Higher-Order Flow Analysis with Call/Return Matching

Dimitrios Vardoulakis
Northeastern University
dimvar@ccs.neu.edu

Abstract

Function call and return is the fundamental control-flow mechanism in higher-order languages. However, existing higher-order flow analyses do not handle call and return well: they remember only a bounded number of pending calls because they approximate programs with control-flow graphs. Call/return mismatch results in imprecision and increases the analysis time.

We present CFA2, a higher-order flow analysis that can match an unbounded number of calls and returns. CFA2 is broadly applicable: it handles features such as first-class functions, objects, dynamic typing, tail calls and first-class continuations. We implemented CFA2 for Scheme and JavaScript and found it to be more precise than kCFA-style analyses. In addition, CFA2 scales to large codebases, such as the Firefox add-ons.

1. Introduction to higher-order flow analysis

Flow analysis is a tool for discovering properties of the runtime behavior of a program without actually running it [18, preface].

Flow analysis is woven into the evolution of programming languages; it is a key technology for creating faster, more reliable, easy-to-use languages. We can classify the applications of flow analysis in four categories:

Optimization This is perhaps the most common use of flow analysis in compilers. The list of optimizations that require knowledge of a program's control flow is never-ending. Examples include the classic data-flow optimizations (constant folding, common-subexpression elimination, dead-code elimination, *etc*), loop optimizations (induction-variable elimination, loop-invariant code motion, loop fusion, *etc*) and stack-based optimizations (escape analysis and stack allocation of closures, records, objects, *etc*).

Debugging Flow analysis can help find errors that are not caught by the type system, such as null dereferences and memory leaks. In dynamic languages, it can catch many type-like errors statically, such as calling a function with the wrong number or kind of arguments.

Verification People have also used flow analysis to show the absence of certain errors: the program will never throw an exception, or will never deadlock, or will not leak trusted data to untrusted parts of the system, *etc*.

Program understanding Last, flow analysis can be incorporated in an IDE to ease development. It allows programmers to ask queries such as which functions can be called at a particular call site. It can also show debugging information and code-completion information.

In this article, we examine flow analysis in the context of higher-order languages, *i.e.*, languages that allow computation to be packaged up as a value. We focus on functional languages, but the results are transferable to object-oriented languages.

1.1 The challenge of higher-order flow analysis

In order to perform flow analysis of a program p, we must first turn p into a form that makes control flow explicit, usually a graph [11, 22]. Each node in the graph represents a program point plus some amount of abstracted environment and control context. Every path in the graph is considered a valid execution. Thus, executions are strings in a *regular* language.

In a higher-order language, functions are values, so we run into call sites like `(x e)`. To create a control-flow graph, we must know what functions can flow to `x` at runtime, for which we must do flow analysis. But to do flow analysis, we need a control-flow graph(!) This chicken-and-egg problem prevented higher-order flow analysis from evolving at the same pace as first-order flow analysis.

In his dissertation, Shivers proposed kCFA [23], a family of analyses that solves the aforementioned problem because it creates a control-flow graph and analyzes it *at the same time*.

Contexts in kCFA are represented by call strings [22]. For $k = 0$, we get a *monovariant* analysis: invocations of a function from different calling contexts are not distinguished. For $k > 0$, the analysis is *polyvariant*: it creates contexts of length k that represent the last k function activations and it distinguishes invocations of a function that happen in different contexts.

2. Limitations of kCFA and related analyses

Shivers's work made higher-order flow analysis feasible. However, kCFA has limitations. In a functional language, the dominant control-flow mechanism is function call and return. The execution traces that match calls with returns are strings in a *context-free* language. Approximating this control flow with regular-language techniques permits execution paths that break call/return nesting. Call/return mismatch affects the analysis in several ways.

2.1 Imprecise data-flow information

In kCFA, a function may be called from one program point but return to a different one, which results in spurious flow of data. We show an example for $k = 0$. (Similar examples can be written for any k.)

```
(define app (λ (f e) (f e)))

(define id (λ (x) x))

(let* ((n1 (app id 1))
       (n2 (app id 2)))
  (+ n1 n2))
```

This Scheme program defines the apply and identity functions, binds `n1` to 1 and `n2` to 2 and adds them. 0CFA creates the graph

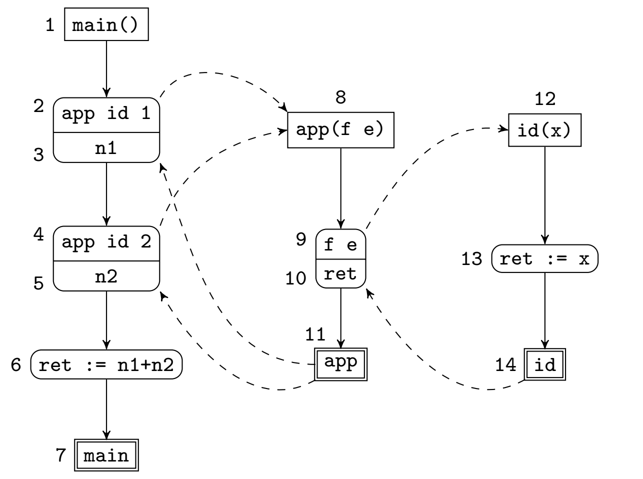

Figure 1. Simple control-flow graph

shown in Fig. 1. All paths in this graph are considered valid executions by 0CFA. Thus, we can bind `n1` to 2 by calling `app` from 4 and returning to 3. Also, we can bind `n2` to 1 by calling `app` from 2 and returning to 5. At 6, 0CFA thinks that each variable can be bound to either 1 or 2.

The `app` example is an atypical program, but it illustrates an issue found in all real-world programs. It is common to have a few functions that are called from tens or hundreds of program points. If an analysis does not do a good job of distinguishing different calling contexts, widely used functions pollute the analysis results.

2.2 Inability to calculate stack change

Besides the spurious flow of data, call/return mismatch results in spurious control flow. As a consequence, we cannot accurately calculate stack changes between program points. The `app` example is a straight-line program, but 0CFA thinks it has a loop (there is a path from 4 to 3). Recursive programs make stack-change calculation even harder.

Some optimizations, however, require accurate information about stack change. For instance:

- Most compound data are heap allocated in the general case. Examples include: closure environments, `cons` pairs, records, objects, *etc*. If we can show statically that such a piece of data is only passed downward, we can allocate it on the stack and reduce garbage-collection overhead.
- Often, continuations captured by `call/cc` do not escape upward. In this case, we do not need to copy the stack into the heap.

Such optimizations are better performed with pushdown analyses.

2.3 The environment problem and fake rebinding

In higher-order languages, many bindings of the same variable can be simultaneously live. Determining at compile time whether two references to some variable will be bound in the same runtime environment is referred to as the *environment problem* [14, 23]. Consider the following program:

```
(let ((f (λ(x thunk)
           (if (number? x)
               (thunk)
               (λ₁() x)))))
  (f 0 (f "foo" "bar")))
```

In the inner call to `f`, `x` is bound to `"foo"` and λ_1 is returned. We call `f` again; this time, `x` is 0, so we jump through `(thunk)` to λ_1, and reference `x`, which, despite the just-completed test, is not a number: it is the string `"foo"`.

Thus, during static analysis, it is generally unsafe to assume that a reference has some property just because an earlier reference had that property. This has an unfortunate consequence: when two references are bound in the same environment, kCFA does not detect it, and it allows paths in which the references are bound to different abstract values. We call this phenomenon *fake rebinding*.

```
(define (compose-same f x) (f (f x)))
```

In `compose-same`, both references to `f` are always bound in the same environment (the top stack frame). However, if multiple closures flow to `f`, kCFA may call one closure at the inner call site and a different closure at the outer call site.

2.4 Polyvariant versions of kCFA are intractable

kCFA for $k > 0$ is expensive, both in theory [26] and in practice [24]. Counterintuitively, imprecision in higher-order flow analyses can increase their running time: imprecision induces spurious paths, along which the analysis must flow data, thus creating further spurious paths, and so on, in a vicious cycle which creates extra work whose only function is to degrade precision [32].

With 20 years of hindsight, we can now say that imprecision in kCFA happens because call strings are not a good abstraction of calling context. With a good abstraction, *a function usually behaves differently in different contexts*, so redundancy is minimized. With call strings, each program point potentially appears in a large number of contexts and the analysis results for many of them are the same [13]. Researchers have proposed BDDs as a way to tame redundancy [31, 33]. We believe it is better to use a different abstraction to avoid introducing redundancy in the first place.

2.5 The root cause: call/return mismatch

A closer look to the shortcomings of kCFA reveals that most of them are caused by call/return mismatch. Specifically,

- Call/return mismatch causes spurious data flows which decrease precision.
- Call/return mismatch causes spurious control flows which hinder effective reasoning about stack change.
- Imprecision induced by call/return mismatch creates extra work that slows down the analysis.

Only fake rebinding is not directly caused by call/return mismatch; it happens because kCFA does not solve the environment problem. In the next section, we show that CFA2 uses a single mechanism, a stack, for call/return matching and to avoid most fake rebinding.

After kCFA, there has been a lot of subsequent work devoted to finding good abstractions of context (*e.g.*, [3, 15, 16, 20, 25, 32]). These analyses provided insights on the different kinds of contexts and improved the state of the art of higher-order flow analysis. However, being finite-state, they all experience the limitations of call/return mismatch.

3. Pushdown models

Pushdown models of programs allow unbounded call/return matching. The key insight is that we can abstract a program to a pushdown automaton (or equivalent) instead of a finite-state machine. By pushing return points on the stack, we eliminate call/return mismatch. Such models have long been used for first-order languages. Examples include the functional approach of Sharir and

Pnueli [22], the tabulation algorithm of Reps *et al.* [21], Recursive State Machines [4] and Pushdown Systems [5, 10].

Pushdown models for higher-order languages have remained relatively unexplored. A notable exception is Mossin's type-based analysis, which uses polymorphic subtyping for polyvariance [17]. It works by reducing flow analysis to a type-inference problem. The analysis uses an annotated type system based on let-polymorphism. Thus, it is precise for flows to different references of let-bound variables, but not for λ-bound variables. For instance, it cannot find that `n1` and `n2` are bound to constants in the `app` example.

Another approach is Kobayashi's work on higher-order model checking [12]. He models a program by a higher-order recursion scheme $\mathcal{G}$, expresses the property of interest in the modal μ-calculus and checks if the infinite tree generated by $\mathcal{G}$ satisfies the property. This technique can do flow analysis, since flow analysis can be encoded as a model-checking problem, and it gives precise results. However, it only applies to pure, typed languages and it is computationally expensive.

4. The CFA2 analysis

CFA2 [27, 28] is pushdown higher-order flow analysis, based on the functional approach [22] and the tabulation algorithm [21]. In contrast to previous analyses, CFA2 is completely general. It can analyze typed and untyped languages, with mutation and expressive control constructs such as tail calls and first-class continuations.

We formulate CFA2 as an abstract interpretation [8, 9]. We want to analyze programs written in some Turing-complete language L. We can describe the semantics of L with a transition relation $\rightarrow$ between program states. Since L is Turing-complete, we cannot use $\rightarrow$ directly for static analysis. Thus, we define an alternate semantics, called the abstract semantics, which is a computable approximation of $\rightarrow$.

Every abstract program state has a stack of unbounded height and a heap, which is an environment of bounded size, similar to the environments in finite-state analyses. CFA2 pushes return points on the stack to pass the results of function calls to the right calling context. However, the stack is more than a control structure for return point information; it is also an environment structure—it contains bindings. CFA2 leverages the stack to do precise variable lookups.

CFA2 has a novel approach to variable binding: *two references to the same variable need not be looked up in the same binding environment*. We split references into two categories: stack and heap references. If a reference appears at the same nesting level as its binder, then it is a stack reference, otherwise it is a heap reference. For example, the program `(λ(x)(λ(y)(x (x y))))` has a stack reference to `y` and two heap references to `x`.

Intuitively, only heap references may escape. When we call a function, we push a frame for its arguments, so we know that stack references are always bound in the top frame. When control reaches a heap reference, its frame is either deeper in the stack, or it has been popped. We look up stack references in the top frame, and heap references in the heap.

Let's return to the `app` example to see how CFA2 addresses the limitations of finite-state analyses. For the first call to `app`, we use a stack frame that binds `e` to 1. This frame is popped after the call. At the second call, we create a frame that binds `e` to 2. Thus, the use of frames as disposable binding environments (they can be pushed and popped) has the effect of analyzing different calls to a function in different environments. CFA2 finds that both variables are bound to constants.

CFA2 also sees a more precise picture of the stack than 0CFA; it finds that there is a single execution path:

1 2 8 9 12 13 14 10 11 3 4 8 9 12 13 14 10 11 5 6 7

Last, we can filter certain bindings off the stack to avoid fake rebinding. Assume that two lambdas can flow to the formal parameter `f` of `compose-same`, so `f` is bound to the abstract value $\{\lambda_1, \lambda_2\}$. Since both references to `f` are stack references, they are always bound to the same value. Therefore, if we call λ_1 at the inner call site, we remove λ_2 from the set to avoid calling it at the outer call site.

4.1 Analyzing recursive programs

If the analyzed program is recursive, the size of the stack is not bounded, so there are infinitely many abstract states. To ensure termination, pushdown-reachability algorithms use a dynamic-programming technique called *summarization* [6, 22]. For every function f, a summary is a pair (arg, ret), which expresses that if f is called with an abstract value arg it returns an abstract value ret. We use summaries at call sites to simulate the effect of the call.

In a language with tail calls, traditional summarization is not enough. Consider what happens if we allow tail calls in the `app` example. Then, `id` would return directly to `main`, not to `app`. For this reason, we generalize summarization to allow *cross-procedure* summaries. We create summaries from the entry of `app` to the exit of `id` and use them at call sites 2 and 4.

4.2 Handling first-class control

Pushdown models require that calls and returns in the analyzed program nest properly. However, many control constructs, some of them in mainstream programming languages, break call/return nesting. Examples include generators (Python, JavaScript), coroutines (Lua, Simula67) and first-class continuations (Scala, Scheme, SML/NJ).

CFA2 is the first pushdown analysis to handle such constructs. We have created an intermediate representation of programs called Restricted Continuation-Passing Style (RCPS) [29]. RCPS can express first-class control constructs, but also permits effective reasoning about the stack. We have generalized CFA2 to RCPS programs and proved it correct [30].

5. Evaluation

5.1 Scheme implementation

We wrote a prototype implementation of CFA2, 0CFA and 1CFA for the Twobit Scheme compiler [7] and compared their precision by doing constant propagation and folding on a small set of benchmarks [27]. We also measured the size of the abstract state space: a small state space indicates that a well-engineered implementation of CFA2 is likely to be fast.

The results appear in Fig. 2. CFA2 is the most precise; it finds many more constants than 0CFA and 1CFA. Also, its state space is comparable to that of 0CFA and much smaller than the state space of 1CFA.

5.2 DoctorJS

In collaboration with Mozilla, we wrote a more mature implementation of CFA2 for JavaScript. This implementation is the core of DoctorJS, Mozilla's suite of static-analysis tools for JavaScript. DoctorJS is open source and available from

`http://github.com/mozilla/doctorjs`

DoctorJS uses Narcissus for lexing and parsing [1]. It performs various rewrites on the AST produced by Narcissus and then does abstract interpretation. The implementation of CFA2 in DoctorJS differs from traditional summarization: it is based on big-step operational semantics. This way it can use the results of expressions

	0CFA		1CFA		CFA2	
	states	con.	states	con.	states	con.
`len`	81	0	126	0	55	2
`rev-iter`	121	0	198	0	82	4
`len-Y`	199	0	356	0	131	2
`tree-count`	293	2	2856	6	183	10
`ins-sort`	509	0	1597	0	600	4
`DFS`	1337	8	6890	8	1719	16
`flatten`	1520	0	6865	0	478	5
`sets`	3915	0	54414	0	4251	4
`church-num`	19130	0	19411	0	22671	0

Figure 2. Scheme benchmark results

	LOC	time (ms)	mem (MB)	viol.
Commentblocker	762	135	5.5	23
Flashblock	936	183	8.7	2
Imtranslator	1335	293	3.4	6
Flagfox	2056	556	5.1	5
Greasemonkey	5764	1103	23.9	10
Flashgot	9730	5323	24.5	12
Video download helper	12916	2997	57.6	6
Web developer	20493	5993	33.6	86
FoxyTunes	35200	8401	109.8	49
ReminderFox	35980	194050	72.8	1
Firebug	80480	30696	271.8	140

Figure 3. Analysis results for some Firefox add-ons

in place, do less caching and find return points quickly. It will be described in detail in the author's forthcoming dissertation.

One application of DoctorJS is type inference of JavaScript programs. After flow analysis is done, we post-process the results, turn the abstract values into types and write the types in a file in ctags format. People have incorporated our analysis in tools to ease program development: the tagbar code browser for Vim [2] and Murray and Bigham's tool for Automatic Function Definition [19].

In the past months, we used DoctorJS to analyze Firefox add-ons, as part of the Electrolysis project at Mozilla.[1] Electrolysis was an experimental architecture for Firefox that would limit the ways in which an add-on can interact with a webpage. For each add-on, we ran DoctorJS to see how often the add-on uses patterns that would not be allowed by Electrolysis. A large number of violations means that the new architecture would break a lot of existing code.

We analyzed 6997 add-ons, a total of 17.3 MLOC, and found 40138 violations. We ran the experiments on an Intel E8400 (Core 2 Duo 3GHz) with 2GB RAM. The median size of an add-on is 444 LOC, the median analysis time is 91 ms and the median memory consumption is 3.4 MB. The analysis timeouts for less than 10% of the add-ons, with a timeout of 5 minutes. Currently, the major cause of slowdown is how we implement the abstract heap, which is orthogonal to call/return matching.

We inspected the results manually for some popular add-ons and present them in Fig. 3. For large add-ons such as Firebug, the analysis scales well and uses little memory. Also, it is low on false positives. For the add-ons we inspected, 303 of the 340 violations found by DoctorJS are true violations.

6. Conclusions

In this article, we gave an overview of CFA2, a higher-order flow analysis with unbounded call/return matching. We have formulated CFA2 as an abstract interpretation and formally established its correctness. The expressive features of higher-order languages required substantial generalizations to summary-based analyses. Our experiments demonstrate that CFA2 is precise and it scales to large, real-world applications.

[1] `http://wiki.mozilla.org/Electrolysis`

References

[1] The Narcissus meta-circular JavaScript interpreter. `http://github.com/mozilla/narcissus`.

[2] Tagbar: the Vim class outline viewer. `http://majutsushi.github.com/tagbar`.

[3] Ole Agesen. The Cartesian Product Algorithm: simple and precise type inference of parametric polymorphism. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 2–26, 1995.

[4] Rajeev Alur, Michael Benedikt, Kousha Etessami, Patrice Godefroid, Thomas W. Reps, and Mihalis Yannakakis. Analysis of Recursive State Machines. *Transactions on Programming Languages and Systems*, 27(4):786–818, 2005.

[5] Ahmed Bouajjani, Javier Esparza, and Oded Maler. Reachability analysis of pushdown automata: Application to model-checking. In *International Conference on Concurrency Theory*, pages 135–150, 1997.

[6] Swarat Chaudhuri. Subcubic algorithms for Recursive State Machines. In *Principles of Programming Languages*, pages 159–169, 2008.

[7] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for Scheme. In *LISP and Functional Programming*, pages 128–139, 1994.

[8] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252, 1977.

[9] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages (POPL)*, pages 269–282, 1979.

[10] Alain Finkel, Bernard Willems, and Pierre Wolper. A direct symbolic approach to model checking pushdown systems. *Electronic Notes in Theoretical Computer Science*, 9, 1997.

[11] Gary A. Kildall. A unified approach to global program optimization. In *Principles of Programming Languages*, pages 194–206, 1973.

[12] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Principles of Programming Languages*, pages 416–428, 2009.

[13] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Transactions on Software Engineering and Methodology*, 18(1):1–53, 2008.

[14] Matthew Might. *Environment analysis of higher-order languages*. PhD thesis, Georgia Institute of Technology, 2007.

[15] Matthew Might, Yannis Smaragdakis, and David Van Horn. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Programming Language Design and Implementation (PLDI)*, pages 305–315, 2010.

[16] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, 2005.

[17] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[18] Steven Muchnick and Neil Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[19] Kyle I. Murray and Jeffrey P. Bigham. Beyond autocomplete: Automatic function definition. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 259–260, 2011.

[20] Hanne Riis Nielson and Flemming Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Principles of Programming Languages (POPL)*, pages 332–345, 1997.

[21] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Principles of Programming Languages (POPL)*, pages 49–61, 1995.

[22] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Muchnick and Jones [18].

[23] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, 1991.

[24] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In *SIGPLAN Notices, special issue: 20 years of PLDI (1979 - 1999): a selection*. ACM, 2004.

[25] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In *Principles of Programming Languages (POPL)*, pages 17–30, 2011.

[26] David Van Horn and Harry Mairson. Deciding k-CFA is complete for EXPTIME. In *International Conference on Functional Programming (ICFP)*, pages 275–282, 2008.

[27] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. In *European Symposium on Programming*, pages 570–589, 2010.

[28] Dimitrios Vardoulakis and Olin Shivers. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science*, 7(2:3):1–39, May 2011.

[29] Dimitrios Vardoulakis and Olin Shivers. Ordering multiple continuations on the stack. In *Workshop on Partial Evaluation and Program Manipulation*, pages 13–22, 2011.

[30] Dimitrios Vardoulakis and Olin Shivers. Pushdown flow analysis of first-class control. In *International Conference on Functional Programming*, pages 69–80, 2011.

[31] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Programming Language Design and Implementation (PLDI)*, pages 131–144, 2004.

[32] Andrew Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *Transactions on programming languages and systems (TOPLAS)*, 20(1):166–207, 1998.

[33] Jianwen Zhu. Symbolic pointer analysis. In *International Conference on Computer-aided Design (ICCAD)*, pages 150–157, 2002.