CFA2: a Context-Free Approach to Control-Flow Analysis

Dimitrios Vardoulakis and Olin Shivers

Northeastern University
{dimvar,shivers}@ccs.neu.edu

Abstract. In a functional language, the dominant control-flow mechanism is function call and return. Most higher-order flow analyses, including k-CFA, 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 introduces precision-degrading spurious control-flow paths and increases the analysis time.
We describe CFA2, the first flow analysis with precise call/return matching in the presence of higher-order functions and tail calls. We formulate CFA2 as an abstract interpretation of programs in continuation-passing style and describe a sound and complete summarization algorithm for our abstract semantics. A preliminary evaluation shows that CFA2 gives more accurate data-flow information than 0CFA and 1CFA.

1 Introduction

Higher-order functional programs can be analyzed using analyses such as the k-CFA family [1]. These algorithms approximate the valid control-flow paths through the program as the set of all paths through a finite graph of abstract machine states, where each state represents a program point plus some amount of abstracted environment and control context.

In fact, this is not a particularly tight approximation. The set of paths through a finite graph is a regular language. However, the execution traces produced by recursive function calls are strings in a *context-free language*. Approximating this control flow with regular-language techniques permits execution paths that do not properly match calls with returns. This is particularly harmful when analyzing higher-order languages, since flowing functional values down these spurious paths can give rise to further "phantom" control-flow structure, along which functional values can then flow, and so forth, in a destructive spiral that not only degrades precision but drives up the cost of the analysis.

To appear in the 19th European Symposium on Programming (ESOP), March 2010, Cyprus.

Pushdown models of programs can match an unbounded number of calls and returns, tightening up the set of possible executions to strings in a context-free language. Such models have long been used for first-order languages. The functional approach of Sharir and Pnueli [2] computes transfer-functions for whole procedures by composing transfer-functions of their basic blocks. Then, at a call-node these functions are used to compute the data-flow value of the corresponding return-node directly. This "summary-based" technique has seen widespread

use [3, 4]. Other pushdown models include Recursive State Machines [5] and Pushdown Systems [6].

In this paper, we propose CFA2, a pushdown model of higher-order programs. Our contributions can be summarized as follows:

- CFA2 is a flow analysis with precise call/return matching that can be used in the compilation of both typed and untyped languages. No existing analysis for functional languages enjoys all of these properties. k-CFA and its variants do not provide call/return matching (section 3.1). Rehof and Fähndrich's analysis [7] supports limited call/return matching and applies to typed languages only (section 7).
- CFA2 uses a stack and a heap for variable binding. Variable references are looked up in one or the other, depending on where they appear in the source code. As it turns out, most references in typical programs are read from the stack, which results in significant precision gains. Also, CFA2 can filter certain bindings off the stack to sharpen precision (section 4). k-CFA with abstract garbage collection [8] cannot infer that it is safe to remove these bindings. Last, the stack makes CFA2 resilient to syntax changes like η-expansion. It is well known that k-CFA is sensitive to such changes [9, 10].
- We formulate CFA2 as an abstract interpretation of programs in continuation-passing style (CPS). The abstract semantics uses a stack of unbounded height. Hence, the abstract state space is infinite, unlike k-CFA. To analyze the state space, we extend the tabulation algorithm of Reps et al. [3]. The resulting algorithm is a search-based variant of summarization that can handle higher-order functions and tail recursion. Currently, CFA2 does not handle first-class-control operators such as `call/cc` (section 5).
- We have implemented 0CFA, 1CFA and CFA2 in the Twobit Scheme compiler [11]. Our experimental results show that CFA2 is more precise than 0CFA and 1CFA. Also, CFA2 usually visits a smaller state space (section 6).

2 Preliminary definitions and notational conventions

We begin with a description of our CPS language and its small-step semantics. For brevity, we develop the theory of CFA2 in the untyped λ-calculus. Primitive data, explicit recursion and side-effects can be easily added using standard techniques [1, ch. 3] [12, ch. 9]. Compilers that use CPS [13, 14] usually partition the terms in a program in two disjoint sets, the user and the continuation set, and treat user terms differently from continuation terms.

We adopt this partitioning for our language (Fig. 1). Variables, lambdas and calls are given labels from *ULab* or *CLab*. Labels are pairwise distinct. User lambdas take a user argument and the current continuation; continuation lambdas take only a user argument. We apply an additional syntactic constraint: the only continuation variable that can appear free in the body of a user lambda $(\lambda_l\,(u\,k)\ \mathit{call})$ is k. This simple constraint forbids first-class control [15]. Intuitively, we get such a program by CPS-converting a direct-style program without `call/cc`. We refer to this variant of CPS as "Restricted CPS" (RCPS).

$$
\begin{array}{rcl}
v \in \mathit{Var} & = & \mathit{UVar} + \mathit{CVar} \\
u \in \mathit{UVar} & = & \text{a set of identifiers} \\
k \in \mathit{CVar} & = & \text{a set of identifiers} \\
\psi \in \mathit{Lab} & = & \mathit{ULab} + \mathit{CLab} \\
l \in \mathit{ULab} & = & \text{a set of labels} \\
\gamma \in \mathit{CLab} & = & \text{a set of labels} \\
\mathit{lam} \in \mathit{Lam} & = & \mathit{ULam} + \mathit{CLam} \\
\mathit{ulam} \in \mathit{ULam} & ::= & (\lambda_l\,(u\,k)\;\mathit{call})
\end{array}
\qquad
\begin{array}{rcl}
\mathit{clam} \in \mathit{CLam} & ::= & (\lambda_\gamma\,(u)\;\mathit{call}) \\
\mathit{call} \in \mathit{Call} & = & \mathit{UCall} + \mathit{CCall} \\
\mathit{ucall} \in \mathit{UCall} & ::= & (f\,e\,q)^l \\
\mathit{ccall} \in \mathit{CCall} & ::= & (q\,e)^\gamma \\
g \in \mathit{Exp} & = & \mathit{UExp} + \mathit{CExp} \\
f, e \in \mathit{UExp} & = & \mathit{ULam} + \mathit{UVar} \\
q \in \mathit{CExp} & = & \mathit{CLam} + \mathit{CVar} \\
\mathit{pr} \in \mathit{Program} & ::= & \mathit{ULam}
\end{array}
$$

Fig. 1. Partitioned CPS

We assume that all variables in a program have distinct names. Concrete syntax enclosed in $[\![\cdot]\!]$ denotes an item of abstract syntax. Functions with a '?' subscript are predicates, e.g., $\mathit{Var}_?(e)$ returns true if e is a variable and false otherwise. Labels can be split into disjoint sets according to the innermost user lambda that contains them. For example, in the following program, which has three user lambdas, these sets are $\{1, 6, 4, 8\}$, $\{2, 9, 5, 10\}$ and $\{3, 7\}$.

```
(λ1(u1 k1) 6((λ2(u2 k2) 9((λ5(u5) 10(k2 u1)) u2))
             (λ3(u3 k3) 7(k3 u3))
             (λ4(u4) 8(k1 u4))))
```

The "label to variable" map $LV(\psi)$ returns all the variables bound by any lambdas that belong in the same set as ψ, e.g., $LV(8) = \{\mathtt{u1}, \mathtt{k1}, \mathtt{u4}\}$ and $LV(5) = \{\mathtt{u2}, \mathtt{k2}, \mathtt{u5}\}$. We use this map to model stack behavior, because all the continuation lambdas that "belong" to a given user lambda λ_l get closed by extending λ_l's stack frame (cf. section 4). Notice that, for any ψ, $LV(\psi)$ contains exactly one continuation variable.

We use two notations for tuples, $(e_1, \ldots, e_n)$ and $\langle e_1, \ldots, e_n \rangle$, to avoid confusion when tuples are deeply nested. We use the latter for lists as well; ambiguities will be resolved by the context. Lists are also described by a head-tail notation, e.g., $3 :: \langle 1, 3, -47 \rangle$.

The semantics of RCPS appears in Fig. 2. Execution traces alternate between *Eval* and *Apply* states. At an *Eval* state, we evaluate the subexpressions of a call site before performing a call. At an *Apply* state, we perform the call.

The last component of each state is a *time*, which is a sequence of call sites. *Eval* to *Apply* transitions increment the time by recording the label of the corresponding call site. *Apply* to *Eval* transitions leave the time unchanged. Thus, the time t of a state reveals the call sites along the execution path to that state.

Times indicate points in the execution when variables are bound. The binding environment β is a partial function that maps variables to their binding times. The variable environment ve maps variable-time pairs to values. To find the value of a variable v, we look up the time v was put in β, and use that to search for the actual value in ve.

Let's look at the transitions more closely. At a *UEval* state with call site $(f\,e\,q)^l$, we evaluate f, e and q using the function $\mathcal{A}_{cs}$. Lambdas are paired up

$$
\begin{array}{rl}
\varsigma \in \mathit{State} = & \mathit{Eval} + \mathit{Apply} \\
\varsigma \in \mathit{Eval} = & \mathit{UEval} + \mathit{CEval} \\
\varsigma \in \mathit{UEval} = & \mathit{UCall} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Time} \\
\varsigma \in \mathit{CEval} = & \mathit{CCall} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Time} \\
\varsigma \in \mathit{Apply} = & \mathit{UApply} + \mathit{CApply} \\
\varsigma \in \mathit{UApply} = & \mathit{UClos} \times \mathit{UClos} \times \mathit{CClos} \times \\
& \mathit{VEnv} \times \mathit{Time}
\end{array}
\qquad
\begin{array}{rl}
\varsigma \in \mathit{CApply} = & \mathit{CClos} \times \mathit{UClos} \times \mathit{VEnv} \times \mathit{Time} \\
\mathit{Clos} = & \mathit{UClos} + \mathit{CClos} \\
\mathrm{d} \in \mathit{UClos} = & \mathit{ULam} \times \mathit{BEnv} \\
\mathrm{c} \in \mathit{CClos} = & (\mathit{CLam} \times \mathit{BEnv}) + \mathit{halt} \\
\beta \in \mathit{BEnv} = & \mathit{Var} \rightharpoonup \mathit{Time} \\
\mathit{ve} \in \mathit{VEnv} = & \mathit{Var} \times \mathit{Time} \rightharpoonup \mathit{Clos} \\
t \in \mathit{Time} = & \mathit{Lab}^*
\end{array}
$$

$$
\mathcal{A}_{cs}(g, \beta, \mathit{ve}) \triangleq \begin{cases} (g, \beta) & \mathit{Lam}_?(g) \\ \mathit{ve}(g, \beta(g)) & \mathit{Var}_?(g) \end{cases}
$$

UEval to *UApply*:
$$
\begin{array}{l}
(\llbracket (f\ e\ q)^l \rrbracket, \beta, \mathit{ve}, t) \ \rightarrow\ (\mathit{proc}, \mathrm{d}, \mathrm{c}, \mathit{ve}, l :: t) \\
\mathit{proc} = \mathcal{A}_{cs}(f, \beta, \mathit{ve}) \\
\mathrm{d} = \mathcal{A}_{cs}(e, \beta, \mathit{ve}) \\
\mathrm{c} = \mathcal{A}_{cs}(q, \beta, \mathit{ve})
\end{array}
$$

CEval to *CApply*:
$$
\begin{array}{l}
(\llbracket (q\ e)^\gamma \rrbracket, \beta, \mathit{ve}, t) \ \rightarrow\ (\mathit{proc}, \mathrm{d}, \mathit{ve}, \gamma :: t) \\
\mathit{proc} = \mathcal{A}_{cs}(q, \beta, \mathit{ve}) \\
\mathrm{d} = \mathcal{A}_{cs}(e, \beta, \mathit{ve})
\end{array}
$$

UApply to *Eval*:
$$
\begin{array}{l}
(\mathit{proc}, \mathrm{d}, \mathrm{c}, \mathit{ve}, t) \ \rightarrow\ (\mathit{call}, \beta', \mathit{ve}', t) \\
\mathit{proc} = \langle \llbracket (\lambda_l\ (u\ k)\ \mathit{call}) \rrbracket, \beta \rangle \\
\beta' = \beta[u \mapsto t][k \mapsto t] \\
\mathit{ve}' = \mathit{ve}[(u, t) \mapsto \mathrm{d}][(k, t) \mapsto \mathrm{c}]
\end{array}
$$

CApply to *Eval*:
$$
\begin{array}{l}
(\mathit{proc}, \mathrm{d}, \mathit{ve}, t) \ \rightarrow\ (\mathit{call}, \beta', \mathit{ve}', t) \\
\mathit{proc} = \langle \llbracket (\lambda_\gamma\ (u)\ \mathit{call}) \rrbracket, \beta \rangle \\
\beta' = \beta[u \mapsto t] \\
\mathit{ve}' = \mathit{ve}[(u, t) \mapsto \mathrm{d}]
\end{array}
$$

Fig. 2. Concrete semantics and domains for Restricted CPS

with β to become closures, while variables are looked up in *ve* using β. We add the label l in front of the current time and transition to a *UApply* state.

From *UApply* to *Eval*, we bind the formals of a procedure $\langle \llbracket (\lambda_l\ (u\ k)\ \mathit{call}) \rrbracket, \beta \rangle$ to the arguments and jump to its body. The new binding environment β' is an extension of the procedure's environment, with u and k mapped to the current time. The new variable environment ve' maps (u, t) to the user argument d, and (k, t) to the continuation argument c.

The remaining two transitions are similar. We use *halt* to denote the top-level continuation of a program *pr*. The initial state $\mathcal{I}(\mathit{pr})$ is $((\mathit{pr}, \emptyset), \mathit{input}, \mathit{halt}, \emptyset, \langle\rangle)$, where *input* is a closure of the form $\langle \llbracket (\lambda_l\ (u\ k)\ \mathit{call}) \rrbracket, \emptyset \rangle$. Note that the initial time is the empty sequence of call sites.

In the terminology of abstract interpretation, this semantics is called the *concrete* semantics. In order to find properties of a program at compile time, one needs to derive a computable approximation of the concrete semantics, called the *abstract* semantics. CFA2 and k-CFA are such approximations.

CPS-based compilers may or may not use a control stack for the final code. Steele's view, illustrated in the Rabbit compiler [13], is that argument evaluation pushes stack and function calls are GOTOs. Since arguments in CPS are not calls, argument evaluation is always trivial and Rabbit never needs to push stack. By this approach, every call in CPS is a tail call.

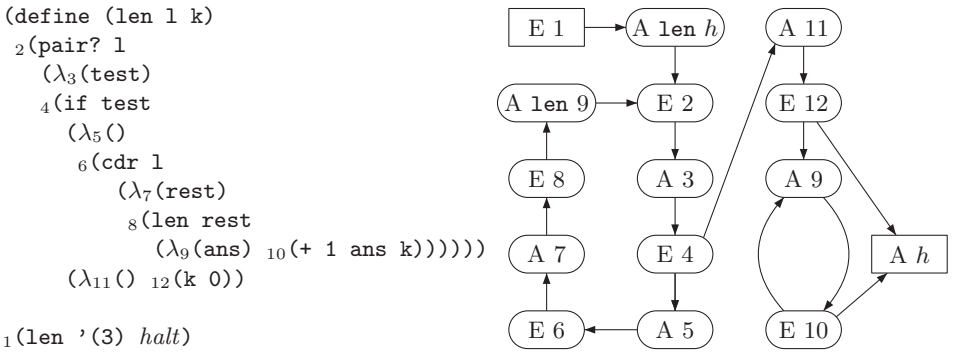

Fig. 3. 0CFA on `len`

An alternative style was used in the Orbit compiler [14]. At every function call, Orbit pushes a frame for the arguments. By this approach, tail calls are only the calls where the continuation argument is a variable. These CPS call sites were in tail position in the initial direct-style program. *CEval* states where the operator is a variable are calls to the current continuation with a return value. Orbit pops the stack at tail calls and before calling the current continuation.

We will see later that the abstract semantics of CFA2 uses a stack, like Orbit. However, CFA2 computes safe flow information which can be used by both aforementioned approaches. The workings of the abstract interpretation are independent of what style an implementor chooses for the final code.

3 Limitations of *k*-CFA

In this section, we discuss the main causes of imprecision and inefficiency in k-CFA. Our motivation in developing CFA2 is to create a higher-order flow analysis that overcomes these limitations.

We assume some familiarity with k-CFA, and abstract interpretation in general. Detailed descriptions on these topics can be found in [1,12]. We use Scheme syntax for our example programs.

3.1 *k*-CFA does not match calls and returns

In order to make the state space of k-CFA finite, Shivers chose a mechanism similar to the call-strings of Sharir and Pnueli [2]. Thus, recursive programs introduce approximation by folding an unbounded number of recursive calls down to a fixed-size call-string. In effect, by applying k-CFA on a higher-order program, we turn it into a finite-state machine. Taken to the extreme, when k is zero, a function can return to any of its callers, not just to the last one.

For example, consider the function that computes the length of a list, written in CPS (Fig. 3). 0CFA on `len` produces the graph in Fig. 3. *Eval* states (marked

with "E") mention the corresponding call site. *Apply* states are marked with "A". *UApply* states mention the callee and the continuation argument. The continuation variable `k` is bound to either *halt* or λ_9. The cycle on the left is taken when the test is true, and it leads to a recursive call. The cycle on the right is taken by returning to λ_9 after a recursive call. Every path from the start to the end node is a valid 0CFA execution. In particular, we cannot exclude the path that recurs four times but applies λ_9 twice. By following such a path, the program will terminate with a *non-empty* stack. It is clear that k-CFA cannot help much with optimizations that require accurate calculation of the stack change between program states, such as stack allocation of closure environments.

Spurious flows caused by call/return mismatch affect traditional data-flow information as well. For instance, 0CFA-constant-propagation for the program below cannot spot that `n2` is the constant 2, because 1 also flows to `x` and is mistakenly passed to the continuation λ_2. 1CFA helps in this example, but repeated η-expansion of the identity function can trick k-CFA for any k.

```
(let ((id (λ(x k) (k x))))
  (id 1 (λ1(n1) (id 2 (λ2(n2) (+ n1 n2 halt))))))
```

In a non-recursive program, a large enough k can provide accurate call/return matching, but this is not desirable because the analysis becomes intractably slow even when k is 1 [10]. Moreover, the ubiquity of recursion in higher-order programs calls for a static analysis that can match an unbounded number of calls and returns. This can be done if we approximate programs using pushdown models instead of finite-state machines.

3.2 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 run-time environment is referred to as the *environment problem* [1]. For example, trace through the execution of the following direct-style code:

```
(let ((f (λ(x thunk) (if (integer? 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 an integer, so we jump through `(thunk)` to `(λ1() x)`, and reference `x`, which, despite the just-completed test, is *not* an integer: it is the earlier-bound string `"foo"`. Thus, during abstract interpretation, it is generally *unsafe* to assume that a variable reference has some property just because an earlier reference had that property.

This has an unfortunate consequence: sometimes an earlier reference provides *safe* information about the reference at hand and k-CFA does not spot it:

```
(define (compose-same f x) 2(f 1(f x)))
```

In `compose-same`, both references to `f` are always bound at the same time. However, if multiple closures flow to `f`, k-CFA may call one closure at call site 1 and a different closure at call site 2. This flow never happens at run time.

CFA2 tackles this problem by treating references for a variable v differently from one another, depending on their location in the source code. If v appears in a static context where we know the current stack frame is its environment record, we can be precise. If v appears free in some possibly escaping lambda, we cannot predict its extent so we fall back to a conservative approximation.

3.3 Imprecision increases the running time of the analysis

It is known that k-CFA for $k > 0$ is not a cheap analysis, both in theory [10] and in practice [16]. Counterintuitively, imprecision in higher-order control-flow analyses can increase their running time: imprecision induces spurious control-flow paths, along which the analysis must then flow data, thus creating further spurious paths, and so on, in a vicious cycle which creates extra work whose only function is to degrade the precision of the analysis. This is why techniques that aggressively prune the search space, such as abstract garbage collection [8], not only increase the precision, but can also improve the speed of the analysis.

In the previous subsections, we saw examples of information known at compile time that k-CFA cannot exploit. CFA2 uses this information. The enhanced precision of CFA2 has a positive effect on its running time (cf. section 6).

4 The CFA2 semantics

4.1 Abstract semantics

The CFA2 semantics is an abstract interpreter that executes a program in RCPS, using a stack for variable binding and return-point information.

We describe the stack-management policy with an example. Assume that we run the `len` program of section 3. When calling `(len '(3)` *halt*`)` we push a frame $[\texttt{l} \mapsto \texttt{(3)}][\texttt{k} \mapsto \textit{halt}]$ on the stack. The test `(pair? l)` is true, so we add the binding $[\texttt{test} \mapsto \textit{true}]$ to the top frame and jump to the true branch. We take the `cdr` of `l` and add the binding $[\texttt{rest} \mapsto \texttt{()}]$ to the top frame. We call `len` again, push a new frame for its arguments and jump to its body. This time the test is false, so we extend the top frame with $[\texttt{test} \mapsto \textit{false}]$ and jump to the false branch. The call to `k` is a function return, so we pop a frame and pass `0` to λ_9. Call site 10 is also a function return, so we pop the remaining frame and pass `1` to the top-level continuation *halt*.

In general, we push a frame at function entries and pop at tail calls and at function returns. Results of intermediate computations are stored in the top frame. This policy enforces two invariants about the abstract interpreter. First, when executing inside a user function $(\lambda_l (u\, k)\ \textit{call})$, the domain of the top frame is a subset of $LV(l)$. Second, the frame below the top frame is the environment of the current continuation.

$\widehat{UEval}$ to $\widehat{UApply}$:

$$([\![(f\ e\ q)^l]\!], st, h) \rightsquigarrow (ulam, \hat{\mathrm{d}}, \hat{\mathrm{c}}, st', h)$$
$$ulam \in \mathcal{A}_u(f, st, h)$$
$$\hat{\mathrm{d}} = \mathcal{A}_u(e, st, h)$$
$$\hat{\mathrm{c}} = \mathcal{A}_k(q, st)$$
$$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \wedge (H_?(f) \vee Lam_?(f)) \\ setTop([f \mapsto \{ulam\}], st) & Lam_?(q) \wedge S_?(f) \end{cases}$$

$\widehat{UApply}$ to $\widehat{Eval}$:

$$([\![(\lambda_l\ (u\ k)\ call)]\!], \hat{\mathrm{d}}, \hat{\mathrm{c}}, st, h) \rightsquigarrow (call, st', h')$$
$$st' = push([u \mapsto \hat{\mathrm{d}}][k \mapsto \hat{\mathrm{c}}], st)$$
$$h' = \begin{cases} h \sqcup [u \mapsto \hat{\mathrm{d}}] & H_?(u) \\ h & S_?(u) \end{cases}$$

$\widehat{CEval}$ to $\widehat{CApply}$:

$$([\![(q\ e)^\gamma]\!], st, h) \rightsquigarrow (clam, \hat{\mathrm{d}}, st', h)$$
$$clam = \mathcal{A}_k(q, st)$$
$$\hat{\mathrm{d}} = \mathcal{A}_u(e, st, h)$$
$$st' = \begin{cases} pop(st) & Var_?(q) \\ st & Lam_?(q) \end{cases}$$

$\widehat{CApply}$ to $\widehat{Eval}$:

$$([\![(\lambda_\gamma\ (u)\ call)]\!], \hat{\mathrm{d}}, st, h) \rightsquigarrow (call, st', h')$$
$$st' = setTop([u \mapsto \hat{\mathrm{d}}], st)$$
$$h' = \begin{cases} h \sqcup [u \mapsto \hat{\mathrm{d}}] & H_?(u) \\ h & S_?(u) \end{cases}$$

$$\mathcal{A}_u(e, st, h) \triangleq \begin{cases} \{e\} & Lam_?(e) \\ st(e) & S_?(e) \\ h(e) & H_?(e) \end{cases}$$

$$\mathcal{A}_k(q, st) \triangleq \begin{cases} q & Lam_?(q) \\ st(q) & Var_?(q) \end{cases}$$

Abstract domains:

$$\begin{aligned} \hat{\varsigma} \in \widehat{UEval} &= UCall \times Stack \times Heap \\ \hat{\varsigma} \in \widehat{UApply} &= ULam \times \widehat{UClos} \times \widehat{CClos} \times Stack \times Heap \\ \hat{\varsigma} \in \widehat{CEval} &= CCall \times Stack \times Heap \\ \hat{\varsigma} \in \widehat{CApply} &= \widehat{CClos} \times \widehat{UClos} \times Stack \times Heap \\ \hat{\mathrm{d}} \in \widehat{UClos} &= Pow(ULam) \\ \hat{\mathrm{c}} \in \widehat{CClos} &= CLam + halt \\ fr, tf \in Frame &= (UVar \rightharpoonup \widehat{UClos}) \cup (CVar \rightharpoonup \widehat{CClos}) \\ st \in Stack &= Frame^* \\ h \in Heap &= UVar \rightharpoonup \widehat{UClos} \end{aligned}$$

Stack operations:

$$\begin{aligned} pop(tf :: st) &\triangleq st \\ push(fr, st) &\triangleq fr :: st \\ (tf :: st)(v) &\triangleq tf(v) \\ setTop([u \mapsto \hat{\mathrm{d}}], tf :: st) &\triangleq tf[u \mapsto \hat{\mathrm{d}}] :: st \end{aligned}$$

Fig. 4. Abstract semantics and relevant definitions

Each variable v in our example was looked up in the top frame, because each lookup happened while we were executing inside the lambda that binds v. This is not always the case; in the first snippet of section 3.2 there is a reference to `x` inside λ_1. When control reaches that reference, the top frame does not belong to the user lambda that binds `x`. CFA2 uses a *heap* to look up such references. The following definition makes these concepts precise.

Definition 1 (Stack and heap references).

- *Let ψ be a call site that refers to a variable v. The predicate $S_?(v)$ holds iff $v \in LV(\psi)$. We call v a* ***stack reference****.*
- *Let ψ be a call site that refers to a variable v. The predicate $H_?(v)$ holds iff $v \notin LV(\psi)$. We call v a* ***heap reference****.*
- *v is a* ***stack variable*** *iff all its references satisfy $S_?$.*
- *v is a* ***heap variable*** *iff some of its references satisfy $H_?$.*

Put differently, if the innermost user lambda that contains ψ is the one that binds v, then v is a stack reference. In addition, if v is bound by a continuation lambda λ_γ, and the innermost user lambda that contains ψ also contains λ_γ, then v is a stack reference. Intuitively, only heap references may escape. We look up stack references in the top frame, and heap references in the heap. Stack lookups below the top frame never happen.

The CFA2 semantics appears in Fig. 4. An abstract value is either an abstract user closure (member of the set $\widehat{UClos}$) or an abstract continuation closure (member of $\widehat{CClos}$). An abstract user closure is a set of user lambdas. An abstract continuation closure is either a continuation lambda or *halt*. A frame is a map from variables to abstract values, and a stack is a sequence of frames. All stack operations except *push* are defined for non-empty stacks only. A heap is a map from variables to abstract values. It contains only user bindings because in RCPS every continuation variable is a stack variable.

On transition from a $\widehat{UEval}$ state $\hat{\varsigma}$ to a $\widehat{UApply}$ state $\hat{\varsigma}'$, we first evaluate f, e and q. We evaluate user terms using $\mathcal{A}_u$ and continuation terms using $\mathcal{A}_k$. We non-deterministically choose one of the lambdas that flow to f as the operator in $\hat{\varsigma}'$.The change to the stack depends on q and f. If q is a variable, the call is a tail call so we pop the stack (case 1). If q is a lambda, it evaluates to a new continuation closure whose environment is the top frame, hence we do not pop the stack (cases 2, 3). Moreover, if f is a lambda or a heap reference then we leave the stack unchanged. However, if f is a stack reference, we set f's value on the top frame to be $\{ulam\}$, possibly forgetting other lambdas that may flow to f. This "stack filtering" prevents fake rebinding (cf. section 3.2): when we return to $\hat{\mathrm{c}}$, we may reach more stack references of f. These references and the current one are all bound at the same time. Since we are committing to *ulam* in this transition, these references must also be bound to *ulam*.

In the $\widehat{UApply}$-to-$\widehat{Eval}$ transition, we push a frame for the procedure's arguments. In addition, if u is a heap variable we must update its binding in the heap. The join operation $\sqcup$ is defined in the usual way.

In a $\widehat{CEval}$-to-$\widehat{CApply}$ transition, we are preparing for a call to a continuation so we must reset the stack to the stack of its birth. When q is a variable, the $\widehat{CEval}$ state is a function return and the continuation's environment is the second stack frame. Therefore, we pop a frame before calling *clam*. When q is a lambda, it is a newly created closure thus the stack does not change. Note that the transition is deterministic, unlike $\widehat{UEval}$-to-$\widehat{UApply}$. Since we always know which continuation we are about to call, call/return mismatch *never* happens. For

$$|([\![(h_1 \ldots h_n)^{\psi}]\!], \beta, ve, t)|_{ca} = ([\![(h_1 \ldots h_n)^{\psi}]\!],\ \mathit{toStack}(\mathit{LV}(\psi), \beta, ve),\ |ve|_{ca})$$

$$|(\langle [\![(\lambda_l\,(u\,k)\ \mathit{call})]\!], \beta\rangle, \mathrm{d}, \mathrm{c}, ve, t)|_{ca} = ([\![(\lambda_l\,(u\,k)\ \mathit{call})]\!], |\mathrm{d}|_{ca}, |\mathrm{c}|_{ca}, \mathit{st}, |ve|_{ca})$$

$$\text{where } \mathit{st} = \begin{cases} \langle\rangle & \mathrm{c} = \mathit{halt} \\ \mathit{toStack}(\mathit{LV}(\gamma), \beta', ve) & \mathrm{c} = ([\![(\lambda_\gamma(u')\mathit{call}')]\!], \beta') \end{cases}$$

$$|(\langle [\![(\lambda_\gamma\,(u)\ \mathit{call})]\!], \beta\rangle, \mathrm{d}, ve, t)|_{ca} = ([\![(\lambda_\gamma\,(u)\ \mathit{call})]\!],\ |\mathrm{d}|_{ca},\ \mathit{toStack}(\mathit{LV}(\gamma), \beta, ve),\ |ve|_{ca})$$

$$|(\mathit{halt}, \mathrm{d}, ve, t)|_{ca} = (\mathit{halt},\ |\mathrm{d}|_{ca},\ \langle\rangle,\ |ve|_{ca})$$

$$|([\![(\lambda_l\,(u\,k)\ \mathit{call})]\!],\ \beta)|_{ca} = \{[\![(\lambda_l\,(u\,k)\ \mathit{call})]\!]\}$$

$$|([\![(\lambda_\gamma\,(u)\ \mathit{call})]\!],\ \beta)|_{ca} = [\![(\lambda_\gamma\,(u)\ \mathit{call})]\!]$$

$$|\mathit{halt}|_{ca} = \mathit{halt}$$

$$|ve|_{ca} = \{\,(u, \bigsqcup_t |ve(u,t)|_{ca})\ :\ H_?(u)\}$$

$$\mathit{toStack}(\{u_1, \ldots, u_n, k\}, \beta, ve) \triangleq$$

$$\begin{cases} \langle[\overline{u_i \mapsto \hat{\mathrm{d}}_i}][k \mapsto \mathit{halt}]\rangle & \hat{\mathrm{d}}_i = |ve(u_i, \beta(u_i))|_{ca} \wedge \\ & \mathit{halt} = ve(k, \beta(k)) \\ [\overline{u_i \mapsto \hat{\mathrm{d}}_i}][k \mapsto [\![(\lambda_\gamma\,(u)\ \mathit{call})]\!]] :: \mathit{toStack}(\mathit{LV}(\gamma), \beta', ve) & \hat{\mathrm{d}}_i = |ve(u_i, \beta(u_i))|_{ca} \wedge \\ & ([\![(\lambda_\gamma\,(u)\ \mathit{call})]\!], \beta') = ve(k, \beta(k)) \end{cases}$$

Fig. 5. From concrete states to abstract states

instance, the function `len` may be called from many places in a program, so multiple continuations may flow to `k`. But, by retrieving `k`'s value from the stack, we always return to the correct continuation.

In the $\widehat{\mathit{CApply}}$-to-$\widehat{\mathit{Eval}}$ transition, our stack policy dictates that we extend the top frame with the binding for the continuation's parameter u. If u is a heap variable, we also update the heap.

4.2 Correctness of CFA2

In this section, we show that the CFA2 semantics safely approximates the concrete semantics. First, we define a map $|\cdot|_{ca}$ from concrete to abstract states. Next, we show that if a state ς transitions to ς' in the concrete semantics, the abstract counterpart $|\varsigma|_{ca}$ of ς transitions to a state $\hat{\varsigma}'$ which approximates $|\varsigma'|_{ca}$. By proving this, we ensure that the possible behaviors of the abstract interpreter include the actual run-time behavior of the program.

The map $|\cdot|_{ca}$ appears in Fig. 5. The abstraction of an *Eval* state ς of the form $([\![(h_1 \ldots h_n)^{\psi}]\!], \beta, ve, t)$ is an $\widehat{\mathit{Eval}}$ state $\hat{\varsigma}$ with the same call site. Since ς does not have a stack, we must expose stack-related information hidden in β and ve. Assume that λ_l is the innermost user lambda that contains ψ. To reach ψ, control passed from a $\widehat{\mathit{UApply}}$ state $\hat{\varsigma}'$ over λ_l. According to our stack policy, the top frame must contain bindings for the formals of λ_l and any temporaries added

along the path from $\hat{\varsigma}'$ to $\hat{\varsigma}$. Therefore, the domain of the top frame is a subset of $LV(l)$, i.e., a subset of $LV(\psi)$. For each user variable $u_i \in (LV(\psi) \cap \mathrm{dom}(\beta))$, the top frame contains $[u_i \mapsto |ve(u_i, \beta(u_i))|_{ca}]$. Let k be the sole continuation variable in $LV(\psi)$. If $ve(k, \beta(k))$ is *halt* (the return continuation is the top-level continuation), the rest of the stack is empty. If $ve(k, \beta(k))$ is $([\![(\lambda_\gamma\,(u)\ \mathit{call})]\!], \beta')$, the second frame is for the user lambda in which λ_γ was born, and so forth: proceeding through the stack, we add a frame for each live activation of a user lambda until we reach the top-level continuation.

The abstraction of a *UApply* state over $\langle [\![(\lambda_l\,(u\,k)\ \mathit{call})]\!], \beta\rangle$ is a $\widehat{\mathit{UApply}}$ state $\hat{\varsigma}$ whose operator is $[\![(\lambda_l\,(u\,k)\ \mathit{call})]\!]$. The stack of $\hat{\varsigma}$ is the stack in which the continuation argument was created, and we compute it using *toStack* as above.

Abstracting a *CApply* is similar to the *UApply* case, only now the top frame is the environment of the continuation operator. Note that the abstraction maps drop the time of the concrete states, since the abstract states do not use times.

We can now state our simulation theorem. The proof proceeds by case analysis on the concrete transition relation. The relation $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$ is a partial ordering on abstract states and can be read as "$\hat{\varsigma}_1$ is more precise than $\hat{\varsigma}_2$". The proof and the definition of $\sqsubseteq$ can be found in [17].

Theorem 1 (Simulation). *If $\varsigma \rightarrow \varsigma'$ and $|\varsigma|_{ca} \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $|\varsigma'|_{ca} \sqsubseteq \hat{\varsigma}'$.*

5 Computing CFA2

In the previous section we saw how CFA2 addresses the problems of k-CFA, but did not discuss how to explore its state space. Since the size of the stack is unbounded, the state space of CFA2 is infinite and the standard workset algorithms for k-CFA [1, 12] will diverge. For this reason, we have designed a new algorithm based on summarization, a dynamic-programming technique widely used in the interprocedural analysis of first-order programs [2–4] and in context-free language (CFL) reachability algorithms [18].

The difficulty with analyzing programs in a way that respects call/return matching is that the reachable program points from a point n do not depend solely on n, but on the stack contents as well. The intuition behind summarization is to flow facts from n with an *empty* stack to another point n' in the same procedure. We say that n' is *same-context reachable* from n. These facts are then suitably combined to get flow facts for the whole program.

Let's do the simplest data-flow analysis for the *first-order* program of Fig. 6, namely find which nodes are reachable from the entry of the main function. We will do so by using *path edges*, i.e., edges whose source is the entry of a procedure and target is some program point in the same procedure. Path edges represent intraprocedural paths, hence the name. We write n_f for the entry node and x_f for the exit node of a procedure f. Solid arrows are intraprocedural steps. Dotted arrows go from call nodes to the corresponding return nodes. Dashed arrows go from call nodes to entries and from exits to return nodes.

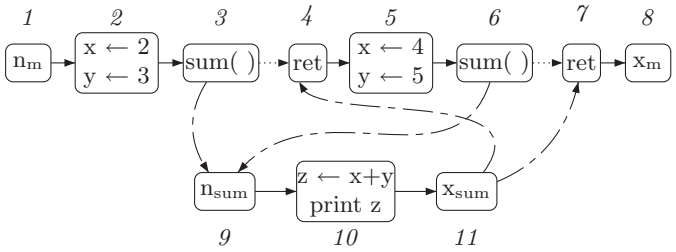

Fig. 6. Interprocedural flow-graph for a simple program

We first scan the program to identify the call sites of each procedure and then start the reachability analysis. Obviously, from 1 we can go to 2 and then to 3, so we record $\langle 1,1\rangle$, $\langle 1,2\rangle$ and $\langle 1,3\rangle$. Then 3 calls sum, so we jump to its body. Analysing sum produces $\langle 9,9\rangle$, $\langle 9,10\rangle$ and $\langle 9,11\rangle$. Node 11 is an exit reachable from 9, so each caller of sum can reach its corresponding return point. We keep track of this fact by recording the *summary* edges $\langle 3,4\rangle$ and $\langle 6,7\rangle$. Now 4 is reachable from 1, so we discover a new path edge $\langle 1,4\rangle$. We go on to discover $\langle 1,5\rangle$ and $\langle 1,6\rangle$. Reachability inside sum does not depend on its calling context, so from the summary edge $\langle 6,7\rangle$ we infer that we can reach 7, so we add $\langle 1,7\rangle$ to the set of path edges. Finally, we record $\langle 1,8\rangle$ which is the end of the program.

We cannot apply summarization to higher-order languages out of the box, because we do not know the call sites of a function by looking at a program's source code. We need a *search-based* variant of summarization, that records callers as it discovers them. Specifically, in the previous example we can record the call $\langle 3,9\rangle$ when we reach 3. On reaching 11, we record the summary edge $\langle 9,11\rangle$. To find possible return points for sum, we look at the set of callers. Since 3 calls 9, 11 can return to 4. Later, when we reach 6, we look at the set of summaries and see that sum reaches its exit, so 6 can reach 7. Note that our search-based variant of summarization uses entry-to-exit summaries instead of call-to-return summaries.

5.1 Local semantics

Summarization-based algorithms operate on a finite set of program points. Hence, we cannot use (an infinite number of) abstract states as program points. For this reason, we introduce *local states* and define a map $|\cdot|_{al}$ from abstract to local states (Fig. 7). Intuitively, a local state is like an abstract state but with a single frame instead of a stack. Discarding the rest of the stack makes the local state space finite; keeping the top frame allows precise lookups for stack references.

Essentially, the local semantics describes executions that do not touch the rest of the stack (in other words, executions where functions do not return). Thus, a $\widetilde{CEval}$ state with call site $(k\,e)^\gamma$ has no successor in this semantics. Since functions do not call their continuations, the frames of local states contain only user bindings. Local steps are otherwise similar to abstract steps. The

$\widetilde{UEval}$ to $\widetilde{UApply}$:
$([\![(f\ e\ q)^l]\!], tf, h) \twoheadrightarrow (ulam, \hat{\mathrm{d}}, h)$
$ulam \in \hat{\mathcal{A}}_u(f, tf, h)$
$\hat{\mathrm{d}} = \hat{\mathcal{A}}_u(e, tf, h)$

$\widetilde{UApply}$ to $\widetilde{Eval}$:
$([\![(\lambda_l\ (u\ k)\ call)]\!], \hat{\mathrm{d}}, h) \twoheadrightarrow (call, [u \mapsto \hat{\mathrm{d}}], h')$
$h' = \begin{cases} h \sqcup [u \mapsto \hat{\mathrm{d}}] & H_?(u) \\ h & S_?(u) \end{cases}$

$\widetilde{CEval}$ to $\widetilde{CApply}$:
$([\![(clam\ e)^\gamma]\!], tf, h) \twoheadrightarrow (clam, \hat{\mathrm{d}}, tf, h)$
$\hat{\mathrm{d}} = \hat{\mathcal{A}}_u(e, tf, h)$

$\widetilde{CApply}$ to $\widetilde{Eval}$:
$([\![(\lambda_\gamma\ (u)\ call)]\!], \hat{\mathrm{d}}, tf, h) \twoheadrightarrow (call, tf', h')$
$tf' = tf[u \mapsto \hat{\mathrm{d}}]$
$h' = \begin{cases} h \sqcup [u \mapsto \hat{\mathrm{d}}] & H_?(u) \\ h & S_?(u) \end{cases}$

$$\hat{\mathcal{A}}_u(e, tf, h) \triangleq \begin{cases} \{e\} & Lam_?(e) \\ tf(e) & S_?(e) \\ h(e) & H_?(e) \end{cases}$$

Local domains:

$$\begin{aligned} \tilde{\varsigma} \in \widetilde{Eval} &= Call \times \widetilde{Stack} \times Heap \\ \tilde{\varsigma} \in \widetilde{UApply} &= ULam \times \widetilde{UClos} \times Heap \\ \tilde{\varsigma} \in \widetilde{CApply} &= \widetilde{CClos} \times \widetilde{UClos} \times \widetilde{Stack} \times Heap \\ \widetilde{Frame} &= UVar \rightharpoonup \widetilde{UClos} \\ \widetilde{Stack} &= \widetilde{Frame} + \langle\rangle \end{aligned}$$

Abstract to local maps:

$$\begin{aligned} |(call, st, h)|_{al} &= (call, |st|_{al}, h) \\ |(ulam, \hat{\mathrm{d}}, \hat{\mathrm{c}}, st, h)|_{al} &= (ulam, \hat{\mathrm{d}}, h) \\ |(\hat{\mathrm{c}}, \hat{\mathrm{d}}, st, h)|_{al} &= (\hat{\mathrm{c}}, \hat{\mathrm{d}}, |st|_{al}, h) \\ |tf :: st'|_{al} &= \{\, (u, tf(u)) \,:\, UVar_?(u)\} \\ |\langle\rangle|_{al} &= \langle\rangle \end{aligned}$$

Fig. 7. Local semantics

metavariable $\tilde{\varsigma}$ ranges over local states. We define the map $|\cdot|_{cl}$ from concrete to local states to be $|\cdot|_{al} \circ |\cdot|_{ca}$.

We can now see the emerging connection between local semantics and summarization: the local semantics is used for intraprocedural steps and function calls, and we discover return points by recording callers and summary edges.

Next, our algorithm needs to distinguish between different kinds of local states: entries, exits, calls, returns and inner states. CPS lends itself naturally to such a categorization:

- A $\widetilde{UApply}$ state corresponds to an **entry** node—control is about to enter the body of a function.
- A $\widetilde{CEval}$ state where the operator is a variable is an **exit** node—a function is about to pass its result to its context.
- A $\widetilde{UEval}$ state where the continuation argument is a variable is also an **exit**—at tail calls control does not return to the caller.
- A $\widetilde{UEval}$ state where the continuation argument is a lambda is a **call**.
- A $\widetilde{CEval}$ state where the operator is a lambda is an **inner** state.
- A $\widetilde{CApply}$ state is a **return** if its predecessor is an exit, or an **inner** state if its predecessor is also an inner state. Our algorithm will not need to distinguish between the two kinds of $\widetilde{CApply}$s; the difference is just conceptual.

Last, we generalize the notion of summary edges to handle tail recursion. In the following, we rewrite `sum` in a functional style and place it in a context where it gets called three times:

```
(let ((sum (λ(x y k) (+ x y (λ(z) 1(print z k))))))
 ...2(sum 2 3 (λ3(u1) call3))...
 ...4(sum 4 5 (λ5(u2) call5))...
 ...((λ6(n k2) 7(sum n 1 k2)) 9 (λ8(u3) call8))...)
```

The first time (site 2), we record a summary edge from the entry of `sum` to its exit at call site 1, and return to λ_3. Then, at the second call (site 4) we use the summary edge to find that `sum` will pass its result to λ_5. The third call is a tail call, so no continuation is born at call site 7. Upon return from `sum`, we must be careful to pass the result to λ_8. Also, we must restore the environment of the call to λ_6, *not* the environment of the tail call. We achieve these by recording a "cross-procedure" summary edge from the entry of λ_6 to call site 1. This transitive nature of summaries is essential for tail recursion.

5.2 Summarization

The algorithm for CFA2 is shown in Fig. 8. It is a search-based summarization for higher-order programs with tail calls. Its goal is to compute which local states are reachable from the initial state of a program through paths that respect call/return matching.

An edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ is an ordered pair of local states. We call $\tilde{\varsigma}_1$ the *source* and $\tilde{\varsigma}_2$ the *target* of the edge. The results of the analysis are stored in the set *Seen*. It contains path edges (from a procedure entry to a state in the same procedure) and summary edges (from an entry to a $\widetilde{CEval}$ exit, not necessarily in the same procedure). The target of an edge in *Seen* is reachable from the source and from the initial state (cf. theorem 2). Summaries are also stored in *Summary*.

The workset W contains path edges and summaries to be examined. *Final* records $\widetilde{CApply}$ states that call *halt* with a return value for the whole program. *Callers* contains triples $\langle \tilde{\varsigma}_1, \tilde{\varsigma}_2, \tilde{\varsigma}_3 \rangle$, where $\tilde{\varsigma}_1$ is an entry, $\tilde{\varsigma}_2$ is a call in the same procedure and $\tilde{\varsigma}_3$ is the entry of the callee. *TCallers* contains triples $\langle \tilde{\varsigma}_1, \tilde{\varsigma}_2, \tilde{\varsigma}_3 \rangle$, where $\tilde{\varsigma}_1$ is an entry, $\tilde{\varsigma}_2$ is a tail call in the same procedure and $\tilde{\varsigma}_3$ is the entry of the callee. The initial state $\tilde{\mathcal{I}}(pr)$ is defined as $|\mathcal{I}(pr)|_{cl}$. The helper function $succ(\tilde{\varsigma})$ returns the successor(s) of $\tilde{\varsigma}$ according to the local semantics.

At every iteration, we remove an edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ from W and branch depending on $\tilde{\varsigma}_2$. If $\tilde{\varsigma}_2$ is an entry, a return or an inner state (line 6), then its successor $\tilde{\varsigma}_3$ is a state in the same procedure. Since $\tilde{\varsigma}_2$ is reachable from $\tilde{\varsigma}_1$, $\tilde{\varsigma}_3$ is also reachable from $\tilde{\varsigma}_1$. If we have not already recorded the edge $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$, we do it now (line 25).

If $\tilde{\varsigma}_2$ is a call (line 8) then $\tilde{\varsigma}_3$ is an entry of a new procedure, so we propagate $(\tilde{\varsigma}_3, \tilde{\varsigma}_3)$ instead of $(\tilde{\varsigma}_1, \tilde{\varsigma}_3)$ (line 10). Next, we record the call in *Callers*. If an exit $\tilde{\varsigma}_4$ is reachable from $\tilde{\varsigma}_3$, it should return its result to the continuation born at $\tilde{\varsigma}_2$ (line 12). The function `Update` is responsible for computing the return state. We find the return value $\hat{\mathrm{d}}$ by evaluating the expression e_4 passed to the continuation

```
01    Summary, Callers, TCallers, Final ⟵ ∅
02    Seen, W ⟵ {(Ĩ(pr), Ĩ(pr))}
03    while W ≠ ∅
04      remove (ς̃1, ς̃2) from W
05      switch ς̃2
06        case ς̃2 of Entry, CApply, Inner-CEval
07          for each ς̃3 in succ(ς̃2) Propagate(ς̃1, ς̃3)
08        case ς̃2 of Call
09          for each ς̃3 in succ(ς̃2)
10            Propagate(ς̃3, ς̃3)
11            insert (ς̃1, ς̃2, ς̃3) in Callers
12            for each (ς̃3, ς̃4) in Summary  Update(ς̃1, ς̃2, ς̃3, ς̃4)
13        case ς̃2 of Exit-CEval
14          if ς̃1 = Ĩ(pr) then
15            Final(ς̃2)
16          else
17            insert (ς̃1, ς̃2) in Summary
18            for each (ς̃3, ς̃4, ς̃1) in Callers  Update(ς̃3, ς̃4, ς̃1, ς̃2)
19            for each (ς̃3, ς̃4, ς̃1) in TCallers Propagate(ς̃3, ς̃2)
20        case ς̃2 of Exit-TC
21          for each ς̃3 in succ(ς̃2)
22            Propagate(ς̃3, ς̃3)
23            insert (ς̃1, ς̃2, ς̃3) in TCallers
24            for each (ς̃3, ς̃4) in Summary Propagate(ς̃1, ς̃4)

      Propagate(ς̃1, ς̃2) ≜
25      if (ς̃1, ς̃2) not in Seen then insert (ς̃1, ς̃2) in Seen and W

      Update(ς̃1, ς̃2, ς̃3, ς̃4) ≜
26      ς̃1 of the form  ([[(λ_{l1}(u1 k1) call1)]] , d̂1, h1)
27      ς̃2 of the form  ([[(f e2 (λ_{γ2} (u2) call2))^{l2}]], tf2, h2)
28      ς̃3 of the form  ([[(λ_{l3}(u3 k3) call3)]] , d̂3, h2)
29      ς̃4 of the form  ([[(k4 e4)^{γ4}]], tf4, h4)
30      d̂ ⟵  Â_u(e4, tf4, h4)
31      tf ⟵  { tf2[f ↦ {[[(λ_{l3}(u3 k3) call3)]]}]   S?(f)
              { tf2                                    H?(f) ∨ Lam?(f)
32      ς̃ ⟵  ([[(λ_{γ2}(u2) call2)]], d̂, tf, h4)
33      Propagate(ς̃1, ς̃)

      Final(ς̃) ≜
34      ς̃ of the form  ([[(k e)^γ]], tf, h)
35      insert  (halt, Â_u(e, tf, h), ⟨⟩, h) in Final
```

Fig. 8. CFA2 workset algorithm

(lines 29-30). Since we are returning to λ_{γ_2}, we must restore the environment of its creation which is tf_2 (possibly with stack filtering, line 31). The new state $\tilde{\varsigma}$ is the corresponding return node of $\tilde{\varsigma}_2$, so we propagate $(\tilde{\varsigma}_1, \tilde{\varsigma})$ (lines 32-33).

If $\tilde{\varsigma}_2$ is a $\widetilde{CEval}$ exit and $\tilde{\varsigma}_1$ is the initial state (lines 14-15), then $\tilde{\varsigma}_2$'s successor is a final state (lines 34-35). If $\tilde{\varsigma}_1$ is some other entry, we record the edge in *Summary* and pass the result of $\tilde{\varsigma}_2$ to the callers of $\tilde{\varsigma}_1$ (lines 17-18). Last, consider the case of a tail call $\tilde{\varsigma}_4$ to $\tilde{\varsigma}_1$ (line 19). No continuation is born at $\tilde{\varsigma}_4$. Thus, we must find where $\tilde{\varsigma}_3$ (the entry that led to the tail call) was called from.

Then again, it is possible that all calls to $\tilde{\varsigma}_3$ are tail calls, in which case we keep searching further back in the call chain to find a return point. We do this backward search by transitively adding a summary edge from $\tilde{\varsigma}_3$ to $\tilde{\varsigma}_2$ (line 25).

If $\tilde{\varsigma}_2$ is a tail call (line 20), we find its successors and record the call in *TCallers* (lines 21-23). If a successor of $\tilde{\varsigma}_2$ goes to an exit, we propagate a summary transitively (line 24).

The local state space is finite, so there is a finite number of path and summary edges. We record edges as seen when we insert them in W, which ensures that no edge is inserted in W twice. Therefore, the algorithm terminates.

We obviously cannot visit an infinite number of abstract states. To establish the soundness of our flow analysis, we show that if an abstract state $\hat{\varsigma}$ is reachable from the initial state, then the algorithm visits $|\hat{\varsigma}|_{al}$ (cf. theorem 2). For instance, CFA2 on `len` (cf. section 3) will tell us that we reach program point 10, *not* that we reach 10 with a stack of size 1, 2, 3 etc.

Soundness guarantees that the CFA2 algorithm does not miss any flows, but it could also compute flows that do not happen in the abstract semantics. For example, a sound but useless algorithm would add all pairs of local states in *Seen*. We establish the completeness of our algorithm by proving that every visited edge has a corresponding abstract flow (cf. theorem 3).

The theorems use two definitions. The first associates a state $\hat{\varsigma}$ with its *corresponding entry*, i.e., the entry of the procedure that contains $\hat{\varsigma}$. The second finds all entries that reach the corresponding entry of $\hat{\varsigma}$ through tail calls. We include the proofs of the theorems in [17].

Definition 2. *The Corresponding Entry $CE_p(\hat{\varsigma})$ of a state $\hat{\varsigma}$ in a path p is:*

- $\hat{\varsigma}$, *if* $\hat{\varsigma}$ *is an Entry*
- $\hat{\varsigma}_1$, *if* $\hat{\varsigma}$ *is not an Entry,* $\hat{\varsigma}_2$ *is not an Exit-CEval,* $p \equiv p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, *and* $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}_1$
- $\hat{\varsigma}_1$, *if* $\hat{\varsigma}$ *is not an Entry,* $p \equiv p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow^+ \hat{\varsigma}_2 \rightsquigarrow \hat{\varsigma}_3 \rightsquigarrow^+ \hat{\varsigma}_4 \rightsquigarrow \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_2$ *is a Call and* $\hat{\varsigma}_4$ *is an Exit-CEval,* $CE_p(\hat{\varsigma}_2) = \hat{\varsigma}_1$, *and* $\hat{\varsigma}_3 \in CE^*_p(\hat{\varsigma}_4)$

Definition 3. *For a state $\hat{\varsigma}$ and a path p, $CE^*_p(\hat{\varsigma})$ is the smallest set such that:*

- $CE_p(\hat{\varsigma}) \in CE^*_p(\hat{\varsigma})$
- $CE^*_p(\hat{\varsigma}_1) \subseteq CE^*_p(\hat{\varsigma})$, *when* $p \equiv p_1 \rightsquigarrow \hat{\varsigma}_1 \rightsquigarrow \hat{\varsigma}_2 \rightsquigarrow^* \hat{\varsigma} \rightsquigarrow p_2$, $\hat{\varsigma}_1$ *is a Tail Call,* $\hat{\varsigma}_2$ *is an Entry, and* $\hat{\varsigma}_2 = CE_p(\hat{\varsigma})$

Theorem 2 (Soundness). *If $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}$ then, after summarization:*

- *if* $\hat{\varsigma}$ *is not a final state then* $(|CE_p(\hat{\varsigma})|_{al}, |\hat{\varsigma}|_{al}) \in Seen$
- *if* $\hat{\varsigma}$ *is a final state then* $|\hat{\varsigma}|_{al} \in Final$
- *if* $\hat{\varsigma}$ *is an Exit-CEval and* $\hat{\varsigma}' \in CE^*_p(\hat{\varsigma})$ *then* $(|\hat{\varsigma}'|_{al}, |\hat{\varsigma}|_{al}) \in Seen$

Theorem 3 (Completeness). *After summarization:*

- *For each* $(\tilde{\varsigma}_1, \tilde{\varsigma}_2)$ *in Seen, there exist* $\hat{\varsigma}_1$, $\hat{\varsigma}_2$ *and* p *such that* $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^* \hat{\varsigma}_1 \rightsquigarrow^* \hat{\varsigma}_2$ *and* $\tilde{\varsigma}_1 = |\hat{\varsigma}_1|_{al}$ *and* $\tilde{\varsigma}_2 = |\hat{\varsigma}_2|_{al}$ *and* $\hat{\varsigma}_1 \in CE^*_p(\hat{\varsigma}_2)$
- *For each* $\tilde{\varsigma}$ *in Final, there exist* $\hat{\varsigma}$ *and* p *such that* $p \equiv \hat{\mathcal{I}}(pr) \rightsquigarrow^+ \hat{\varsigma}$ *and* $\tilde{\varsigma} = |\hat{\varsigma}|_{al}$ *and* $\hat{\varsigma}$ *is a final state.*

	$S_?$	$H_?$	0CFA		1CFA		CFA2	
			visited	constants	visited	constants	visited	constants
`len`	9	0	81	0	126	0	55	2
`rev-iter`	17	0	121	0	198	0	82	4
`len-Y`	15	4	199	0	356	0	131	2
`tree-count`	33	0	293	2	2856	6	183	10
`ins-sort`	33	5	509	0	1597	0	600	4
`DFS`	94	11	1337	8	6890	8	1719	16
`flatten`	37	0	1520	0	6865	0	478	5
`sets`	90	3	3915	0	54414	0	4251	4
`church-nums`	46	23	19130	0	19411	0	22671	0

Fig. 9. Benchmark results

6 Evaluation

We implemented CFA2, 0CFA and 1CFA for the Twobit Scheme compiler [11] and used them to do constant propagation and folding. In this section we report on some initial measurements and comparisons.

We compared the effectiveness of the analyses on a small set of benchmarks (Fig. 9). We measured the number of stack and heap references in each program and the number of constants found by each analysis. We also recorded what goes in the workset in each analysis, i.e., the number of abstract states visited by 0CFA and 1CFA, and the number of path and summary edges visited by CFA2. The running time of an abstract interpretation is proportional to the amount of things inserted in the workset.

We chose programs that exhibit a variety of control-flow patterns. `Len` computes the length of a list recursively. `Rev-iter` reverses a list tail-recursively. `Len-Y` computes the length of a list using the Y-combinator instead of explicit recursion. `Tree-count` counts the nodes in a binary tree. `Ins-sort` sorts a list of numbers using insertion-sort. `DFS` does depth-first search of a graph. `Flatten` turns arbitrarily nested lists into a flat list. `Sets` defines the basic set operations and tests De Morgan's laws on sets of numbers. `Church-nums` tests distributivity of multiplication over addition for a few Church numerals.

CFA2 finds the most constants, followed by 1CFA. 0CFA is the least precise. CFA2 is also more efficient at exploring its abstract state space. In five out of nine cases, it visits fewer paths than 0CFA does states. The visited set of CFA2 can be up to 3.2 times smaller (`flatten`), and up to 1.3 times larger (`DFS`) than the visited set of 0CFA. 1CFA is less efficient than both 0CFA (9/9 cases) and CFA2 (8/9 cases). The visited set of 1CFA can be significantly larger than that of CFA2 in some cases (15.6 times in `tree-count`, 14.4 times in `flatten`, 12.8 times in `sets`).

Naturally, the number of stack references in a program is much higher than the number of heap references; most of the time, a variable is referenced only by the lambda that binds it. Thus, CFA2 uses the precise stack lookups more often than the imprecise heap lookups.

7 Related work

We were particularly influenced by Chaudhuri's paper on subcubic algorithms for recursive state machines [4]. His clear and intuitive description of summarization helped us realize that we can use it to explore the state space of CFA2.

Reps et al. [3] used summarization to reduce certain data-flow problems for first-order languages to a graph-reachability problem. Our workset algorithm is a variant of their tabulation algorithm, extended for tail recursion and higher-order functions. The reader may have noticed that CFA2 essentially produces a pushdown system. Then, one may wonder why we designed a new algorithm instead of using an existing one like $post^*$ [6, ch. 3]. The reason is that callers cannot be identified syntactically in higher-order languages. Hence, algorithms that analyze higher-order programs must be based on search. The tabulation algorithm can be changed to use search fairly naturally. It is unclear to us how to do that for $post^*$. In a way, CFA2 creates a pushdown system and analyzes it *at the same time*, much like what k-CFA does with control-flow graphs.

Melski and Reps [19] reduced Heintze's set-constraints [20] to an instance of CFL reachability, which they solve using summarization. Therefore, their solution has the same precision as 0CFA.

CFL reachability has also been used for points-to analysis of imperative higher-order languages. For instance, Sridharan and Bodík's points-to analysis for Java [21] uses CFL reachability to match writes and reads to object fields. Precise call/return matching is achieved only for programs without recursive methods. Hind's survey [22] discusses many other variants of points-to analysis.

Debray and Proebsting [23] used ideas from parsing theory to design an interprocedural analysis for first-order programs with tail calls. They describe control-flow with a context-free grammar. Then, the FOLLOW set of a procedure represents its possible return points. Our approach is different on the surface, but similar in spirit; we handle tail calls by computing summaries transitively.

Analyses that match an unbounded number of calls and returns have been neglected by the functional language community. The type-based flow analysis of Rehof and Fähndrich [7] is a notable exception. They encode flow information in a type system and then recast the type inference problem to an instance of CFL reachability. The type system uses let-polymorphism. As a result, it provides precise call/return matching for let- and letrec-bound variables but not for lambda-bound variables. For instance, if we lambda-bind `id` in our earlier example, their type system will not find `n2` to be constant:

```
((λ(id) (let ((n1 (id 1))
              (n2 (id 2)))
          (+ n1 n2)))
 (λ(x) x))
```

CFA2 does not distinguish between let and lambda; in fact, the AST of Twobit contains no lets.

Midtgaard and Jensen [24] created a flow analysis for direct-style higher-order programs that keeps track of "return flow". They point out that continuations

make return-point information explicit in CPS and show how to recover this information in direct-style. They do not address unbounded call/return matching.

Might and Shivers [8] proposed ΓCFA (abstract garbage collection) and μCFA (abstract counting) to increase precision in k-CFA. ΓCFA removes unreachable bindings from the variable environment; μCFA counts how many times a variable is bound during the analysis. The two techniques combined reduce spurious flows and improve environment information. Stack references in CFA2 have a similar effect, because different calls to the same function use different frames. However, we can use ΓCFA and μCFA to improve precision in the heap.

Recently, Kobayashi [25] proposed a way to statically verify properties of typed higher-order programs using model-checking. 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. The target language of this work is the simply-typed lambda calculus. Programs in a Turing-complete language must be approximated in the simply-typed lambda calculus in order to be model-checked.

8 Conclusions

In this paper we propose CFA2, a pushdown model of higher-order programs, and prove it correct. CFA2 provides precise call/return matching and has a better approach to variable binding than k-CFA. Our evaluation shows that CFA2 gives more accurate data-flow information than 0CFA and 1CFA.

CFA2 is monovariant in the heap. It can be easily extended with call-strings polyvariance, like k-CFA, to produce a family of analyses CFA2.0, CFA2.1 and so on. Then, any instance of CFA2.k would be strictly more precise than the corresponding instance of k-CFA. Another possibility is to add contours in the style of Agesen [26] or Wright and Jagannathan [9]. Note that CFA2 already has most of the above polyvariance "accidentally", because of the stack lookups.

We believe that pushdown models are a better tool for higher-order flow analysis than control-flow graphs, and are working on providing more empirical support to this thesis. We plan to use CFA2 for environment analysis and stack-related optimizations. We also plan to add support for `call/cc` in CFA2.

Acknowledgements. Thanks to Will Clinger and Felix Klock for help with Twobit, and to Manuel Fähndrich and Naoki Kobayashi for clarifications on their work. Comments from Bryan Chadwick, Matthias Felleisen, Felix Klock, Aaron Turon and the anonymous referees greatly improved the paper.

References

1. Shivers, O.: Control-Flow Analysis of Higher-Order Languages. PhD thesis, Carnegie-Mellon University (1991)
2. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Program Flow Analysis, Theory and Application. Prentice Hall (1981)

3. Reps, T.W., Horwitz, S., Sagiv, S.: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: Principles of Programming Languages. (1995) 49–61
4. Chaudhuri, S.: Subcubic Algorithms for Recursive State Machines. In: Principles of Programming Languages. (2008) 159–169
5. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T.W., Yannakakis, M.: Analysis of Recursive State Machines. Transactions on Programming Languages and Systems **27**(4) (2005) 786–818
6. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis, Technische Universität München (2002)
7. Rehof, J., Fähndrich, M.: Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In: Principles of Programming Languages. (2001) 54–66
8. Might, M., Shivers, O.: Improving Flow Analyses via ΓCFA: Abstract Garbage Collection and Counting. In: International Conference on Functional Programming. (2006) 13–25
9. Wright, A., Jagannathan, S.: Polymorphic Splitting: An Effective Polyvariant Flow Analysis. Trans. on Programming Languages and Systems **20**(1) (1998) 166–207
10. Van Horn, D., Mairson, H.G.: Deciding k-CFA is complete for EXPTIME. In: International Conference on Functional Programming. (2008) 275–282
11. Clinger, W.D., Hansen, L.T.: Lambda, the Ultimate Label or a Simple Optimizing Compiler for Scheme. In: LISP and Functional Programming. (1994) 128–139
12. Might, M.: Environment Analysis of Higher-Order Languages. PhD thesis, Georgia Institute of Technology (2007)
13. Steele, G.L.: Rabbit: A Compiler for Scheme. Master's thesis, MIT (1978)
14. Kranz, D.: ORBIT: An Optimizing Compiler for Scheme. PhD thesis, Yale University (1988)
15. Sabry, A., Felleisen, M.: Reasoning About Programs in Continuation-Passing Style. In: LISP and Functional Programming. (1992) 288–298
16. Shivers, O.: Higher-Order Control-Flow Analysis in Retrospect: Lessons Learned, Lessons Abandoned. In: Best of PLDI. (2004) 257–269
17. Vardoulakis, D., Shivers, O.: CFA2: a Context-Free Approach to Control-Flow Analysis. Technical Report NU-CCIS-10-01, Northeastern University (2010)
18. Yannakakis, M.: Graph-Theoretic Methods in Database Theory. In: Principles of Database Systems. (1990) 230–242
19. Melski, D., Reps, T.: Interconvertibility of a Class of Set Constraints and Context-Free-Language Reachability. Theoretical Comp. Sci. **248**(1-2) (2000) 29–98
20. Heintze, N.: Set-based program analysis. PhD thesis, Carnegie-Mellon Univ. (1992)
21. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Programming Language Design and Implementation. (2006) 387–400
22. Hind, M.: Pointer analysis: haven't we solved this problem yet? In: Program Analysis For Software Tools and Engineering. (2001) 54–61
23. Debray, S.K., Proebsting, T.A.: Interprocedural Control Flow Analysis of First-Order Programs with Tail-Call Optimization. Transactions on Programming Languages and Systems **19**(4) (1997) 568–585
24. Midtgaard, J., Jensen, T.: Control-flow analysis of function calls and returns by abstract interpretation. In: International Conference on Functional Programming. (2009) 287–298
25. Kobayashi, N.: Types and higher-order recursion schemes for verification of higher-order programs. In: Principles of Programming Languages. (2009) 416–428
26. Agesen, O.: The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In: European Conference on Object-Oriented Programming. (1995) 2–26