# Context-Sensitive Demand-Driven Control-Flow Analysis

ANONYMOUS AUTHOR(S)

By decoupling and decomposing control flows, demand control-flow analysis (CFA) is able to resolve only those segments of flows it determines necessary to resolve a given query. Thus, it presents a much more flexible interface and pricing model for CFA, making many useful applications practical. At present, the only realization of demand CFA is demand 0CFA, which is context-insensitive. This paper presents a context-sensitive demand CFA hierarchy, Demand $m$-CFA, based on the top-$m$-stack-frames abstraction of $m$-CFA. We evaluate the scalability of Demand $m$-CFA in contrast to the scalability of $m$-CFA and find that Demand $m$-CFA resolves many non-trivial control flows in constant time regardless of program size, which make it what we term a *demand-scalable* analysis. We also show that in the case of singleton flow sets, Demand $m$-CFA resolves a similar number of singleton flow sets as an *exponential* formulation of $m$-CFA, but in constant time.

**ACM Reference Format:**
Anonymous Author(s). 2024. Context-Sensitive Demand-Driven Control-Flow Analysis. In *Proceedings of International Conference on Functional Programming (ICFP '24).* ACM, New York, NY, USA, 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 GETTING INTO THE FLOW

Conventional control-flow analysis is tactless—unthinking and inconsiderate.

To illustrate, consider the program fragment on the right which defines the recursive `fold` function. As this function iterates, it evolves the index `n` using the function `f` and the accumulator `a` using the function `g`, all arguments to `fold` itself. The values of `f` and `g` flow in parallel within the fold itself, each (1) being bound in the initial call, (2) flowing to its corresponding parameter, and (3) being called directly once per iteration. But their flows don't completely overlap: `f`'s value's flow begins at `sub1` whereas `g`'s value's comes from a reference to `h` and `f`'s value's flow branches into the call to `g`.

```
(letrec ([fold (λ (f g n a)
                 (if (zero? n)
                   a
                   (fold f g (f n) (g f n a))))])
  (fold sub1 h 42 1))
```

Now consider a tool focused on the call `(f n)` and seeking the value of `f` in order to, say, expand `f` inline. Only the three flow segments identified above respective to `f` are needed to fully determine this value—and know that it is fully-determined. Yet conventional control-flow analysis (CFA) is *exhaustive*, insistent on determining every segment of every flow, starting from the program's entry point.[1] In the account it produces, the segmentation of individual flows and independence of

[1]Exhaustive CFA can be made to work with program components where free variables are treated specially (e.g. using Shivers' escape technique [Shivers 1991, Ch. 3]). This special treatment does not change the fundamental *exhaustive* nature of the analysis nor bridge the shortcomings we describe.


*ICFP '24, Sept 2024, Milan, Italy*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

distinct flows are completely implicit. To obtain `f`'s value with a conventional CFA, the user must be willing to pay for `g`'s—and any other values incidental to it—as well.

Inspired by demand dataflow analysis [Duesterwald et al. 1997], a *demand* CFA does not determine every segment of every flow but only those segments which contribute to the values of specified program points. Moreover, because its segmentation of flows is explicit, it only need analyze each segment once and can reuse the result in any flow which contains the segment. In this example, a supporting demand CFA would work backwards from the reference to `f` to determine its value, and would consider only the three flow segments identified above to do so.

### 1.1 Practicality and Scalability of Control Flow Analyses

The interface and pricing model exhibited by demand CFA make many useful applications practical. Horwitz et al. [1995] identify several ways this practicality is realized: First, one can restrict analysis to a program's critical path or only where certain features are used. Second, one can analyze more often, and interleave analysis with other tools. For example, a demand analysis does not need to worry about optimizing transformations invalidating analysis results since one can simply re-analyze the transformed points. Finally, one can let a user drive the analysis, even interactively, to enhance, e.g., an IDE experience.

We claim that demand CFA is what we term a *demand-scalable* analysis. We characterize such analyses as (1) being able to answer many relevant questions about a program in constant time or effort, and (2) being robust to increases in program size. *Demand-scalable* analyses are focused on the information gleaned from the analysis regardless of the underlying computational complexity, and opt for the usage of timeouts or early stopping criteria to keep the analysis practical. Additionally, we theorize that *demand-scalable* analyses are much better suited than monolithic analyses for integration in modern compilers which typically involve incremental recompilation, language servers, linters, debuggers and other tools, which can each benefit from additional semantic information.

### 1.2 Towards Context Sensitivity

Presently, the only realization of demand CFA is Demand 0CFA [Germane et al. 2019] which is context-*insensitive*. (We offer some intuition about Demand 0CFA's operation in § 2 and review it in § 4.) However, the benefits to a context-*sensitive* demand CFA are clear. A demand CFA would enjoy increased precision and also, in some cases, a reduced workload (which we discuss at an intuitive level in § 2). The primary difficulty is to determine control flow about an arbitrary program point, in an arbitrary context. Thus, the task of a context-sensitive demand CFA is not only to correctly maintain the context but to *discover* the contexts in which evaluation occurs. To achieve this requires a compatible choice of context, context representation, and even environment representation, as we discuss in § 5.

After surmounting these issues, we arrive at Demand $m$-CFA (§ 6), a hierarchy of demand CFA that exhibits context sensitivity. At a high level, Demand $m$-CFA achieves context sensitivity by permitting indeterminate contexts, which stand for any context, and instantiating them when further information is discovered. It then uses instantiated contexts to filter its resolution of control flow to ensure that its view of evaluation remains consistent with respect to context. (We offer intuition about these operations in § 2 as well.) Demand $m$-CFA is sound with respect to a concrete albeit demand-driven semantics called *demand evaluation* (§ 7), which is itself sound with respect to a standard call-by-value semantics.

Demand $m$-CFA is comprehensive in the sense that it discovers all contexts to the extent necessary for evaluation. It achieves this by carefully ensuring at certain points that it proceeds only when the context is known, even if it is not strictly necessary to produce a value. We find this disposition

toward analysis fairly effective: in some cases, it produces effectively-exhaustive, identically-precise results as an exhaustive analysis at the same level of context sensitivity but *at a constant price*.

Although Demand $m$-CFA requires a fair amount of technical machinery to formulate, its implementation is very straightforward using the *Abstracting Definitional Interpreters* (ADI) technique [Darais et al. 2017]. To illustrate its directness, we reproduce and discuss the core of Demand $m$-CFA's implementation in § 8. One virtue of using the ADI approach is that it endows the implemented analyzer with "pushdown precision" with respect to the reference semantics—which, for our analyzer, are the demand semantics. However, as we discuss in § 8, Demand $m$-CFA satisfies the *Pushdown for Free* criteria [Gilray et al. 2016] which ensures that it has pushdown precision with respect to the direct semantics as well.

This paper makes the following contributions:

- a new formalism for Demand 0CFA which can be implemented straightforwardly using contemporary techniques [Darais et al. 2017; Wei et al. 2018] (§ 4);
- Demand $m$-CFA (§ 6), a hierarchy of context-sensitive demand CFA and a proof of its soundness (§ 7);
- an empirical evaluation of the scalability and precision of Demand $m$-CFA(§ 9);

## 2 DEMAND CFA, INTUITIVELY

A user interacts with demand CFA by submitting queries, which the analyzer resolves online. There are two types of queries: An *evaluation* query yields the values to which a specified expression may evaluate. A *trace* query yields the sites at which the value of a specified expression may be called. Some queries are trivially resolved, such as those for the value of a constructor application. Typically, however, the control flow within a query depends on adjacent flows, for which the analyzer issues subqueries.

We illustrate the operation of a demand CFA considering queries over the program

```
(let ([f (λ (x) x)]) (+ (f 42) (f 35)))
```

which is written in an applicative functional language with Lisp syntax.

### 2.1 Without Context Sensitivity

```
q0  evaluate (f 35)
q1      evaluate f
q2      evaluate (λ (x) x)
        ⇒ (λ (x) x)
q3  evaluate x
q4      trace (λ (x) x)
q5      trace f in (f 42)
        ⇒ (f 42)
q7  evaluate 42
    ⇒ 42
q6      trace f in (f 35)
        ⇒ (f 35)
q8  evaluate 35
    ⇒ 35
```

As many readers are likely unfamiliar with demand CFA, we'll first look at how demand 0CFA, the context-*in*sensitive embodiment of demand CFA, resolves queries.

Suppose that a user submits an evaluation query $q_0$ on the expression `(f 35)`. Since `(f 35)` is a function application, demand 0CFA issues a subquery $q_1$ to evaluate the operator `f`. For each procedure value of `f`, demand 0CFA will issue a subquery to determine the value of its body as the value of $q_0$. To the left is a trace of the queries that follow $q_0$. Indented queries denote subqueries whose results are used to continue resolution of the superquery. A subsequent query at the same indentation level is a query in "tail position", whose results are those of a preceding query. A query often issues multiple queries in tail position, as this example demonstrates. The operator `f` is a reference, so demand 0CFA walks the syntax to find where `f` is bound. Upon finding it bound by a `let` expression, demand 0CFA issues a subquery $q_2$ to evaluate its bound expression `(λ (x) x)`. The expression `(λ (x) x)` is a $\lambda$ term—a value—which $q_2$ propagates directly to $q_1$. Once $q_1$ receives it, demand 0CFA issues a subquery $q_3$ for the

evaluation of its body. Its body `x` is a reference, so demand 0CFA walks the syntax to discover that it is $\lambda$-bound and therefore that its value is the value of the argument at the application of `(λ (x) x)`. That this call to `(λ (x) x)` originated at `(f 35)` is contextual information, to which demand 0CFA is insensitive. Consequently, demand 0CFA issues a trace query $q_4$ to find all the application sites of `(λ (x) x)`. Because it is an expression bound to `f`, demand 0CFA issues a subqueries $q_5$ and $q_6$ to find the use sites of both references to $f$. Each of these subqueries resolves immediately since each of the references is in operator position and their results are propagated to $q_4$. For each result, $q_3$ issues a subquery—$q_7$ and $q_8$—to evaluate the arguments, each of which is a numeric literal, whose value is immediately known. Each query propagates its results to $q_3$ which propagates them to $q_0$ which returns them to the user. Thus, demand 0CFA concludes that `(f 35)` may evaluate to 42 or 35.

## 2.2 With Context Sensitivity

We'll now look at how Demand $m$-CFA, a context-sensitive demand CFA, resolves queries. As is typical, Demand $m$-CFA uses an environment to record the binding context of each in-scope variable. Hence, in this setting, queries and results include not only expressions, but environments as well. We will also see that Demand $m$-CFA does not need a timestamp to record the "current" context, a fact we discuss further in § 5.5.

Let's consider the same evaluation query $q_0$ over `(f 35)`, this time in the top-level environment $\langle\rangle$. Like Demand 0CFA, Demand $m$-CFA issues the subquery $q_1$ to determine the operator `f`, also in $\langle\rangle$. After it discovers `(λ (x) x)` to be the binding expression of `f`, it issues an evaluation query over it ($q_2$) again in $\langle\rangle$. The result of $q_2$ is `(λ (x) x)` in $\langle\rangle$, essentially a closure. As before, this result is passed first to $q_1$ and then to $q_0$ at which point Demand $m$-CFA constructs $q_3$, an evaluation query over its body. The query's environment $\langle$`(f 35)`$\rangle$ records the context in which the parameter `x` was bound. In order to evaluate `x`, Demand $m$-CFA issues a *caller* query $q_3'$ to determine the caller of `(λ (x) x)` that yielded the environment $\langle$`(f 35)`$\rangle$. It then issues the trace query $q_4$, this time subordinate to $q_3'$, which issues $q_5$ and $q_6$ and results in the same two applications of `f`. However, when $q_3'$ receives a caller from $q_4$, Demand $m$-CFA ensures that the caller could produce the binding context of the parameter in $q_3'$'s environment. If so, $q_3'$ yields the result to $q_3$; if not, it cuts off the resolution process for that path. In this case, $q_5$'s result `(f 42)` is not compatible with $q_3'$, and Demand $m$-CFA ceases resolving it rather than issuing $q_7$. However, $q_6$'s result `(f 35)` is compatible, and its resolution continues, issuing $q_8$. Resolution of $q_8$ occurs immediately and its result is propagated to the top-level query.

```
q0  evaluate (f 35) in ⟨⟩
q1      evaluate f in ⟨⟩
q2      evaluate (λ (x) x) in ⟨⟩
        ⇒ (λ (x) x) in ⟨⟩
q3  evaluate x in ⟨(f 35)⟩
q3'     caller x in ⟨(f 35)⟩
q4          trace (λ (x) x) in ⟨⟩
q5          trace f in ⟨⟩
            ⇒ (f 42) in ⟨⟩
        ⇒ fail
q6          trace f in ⟨⟩
            ⇒ (f 35) in ⟨⟩
        ⇒ (f 35) in ⟨⟩
q8  evaluate 35 in ⟨⟩
    ⇒ 35 in ⟨⟩
```

This example illustrates how Demand $m$-CFA uses the context information recorded in the environment to filter out discovered callers of a particular closure. Not only does this filtering increase precision in the expected way, but, in this example, it also prevents Demand $m$-CFA from issuing a spurious query ($q_7$). This behavior is an example of the well-known phenomenon of high precision keeping the analyzer's search space small [Might and Shivers 2006].

### 2.3 ...And Indeterminacy

Each environment in the previous section was fully determined. Typically, Demand $m$-CFA resolves queries and produces results with environments that are—at least partially—indeterminate. For instance, to obtain all of the values to which `x`, the body of `(λ (x) x)`, may evaluate, a user may issue the query evaluate `x` in $\langle ? \rangle$ where ? is a "wildcard" context to be instantiated with each context the analyzer discovers. (Though each context in the environment is indeterminate, the shape of the environment itself is determined by the lexical binding structure, which we discuss further in § 5.3.)

$q_0$ evaluate `x` in $\langle ? \rangle$
$q_0'$ caller `x` in $\langle ? \rangle$
$q_1$ trace `(λ (x) x)` in $\langle \rangle$
$q_2$ trace `f` in `(f 42)` in $\langle \rangle$
$\Rightarrow$ `(f 42)` in $\langle \rangle$
$\Rightarrow$ `(f 42)` in $\langle \rangle$
$q_4$ evaluate `x` in $\langle$`(f 42)`$\rangle$
$q_5$ evaluate `42` in $\langle \rangle$
$\Rightarrow$ `42` in $\langle \rangle$
$q_3$ trace `f` in `(f 35)`
$\Rightarrow$ `(f 35)`
$q_6$ evaluate `x` in $\langle$`(f 35)`$\rangle$
$q_7$ evaluate `35` in $\langle \rangle$
$\Rightarrow$ `35` in $\langle \rangle$

Once issued, resolution of `x`'s evaluation again depends on a caller query $q_0'$. However, because the parameter `x`'s context is unknown, rather than filtering out callers, the caller query will cause ? to be instantiated with a context derived from each caller. As before, Demand $m$-CFA dispatches a trace query $q_1$ which then traces occurrences of `f` via $q_2$ and $q_3$. This query locates the call sites `(f 42)` in $\langle \rangle$ and `(f 35)` in $\langle \rangle$ Once $q_2$ delivers the result `(f 42)` in $\langle \rangle$ to $q_1$ and then $q_0'$, Demand $m$-CFA *instantiates* $q_0$ with this newly-discovered caller to form $q_4$, whose result is $q_0$'s also. After creating $q_3$, it continues with its resolution by issuing $q_4$ to evaluate the argument `42` in $\langle \rangle$. Its result of 42 propagates from $q_4$ to $q_3$ to $q_0$; from $q_0$, one can see all instantiations of it as well every result of those instantiations. The instantiation from $q_3$ proceeds similarly.

## 3 LANGUAGE AND NOTATION

In order to keep otherwise-identical expressions distinct, many presentations of CFA uniquely label program sub-expressions.[2] This approach would be used, for example, to disambiguate the two references to `f` in the program in § 2. Demand $m$-CFA extensively consults the syntactic context of each expression, which uniquely determines it, and uses it as a de facto label.

The syntactic context $C$ of an instance of an expression $e$ within a program $pr$ is $pr$ itself with a hole $\square$ in place of the selected instance of $e$. For example, the program $\lambda x.(x\, x)$ contains two references to $x$, one with syntactic context $\lambda x.(\square\, x)$ and the other with $\lambda x.(x\, \square)$.

In the unary $\lambda$ calculus, expressions $e$ adhere to the grammar on the left and syntactic contexts $C$ adhere to the grammar on the right.

$$e ::= x \mid \lambda x.e \mid (e\, e) \qquad\qquad C ::= \lambda x.C \mid (C\, e) \mid (e\, C) \mid \square.$$

The composition $C[e]$ of a syntactic context $C$ with an expression $e$ consists of $C$ with $\square$ replaced by $e$. In other words, $C[e]$ denotes the program itself but with a focus on $e$. For example, we focus on the reference to $x$ in operator position in the program $\lambda x.(x\, x)$ with $\lambda x.([x]\, x)$.

We typically leave the context unspecified, referring to, e.g., a reference to $x$ by $C[x]$ and two distinct references to $x$ by $C_0[x]$ and $C_1[x]$ (where $C_0 \neq C_1$). The immediate syntactic context of an expression is often relevant, however, and we make it explicit by a double composition $C_0[C_1[e]]$. For example, we use $C[([x]\, x)]$ to focus on the expression $x$ in the operator context $(\square\, x)$ in the context $C$.

[2] Others operate over a form which itself names all intermediate results, such as CPS or $\mathcal{A}$-normal form, and identify each expression by its associated (unique) name.

$$\text{bind} : \mathit{Var} \times \mathit{Exp} \to \mathit{Exp}$$

$$\begin{aligned}
\text{bind}(x, C[([e_0]\ e_1)]) &= \text{bind}(x, C[(e_0\ e_1)]) \\
\text{bind}(x, C[(e_0\ [e_1])]) &= \text{bind}(x, C[(e_0\ e_1)]) \\
\text{bind}(x, C[\lambda y.[e]]) &= \text{bind}(x, C[\lambda y.e]) \text{ where } y \neq x \\
\text{bind}(x, C[\lambda x.[e]]) &= C[\lambda x.[e]]
\end{aligned}$$

Fig. 1. The bind metafunction

## 4 DEMAND 0CFA

Demand 0CFA has two modes of operation, *evaluation* and *tracing*, which users access by submitting evaluation or trace queries, respectively. A query, in addition to its type, designates a program expression over which the query should be resolved. An evaluation query resolves the values to which the designated expression may evaluate and a trace query resolves the sites which may apply the value of the designated expression. These modes are essentially dual and reflect the dual perspective of exhaustive CFA as either (1) the $\lambda x.e$ which may be applied at a given site $(e_0\ e_1)$, or (2) the $(e_0\ e_1)$ at which a given $\lambda x.e$ may be applied. However, in contrast to exhaustive CFA, demand 0CFA is designed to resolve evaluation queries over arbitrary program expressions. (It is also able to resolve trace queries over arbitrary program expressions, but exhaustive CFAs have no counterpart to this functionality.) The evaluation and trace modes of operation are effected by the big-step relations $\Downarrow_{eval}$ and $\Rightarrow_{expr}$, respectively, which are defined mutually inductively. These relations are supported by auxiliary relations $\Downarrow_{call}$ and $\Rightarrow_{find}$. Figure 2 presents the definitions of all of these relations.

The judgement $C[e] \Downarrow_{eval} C_v[\lambda x.e_v]$ denotes that the expression $e$ (residing in syntactic context $C$) evaluates to (a closure over) $\lambda x.e_v$. (In a context-insensitive analysis, we may represent a closure by the $\lambda$ term itself.) Demand 0CFA arrives at such a judgement, as an interpreter does, by considering the type of expression being evaluated. The *Lam* rule captures the intuition that a $\lambda$ term immediately evaluates to itself. The *App* rule captures the intuition that an application evaluates to whatever the body of its operator does. Hence, if the operator $e_0$ evaluates to $\lambda x.e$, and $e$ evaluates to $\lambda y.e_v$, then the application $(e_0\ e_1)$ evaluates to $\lambda y.e_v$ as well. Notice that the *App* does not evaluate the argument; if the argument is needed, indicated by a reference to the operator's parameter $x$ during evaluation of its body, the *Ref* rule obtains it. The *Ref* rule captures the intuition that a reference to a parameter $x$ takes on the value of the argument at each call site where the $\lambda$ which binds $x$ is called. The bind metafunction determines the binding configuration of $x$ by walking outward on the syntax tree until it encounters $x$'s binder. Figure 1 presents its definition, and we note the absence of a rule for the case where $x$ is unbound, since we define programs as closed expressions.

A judgement $C[\lambda x.[e]] \Downarrow_{call} C'[(e_0\ e_1)]$ denotes that the application $(e_0\ e_1)$ applies $\lambda x.e$, thereby binding $x$. Demand 0CFA arrives at this judgment by the *Call* rule which uses the $\Rightarrow_{expr}$ relation to determine it. In demand 0CFA, this relation is only a thin wrapper over $\Rightarrow_{expr}$, but it becomes more involved in context-sensitive demand CFA. We include it here for consistency.

A judgement $C[e] \Rightarrow_{expr} C'[(e_0\ e_1)]$ denotes that the value of the expression $e$ is applied at $(e_0\ e_1)$. Demand 0CFA arrives at such a judgement by considering the type of the syntactic context to which the value flows. The *Rator* rule captures the intuition that, if $\lambda x.e$ flows to ope*rator* position $e_0$ of $(e_0\ e_1)$, it is applied by $(e_0\ e_1)$. The *Body* rule captures the intuition that if a value flows to the body of a $\lambda$ term, then it flows to each of its callers as well. The *Rand* rule captures the intuition that a value in ope*rand* position is bound by the formal parameter of each operator value and hence

$$\textsc{Lam}\ \frac{}{C[\lambda x.e] \Downarrow_{eval} C[\lambda x.e]} \qquad \textsc{App}\ \frac{C[([e_0]\ e_1)] \Downarrow_{eval} C'[\lambda x.e] \qquad C'[\lambda x.[e]] \Downarrow_{eval} C_v[\lambda x.e_v]}{C[(e_0\ e_1)] \Downarrow_{eval} C_v[\lambda x.e_v]}$$

$$\textsc{Ref}\ \frac{C'[e_x] = \mathsf{bind}(x, C[x]) \qquad C'[e_x] \Downarrow_{call} C''[(e_0\ e_1)] \qquad C''[(e_0\ [e_1])] \Downarrow_{eval} C_v[\lambda x.e]}{C[x] \Downarrow_{eval} C_v[\lambda x.e]}$$

$$\textsc{Rator}\ \frac{}{C[([e_0]\ e_1)] \Rightarrow_{expr} C[(e_0\ e_1)]} \qquad \textsc{Body}\ \frac{C[\lambda x.[e]] \Downarrow_{call} C'[(e_0\ e_1)] \qquad C'[(e_0\ e_1)] \Rightarrow_{expr} C''[(e_2\ e_3)]}{C[\lambda x.[e]] \Rightarrow_{expr} C''[(e_2\ e_3)]}$$

$$\textsc{Rand}\ \frac{C[([e_0]\ e_1)] \Downarrow_{eval} C'[\lambda x.e] \qquad x, C'[\lambda x.[e]] \Rightarrow_{find} C_x[x] \qquad C_x[x] \Rightarrow_{expr} C'[(e_2\ e_3)]}{C[(e_0\ [e_1])] \Rightarrow_{expr} C'[(e_2\ e_3)]}$$

$$\textsc{Call}\ \frac{C[\lambda x.e] \Rightarrow_{expr} C'[(e_0\ e_1)]}{C[\lambda x.[e]] \Downarrow_{call} C'[(e_0\ e_1)]} \qquad \textsc{Find-Ref}\ \frac{}{x, C[x] \Rightarrow_{find} C[x]} \qquad \textsc{Find-Rator}\ \frac{x, C[([e_0]\ e_1)] \Rightarrow_{find} C_x[x]}{x, C[(e_0\ e_1)] \Rightarrow_{find} C_x[x]}$$

$$\textsc{Find-Rand}\ \frac{x, C[(e_0\ [e_1])] \Rightarrow_{find} C_x[x]}{x, C[(e_0\ e_1)] \Rightarrow_{find} C_x[x]} \qquad \textsc{Find-Body}\ \frac{x \neq y \qquad x, C[\lambda y.[e]] \Rightarrow_{find} C_x[x]}{x, C[\lambda y.[e]] \Rightarrow_{find} C_x[x]}$$

Fig. 2. Demand 0CFA Relations

to each reference to the formal parameter in the operator's body. If the operator $e_f$ evaluates to $\lambda x.e$, then the value of $e_a$ flows to each reference to $x$ in $e$.

The $\Rightarrow_{find}$ relation associates a variable $x$ and expression $e$ with each reference to $x$ in $e$. *Find-Ref* finds $e$ itself if $e$ is a reference to $x$. *Find-Rator* and *Find-Rand* find references to $x$ in $(e_0\ e_1)$ by searching the ope*rator* $e_0$ and ope*rand* $e_1$, respectively. *Find-Body* finds references to $x$ in $\lambda x.e$ taking care that $x \neq y$ so that it does not find shadowed references.

### 4.1 Reachability

All but the most naïve exhaustive CFAs compute reachability at the same time as control flow. For instance, when analyzing the program $(\lambda x.\lambda y.x\ \lambda z.z)$, such CFAs do not evaluate the reference $x$ as it occurs in $\lambda y.x$ which is never applied.

Demand 0CFA, however, considers reachability not for the sake of control but for data. In this example, the caller of $\lambda y.x$ is not needed for evaluation of $x$, so demand 0CFA remains oblivious to the fact that $\lambda y.x$ is never called. If, however, the reference $x$ were replaced with $y$ so that the program was $(\lambda x.\lambda y.y\ \lambda z.z)$, evaluation of $y$ would depend on the caller of $\lambda y.y$. Unable to find a caller in this case, demand 0CFA would report that $y$ obtains no value.

By ignoring control that does not influence the sought-after data, Demand 0CFA avoids exploring each path which transported the data, instead relying on the discipline of lexical scope to correspond binding and use. This policy does mean that Demand 0CFA sometimes analyzes dead code as if it were live. From a practical standpoint, this is harmless, since any conclusion about genuinely dead code is vacuously true. However, it is possible for Demand 0CFA to include, e.g., dead references

in its trace of a value via a binding, which potentially compromises precision. We empirically investigate the extent to which precision is compromised in § 9.

## 5 ADDING CONTEXT SENSITIVITY

A context-*insensitive* CFA is characterized by each program variable having a single entry in the store, shared by all bindings to it. A context-sensitive CFA considers the context in which each variable is bound and requires only bindings made in the same context to share a store entry. By extension, a context-sensitive CFA evaluates an expression under a particular environment, which identifies the context in which free variables within the expression are bound. Together, an expression and its enclosing environment constitute a *configuration*.[3] In order to introduce context sensitivity to demand 0CFA, we extend $\Downarrow_{eval}$, $\Rightarrow_{expr}$, and $\Downarrow_{call}$ to relate not just expressions to expressions, but configurations to configurations. (Like Demand 0CFA, our context-sensitive demand CFA will not materialize the store in the semantics, but it can be recovered if desired.)

However, demand CFA differs from exhaustive CFA in that the precise environment in which an expression is evaluated or traced may not be completely determined. That is, the context in which each environment variable is bound may not be fully known. For instance, users typically want control-flow information about an expression which is sound with respect to all its evaluations and so indicate in queries by leaving the environment completely indeterminate. In this case, demand CFA should instantiate the environment (only) as necessary to distinguish evaluations in different contexts. Demand CFA thus requires a choice of context, context representation, and environment representation which support its ability to do so. In this section, we examine each of these choices in turn.

### 5.1 Choosing the context

To formulate context-sensitive demand CFA in the most general setting possible, we will avoid sensitivities to properties not present in our semantics, such as types. Since we focus on an untyped $\lambda$ calculus, the most straightforward choice is call-site sensitivity.

The canonical call-site sensitivity is that of $k$-CFA [Shivers 1991] which sensitizes each binding to the last $k$ call sites encountered in evaluation. However, this form of call-site sensitivity works against the parsimonious nature of demand CFA. To make this concrete, consider that, in the fragment `(begin (f x) (g y))`, the binding of `g`'s parameter under a last $k$ call-site sensitivity will depend on the particular calls made during the evaluation of `(f x)`. If the value of `(f x)` is not otherwise demanded, this dependence either provokes demand analysis to discover more of the context or requires that the portion of the context contributed by `(f x)` be left indeterminate, thereby sacrificing precision.

A more fitting call-site sensitivity would allow demand CFA to discover more of the context through its course of operation. A natural fit, it turns out, is $m$-CFA's call-site sensitivity which models the top-$m$ stack frames.

*5.1.1 m-CFA's context abstraction.* The $m$-CFA hierarchy [Might et al. 2010] is the result of an investigation into the polynomial character of $k$-CFA in an object-oriented setting versus its exponential character in a functional setting. The crucial discovery of that investigation was that OO-oriented $k$-CFA induces flat environments whereas functional-oriented $k$-CFA induces nested environments. Specifically, OO-oriented $k$-CFA re-binds object (closure) variables in the allocation context which collapses the exponential environment space to a polynomial size. From this insight they derive $m$-CFA, which simulates the OO binding discipline even when analyzing functional

[3] Configurations in exhaustive CFAs include a timestamp as well. We discuss its omission from demand CFA configurations in § 5.5.

programs; that is, when applied, $m$-CFA binds a closure's arguments and rebinds its environment's bindings all in the binding context of the application. Under this policy, within a given environment, all bindings are in the same context and, consequently, the analysis can represent that environment simply as that binding context.

However, this binding policy amplifies a weakness of the $k$-most-recent-call-sites abstraction of $k$-CFA. Consider a $[k = 2]$CFA analysis of a call to `f` defined by

```
(define (f x) (log "called f") (g x))
```

and suppose that `log` doesn't itself make any calls. When control reaches `(g x)`, the most-recent call is always `(log "called f")` so the binding context of `g`'s parameter, the most-recent two calls `(log "called f")` and `(g x)`, determine only one point in the program—the body of `f`. In $k$-CFA, only `g`'s parameter binding suffers this abbreviated precision; the bindings in `g`'s closure environment refer to the context in which they were introduced into it. In $m$-CFA, however, the bindings of `g`'s closure environment are re-bound in the context of `g`'s application as `g` is applied and once-distinct bindings are merged together in a semi-degenerate context.

To accommodate this binding policy, Might et al. [2010] use the top-$m$ stack frames as a binding context rather than the last $k$ call sites as the former is unaffected by static sequences of calls. Hence, when control reaches `(g x)` in $[m = 2]$-CFA analysis, the binding context is that of `f`'s entry and `g`'s parameter is bound in the context of `(g x)` and `f`'s caller—no context is wasted.

Because we're using $m$-CFA's top-$m$-stack-frames context abstraction, we call our context-sensitive demand CFA *Demand $m$-CFA*. It is important to keep in mind, however, that we do *not* adopt its re-binding policy, which is the essence of $m$-CFA.

### 5.2 Representing the top-$m$ stack frames

Now that we have identified a context abstraction, we must choose a representation for it which will allow us to model incomplete knowledge. One choice would be an $m$-length vector of possibly-indeterminate call sites which Demand $m$-CFA could fill in as it discovers contexts. However, this representation fails to capture a useful invariant.

To illustrate, suppose we are evaluating `x` in the function `(λ (x) x)` and that we have no knowledge about the binding context of `x`. In order to determine `x`'s value, we must determine the sites that call `(λ (x) x)`. If our analyzer determines that one such site is `(f 42)`, then it learns that the *top* frame—the first of the top-$m$ frames—is `(f 42)`. The next frame is whatever the top frame of `(f 42)`'s context is, which may be indeterminate. Thus, the analyzer's knowledge of the top-$m$ frames always increases from the top down. (This invariant holds when the analyzer enters a call as well, since it necessarily knows the call site when doing so.)

We devise a context representation that captures this invariant. A context $cc^m$ representing $m$ stack frames is either completely indeterminate, a pair of a call $C[(e_0\ e_1)]$ and a context $cc^{m-1}$ of $m - 1$ frames, or the stack bottom $()$ (which occurs at the top level of evaluation). A context $cc^0$ of zero frames (which is a different notion than the stack bottom) is simply the degenerate $\square$. Formally, we have

$$\mathit{Context}_0 \ni cc^0 ::= \square \qquad \mathit{Context}_m \ni cc^m ::= ? \mid (C[(e_0\ e_1)], cc^{m-1}) \mid ()$$

With the context representation chosen, we can now turn to the environment representation.

### 5.3 More-Orderly Environments

In a lexically-scoped language, the environment at the reference `x` in the fragment

```
(define f (λ (x y) ... (λ (z) ... (λ (w) ... x ...) ...) ...))
```

contains bindings for x, y, z, and w. Exhaustive CFAs typically model this environment as a finite map from variables to contexts (i.e., the type $Var \to Context$). For instance, $k$-CFA uses this model with $Binding = Contour$ where a $contour\ c \in Contour = Call^{\leq k}$ is the $k$-most-recent call sites encountered during evaluation (and $Call$ is the set of call sites in the analyzed program). Hence, $k$-CFA models the environment at the reference x as $[\mathtt{x} \mapsto c_0, \mathtt{y} \mapsto c_1, \mathtt{z} \mapsto c_2, \mathtt{w} \mapsto c_3]$ for some contours $c_0$, $c_1$, $c_2$, and $c_3$. $m$-CFA uses a similar representation, though it's contours represent the top-$m$ stack frames.

While this representation captures the structure necessary for $k$-CFA (or $m$-CFA), we observe that due to lexical-scoping we can use a variable's de Bruijn index, which is statically determined by the program syntax, and model environments as a sequence $Context^*$, splitting the environment by the context associated with each lexical set of variables. For instance, we model the environment at the reference x as $[c1, c2, c3]$ where $c1$ represents the m-stack frames that led to calling the outermost $\lambda$, $c2$ the m-stack frames for the middle $\lambda$, and 'c3' the m-stack frames for the innermost $\lambda$. This representation discards none of the environment structure of $k$-CFA and captures more of the structure inherent in evaluation, although the selection of contours differs in the same way as $m$-CFA.

Given this environment representation, we make one final tweak to the definition of contexts: we will qualify an indeterminate context ? with the parameter of the function whose context it represents, and assume programs are alphatized.[4] This way, an environment of even completely indeterminate contexts still determines the expression it closes. For instance, we represent the indeterminate environment of y in `(λ (x) ((λ (y) y) (λ (z) z)))` by $\langle ?_{\mathtt{y}}, ?_{\mathtt{x}} \rangle$ which is distinct from the indeterminate environment of z, which we represent by $\langle ?_{\mathtt{z}}, ?_{\mathtt{x}} \rangle$, even though they have the same shape.

### 5.4 Instantiating Contexts

Demand $m$-CFA, at certain points during resolution, discovers information about the context in which the resolved flow occurs, and must instantiate relevant environments with that information. For instance, in the program

```
(let ([apply (λ (f) (λ (x) (f x)))])
  (+ ((apply add1) 42)
     ((apply sub1) 35)))
```

suppose that Demand $m$-CFA is issued an evaluation query for `(f x)` in the environment $\langle ?_{\mathtt{x}}, ?_{\mathtt{f}} \rangle$, i.e., the fully-indeterminate environment. With our global view, we can see that `(f x)` evaluates to 43 and 34. Let's consider how Demand $m$-CFA would arrive at the same conclusion.

First, it would trace `(λ (f) (λ (x) (f x)))` in the environment $\langle\rangle$ to the binding `apply` and then to the call sites `(apply add1)` in $\langle\rangle$ and `(apply sub1)` in $\langle\rangle$. Each of these sites provides a context in which f is bound, so the evaluation of `(f x)` continues in two instantiations of $\langle ?_{\mathtt{x}}, ?_{\mathtt{f}} \rangle$: the first to $\langle ?_{\mathtt{x}}, (\mathtt{apply\ add1}) :: ?_{tl} \rangle$ and the second to $\langle ?_{\mathtt{x}}, (\mathtt{apply\ sub1}) :: ?_{tl} \rangle$, where $?_{tl}$ is a dummy variable for the top-level context (the stack bottom). To evaluate either add1 or sub1, its argument is needed, which flows through x. Then the two callers of `(λ (x) (f x))` must be resolved. When `(λ (x) (f x))` flows to the result of `(apply add1)`, it is applied immediately at `((apply add1) 42)`. Similarly, when it flows to the result of `(apply sub1)`, it is applied immediately at `((apply sub1) 35)`. These two call sites provide the binding contexts for x.

We might be tempted at this point to blindly instantiate $?_{\mathtt{x}}$ with each of these sites. However, in doing so, we will get *six* environments, instantiating $?_{\mathtt{x}}$ in each of $\langle ?_{\mathtt{x}}, ?_{\mathtt{f}} \rangle$, $\langle ?_{\mathtt{x}}, (\mathtt{apply\ add1}) :: ?_{tl} \rangle$,

[4] In practice, we use the syntactic context of the body instead of the parameter, which is unique even if the program is not alphatized.

$$\mathsf{bind} : \mathit{Var} \times \mathit{Exp} \times \mathit{Env} \to \mathit{Exp} \times \mathit{Env}$$

$$\begin{aligned}
\mathsf{bind}(x, C[([e_0]\ e_1)], \hat{\rho}) &= \mathsf{bind}(x, C[(e_0\ e_1)], \hat{\rho}) \\
\mathsf{bind}(x, C[(e_0\ [e_1])], \hat{\rho}) &= \mathsf{bind}(x, C[(e_0\ e_1)], \hat{\rho}) \\
\mathsf{bind}(x, C[\lambda y.[e]], \hat{cc} :: \hat{\rho}) &= \mathsf{bind}(x, C[\lambda y.e], \hat{\rho}) \text{ where } y \neq x \\
\mathsf{bind}(x, C[\lambda x.[e]], \hat{\rho}) &= (C[\lambda x.[e]], \hat{\rho})
\end{aligned}$$

Fig. 3. The bind metafunction

and $\langle ?_{\mathtt{x}}, (\mathtt{apply\ sub1}) ::?_{tl} \rangle$ with each of $((\mathtt{apply\ add1})\ 42) ::?_{tl}$ and $((\mathtt{apply\ sub1})\ 35) ::?_{tl}$. In particular, we obtain

$$\langle ((\mathtt{apply\ sub1})\ 35) ::?_{tl}, (\mathtt{apply\ add1}) ::?_{tl} \rangle \qquad \langle ((\mathtt{apply\ add1})\ 42) ::?_{tl}, (\mathtt{apply\ sub1}) ::?_{tl} \rangle$$

which do not correspond to any environment which arises in an exhaustive analysis.

The issue is that blindly instantiating indeterminate contexts ignores the evaluation path taken to arrive at the call site. In particular, it ignores where the nesting $\lambda$ is applied, which must be applied first (as we observed in the previous section). In this example, we must consider the context of `(λ (f) (λ (x) (f x)))` before we instantiate the context of `(λ (x) (f x))`.

The solution, then, is to not substitute a indeterminate context with a more-determined context, but an entire environment headed by an indeterminate context with that same environment headed by the more-determined one. This policy would, in this example, lead to all occurrences of

$$\langle ?_{\mathtt{x}}, (\mathtt{apply\ add1}) ::?_{tl} \rangle \quad \text{being substituted with} \quad \langle ((\mathtt{apply\ add1})\ 42) ::?_{tl}, (\mathtt{apply\ add1}) ::?_{tl} \rangle$$

and

$$\langle ?_{\mathtt{x}}, (\mathtt{apply\ sub1}) ::?_{tl} \rangle \quad \text{being substituted with} \quad \langle ((\mathtt{apply\ sub1})\ 35) ::?_{tl}, (\mathtt{apply\ sub1}) ::?_{tl} \rangle$$

and no others, which is precisely what we would hope.

This policy is effective even when the result of the function does not depend on both values. For instance, when Demand $m$-CFA evaluates `x` in `(λ (f) (λ (x) x))`, it must still determine the caller of `(λ (f) (λ (x) x))` to determine the downstream caller of `(λ (x) x)`.

### 5.5 Whence the timestamp?

In addition to introducing environments, context-sensitivity also typically introduces "timestamps" which serve as snapshots of the context at each evaluation step. For instance, classic $k$-CFA evaluates *configurations*, consisting of an expression, environment, and timestamp, to results.

With a top-$m$-stack-frames abstraction, the "current context" is simply the context of the variable(s) most recently bound in the environment. Lexical scope makes identifying these variables easy as does our representation of the environment as a sequence of binding contexts for the context itself. In other words, with such an abstraction, the environment uniquely determines the timestamp, and our configurations consisting of an expression paired with its environment can be viewed to include the timestamp as well.

With our context identified as well as its and the environment's representation, we are ready to define Demand $m$-CFA.

## 6 DEMAND $m$-CFA

Demand $m$-CFA augments Demand 0CFA with environments and environment-instantiation mechanisms which together provide context sensitivity. The addition of environments pervades $\Downarrow^m_{eval}$,

$$\textsc{Lam}\ \frac{}{C[\lambda x.e], \hat{\rho} \Downarrow^m_{eval} C[\lambda x.e], \hat{\rho}} \qquad \textsc{Rator}\ \frac{}{C[([e_0]\ e_1)], \hat{\rho} \Rightarrow^m_{expr} C[(e_0\ e_1)], \hat{\rho}}$$

$$\textsc{Ref}\ \frac{(C_x[e_x], \hat{\rho}_x) = \mathsf{bind}(x, C[x], \hat{\rho}) \quad C_x[e_x], \hat{\rho}_x \Downarrow^m_{call} C'[(e_0\ e_1)], \hat{\rho}' \quad C'[(e_0\ [e_1])], \hat{\rho}' \Downarrow^m_{eval} C_v[\lambda x.e], \hat{\rho}_v}{C[x], \hat{\rho} \Downarrow^m_{eval} C_v[\lambda x.e], \hat{\rho}_v}$$

$$\textsc{App}\ \frac{C[([e_0]\ e_1)], \hat{\rho} \Downarrow^m_{eval} C'[\lambda x.e], \hat{\rho}' \quad C'[\lambda x.[e]], \mathsf{succ}_m(C[(e_0\ e_1)], \hat{\rho}) :: \hat{\rho}' \Downarrow^m_{eval} C_v[\lambda x.e_v], \hat{\rho}_v}{C[(e_0\ e_1)], \hat{\rho} \Downarrow^m_{eval} C_v[\lambda x.e_v], \hat{\rho}_v}$$

$$\textsc{Rand}\ \frac{C[([e_0]\ e_1)], \hat{\rho} \Downarrow^m_{eval} C'[\lambda x.e], \hat{\rho}' \quad x, C'[\lambda x.[e]], \mathsf{succ}_m(C[(e_0\ e_1)], \hat{\rho}) :: \hat{\rho}' \Rightarrow^m_{find} C_x[x], \hat{\rho}_x \quad C_x[x], \hat{\rho}_x \Rightarrow^m_{expr} C''[(e_2\ e_3)], \hat{\rho}''}{C[(e_0\ [e_1])], \hat{\rho} \Rightarrow^m_{expr} C''[(e_2\ e_3)], \hat{\rho}''}$$

$$\textsc{Body}\ \frac{C[\lambda x.[e]], \hat{\rho} \Downarrow^m_{call} C'[(e_0\ e_1)], \hat{\rho}' \quad C'[(e_0\ e_1)], \hat{\rho}' \Rightarrow^m_{expr} C''[(e_2\ e_3)], \hat{\rho}''}{C[\lambda x.[e]], \hat{\rho} \Rightarrow^m_{expr} C''[(e_2\ e_3)], \hat{\rho}''}$$

$$\textsc{Find-Ref}\ \frac{}{x, C[x], \hat{\rho} \Rightarrow^m_{find} C[x], \hat{\rho}} \qquad \textsc{Find-Rator}\ \frac{x, C[([e_0]\ e_1)], \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x}{x, C[(e_0\ e_1)], \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x}$$

$$\textsc{Find-Rand}\ \frac{x, C[(e_0\ [e_1])], \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x}{x, C[(e_0\ e_1)], \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x} \qquad \textsc{Find-Body}\ \frac{x \neq y \quad x, C[\lambda y.[e]], ?_y :: \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x}{x, C[\lambda y.[e]], \hat{\rho} \Rightarrow^m_{find} C_x[x], \hat{\rho}_x}$$

Fig. 4. Demand $m$-CFA Resolution

$\Rightarrow^m_{expr}$, and $\Rightarrow^m_{find}$ which are otherwise identical to their Demand 0CFA counterparts; these enriched relations are presented in Figure 4.

When a call is entered, which occurs in the *App* and *Rand* rules, a new environment is synthesized using the $\mathsf{succ}_m$ metafunction which determines the binding context of the call as

$$\mathsf{succ}_m(C[(e_0\ e_1)], \hat{cc} :: \hat{\rho}) = \lfloor C[(e_0\ e_1)] :: \hat{cc} \rfloor_m$$

where $\lfloor \cdot \rfloor_m$ is defined

$$\lfloor \hat{cc} \rfloor_0 = \square \qquad \lfloor ?_x \rfloor_m = ?_x \qquad \lfloor C[(e_0\ e_1)] :: \hat{cc} \rfloor_m = C[(e_0\ e_1)] :: \lfloor \hat{cc} \rfloor_{m-1}$$

The bind metafunction, which locates the binding configuration of a variable reference, is lifted to accommodate environments as well; its definition is presented in Figure 3.

However, the $\Downarrow^m_{call}$ relation changes substantially.

The particular changes to the $\Downarrow^m_{call}$ require us to make the reachability relation $\Uparrow^m_{reach}$ explicit. We define reachability over queries themselves; the judgment $q \Uparrow^m_{reach} q'$ captures that, if query $q$ is

$$\textsc{Reflexivity}\ \frac{}{q \Uparrow^m_{reach} q} \qquad \textsc{Ref-Caller}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[x], \hat\rho) \qquad (C'[e], \hat\rho') = \mathsf{bind}(x, C[x], \hat\rho)}{q \Uparrow^m_{reach} \mathsf{call}(C'[e], \hat\rho')}$$

$$\textsc{Ref-Argument}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[x], \hat\rho) \qquad (C'[e], \hat\rho') = \mathsf{bind}(x, C[x], \hat\rho) \qquad C'[e], \hat\rho' \Downarrow^m_{call} C''[(e_0\ e_1)], \hat\rho''}{q \Uparrow^m_{reach} \mathsf{eval}(C''[(e_0\ [e_1])], \hat\rho'')}$$

$$\textsc{App-Operator}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[(e_0\ e_1)], \hat\rho)}{q \Uparrow^m_{reach} \mathsf{eval}(C[([e_0]\ e_1)], \hat\rho)} \qquad \textsc{App-Body}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[(e_0\ e_1)], \hat\rho) \qquad C[([e_0]\ e_1)], \hat\rho \Downarrow^m_{eval} C'[\lambda x.e], \hat\rho'}{q \Uparrow^m_{reach} \mathsf{eval}(C'[\lambda x.[e]], \mathsf{succ}_m(C[(e_0\ e_1)], \hat\rho) :: \hat\rho')}$$

$$\textsc{Call-Trace}\ \frac{q \Uparrow^m_{reach} \mathsf{call}(C[\lambda x.[e]], \hat{cc} :: \hat\rho)}{q \Uparrow^m_{reach} \mathsf{expr}(C[\lambda x.e], \hat\rho)} \qquad \textsc{Rand-Operator}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[(e_0\ [e_1])], \hat\rho)}{q \Uparrow^m_{reach} \mathsf{eval}(C[([e_0]\ e_1)], \hat\rho)}$$

$$\textsc{Rand-Body}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[(e_0\ [e_1])], \hat\rho) \qquad C[([e_0]\ e_1)], \hat\rho \Downarrow^m_{eval} C'[\lambda x.e], \hat\rho' \qquad x, C'[\lambda x.[e]], \mathsf{succ}_m(C[(e_0\ e_1)], \hat\rho) :: \hat\rho' \Rightarrow^m_{find} C_x[x], \hat\rho_x}{q \Uparrow^m_{reach} \mathsf{expr}(C_x[x], \hat\rho_x)}$$

$$\textsc{Body-Caller-Find}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[\lambda x.[e]], \hat\rho)}{q \Uparrow^m_{reach} \mathsf{call}(C[\lambda x.[e]], \hat\rho)} \qquad \textsc{Body-Caller-Trace}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[\lambda x.[e]], \hat\rho) \qquad C[\lambda x.[e]], \hat\rho \Downarrow^m_{call} C'[(e_0\ e_1)], \hat\rho'}{q \Uparrow^m_{reach} \mathsf{expr}(C'[(e_0\ e_1)], \hat\rho')}$$

Fig. 5. Demand $m$-CFA Reachability

$$\textsc{Known-Call}\ \frac{q \Uparrow^m_{reach} \mathsf{call}(C[\lambda x.[e]], \hat{cc}_0 :: \hat\rho) \qquad C[\lambda x.e], \hat\rho \Rightarrow^m_{expr} C'[(e_0\ e_1)], \hat\rho' \qquad \hat{cc}_1 := \mathsf{succ}_m(C'[(e_0\ e_1)], \hat\rho') \qquad \hat{cc}_0 \sqsubset \hat{cc}_1}{C[\lambda x.[e]], \hat{cc}_0 :: \hat\rho \Downarrow^m_{call} C'[(e_0\ e_1)], \hat\rho'}$$

$$\textsc{Unknown-Call}\ \frac{q \Uparrow^m_{reach} \mathsf{call}(C[\lambda x.[e]], \hat{cc}_0 :: \hat\rho) \qquad C[\lambda x.e], \hat\rho \Rightarrow^m_{expr} C'[(e_0\ e_1)], \hat\rho' \qquad \hat{cc}_1 := \mathsf{succ}_m(C'[(e_0\ e_1)], \hat\rho') \qquad \hat{cc}_1 \sqsubset \hat{cc}_0}{\hat{cc}_1 :: \hat\rho / \hat{cc}_0 :: \hat\rho}$$

Fig. 6. Demand $m$-CFA Call Discovery

reachable in analysis, then $q'$ is also, where $q$ is of the form

$$\mathit{Query} \ni q ::= \mathsf{eval}(C[e], \hat\rho) \mid \mathsf{expr}(C[e], \hat\rho) \mid \mathsf{call}(C[e], \hat\rho)$$

Figure 5 presents a formal definition of $\Uparrow^m_{reach}$.

The *Reflexivity* rule ensures that the top-level query is considered reachable. The *Ref-Caller* and *Ref-Argument* rules establish reachability corresponding to the *Ref* rule of $\Downarrow^m_{eval}$: *Ref-Caller* makes the caller query reachable and, if it succeeds, *Ref-Argument* makes the ensuing evaluation query

$$\textsc{Instantiate-Reachable-Eval}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[e], \hat{\rho}) \qquad \hat{\rho}_1/\hat{\rho}_0}{q \Uparrow^m_{reach} \mathsf{eval}(C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0])} \qquad \textsc{Instantiate-Eval}\ \frac{\hat{\rho}_1/\hat{\rho}_0 \qquad C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0] \Downarrow^m_{eval} C_v[\lambda x.e], \hat{\rho}_v}{C[e], \hat{\rho} \Downarrow^m_{eval} C_v[\lambda x.e], \hat{\rho}_v}$$

$$\textsc{Instantiate-Reachable-Expr}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[e], \hat{\rho}) \qquad \hat{\rho}_1/\hat{\rho}_0}{q \Uparrow^m_{reach} \mathsf{expr}(C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0])} \qquad \textsc{Instantiate-Expr}\ \frac{\hat{\rho}_1/\hat{\rho}_0 \qquad C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0] \Rightarrow^m_{expr} C'[(e_0\ e_1)], \hat{\rho}'}{C[e], \hat{\rho} \Rightarrow^m_{expr} C'[(e_0\ e_1)], \hat{\rho}'}$$

$$\textsc{Instantiate-Reachable-Call}\ \frac{q \Uparrow^m_{reach} \mathsf{call}(C[e], \hat{\rho}) \qquad \hat{\rho}_1/\hat{\rho}_0}{q \Uparrow^m_{reach} \mathsf{call}(C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0])} \qquad \textsc{Instantiate-Call}\ \frac{\hat{\rho}_1/\hat{\rho}_0 \qquad C[e], \hat{\rho}[\hat{\rho}_1/\hat{\rho}_0] \Downarrow^m_{call} C'[(e_0\ e_1)], \hat{\rho}'}{C[e], \hat{\rho} \Downarrow^m_{call} C'[(e_0\ e_1)], \hat{\rho}'}$$

$$\textsc{App-Body-Instantiation}\ \frac{q \Uparrow^m_{reach} \mathsf{eval}(C[(e_0\ e_1)], \hat{\rho}) \qquad C[([e_0]\ e_1)], \hat{\rho} \Downarrow^m_{eval} C'[\lambda x.e], \hat{\rho}'}{\mathsf{succ}_m(C[(e_0\ e_1)], \hat{\rho}) :: \hat{\rho}'/?_x :: \hat{\rho}'}$$

$$\textsc{Rand-Body-Instantiation}\ \frac{q \Uparrow^m_{reach} \mathsf{expr}(C[(e_0\ [e_1])], \hat{\rho}) \qquad C[([e_0]\ e_1)], \hat{\rho} \Downarrow^m_{eval} C'[\lambda x.e], \hat{\rho}'}{\mathsf{succ}_m(C[(e_0\ e_1)], \hat{\rho}) :: \hat{\rho}'/?_x :: \hat{\rho}'}$$

Fig. 7. Demand $m$-CFA Instantiation

reachable. *App-Operator* and *App-Body* do the same for the *App* rule of $\Downarrow^m_{eval}$, making, respectively, the operator evaluation query reachable and, if it yields a value, the body evaluation query reachable. *Rand-Operator* makes the evaluation query of the *Rand* rule reachable and *Rand-Body* makes the trace query of any references in the operator body reachable. *Body-Caller-Find* makes the caller query of *Body* reachable; if a caller is found, *Body-Caller-Trace* makes the trace query of that caller reachable. Finally, *Call-Trace* makes sure that the trace query of an enclosing $\lambda$ of a caller query is reachable.

Now we are in a position to discuss the definition of $\Downarrow^m_{call}$, presented in Figure 6.

Unlike $\Downarrow^m_{eval}$ and $\Rightarrow^m_{expr}$, $\Downarrow^m_{call}$ is defined in terms of reachability. The *Known-Call* rule says that, if a caller query is reachable, the ensuing trace query of its enclosing $\lambda$ yields a caller, and the caller query's binding context refines the discovered binding context of the call, the resultant caller of the trace query is also a result of the caller query. The call is *known* because the caller query has the context of the call already in its environment. If $\hat{cc}_1 \sqsubset \hat{cc}_0$, however, then the result constitutes an *unknown* caller. In this case, *Unknown-Call* considers whether $\hat{cc}_1$ refines $\hat{cc}_0$ in the sense that $\hat{cc}_0$ can be instantiated to form $\hat{cc}_1$. Formally, the refinement relation $\sqsubset$ as the least relation satisfying

$$C'[(e_0\ e_1)] :: \hat{cc} \sqsubset ?_{C[e]} \qquad\qquad C[(e_0\ e_1)] :: \hat{cc}_1 \sqsubset C[(e_0\ e_1)] :: \hat{cc}_0 \Longleftarrow \hat{cc}_1 \sqsubset \hat{cc}_0$$

If $\hat{cc}_1$ refines $\hat{cc}_0$, *Unknown-Call* does not conclude a $\Downarrow^m_{call}$ judgement, but rather an *instantiation* judgement $\hat{cc}_1 :: \hat{\rho}/\hat{cc}_0 :: \hat{\rho}$ which denotes that *any* environment $\hat{cc}_0 :: \hat{\rho}$ may be instantiated to $\hat{cc}_1 :: \hat{\rho}$. It is by this instantiation that *Known-Call* will be triggered. When $\hat{cc}_1$ does not refine $\hat{cc}_0$, the resultant caller is ignored which, in effect, filters the callers to only those which are compatible and ensures that Demand $m$-CFA is indeed context-sensitive.

Figure 7 presents an extension of $\Uparrow^m_{reach}$ which propagates instantiations to all reachable queries. The *Instantiate-Reachable-** rules ensure that if a query of any kind is reachable, then its instantiation is too. When an instantiation $\hat{\rho}_1/\hat{\rho}_0$ does not apply (so that $\hat{\rho}$ is unchanged), each rule reduces to

a trivial inference. The counterpart *Instantiate-** rules, also present in Figure 7, each extend one of $\Downarrow^m_{eval}$, $\Rightarrow^m_{expr}$, and $\Downarrow^m_{call}$ so that, if an instantiated query of that type yields a result, the original, uninstantiated query yields that same result. As discussed at the beginning of this section, Demand $m$-CFA also discovers instantiations when it extends the environment in the *App* and *Rand* rules. The *App-Body-Instantiation* and *Rand-Body-Instantiation* rules capture these cases.

The definition of Demand $m$-CFA in terms of an "evaluation" relation (which includes evaluation, trace, and caller resolution) and a reachability relation follows the full formal approach of *Abstracting Definitional Interpreters* by Darais [2017]. From this correspondence, we can define the Demand $m$-CFA resolution of a given query as the least fixed point of these relations, effectively computable with the algorithm Darais [2017] provides. We discuss this implementation in more depth in § 8.

## 7 DEMAND $m$-CFA CORRECTNESS

Demand $m$-CFA is a hierarchy of demand CFA. Instances higher in the hierarchy naturally have larger state spaces. The size $|\Downarrow^m_{eval}|$ of the $\Downarrow^m_{eval}$ relation satisfies the inequality

$$|\Downarrow^m_{eval}| \leq |Config \times Config| = |Config|^2 = |Exp \times Env|^2 = |Exp|^2|Env|^2 = n^2|Env|^2$$

where $n$ is the size of the program. We then have

$$|Env| \leq |Ctx|^n \leq (|Call| + 1)^{mn} \leq n^{mn}$$

since the size of environments is statically bound and may be indeterminate. Thus, $|\Downarrow^m_{eval}| \leq n^{mn+2}$; the state space of $\Downarrow^m_{eval}$ is finite but with an exponential bound; the state spaces of $\Rightarrow^m_{expr}$ and $\Downarrow^m_{call}$ behave similarly.

### 7.1 Demand $m$-CFA Refinement

Instances higher in the hierarchy are also more precise, which we formally express with the following theorems.

Theorem 7.1 (Evaluation Refinement). *If* $C[e], \hat{\rho}_0 \Downarrow^{m+1}_{eval} C_v[\lambda x.e_v], \hat{\rho}'_0$ *where* $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ *then* $C[e], \hat{\rho}_1 \Downarrow^m_{eval} C_v[\lambda x.e_v], \hat{\rho}'_1$ *where* $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.

Theorem 7.2 (Trace Refinement). *If* $C[e], \hat{\rho}_0 \Rightarrow^{m+1}_{expr} C'[(e_0\ e_1)], \hat{\rho}'_0$ *where* $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ *then* $C[e], \hat{\rho}_1 \Rightarrow^m_{expr} C'[(e_0\ e_1)], \hat{\rho}'_1$ *where* $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.

Theorem 7.3 (Caller Refinement). *If* $C[e], \hat{\rho}_0 \Downarrow^{m+1}_{call} C'[(e_0\ e_1)], \hat{\rho}'_0$ *where* $\hat{\rho}_0 \sqsubseteq \hat{\rho}_1$ *then* $C[e], \hat{\rho}_1 \Downarrow^m_{call} C'[(e_0\ e_1)], \hat{\rho}'_1$ *where* $\hat{\rho}'_0 \sqsubseteq \hat{\rho}'_1$.

These theorems state that refining configurations submitted to Demand $m$-CFA and its successor Demand $m$+1-CFA yield refining configurations. The proof proceeds directly (if laboriously) by induction on the derivations of the relations.

### 7.2 Demand $\infty$-CFA and Demand Evaluation

To show that Demand $m$-CFA is sound with respect to a standard call-by-value (CBV) semantics, we consider the limit of the hierarchy, Demand $\infty$-CFA, in which context lengths are unbounded. From here, we bridge Demand $\infty$-CFA to a CBV semantics with a concrete form of demand analysis called *Demand Evaluation*. Our strategy will be to show that the Demand $\infty$-CFA semantics is equivalent to Demand Evaluation which itself is sound with respect to a standard CBV semantics.

Demand Evaluation is defined in terms of relations $\Downarrow^{de}_{eval}$, $\Rightarrow^{de}_{expr}$, and $\Downarrow^{de}_{call}$ which are counterpart to $\Downarrow^m_{eval}$, $\Rightarrow^m_{expr}$, and $\Downarrow^m_{call}$, respectively. Like their counterparts, $\Downarrow^{de}_{eval}$, $\Rightarrow^{de}_{expr}$, and $\Downarrow^{de}_{call}$ relate configurations to configurations. However, a Demand Evaluation configuration includes a store $\sigma$ from addresses $n$ to calls consisting of a call site and its environment. Demand Evaluation environments,

rather than being a sequence of contexts, are sequences of addresses. Like contexts, an address may denote an indeterminate context (i.e. call) which manifests as an address which is not mapped in the store. Formally, the components of stores and environments are defined

$$(s, n), \sigma \in Store = (Addr \rightharpoonup Call) \times Addr \qquad \rho \in Env = Addr^*$$
$$cc \in Call = App \times Env \qquad n \in Addr = \mathbb{N}$$

A store is a pair consisting of a map from addresses to calls and the next address to use; the initial store is $(\bot, 0)$.

Figure 8 presents the definitions of $\Downarrow^{de}_{eval}$, $\Rightarrow^{de}_{expr}$, and $\Downarrow^{de}_{call}$. Most rules are unchanged from Demand $m$-CFA modulo the addition of stores. Instantiation in Demand Evaluation is captured by creating a mapping in the store. For instance, Demand $m$-CFA's *App* rule "discovers" the caller of the entered call, which effects an instantiation via *App-Body-Instantiation*. In contrast, Demand Evaluation's *App* rule allocates a fresh address $n$ using fresh, maps it to the caller in the store, and extends the environment of the body with it. The fresh metafunction extracts the unused address and returns a store with the next one. Store extension is simply lifted over the next unused address. Formally, they are defined as follows.

$$\text{fresh}((s, n)) := (n, (s, n + 1)) \qquad (s, n)[n_0 \mapsto cc] := (s[n_0 \mapsto cc], n)$$

*Unknown-Call* applies when the address $n$ is unmapped in the store. It instantiates the environment by mapping $n$ with the discovered caller. *Known-Call* uses $\equiv_\sigma$ to ensure that the known and discovered environments are isomorphic in the store. The $\equiv_\sigma$ relation is defined on addresses and lifted elementwise to environments. We have $n_0 \equiv_\sigma n_1$ if and only if $\sigma(n_0) = \bot = \sigma(n_1)$ or $\sigma(n_0) = (C[(e_0\ e_1)], \rho_0), \sigma(n_1) = (C[(e_0\ e_1)], \rho_1)$, and $\rho_0 \equiv_\sigma \rho_1$. If the environments are isomorphic, then all instances of the known environment are substituted with the discovered environment in the store, ensuring that queries in terms of the known are kept up to date. This rule corresponds directly to the instantiation relation of Demand $m$-CFA.

### 7.3 Demand Evaluation Equivalence

In order to show a correspondence between Demand ∞-CFA and Demand Evaluation, we establish a correspondence between the environments of the former and the environment–store pairs of the latter, captured by the judgement $\hat{\rho}\ _{\hat{\rho}} \Leftrightarrow_\rho \rho, \sigma$ defined by the following rules.

$$\frac{\hat{cc}_1\ _{\hat{cc}} \Leftrightarrow_n n_1, \sigma \qquad \ldots \qquad \hat{cc}_k\ _{\hat{cc}} \Leftrightarrow_n n_k, \sigma}{\langle \hat{cc}_1, \ldots, \hat{cc}_k \rangle\ _{\hat{\rho}} \Leftrightarrow_\rho \langle n_1, \ldots, n_k \rangle, \sigma} \qquad \frac{}{\langle\rangle\ _{\hat{cc}} \Leftrightarrow_\rho \langle\rangle, \sigma}$$

$$\frac{C[(e_0\ e_1)] :: \hat{cc}\ _{\hat{cc}} \Leftrightarrow_n n, \sigma}{C[(e_0\ e_1)] :: \hat{cc}\ _{\hat{cc}} \Leftrightarrow_\rho n :: \rho, \sigma} \qquad \frac{\sigma(n) = \bot}{?_x\ _{\hat{cc}} \Leftrightarrow_n n, \sigma} \qquad \frac{\sigma(n) = (C[(e_0\ e_1)], \rho) \qquad \hat{cc}\ _{\hat{cc}} \Leftrightarrow_\rho \rho, \sigma}{C[(e_0\ e_1)] :: \hat{cc}\ _{\hat{cc}} \Leftrightarrow_n n, \sigma}$$

This judgement ensures that each context in the Demand ∞-CFA environment matches precisely with the corresponding address with respect to the store: if the context is indeterminate, the address must not be mapped in the store; otherwise, if the heads of the context are the same, the relation recurs.

Now it is straightforward to express the equivalence between the Demand ∞-CFA relations and Demand Evaluation.

Theorem 7.4 (Evaluation Equivalence). *If* $\hat{\rho}_0\ _{\hat{\rho}} \Leftrightarrow_\rho \rho_0, \sigma_0$ *then* $C[e], \hat{\rho}_0 \Downarrow^{\infty}_{eval} C'[\lambda x.e], \hat{\rho}_1$ *if and only if* $C[e], \rho_0, \sigma_0 \Downarrow^{de}_{eval} C'[\lambda x.e], \rho_1, \sigma_1$ *where* $\hat{\rho}_1\ _{\hat{\rho}} \Leftrightarrow_\rho \rho_1, \sigma_1$.

Theorem 7.5 (Trace Equivalence). *If* $\hat{\rho}_0\ _{\hat{\rho}} \Leftrightarrow_\rho \rho_0, \sigma_0$ *then* $C[e], \hat{\rho}_0 \Rightarrow^{\infty}_{expr} C'[(e_0\ e_1)], \hat{\rho}_1$ *if and only if* $C[e], \rho_0, \sigma_0 \Rightarrow^{de}_{expr} C'[(e_0\ e_1)], \rho_1, \sigma_1$ *where* $\hat{\rho}_1\ _{\hat{\rho}} \Leftrightarrow_\rho \rho_1, \sigma_1$.

$$\textsc{Lam}\ \frac{}{C[\lambda x.e], \rho, \sigma \Downarrow^{de}_{eval} C[\lambda x.e], \rho, \sigma} \qquad \textsc{Rator}\ \frac{}{C[([e_0]\ e_1)], \rho, \sigma \Rightarrow^{de}_{expr} C[(e_0\ e_1)], \rho, \sigma}$$

$$\textsc{Ref}\ \frac{(C_x[e_x], \rho_x) = \mathsf{bind}(x, C[x], \rho) \quad C_x[e_x], \rho_x, \sigma_0 \Downarrow^{de}_{call} C'[(e_0\ e_1)], \rho', \sigma_1 \qquad C'[(e_0\ [e_1])], \rho', \sigma_1 \Downarrow^{de}_{eval} C_v[\lambda x.e], \rho_v, \sigma_2}{C[x], \rho, \sigma_0 \Downarrow^{de}_{eval} C_v[\lambda x.e], \rho_v, \sigma_2}$$

$$\textsc{App}\ \frac{(n, \sigma_2) := \mathsf{fresh}(\sigma_1) \quad \begin{array}{c} C[([e_0]\ e_1)], \rho, \sigma_0 \Downarrow^{de}_{eval} C'[\lambda x.e], \rho', \sigma_1 \\ C'[\lambda x.[e]], n :: \rho', \sigma_2[n \mapsto (C[(e_0\ e_1)], \rho)] \Downarrow^{de}_{eval} C_v[\lambda x.e_v], \rho_v, \sigma_3 \end{array}}{C[(e_0\ e_1)], \rho, \sigma_0 \Downarrow^{de}_{eval} C_v[\lambda x.e_v], \rho_v, \sigma_3}$$

$$\textsc{Rand}\ \frac{(n, \sigma_2) := \mathsf{fresh}(\sigma_1) \quad \begin{array}{c} C[([e_0]\ e_1)], \rho, \sigma_0 \Downarrow^{de}_{eval} C'[\lambda x.e], \rho', \sigma_1 \\ x, C'[\lambda x.[e]], n :: \rho', \sigma_2[n \mapsto (C[(e_0\ e_1)], \rho)] \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_3 \\ C_x[x], \rho_x, \sigma_3 \Rightarrow^{de}_{expr} C''[(e_2\ e_3)], \rho'', \sigma_4 \end{array}}{C[(e_0\ [e_1])], \rho, \sigma_0 \Rightarrow^{de}_{expr} C''[(e_2\ e_3)], \rho'', \sigma_4}$$

$$\textsc{Body}\ \frac{C[\lambda x.[e]], \rho, \sigma_0 \Downarrow^{de}_{call} C'[(e_0\ e_1)], \rho', \sigma_1 \qquad C'[(e_0\ e_1)], \rho', \sigma_1 \Rightarrow^{de}_{expr} C''[(e_2\ e_3)], \rho'', \sigma_2}{C[\lambda x.[e]], \rho, \sigma_0 \Rightarrow^{de}_{expr} C''[(e_2\ e_3)], \rho'', \sigma_2}$$

$$\textsc{Find-Ref}\ \frac{}{x, C[x], \rho, \sigma \Rightarrow^{de}_{find} C[x], \rho, \sigma} \qquad \textsc{Find-Rator}\ \frac{x, C[([e_0]\ e_1)], \rho, \sigma_0 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_1}{x, C[(e_0\ e_1)], \rho, \sigma_0 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_1}$$

$$\textsc{Find-Rand}\ \frac{x, C[(e_0\ [e_1])], \rho, \sigma_0 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_1}{x, C[(e_0\ e_1)], \rho, \sigma_0 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_1}$$

$$\textsc{Find-Body}\ \frac{x \neq y \qquad (n, \sigma_1) := \mathsf{fresh}(\sigma_0) \qquad x, C[\lambda y.[e]], n :: \rho, \sigma_1 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_2}{x, C[\lambda y.[e]], \rho, \sigma_0 \Rightarrow^{de}_{find} C_x[x], \rho_x, \sigma_2}$$

$$\textsc{Unknown-Call}\ \frac{\sigma_0(n) = \bot \qquad C[\lambda x.e], \rho, \sigma_0 \Rightarrow^{de}_{expr} C'[(e_0\ e_1)], \rho', \sigma_1 \qquad \sigma_2 := \sigma_1[n \mapsto (C[(e_0\ e_1)], \rho')]}{C[\lambda x.[e]], n :: \rho, \sigma_0 \Downarrow^{de}_{call} C'[(e_0\ e_1)], \rho', \sigma_2}$$

$$\textsc{Known-Call}\ \frac{\sigma_0(n) = (C[(e_0\ e_1)], \rho') \quad C[\lambda x.e], \rho, \sigma_0 \Rightarrow^{de}_{expr} C'[(e_0\ e_1)], \rho'', \sigma_1 \qquad \rho' \equiv_{\sigma_1} \rho'' \qquad \sigma_2 := \sigma_1[\rho''/\rho']}{C[\lambda x.[e]], n :: \rho, \sigma_0 \Downarrow^{de}_{call} C'[(e_0\ e_1)], \rho'', \sigma_2}$$

Fig. 8. Demand Evaluation

Theorem 7.6 (Caller Equivalence). *If* $\hat{\rho}_0\ \hat{\rho} \Leftrightarrow_\rho\ \rho_0, \sigma_0$ *then* $C[e], \hat{\rho}_0 \Downarrow^\infty_{call} C'[(e_0\ e_1)], \hat{\rho}_1$ *if and only if* $C[e], \rho_0, \sigma_0 \Downarrow^{de}_{call} C'[(e_0\ e_1)], \rho_1, \sigma_1$ *where* $\hat{\rho}_1\ \hat{\rho} \Leftrightarrow_\rho\ \rho_1, \sigma_1$.

These theorems are proved by induction on the derivations, corresponding instantiation of environments on the Demand ∞-CFA side with mapping an address on the Demand Evaluation side.

## 8 IMPLEMENTATION

We have implemented Demand $m$-CFA using the *Abstracting Definitional Interpreters* approach [Darais et al. 2017]. Using this approach, one defines a definitional interpreter of their evaluator in a monadic and open recursive style (so that the analyzer can intercept recursive calls to the evaluator).

The result of the analysis is a map with four types of keys, one corresponding to evaluation queries through `eval`, one to trace queries through `expr`, one to uses of `call`, and one to refinements for each $\rho$ encountered in the analysis. Keys locate the results of the query; if the key is present in the map, then the results it locates are sound with respect to the query.

### 8.1 Pushdown Precision

An important property of an analyzer is whether its search over the control-flow graph respects CFL reachability. In the CFA literature, such analyses are often described as "pushdown" since they construct a pushdown model of control flow (e.g. [Gilray et al. 2016]). One of the features of the ADI approach to analysis construction is that the analyzer's search is disciplined by the continuation of the defining language, which induces a pushdown model naturally.

Demand $m$-CFA exhibits pushdown precision with respect to two different semantics. First, it is pushdown precise with respect to the demand semantics given in § 6, defined as big-step relations, by virtue of its implementation using the ADI technique. Second, it is pushdown precise with respect the the direct semantics as well. The reason here is a result of the insight of Gilray et al. [2016]: a continuation address which consists of a target (function body) expression and its environment is distinct enough to perfectly correspond calls and returns (but not needlessly distinct to increase analysis complexity). In Demand $m$-CFA, the $\Downarrow^m_{call}$ relation serves as a kind of continuation store relating a target body and its environment to caller configurations. Because the $\Downarrow^m_{call}$ relation is used to access caller configurations both to access arguments and return results (via tracing), it fulfills each call and return aspect of control flow.

## 9 EVALUATION

We implemented Demand $m$-CFA for a subset of R6RS Scheme [Sperber et al. 2010] including `let`, `let*`, and `letrec` binding forms; definition contexts in which a sequence of definitions and expressions can be mixed in a mutually-recursive scope; and a few dozen primitives. We also implemented support for algebraic datatypes and matching on those datatypes, which is a particularly elegant extension to the formalism. In fact, the representation of closures and constructors are isomorphic due to the fact that they both have introduction forms which can be represented as a syntactic context (i.e. lambda, or application of the constructor) paired with an environment. Constructors differ in the fact that they are eliminated with match statements. Pattern matching alternates between $\Rightarrow^m_{expr}$ and $\Downarrow^m_{eval}$ modes discovering introduction forms (binding configurations of the constructors) that flow to the scrutinee and evaluating the appropriate clause based on the discriminant. We reuse support for constructors and matching to desugar `if` statements and `cond` statements.

We included in our benchmarks common R6RS benchmarks, as well as a larger example `tic-tac-toe` which computes the minimax algorithm for an AI to play `tic-tac-toe`, and extensively uses matching, custom datatypes and higher-order behavior.

We evaluate Demand $m$-CFA with respect to the following questions:

(1) Is the implementation cost of Demand $m$-CFA comparable to an exhaustive analysis?
(2) Is Demand $m$-CFA *demand-scalable*?
(3) How does the precision compare to an exhaustive $m$-CFA?

To answer (1) we discuss our observations about implementing both in the same ADI style framework, including the difference in lines of code for the core algorithms. For (2) we consider Demand $m$-CFA what fractions of evaluation queries with a cutoff of 5ms from $m$=0 up to $m$=4 return an answer for a variety of program sizes. Finally to answer (3) we determine what percentage of singleton flow sets as compared to the corresponding exhaustive analysis we obtain within our budget of 5ms per query.

### 9.1 Implementation Costs

In an exhaustive CFA the developer chooses an abstraction and an analysis technique prior to implementation. If any primitive is not supported or any source code is not available (i.e. external libraries), the developer must make a hard choice. They must approximate the behavior or throw away the results of the analysis. It is hard to guarantee the soundness of such an analysis. As languages evolve and add new features and primitives, maintaining and evolving the corresponding analyses becomes both a burden and a source of bugs.

In contrast Demand $m$-CFA is formulated such that the analysis of each language feature is specified independently as much as possible. Due to this design the implementation of an analysis should work transparently across language versions as long as (1) the semantics of each implemented feature and its dependencies does not change, and (2) the abstraction does not need to change.

For example, we did not implement the `set!` form of R6RS Scheme which mutates the binding of a given variable, and we did not implement primitives with side effects. This omission does *not* mean that demand CFA fails on programs that uses `set!`. Rather, it means that demand CFA fails on *queries* whose resolution depends on a `set!`'d variable; other queries resolve without issue. Because the use of mutation is relatively rare in functional languages such as Scheme, ML, and OCaml, we expect that relatively few queries encounter mutation.

Concretely, in terms of lines of code needed, our implementation suggests that a demand analysis involves about the same order of magnitude of engineering effort as $m$-CFA (~660 lines of code versus ~430). However, for programs with unsupported features or unimplemented primitives, our implementation of $m$-CFA fails to give any results but our implementation of Demand $m$-CFA gives correct answers for a subset of the queries.

This answers the first question that Demand $m$-CFA requires a comparable implementation effort to an exhaustive analysis, and in particular when considering the number of primitives and language features that modern languages afford.

### 9.2 Scalability

Monolithic analyses such as $m$-CFA require doing an abstract interpretation over the full program. Therefore to discuss scalability of such analyses we typically determine the computational complexity in terms of the program size. 0CFA has a complexity of $O(n^3)$ [Nielson et al. 1999], and $k$-CFA is proven to be exponential [Van Horn and Mairson 2008]. $m$-CFA (with rebinding) has the advantage is that it gives context sensitivity at a polynomial complexity [Might et al. 2010]. However, even with small programs it quickly becomes expensive as shown in Figure 9.

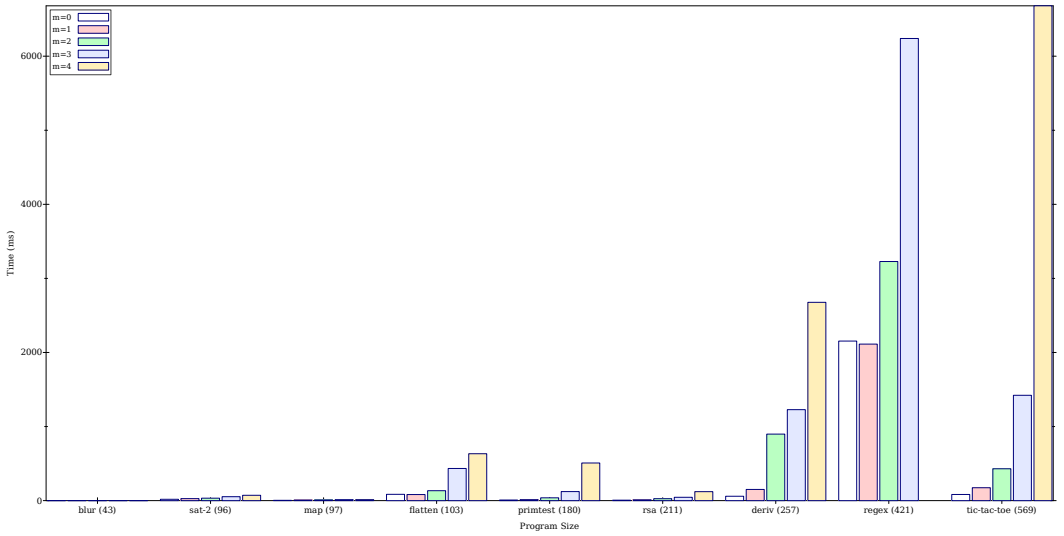


Fig. 9. The scalability of $m$-CFA in practice on simple benchmarks with size of the program (in parethesis) and as we increase $m$

In our results we measure the size of the program as the number of non-trivial syntactic contexts that we could run an evaluation query for, which is closely related to the size of the abstract syntax tree of the program. Trivial queries include lambdas, constants, and references to let bindings that are themselves trivial. These were all omitted from the results unless otherwise stated to determine how Demand $m$-CFA performs in contexts where compiler heuristics would not already trivially understand the control flow.

Demand $m$-CFA has two sources of inherent overhead compared to a monolithic analysis. These are: (1) resolving trace queries in addition to evaluation queries, and (2) instantiating environments.

These apparent disadvantages work to the benefit of Demand $m$-CFA in practice. For example, indeterminate queries allow the analysis to disregard exponential combinations of environments when it is irrelevant to a particular query. In particular we have observed this behavior in the sat-2 benchmark, which induces worst-case behavior in exponential $m$-CFA, due to the exponential combination of nested environments, but where Demand $m$-CFA is able to keep the environments indeterminate for the majority of the queries.

Before deciding to integrate an analysis into their language tooling engineers would like to know the cost benefit tradeoff. Theoretically Demand $m$-CFA presents both costs and benefits at a much more granular level which the compiler engineer can control, but how does that work in practice? To answer this we measure

- the percent of evaluation queries that return within a constant timeout (5ms) per query, as well as
- the percent of singleton flows found as compoared to an exponential $m$-CFA analysis (without rebinding).

We choose exponential $m$-CFA to be able to compare singleton flows against a monolithic analysis with similar environment representation — which should return similar if not identical results. The only thing that makes Demand $m$-CFA's environments different from exponential $m$-CFA's environments is that the latter do not contain indeterminate environments.

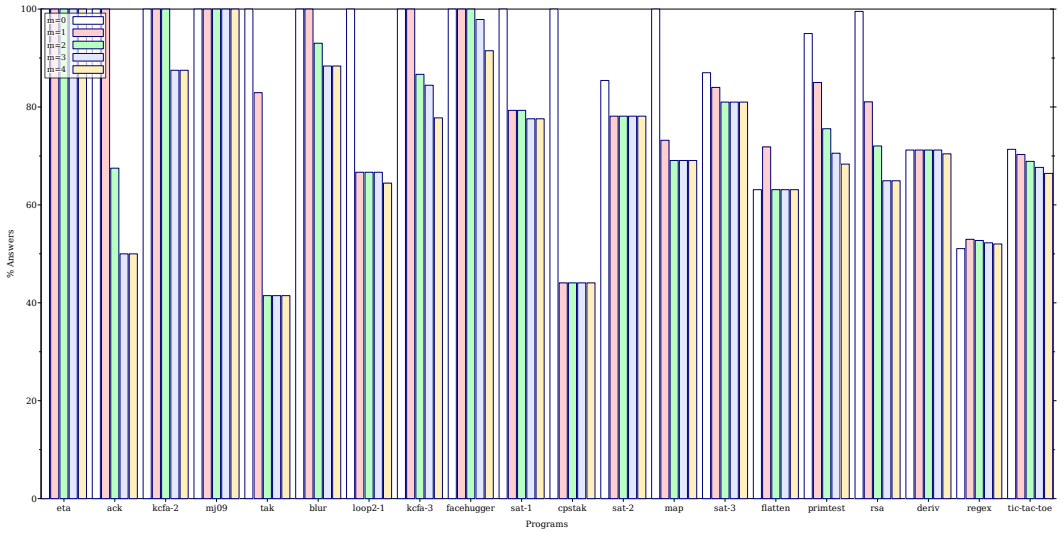


Fig. 10. The percent of answers that Demand $m$-CFA answers based on a 5ms timeout per query. This graph shows all queries (including trivial ones).

As seen in Figure 10, we answer a large majority of all evaluation queries within the specified timeout. When we restrict it to non-trivial queries we get the results in Figure 11, which shows a predictable decrease due to the fact that many flows are lexically obvious (lambdas, constants, etc). When compared to an exhaustive analysis which might timeout or fail, any amount of non-trivial flow at constant cost is welcomed. It is worth noting that increasing the timeout to 15ms only marginally improves the number of returned answers. This matches the intuition that if a query only requires a subset of the entire flow of a program, it should be quick to answer. Importantly we see

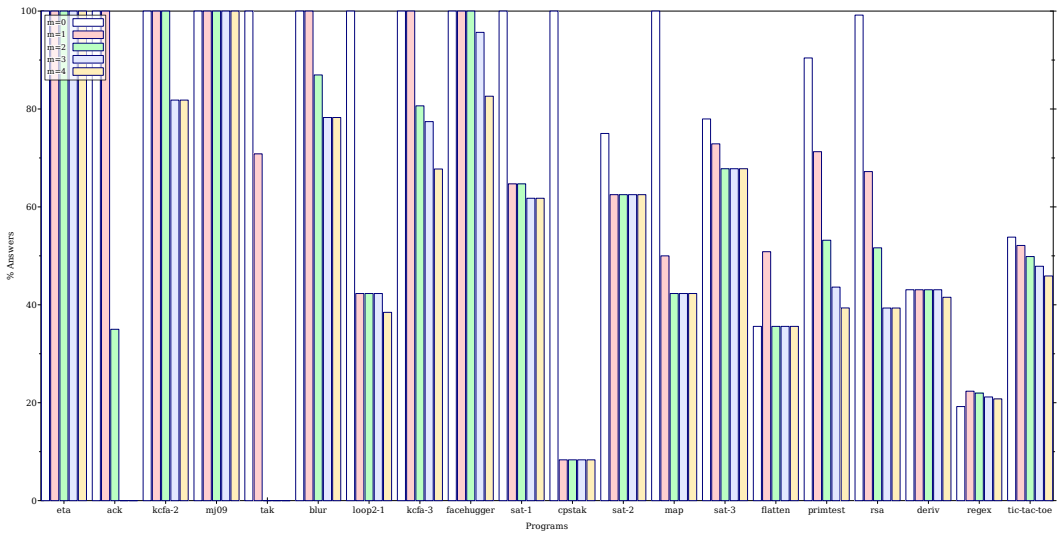


Fig. 11. The percent of non-trivial queries that Demand $m$-CFA answers using a 5ms/query timeout.

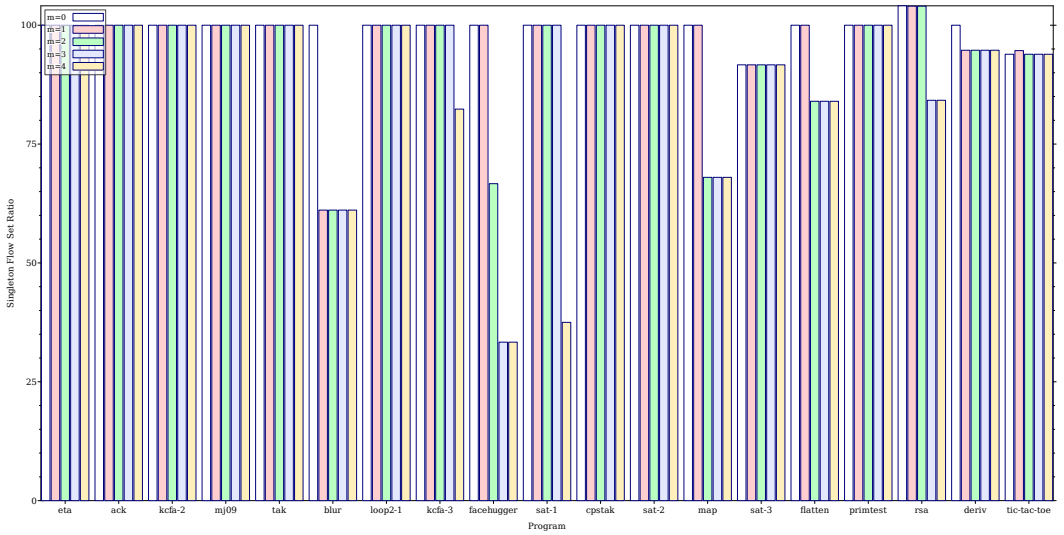


Fig. 12. Ratio of the # singleton flow sets found by Demand $m$-CFA compared to exhaustive $m$-CFA.

that the size of the program does not seem to have a large effect on the tractability of the problem and neither does $m$. This means that our Demand $m$-CFA analysis is indeed *demand-scalable*, answering our second question.

Figure 12 shows that in most cases we resolve much more than half the number of singleton value flows as *exponential* $m$-CFA, but in constant time[5]. This means that not only is Demand $m$-CFA *demand-scalable*, it also produces useful results. In a few cases we actually report more than

[5]To compute flow sets we obtain all configurations that have the same evaluation configuration (without environments) and join the results (also without environments). showing that it had to do more work to arrive at the same conclusion.

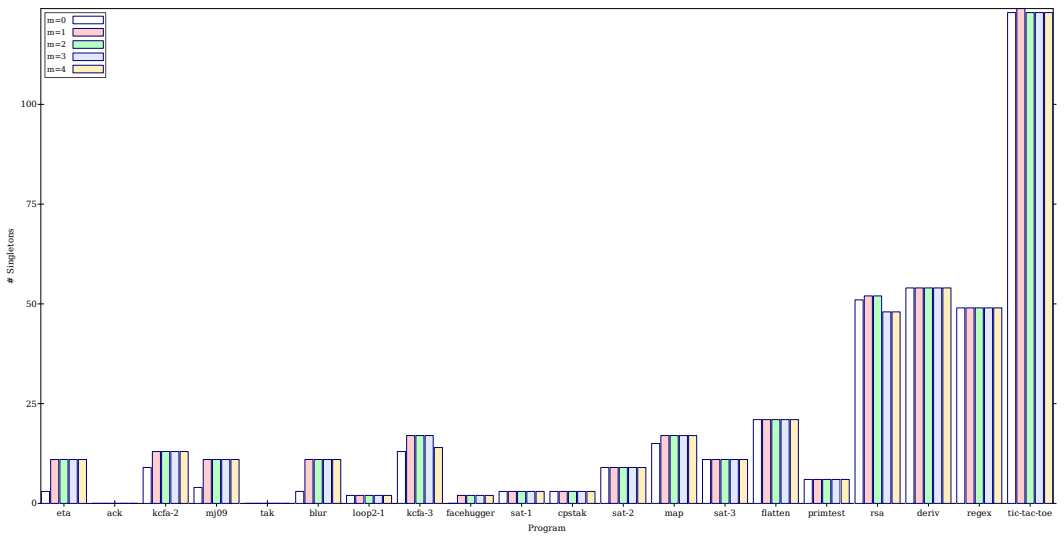


Fig. 13. # of singleton flow sets found by Demand $m$-CFA

100% of the equivalent singleton flow sets of exponential $m$-CFA. This is due to the fact that in a few instances Demand $m$-CFA evaluates queries even for parts of the program that are never seen in the exhaustive analysis.

In Figure 13, we show the total number of singleton flow sets found, however, we see that increasing $m$ does not seem to have the desired effect of increasing precision in many cases. We attribute this partially to the fact that these benchmark programs are small. The fact that we still get many results with high $m$ within the 5ms timeout shows the power of Demand $m$-CFA in allowing us to explore high $m$ at low cost. Of particular note is the omission of regex in Figure 13, this is due to exhaustive $m$-CFA not completing within a generous timeout of 100 seconds for $m = 4$ causing that ratio to be ill-defined. The results are similar to the other large example programs, and Figure 13 shows that we resolve many singleton flows within the timeout period. In some cases a larger $m$ actually shows a decrease in the number of singletons found. In an exhaustive analysis this would be a red flag, due to the fact that the precision of the results should monotonically increase with respect to $m$. However, due to the fact that more configurations have to be evaluated when $m$ gets larger, some queries which used to be resolved within a constant time can timeout as $m$ gets larger. So this behavior is expected in a Demand analysis, and it is remarkable that we don't lose more singletons to timeouts.

### 9.3 Limitations and Future Work

These results are most limited with respect to the benchmark sizes. We intend on scaling up Demand $m$-CFA to handle a full language and larger benchmarks to assuage these concerns. In the meantime we appeal to the intuition of the reader about singleton flow sets. The very nature of singleton flows mean they do not become highly entangled with other differentiated flows. Call-site sensitivity (such as the $m$-CFA abstraction) can help tease apart distinguished flow sets which originate from different call sites. With supporting evidence from our larger benchmarks we believe our approch will scale to larger programs.

Additionally Demand $m$-CFA makes reachability assumptions which can, decrease its precision. For instance, if Demand $m$-CFA is tracing the caller of `f` in the expression `(λ (g) (f 42))` so that it can evaluate the argument, it assumes that `(f 42)` is reachable—i.e., it assumes that `(λ (g) (f 42))` is called. If that assumption is false, then the argument `42` does not actually contribute to the value that Demand $m$-CFA is resolving, and its inclusion is manifest imprecision. We believe this to be the case for several of the benchmarks. Determining callers prior to evaluating would cause the indeterminate environment to be instantiated which could counteract the benefit of keeping the environments mostly indeterminate, and a more nuanced approach is left to future work.

Some aspects of programs, such as the use of dynamic features, inherently limit the information that can be obtained statically. Defensive analysis [Smaragdakis and Kastrinis 2018] provides both a result and an indicator of whether that result is sound for every execution environment. Demand CFA is already defensive in a sense: query resolution fails when it encounters an unsupported language feature. However, integrating defensive analysis would require it to be more principled about its reachability assumptions and the status of its results.

Future work should investigate interesting tradeoffs exposed by Demand $m$-CFA's cost model. This includes: (1) exploring other criteria for terminating queries early, (2) beginning with $m = 0$, rerun queries with higher $m$ only as needed.

Beyond exploring the tradeoffs and implications of the cost model, future work is also needed to: (1) develop theories for common language features such as mutation and higher order control flow such as exceptions and continuations, (2) evaluate Demand $m$-CFA for practical usage in language servers, optimizing compilers, and other analyses, and (3) consider how selective context sensitivity [Li et al. 2020] could be realized given the indeterminate environments of our approach.

## 10 RELATED WORK

The original inspiration for demand CFA is demand dataflow analysis [Horwitz et al. 1995] which refers to dataflow analysis algorithms which allow selective, local, parsimonious analysis of first-order programs driven by the user. Demand CFA seeks algorithms with those same characteristics which operate in the presence of first-class functions. This work extends Demand 0CFA [Germane et al. 2019], currently the sole embodiment of demand CFA, with context sensitivity using the context abstraction of $m$-CFA [Might et al. 2010].

Most closely related is the technique developed in Lifting On-Demand Analysis to Higher-Order Languages [Schoepe et al. 2023]. The approach developed meets all of our criteria above, including context sensitivity, but relegates the context sensitivity to the underlying analyses, and requires multiple demand-driven analyses for the language in question. Our work differs in two major ways: first, it does not require the existence of pre-existing first-order demand-driven forward and backwards analyses, and second, it directly addresses context sensitivity of variables bound in higher order and nested lexical closure environments.

DDPA [Palmer and Smith 2016] is a context-sensitive, demand-driven analysis for higher-order programs so, nominally, it is in precisely the same category as Demand $m$-CFA. However, before resolving any on-demand queries, DDPA must bootstrap a global control-flow graph to support them. Because of this large, fixed, up-front cost, DDPA does not provide the pricing model of a demand analysis and does not make the kinds of applications targeted by demand analysis practical.

Several other "demand-driven" analyses exist for functional programs. Midtgaard and Jensen [2008] present a "demand-driven 0-CFA" derived via a calculational approach which analyzes only those parts of the program on which the program's result depends. Biswas [1997] presents a demand analysis that takes a similar approach to "demand-driven 0-CFA" to analyze first-order functional programs. These analyses are certainly "demand-driven" in the sense the authors intend, but not in the sense that we use it, as a descriptor for generic demand CFA. Demand CFA can be considered a loose extension in spirit which analyzes only those parts of a *higher-order* functional program on which *a selected expression*'s result depends.

Heintze and McAllester [1997] describe the "demand-driven" subtransitive CFA which computes an underapproximation of control flow in linear time and can be transitively closed for an additional quadratic cost. Their analysis exploits type structure and applies to typed programs with bounded type, whereas our formulation neither considers nor requires types.

*Points-to* analysis is the analogue of CFA in an object-oriented setting in the sense that both are fundamental analyses that provide the necessary support for higher-level analysis. Many context-sensitive demand-driven points-to analyses (e.g. Lu et al. [2013]; Shang et al. [2012]; Späth et al. [2016]; Su et al. [2014]) exist, formulated for Java. Though both points-to analysis and CFA target higher-order programs, Might et al. [2010] observed that the explicit object creation in the object-oriented setting induces flat closures whereas implicit closure creation in the functional setting induces nested closures. Moreover, mutation is routine in many object-oriented settings (e.g. Java) in which points-to analyses are expressed. Both nested environments and the prevalence of immutable variables are fundamental to our realization of demand CFA.

Pointer analysis is the imperative analogue of control-flow analysis and it too enjoys a rich literature. When function pointers are present, a demand-driven pointer analysis must be able to reconstruct the call graph on the fly, a requirement shared by demand-driven CFA.

Heintze and Tardieu [2001] present a demand-driven pointer analysis for C capable of constructing the call graph on the fly. They recognize that most call targets are not specified indirectly through pointers and advocate demand-driven analysis to resolve indirect targets when they appear. In the core language of demand $m$-CFA, calls with indirect targets are commonplace; the

language was chosen to emphasize those cases in particular. In a full-featured language, calls with indirect targets—though still more common than in C—are less common, since calls to primitives are typically direct and statically discernible. However, closures are not merely code pointers, but environments too, and we have had to exercise great care to ensure that demand $m$-CFA correctly models their behavior.

Saha and Ramakrishnan [2005] formulate points-to analysis of C as a logic program, so that the incremental-evaluation capabilities of the logic engine yield an incremental analysis. They combine an incremental and demand-driven approach to update the points-to model in response to changes in the program. Our work is similar in that it may be possible to cast it as a logic program and thereby combine an initial exhaustive CFA with an incremental, demand-driven CFA to keep the model synchronized with a changing program.

More recently, Sui and Xue [2016] developed Supa, an on-demand analyzer for C programs which refines value flows during analysis. Supa is designed for low-budget environments and its evaluation explicitly weighs its ultimate precision against its initial budget. Demand $m$-CFA can be positioned similarly where the low-budget environments are compilers and IDEs for functional programs.

## 11 CONCLUSION

This paper presented one strategy for achieving context-sensitive demand CFA, based on the top-$m$-stack-frames context abstraction of $m$-CFA [Might et al. 2010]. This strategy leads to the Demand $m$-CFA hierarchy which exhibits pushdown precision (1) with respect to the demand semantics, by virtue of using the *Abstracting Definitional Interpreters* [Darais et al. 2017] implementation approach, and (2) with respect to the direct semantics, by virtue of using the continuation address identified in *Pushdown for Free* [Gilray et al. 2016]. This leads to the Demand $m$-CFA hierarchy, which, for many singleton flow sets, offers the precision of context sensitivity at a constant price, regardless of program size, which we claim makes Demand $m$-CFA a *demand-scalable* analysis.

## REFERENCES


Sandip K. Biswas. 1997. A demand-driven set-based analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. ACM, New York, NY, USA, 372–385. https://doi.org/10.1145/263699.263753

David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 1, ICFP, Article 12 (Aug. 2017), 25 pages. https://doi.org/10.1145/3110256

David Charles Darais. 2017. *Mechanizing Abstract Interpretation.* Ph. D. Dissertation. University of Maryland, College Park, MD, USA.

Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1997. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 6 (Nov. 1997), 992–1030. https://doi.org/10.1145/267959.269970

Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. 2019. Demand Control-Flow Analysis. In *Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science, Vol. 11388)*, Constantin Enea and Ruzica Piskac (Eds.). Springer, Cham, 226–246. https://doi.org/10.1007/978-3-030-11245-5_11

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

Nevin Heintze and David McAllester. 1997. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada, USA) *(PLDI '97)*. ACM, New York, NY, USA, 261–272. https://doi.org/10.1145/258915.258939

Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. ACM, New York, NY, USA, 24–34. https://doi.org/10.1145/378795.378802

Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand interprocedural dataflow analysis. *ACM SIGSOFT Software Engineering Notes* 20, 4 (1995), 104–115.

Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 42, 2 (2020), 1–40.

Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction (Lecture Notes in Computer Science, Vol. 7791)*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer, Berlin, Heidelberg, 61–81. https://doi.org/10.1007/978-3-642-37051-9_4

Jan Midtgaard and Thomas Jensen. 2008. A Calculational Approach to Control-Flow Analysis by Abstract Interpretation. In *Static Analysis (Lecture Notes in Computer Science, Vol. 5079)*, María Alpuente and Germán Vidal (Eds.). Springer, Berlin, Heidelberg, 347–362. https://doi.org/10.1007/978-3-540-69166-2_23

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming* (Portland, Oregon, USA) *(ICFP '06)*. ACM, New York, NY, USA, 13–25. https://doi.org/10.1145/1159803.1159807

Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. 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* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

Flemming Nielson, Hanne R. Nielson, and Chris Hankin. 1999. *Principles of Program Analysis*. Springer-Verlag, Berlin, Heidelberg.

Zachary Palmer and Scott F. Smith. 2016. Higher-Order Demand-Driven Program Analysis. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 19:1–19:25. https://doi.org/10.4230/LIPIcs.ECOOP.2016.19

Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. ACM, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

Daniel Schoepe, David Seekatz, Ilina Stoilkovska, Sandro Stucki, Daniel Tattersall, Pauline Bolignano, Franco Raimondi, and Bor-Yuh Evan Chang. 2023. Lifting on-demand analysis to higher-order languages. In *International Static Analysis Symposium*. Springer, 460–484.

Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization* (San Jose, California) *(CGO '12)*. ACM, New York, NY, USA, 264–274. https://doi.org/10.1145/2259016.2259050

Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages*. Ph. D. Dissertation. Carnegie Mellon University, Pittsburgh, PA, USA.

Yannis Smaragdakis and George Kastrinis. 2018. Defensive Points-To Analysis: Effective Soundness via Laziness. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.22

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Robby Findler, and Jacob Matthews. 2010. *Revised [6] Report on the Algorithmic Language Scheme* (1st ed.). Cambridge University Press, New York, NY, USA.

Yu Su, Ding Ye, and Jingling Xue. 2014. Parallel Pointer Analysis with CFL-Reachability. In *43rd International Conference on Parallel Processing (ICPP 2014)*. IEEE Computer Society, Los Alamitos, CA, USA, 451–460. https://doi.org/10.1109/ICPP.2014.54

Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 460–473. https://doi.org/10.1145/2950290.2950296

David Van Horn and Harry G. Mairson. 2008. Deciding *k*CFA is complete for EXPTIME. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming* (Victoria, BC, Canada) *(ICFP '08)*. ACM, New York, NY, USA, 275–282. https://doi.org/10.1145/1411204.1411243

Guannan Wei, James Decker, and Tiark Rompf. 2018. Refunctionalization of abstract abstract machines: bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). *Proceedings of the ACM on Programming Languages* 2, ICFP, Article 105 (July 2018), 28 pages. https://doi.org/10.1145/3236800