top element, so the entire path involving it does not contribute to the answer.

[31] The context stack in DDPA corresponds to the call stack in an interpreter, but it operates in the opposite direction.

ISSUE 2: SUBORDINATE LOOKUPS. Consider the example on the margin: the program defines and then calls the constant function k, which captures its argument in a closure. We query DDPA for the return value of the program, and it starts a CFG traversal in reverse, visiting $\hat{8}$ and changing the lookup subject to $\hat{x}$. If the analysis followed the rules described thus far, then it would explore $\hat{5}\to\hat{4}\to\hat{3}\to\hat{2}\to\hat{1}$, reach the start of the program without having gone past the $-\hat{4}$ edge where $\hat{x}$ is bound, and fail to find an answer, for not having accurately expressed the notion of lexical scoping that is requisite to capturing a variable in a closure. The solution to this hurdle in DDPA is to introduce another stack, called the *continuation stack*, that manages subordinate lookups. In our example, before going past the function entrance at $-\hat{6}$, the analysis detects that the query subject is not the function argument, so it must be a non-local variable that was in scope at the function definition; DDPA then pushes the current lookup to the continuation stack and begins to look for the function called at $\hat{6}$: $\hat{4}$. This causes the analysis to explore $\hat{5}\to\hat{7}$, where the subordinate lookup ends, and from there it can resume the original lookup, going past $-\hat{4}$ and finding the answer at $\hat{3}$.[32,33] Besides non-local variable lookups, the analysis of several operations calls for subordinate lookups and uses the continuation stack, for example, record projection, functions with multiple arguments (including, binary operations), and so forth.

$$\texttt{(define (k x) (}\lambda\texttt{ (y) x}^{8}\texttt{)}^{7}\texttt{)}^{1}$$
$$\texttt{((k}^{2}\texttt{ 3}^{3}\texttt{)}^{4}\texttt{ -6}^{5}\texttt{)}^{6}\texttt{ ; }\Rightarrow\texttt{ 3}$$

$$\texttt{(}\widehat{\texttt{define}}\texttt{ (}\hat{\texttt{k}}\texttt{ }\hat{\texttt{x}}\texttt{) (}\hat{\lambda}\texttt{ (}\hat{\texttt{y}}\texttt{) }\hat{\texttt{x}}^{\hat{8}}\texttt{)}^{\hat{7}}\texttt{)}^{\hat{1}}$$
$$\texttt{((}\hat{\texttt{k}}^{\hat{2}}\texttt{ }\widehat{\oplus}^{\hat{3}}\texttt{)}^{\hat{4}}\texttt{ }\widehat{\ominus}^{\hat{5}}\texttt{)}^{\hat{6}}$$

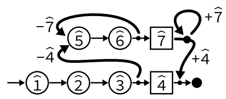


[32] The naïve lookup path mentioned above ($\hat{5}\to\hat{4}\to\hat{3}\to\hat{2}\to\hat{1}$) also visited $\hat{3}$, but failed to discover the answer for having the wrong lookup subject when reaching it.

[33] This strategy is comparable to—and inspired by—**access links** used in compilers for higher-order languages [Aho et al. 2007].

$$\texttt{(define (u x) (x}^{5}\texttt{ x}^{6}\texttt{)}^{7}\texttt{)}^{1}$$
$$\texttt{(u}^{2}\texttt{ u}^{3}\texttt{)}^{4}\texttt{ ; }\Rightarrow\texttt{ }\bot$$

$$\texttt{(}\widehat{\texttt{define}}\texttt{ (}\hat{\texttt{u}}\texttt{ }\hat{\texttt{x}}\texttt{) (}\hat{\texttt{x}}^{\hat{5}}\texttt{ }\hat{\texttt{x}}^{\hat{6}}\texttt{)}^{\hat{7}}\texttt{)}^{\hat{1}}$$
$$\texttt{(}\hat{\texttt{u}}^{\hat{2}}\texttt{ }\hat{\texttt{u}}^{\hat{3}}\texttt{)}^{\hat{4}}$$



ISSUE 3: DECIDABILITY. Consider the example on the margin: the

program is a diverging recursion. Recursion (diverging or otherwise) induces cycles in the CFG, for example, $+\hat{7}$ and $\hat{5}\to\hat{6}\to-\hat{7}\to\hat{5}$, and a graph-traversal-based analysis like the one described thus far would follow these cycles indefinitely without terminating. DDPA encodes the CFG traversals in terms of a pushdown automaton (PDA) with an empty input alphabet, also known as a pushdown system (PDS), and lookups in terms of PDS reachability queries, which are known to be decidable.[34] There is one hurdle, though: a PDS has one stack, but we described two stacks in this section, the context stack and the continuation stack, and they can both grow arbitrarily, which is enough to simulate a Turing machine.[35] Our strategy to suit the analysis encoding to the PDS model is to finitely approximate one of the stacks and embed it in multiple copies of the automaton nodes.[36] The choice of which stack to compromise has distinct effects on precision: approximating the context stack impacts call–return alignment, and approximating the continuation stack impacts operations depending on subordinate lookups, including non-local lookups, record projections, and so forth. In DDPA, we choose the former,[37] and the context stack abstraction is simple but unrefined: it is truncated to a certain configurable maximum size k.[38] When the context stack is exhausted, precision is lost in the form of call–return misalignment, and, unfortunately, recursive programs always trigger this issue in our current system—improving on this is one of the thrusts in this research proposal.[39]

[34] Bouajjani et al. 1997.

[35] The stacks are the left and right sides of the tape.

[36] Similar to the strategy adopted by the standard algorithm for converting a non-deterministic finite automaton into a deterministic one [Aho et al. 2007].

[37] CFA2 [Vardoulakis 2012] does the opposite.

[38] Analogous to k-CFA's strategy for handling contours.

[39] See § 9.

7 *Thrust: Practicality*

PRACTICAL TOOLS based on higher-order program analyses are in short supply; among the few examples there are MLton[40] and Stalin,[41,42] highly-optimizing compilers for ML and Scheme, respectively; and DoctorJS,[43] a type-recovery analysis for JavaScript. But they are limited: MLton and Stalin use relatively imprecise variants of 0CFA, and DoctorJS is unsound. We attribute this scarcity to the poor performance of more sophisticated analyses, and, while there are theoretical firm boundaries on how fast they can operate,[44] we believe that better trade-offs are attainable in practice. Our research does not concentrate on building analysis clients, but on developing theoretical analysis frameworks to support them,[45] and, to this end, we emphasize the examination of algorithmic complexity and, more importantly, running times.

Realistically, we do not anticipate reaching the performance necessary to support real-time interactivity in Integrated Development Environments (IDEs), for example, but we expect our theory to be capable of processing reasonably-sized real-world code bases in ac-

[40] Weeks 2006.

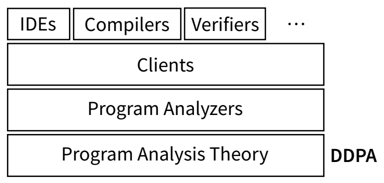
[41] Siskind 1999.

[42] An unfortunate name choice.

[43] Vardoulakis 2012.

[44] k-CFA is complete for exponential time and even 0CFA is $O(n^3)$ [Van Horn 2013].

[45]



ceptable time (hundreds of thousands of lines of code within hours or a few days at most), which should suffice for optimizers and verification tools that run once before project releases, for example. As an instance of a prototypical concrete client, in our tests we use a lightweight static type checker for dynamically-typed languages which asserts, for example, that non-function values are not used as operators at call sites, that record projections only involve existing fields, and so forth.

The next sections propose specific research directions to achieve this overarching goal of practicality.

8 *Thrust: Bidirectional Analysis*

THE PROMINENT BENEFIT of demand-driven analyses is their potential to only analyze program slices relevant to a query while setting aside a large portion of the code base, which is advantageous for clients that are concerned with extra specific targets, for example, a security verification tool trying to discover sensitive information disclosure in an individual function from a cryptographic library. However, beyond the mere ability to disregard parts of the program, preliminary results[46] in our exploration indicate that our backward-lookup approach gives rise to fundamentally different trade-offs when compared to forward techniques in terms of running time and expressiveness: in some test cases DDPA outperforms other state-of-the-art analyses, and in other cases it is defeated; more importantly, precision losses are found at different sites for the competing analyses. The causes for these discrepancies are still unclear, but our initial investigations suggest that they might be indicative of theoretically interesting distinct underlying properties, comparable to those in bottom-up versus top-down parsing[47] and forward versus backward chaining in inference engines,[48] and we propose to further examine this connection in the domain of higher-order program analyses.

[46] See § 11

[47] Aho et al. 2007.

[48] Hayes-Roth et al. 1983.

In particular, we intend to explore the relationship between the value environments maintained by forward analyses during abstract interpretation and an automaton we developed for DDPA's implementation to cache PDS-reachability computations and improve running time, the *Pushdown Reachability* automaton (PDR). The abstract interpretation in our analysis builds the PDR incrementally in a manner that resembles the value environments from forward analyses, but they do not completely align; for example, PDRs may be partially lazily computed and appear to promote more subordinate lookups reuse than value environments. To direct our examination, we wish to reformulate forward analyses, for example, k-CFA, in terms closer to PDRs and *vice versa*; a methodology that has been successful in literature, for example, in developing *m*CFA, which arose from comparing *k*-CFA in object-oriented and functional programming languages.[49] Our vision is to eventually construct a bidirectional analysis that combines the strengths of forward and backward analyses, a fruitful approach in other areas, for example, inference engines.

[49] Might et al. 2010.

9 *Thrust: Expressiveness*

THE MOST SIGNIFICANT deficiency in DDPA is in its treatment of recursive functions: the analysis ingenuously traverses the CFG cycle induced by recursion[50] until it exhausts the context stack, which causes loss of precision and is expensive to compute; to exacerbate the issue, an exhausted context stack percolates and pollutes the analysis of associated stack frames. Revisiting the example in ISSUE 3 from § 6, DDPA's context stack grows $[\widehat{4}] \to [\widehat{4}, \widehat{7}] \to [\widehat{4}, \widehat{7}, \widehat{7}] \to \ldots$ until the k positions are exhausted, when precision is lost, the element $\widehat{4}$ is discarded, and the context stack collapses into $[\widehat{7}, \widehat{7}, \ldots]$, where $\widehat{7}$ appears k times. We observed this negative effect in practice in our preliminary investigations,[51] and we attribute most severe cases of precision loss and substandard running times to it.

[50] Traversing a cycle is equivalent to unrolling a loop [Aho et al. 2007].

[51] See § 11.

We propose to address this situation by considering alternative context abstractions, including models based on regular expressions and graphs. A regular-expression abstraction is similar to the Φ_{CR} function for handling contours in TinyBang[52] and, to a lesser extent, to the stack discipline in ΔCFA:[53] instead of accumulating $\widehat{7}$s and then truncating the stack, it grows $[\widehat{4}] \to [\widehat{4}, \widehat{7}] \to [\widehat{4}, \widehat{7}*]$[54], at which point it has collapsed the cycle, but avoided losing extra precision on $\widehat{4}$. If there are multiple edges in the cycle, then they are all merged together; for example, if the cycle is formed by $\widehat{8}$ and $\widehat{9}$, then the stack element becomes $\{\widehat{8}|\widehat{9}\}*$. Unlike our current context stack, this abstraction is exact for non-recursive programs, because it only approximates calls within a cycle; for example, calls between $\widehat{8}$ and $\widehat{9}$ might misalign when the regular expression suffix is $\{\widehat{8}|\widehat{9}\}*$.

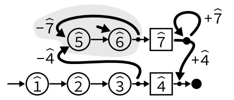
[52] Palmer 2015.

[53] Might 2007.

[54] The $*$ is the Kleene star, which denotes "zero or more of the previous form [Aho et al. 2007]."

As a further refinement, to avoid collapsing recursive cycles, we propose a graph-based context abstraction, in which contexts are CFG slices paired with a pointer: the slices includes the cycles, and the pointer monitors where in the cycle the abstract interpretation is at the moment.[55] For these CFG slices to be sound, there must be no missing wirings, but the purpose of DDPA's abstract interpretation is precisely to discover these wirings, so, to solve this conundrum, we entertain the possibility of running a faster and less precise analysis—for example, 0DDPA, which has no context abstraction whatsoever—to inform the more sophisticated pass. The source of approximation

[55]



Example of CFG slice paired with a pointer to $\widehat{6}$ serving as a context abstraction.

in this model is the potential misalignment between traversals of call and return cycles, because there is no communication mechanism between these traversals besides the pointer in the graph, which cannot transport the required information; in our example, the call and return cycles are, respectively, $\widehat{5}\to\widehat{6}\to-\widehat{7}\to\widehat{5}$ and $+\widehat{7}$, and misalignment would occur if traversing one three times and the other only twice, for example.

Our very early explorations of the regular-expression model suggest that the added precision are accompanied by enormous performance costs,[56] so, as a compromise, we propose exploring an adaptive DDPA variant, in which parameters are adjusted in the course of lookup; for example, the context stack size limit k can increase until recursion is detected, improving precision but not affecting the algorithm decidability, and, for recursive cycles, k is decreased to lessen the harm of the ingenuous traversal described above; additionally, we may transition to the more expensive context abstraction only in select cases.

[56] We anticipate the graph model gives rise to less duplication, and, consequently, will perform better.

```
(define b (random-boolean))
(define m (if b 3 -6))
(define n (if b 4 -7))
(× m n) ; ⇒ 12 or 42
```

$$
\begin{array}{l}
(\widehat{\texttt{define}}\ \hat{\texttt{b}}\ (\widehat{\texttt{random-boolean}}))\\
(\widehat{\texttt{define}}\ \hat{\texttt{m}}\ (\widehat{\texttt{if}}\ \hat{\texttt{b}}\ \widehat{\oplus}\ \widehat{\ominus}))\\
(\widehat{\texttt{define}}\ \hat{\texttt{n}}\ (\widehat{\texttt{if}}\ \hat{\texttt{b}}\ \widehat{\oplus}\ \widehat{\ominus}))\\
(\hat{\times}\ \hat{\texttt{m}}\ \hat{\texttt{n}})
\end{array}
$$

ANOTHER ASPECT of expressiveness is aligning values from different paths. Consider the example on the margin: it starts by choosing a random boolean `b`, then it independently computes two variables `m` and `n` based on `b`, and, finally, it multiplies these two values; the program output is positive regardless of the choice of `b`, because `m` and `n` condition upon the same circumstance so there does not exist a program run in which they disagree in sign. DDPA's lookups for `m` and `n` are unconnected and the results are not correlated, so the analysis does not correctly capture the sign property and answers that the abstract program results in $\{\widehat{\ominus},\widehat{\oplus}\}$. We developed a variant called *Demand-Driven Relative Store Fragment Analysis* (DRSF) to mend this issue, and its defining characteristic is that it does not reply to lookups with only the abstract values, but also with the *relative traces* from the program point associated with the query to the relevant allocation point, as well as partial abstract stores with related bindings. In our example, the lookup for $\hat{\texttt{m}}$ in DRSF returns $\{[\hat{\texttt{m}} \mapsto \widehat{\oplus}, \hat{\texttt{b}} \mapsto \widehat{\texttt{true}}], [\hat{\texttt{m}} \mapsto \widehat{\ominus}, \hat{\texttt{b}} \mapsto \widehat{\texttt{false}}]\}$,[57] and the lookup for $\hat{\texttt{n}}$ would return in a similar fashion, allowing the analysis to discard correlations between stores with disagreeing abstract values for $\hat{\texttt{b}}$.

[57] Abstract traces in the query results not shown here for simplicity—they are empty, in this case, because there are no function calls involved.

DRSF's capabilities go beyond the lookup correlations, though, because the abstract traces in the query results (not shown in the example above) are sufficient to substantiate environment analysis,[58] a form of program analysis that reasons about allocations of abstract bindings upon closure creation and is useful for various compiler optimizations. While DRSF itself has been implemented and tested, building a lightweight client to explore environment analysis on

[58] Might 2007.

top of it is still on the plans. Moreover, we take in consideration structuring the rest of the work in this research proposal on top of DRSF as opposed to DDPA, but this still demands more investigation due to the computational costs associated with the added precision; specifically, DDPA is polynomial and DRSF is exponential, and our benchmarks reveal that the distinction manifests in practice, but weakened (potentially polynomial) variations of DRSF may prove to be better foundations for our research for other reasons—for example, the theory for handling stateful computations is more straightforward in this setting.

THE LAST EXPRESSIVENESS facet we propose to examine is extending the core theory to support more features of practical programming languages: tail-call optimization, sophisticated control flow operators (for example, exceptions and first-class continuations), more basic values, data structures and operations on them (for example, lists), and so forth. This lays the foundations for our goal of testing DDPA with real-world code bases, and requires crucial theoretical advancements; for example, exceptions introduce numerous wirings in the CFG, and managing them efficiently is an open problem. Improvements in this area facilitate the creation of compilers from other programming languages into the core language DDPA analyzes, and will allow us to extend the proof-of-concept Scheme compiler we use in our benchmark suite.[59]

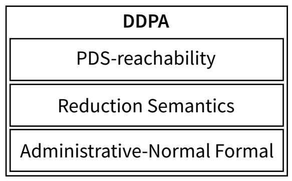
[59] See § 11

10 *Thrust: Other Foundations*

THE THEORETICAL DEVICE that makes variable lookup decidable in DDPA is the PDS-reachability question to which it reduces, in a manner comparable to—and inspired by—PDCFA[60] and similar analyses,[61] but this is far from being the sole option: there are program analyses based on logic programming,[62] grammars, constraints, set theory and so forth.[63] Some of these formulations might offer advantages; for example, encoding lookups in terms of logic programs may permit DDPA to benefit from decades of research and implementation efforts that went into inference engines, including some bidirectional capabilities we wish to explore.[64] We anticipate this might improve our understanding of the trade-offs and the core properties in our analysis, as well as potentially simplify its implementation.

[60] Earl et al. 2010.

[61]



[62] Reps 1994, Saha and Ramakrishnan 2005.

[63] Midtgaard 2012.

[64] See § 8.

Another dimension worthy of examination are the different frameworks for specifying semantics: as it stands, the definition of DDPA's concrete language is based on reduction relations [Felleisen and Hieb 1992], and we propose to explore alternatives including natural (big-step) semantics [Kahn 1987], denotational semantics [Scott 1970], abstract machines [Landin 1966], definitional interpreters [Reynolds 1972], and so forth. Related works in program analysis demonstrate that different computational models have a tendency to induce appealing abstract interpreters; for example, Big CFA2 is based on natural semantics [Vardoulakis 2012], the original *k*-CFA presentation used denotational semantics [Shivers 1991], AAM relies on abstract machines [Van Horn and Might 2011], and there exists recent work on program analysis using definitional interpreters [Darais et al. 2017]; our objective is to comprehend the impact the demand-driven approach has in analyses based on these distinct models. On a related venue, we consider investigating alternative intermediate representations for the analyzed language; for example, DDPA is currently defined using Administrative Normal Form (ANF) [Flanagan et al. 1993], but more sophisticated control operators including exceptions and first-class continuations might be better handled using Continuation-Passing Style (CPS) [Sussman and Steele 1975], which converts them into a single construction, the tail function call.

11 *Preliminary Results*

WE PUBLISHED two peer-reviewed papers on DDPA thus far:[65,66] the first presents the core DDPA theory, and the second introduces DRSF, a path-sensitive variant in which lookups result not only in abstract values, but also in abstract relative traces to their definitions and in partial abstract stores.[67] Besides the contributions to the theory and implementation, the author also led the efforts on benchmarking, which involved finding test cases in the literature, porting them to our analysis' language, and running experiments comparing DDPA and DRSF to other state-of-the-art analyses, for example, P4F[68] and OAAM.[69,70] To accomplish this, we built a compiler for a Scheme subset, including a collection of standard-library functions; moving forward we will examine more realistic test cases beyond micro-benchmarks, and extend our compiler to support more language features and a larger assortment of built-in functions.[71] Our preliminary results indicate that DDPA is comparable to its competitors in performance, as the graph on the margin suggests, but the source of some discrepancies is unclear, and we plan to further investigate this.

[65] Palmer and Smith 2016, Facchinetti et al. 2017.

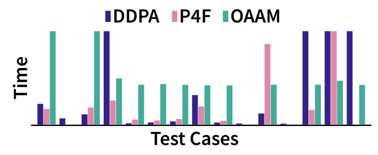
[66] The author of this research proposal was co-author on the first paper artifact and co-author on the second paper and accompanying artifact.

[67] See § 9.

[68] Gilray et al. 2016.

[69] Johnson et al. 2013.

[70]



An impressionistic view of how DDPA performance compares to P4F and OAAM. The test cases range from synthetic micro-benchmarks that stress particular aspects of program analyses, for example, closure creation, to **real-world** programs of 100–200 lines of code.

Regarding expressiveness and DDPA's shortcoming in recursive programs, the author's first qualifying project[72] introduced a language variation and corresponding analysis in which recursive bindings are first-class citizens and do not require encodings in the form of self-passing or the Y combinator; this simplifies compilers from other languages (for example, Scheme and Python) into DDPA core

[71] A collaborator is doing research in this area, targeting a Python subset in a project called CoPylot.

[72] Facchinetti 2016.

language, and reduces program size. The initial implementation of this idea had a negative impact on performance, but there are several improvements to consider in future work; for example, we currently treat all bindings as potentially recursive, but we could identify those that are non-recursive and skip the more sophisticated and expensive analysis on them.

References

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, Boston, 2nd edition, 2007.

Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *Proceedings of the 8th International Conference on Concurrency Theory*, CONCUR '97, pages 135–150. Springer-Verlag, 1997.

Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

David Darais, Nicholas Labich, Phuc C. Nguyen, and David Van Horn. Abstracting Definitional Interpreters. *ICFP 2017*, jul 2017.

Danny Dubé. *Demand-Driven Type Analysis for Dynamically-Typed Functional Languages*. PhD thesis, Universite de Montreal, 2003.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Trans. Program. Lang. Syst.*, 19(6):992–1030, nov 1997.

Christopher Earl, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis of Higher-Order Programs. *Scheme Workshop*, 2010.

Leandro Facchinetti. Practical Demand-Driven Program Analysis with Recursion. Qualifying project report, 2016.

Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. *Relative Store Fragments for Singleton Abstraction*, pages 106–127. 2017.

Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theor. Comput. Sci.*, 103(2):235–271, sep 1992.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247. ACM, 1993.

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown Control-Flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 691–704. ACM, 2016.

K. Gödel. *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. 1931.

Frederick Hayes-Roth, Donald A. Waterman, and Douglas B. Lenat. *Building Expert Systems*. Addison-Wesley Longman Publishing Co., Inc., 1983.

Nevin Heintze and Olivier Tardieu. Demand-Driven Pointer Analysis. In *PLDI*, pages 24–34. ACM, 2001.

Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand Interprocedural Dataflow Analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 104–115. ACM, 1995.

J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing Abstract Abstract Machines. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, pages 443–454. ACM, 2013.

G. Kahn. Natural Semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39. Springer-Verlag, 1987.

P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, mar 1966.

Jan Midtgaard. Control-Flow Analysis of Functional Programs. *ACM Comput. Surv.*, 44(3):10:1–10:33, jun 2012.

Matthew Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and Exploiting the k-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 305–315. ACM, 2010.

Zachary Palmer. *Building a Typed Scripting Language*. PhD thesis, Johns Hopkins University, Baltimore, MD, USA, 2015.

Zachary Palmer and Scott F. Smith. Higher-Order Demand-Driven Program Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56, pages 19:1–19:25, 2016.

Thomas Reps. Demand Interprocedural Program Analysis Using Logic Databases. In *Application of Logic Databases*, 1994.

Thomas Reps. Shape Analysis as a Generalized Path Problem. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 1–11. ACM, 1995.

Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*, pages 49–61. ACM, 1995.

John C. Reynolds. Definitional Interpreters for Higher-order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2*, ACM '72, pages 717–740. ACM, 1972.

H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2): 358–366, 1953.

Diptikalyan Saha and C. R. Ramakrishnan. Incremental and Demand-Driven Points-to Analysis Using Logic Programming. In *PPDP*, pages 117–128. ACM, 2005.

D.S. Scott. *Outline of a Mathematical Theory of Computation*. Programming Research Group, Oxford University Computing Laboratory. Oxford University Computing Laboratory, Programming Research Group, 1970.

Olin Grigsby Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1991.

Jeffrey Mark Siskind. Flow-Directed Lightweight Closure Conversion. Technical report, 1999.

Steven Alexander Spoon. *Demand-Driven Type Inference with Subgoal Pruning*. PhD thesis, dec 2005.

Gerald Jay Sussman and Guy L. Steele, Jr. Scheme: An Interpreter for Extended Lambda Calculus. Technical report, dec 1975.

Alfred Tarski. A Lattice-Theoretical Fixpoint Theorem and its Applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

David Van Horn. *The Complexity of Flow Analysis in Higher-Order Languages*. PhD thesis, nov 2013.

David Van Horn and Matthew Might. Abstracting Abstract Machines: A Systematic Approach to Higher-Order Program Analysis. *Commun. ACM*, 54(9):101–109, sep 2011.

Dimitrios Vardoulakis. *CFA2: Pushdown Flow Analysis for Higher-Order Languages*. PhD thesis, Northeastern University, 2012.

Stephen Weeks. Whole-Program Compilation in MLton. In *Proceedings of the 2006 workshop on ML*. ACM, 2006.