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 an interface and pricing model much more flexible than typical 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 implementation effort, scalability, and precision of Demand m-CFA. We demonstrate that Demand m-CFA (1) resolves a large percent of queries quickly even when we increase context sensitivity and program size, (2) can resolve as many singleton value flows as a formulation of m-CFA with full environment precision, and (3) can be implemented cheaply and integrated into interactive tools such as language servers.

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, 23 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 works backwards from the reference to `f` to determine its value, and considers only the three flow segments identified above to do so.

The interface and pricing model demand CFA offers make many useful applications practical. Horwitz et al. [1995] identify several ways this practicality is realized:

(1) One can restrict analysis to a program's critical path or only where certain features are used.
(2) 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.
(3) One can let a user drive the analysis, even interactively, to enhance, e.g., an IDE experience. We have implemented the demand CFA we present in this paper in Visual Studio Code for the Koka language [kok 2019].

1.1 Adding Context Sensitivity to Demand CFA

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 present a streamlined version of it in §4.) However, context sensitivity would endow demand CFA with the same benefits that it does analyses at large: increased precision and, in some cases, a reduced workload [Might and Shivers 2006] (which we discuss at an intuitive level in §2).

However, the demand setting presents a particular challenge for adding context sensitivity: unlike exhaustive analyses in which the context is fully determined at each point in analysis, a demand analysis is deployed on an arbitrary program point in an undetermined context. Thus, the task of a context-sensitive demand CFA is not only to respect the context as far as it is known, but also to determine unknown contexts as they are discovered relevant to analysis. Achieving this task requires a compatible choice of context, context representation, and even environment representation, as we discuss in §5.

After overcoming this challenge, we arrive at Demand m-CFA (§6), a hierarchy of context-sensitive demand CFA. Demand m-CFA is sound with respect to a concrete albeit demand semantics called *demand evaluation* (§7), which is itself sound with respect to a standard call-by-value semantics.

Demand m-CFA determines the context only to the extent necessary to soundly answer analysis questions, as opposed to determining the entire context. Not only does this allow Demand m-CFA to avoid analysis work, it offers information to the analysis client regarding which aspects of the context are relevant to a particular analysis question, which the client can use to formulate subsequent questions.

We have implemented Demand m-CFA in several settings using the *Abstracting Definitional Interpreters* (ADI) technique [Darais et al. 2017]. We evaluate the implementation cost and performance to empirically assess Demand m-CFA (§8).

We conclude by discussing related (§9) and future work (§10).

2 DEMAND CFA, INTUITIVELY

A user interacts with demand CFA by submitting queries, which the analyzer resolves online. There are two types of queries:

(1) An *evaluation* query seeks the values to which a specified expression may evaluate. In essence, it is asking which values flow to a given expression.
(2) A *trace* query seeks the sites at which the value of a specified expression may be used. In essence, it is asking where values flow from a given expression.

To resolve all but trivial queries, the analysis issues subqueries—of one type or the other—to resolve flows on which the original query depends.

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 Pure Scheme (i.e. purely-functional Scheme).

2.1 Without Context Sensitivity

q_0 evaluate `(f 35)`
q_1 evaluate `f`
q_2 evaluate `(λ (x) x)`
⇒ `(λ (x) x)`
q_3 evaluate `x`
q_4 trace `(λ (x) x)`
q_5 trace `f` in `(f 42)`
⇒ `(f 42)`
q_7 evaluate `42`
⇒ `42`
q_6 trace `f` in `(f 35)`
⇒ `(f 35)`
q_8 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 λ 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 λ-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.

q_0 evaluate `(f 35)` in $\langle\rangle$
q_1 evaluate `f` in $\langle\rangle$
q_2 evaluate `(λ (x) x)` in $\langle\rangle$
⇒ `(λ (x) x)` in $\langle\rangle$
q_3 evaluate `x` in $\langle$`(f 35)`$\rangle$
q_3' caller `x` in $\langle$`(f 35)`$\rangle$
q_4 trace `(λ (x) x)` in $\langle\rangle$
q_5 trace `f` in $\langle\rangle$
⇒ `(f 42)` in $\langle\rangle$
⇒ *fail*
q_6 trace `f` in $\langle\rangle$
⇒ `(f 35)` in $\langle\rangle$
⇒ `(f 35)` in $\langle\rangle$
q_8 evaluate `35` in $\langle\rangle$
⇒ `35` in $\langle\rangle$

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.

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 $\langle\rangle$
⇒ `(f 42)` in $\langle\rangle$
⇒ `(f 42)` in $\langle\rangle$
q_4 evaluate `x` in $\langle$`(f 42)`$\rangle$
q_5 evaluate `42` in $\langle\rangle$
⇒ `42` in $\langle\rangle$
q_3 trace `f` in `(f 35)`
⇒ `(f 35)`
q_6 evaluate `x` in $\langle$`(f 35)`$\rangle$
q_7 evaluate `35` in $\langle\rangle$
⇒ `35` in $\langle\rangle$

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 $\langle$?$\rangle$ (q_0 at left) 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.) 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

We present Demand m-CFA over the unary λ calculus. It is straightforward to extend it to multiple-arity functions, data structures, constants, primitives, conditionals, and recursive forms—which we have in our implementations—but this small language suffices to discuss the essential aspects of context sensitivity.

In order to keep otherwise-identical expressions distinct, many CFA presentations 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. We instead use the syntactic context of each expression, which uniquely determines it, as a de-facto label, since Demand m-CFA consults it extensively.

In the unary λ 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 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 $\lambda x.(x\ \square)$.

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.

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 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 respectively supported by the auxiliary metafunction bind and relation $\Rightarrow_{find}$, which concern variable bindings and are also dual to each other. Anticipating the addition of context sensitivity, we define $\Downarrow_{eval}$ in terms of the relation $\Downarrow_{call}$, which we discuss below. Figure 1 presents the definitions of all of these relations.

[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.

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

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

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

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

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

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

Fig. 1. Demand 0CFA evaluation and trace 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 λ 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 λ 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 λ 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 2 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 becomes more involved with the addition of context sensitivity.

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

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

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

Fig. 2. The bind metafunction

body of a λ 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 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 $\Rightarrow_{\mathit{find}}$ 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.

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. (Like Demand 0CFA, our context-sensitive demand CFA will not materialize the store in the semantics, but it can be recovered if desired.)

From this point, we must determine (1) the appropriate choice of context, (2) its representation, and (3) the appropriate representation of environments which record the context, which we do in this section. Once these are determined, we combine each expression and its enclosing environment to form a *configuration*.[3] We can extend $\Downarrow_{\mathit{eval}}$ and $\Rightarrow_{\mathit{expr}}$ trivially to relate configurations rather than mere expressions.

A key constraint in the demand setting is that a client-issued query is typically issued in a completely indeterminate context. The expectation is that the analysis will both determine the context to the extent necessary to resolve the query (but no more) and also respect the context so that the analysis is in fact context-sensitive. One implication of this expecation is that queries over expressions which are evaluated in multiple contexts should produce multiple results, one for each context. To operate under this constraint, we extend $\Downarrow_{\mathit{call}}$ to instantiate the context and ensure that the analysis respects it.

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 λ calculus, the most straightforward choice is call-site sensitivity.

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

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 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.

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 □. 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 $\mathit{Var} \to \mathit{Context}$). We instead use a variable's de Bruijn index, which is statically determined by the program syntax, and model environments as a sequence $\mathit{Context}^*$ which, for any finite program, has length bounded by the maximum lexical depth therein.

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 $\langle ?_y, ?_x \rangle$ which is distinct from the indeterminate environment of `z`, which we represent by $\langle ?_z, ?_x \rangle$, 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 $\langle ?_x, ?_f \rangle$, 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 $\langle ?_x, ?_f \rangle$: the first to $\langle ?_x, (\texttt{apply add1}) :: ?_{tl} \rangle$ and the second to $\langle ?_x, (\texttt{apply sub1}) :: ?_{tl} \rangle$, 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

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

(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 $?_x$ with each of these sites. However, in doing so, we will get *six* environments, instantiating $?_x$ in each of $\langle ?_x, ?_f \rangle$, $\langle ?_x, (\texttt{apply add1}) ::?_{tl} \rangle$, and $\langle ?_x, (\texttt{apply sub1}) ::?_{tl} \rangle$ with each of $((\texttt{apply add1})\ 42) ::?_{tl}$ and $((\texttt{apply sub1})\ 35) ::?_{tl}$. In particular, we obtain

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

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 λ 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

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

and

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

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 straightforward and our environment representation makes accessing their binding context straightforward as well. Thus, under such an abstraction, the environment uniquely determines the timestamp, and we can omit timestamps from configurations with no loss of information.

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

6 DEMAND m-CFA

Demand m-CFA augments Demand 0CFA with environments and a mechanism to instantiate contexts within environments, which together provide context sensitivity. The addition of environments pervades $\Downarrow^m_{eval}$, $\Rightarrow^m_{expr}$, and $\Rightarrow^m_{find}$ which are otherwise identical to their Demand 0CFA counterparts; these enriched relations are presented in Figure 4. The mechanism is located in $\Downarrow^m_{call}$, which is significantly elaborated relative to its Demand 0CFA counterpart.

In Demand m-CFA, environments are constructed when the analysis follows evaluation forward, such as entering a call, and instantiated when the analysis follows it backward, such as when the caller of an entry configuration is sought.

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

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

Fig. 3. The bind metafunction

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

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

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

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

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

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

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

Fig. 4. Demand m-CFA Resolution

KNOWN-CALL

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

UNKNOWN-CALL

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

Fig. 5. Demand m-CFA Call Discovery

6.1 Following Evaluation Forwards

When a call is entered, which occurs in the *App* and *Rand* rules, a new environment is synthesized using the 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

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

6.2 Following Evaluation Backwards

When the value of a reference is demanded, Demand m-CFA first uses the bind metafunction to locate its binding configuration. Its definition, presented in Figure 3, is lifted from Demand 0CFA's to accommodate environments.

With the binding configuration in hand, Demand m-CFA issues a call query to resolve calls which enter that configuration. The $\Downarrow^m_{call}$ relation resolves such queries. As in Demand 0CFA, $\Downarrow^m_{call}$ defers to $\Rightarrow^m_{expr}$ to determine the call configurations at which a particular closure is applied. In the presence of contexts, however, there are two possibilities when a call configuration is resolved. The first possibility is that the call entry context computed from that configuration precisely matches that of the binding configuration, in which case the call configuration is a caller of the binding configuration; we refer to this as the *Known-Call* case. The second possibility is that the call entry context refines the binding configuration's context, in which case the refined context and environment should be instantiated to the refining context and environment; we refer to this as the *Unknown-Call* case.

6.3 Discovering Callers

We now describe how $\Downarrow^m_{call}$, presented in Figure 5, handles each of these cases. Each case is handled by a corresponding rule. Each rule is predicated on the *reachability* of the call query, which we discuss shortly, and resolution of the corresponding trace query. The *Known-Call* rule says that, if the call entry context of the delivered call matches the binding configuration's, then the delivered call is a known call—*known* because the caller query has the context of the call already in its environment. If $\hat{cc}_1 \neq \hat{cc}_0$, 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 $\sqsubset$ is defined as the least relation satisfying

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

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

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

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

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

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

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

Fig. 6. Demand m-CFA Reachability

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.

The $\Downarrow^m_{call}$ relation relies on a reachability relation $\Uparrow^m_{reach}$ which establishes which queries are (transitively) issued in the course of resolving a top-level query. This relation allows $\Downarrow^m_{call}$ to instantiate only seen queries with refinement results. This relation is defined over queries themselves; the judgment $q \Uparrow^m_{reach} q'$ captures that, if query q is 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 6 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 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*

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

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

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

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

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

Fig. 7. Demand m-CFA Instantiation

makes the trace query of that caller reachable. Finally, *Call-Trace* makes sure that the trace query of an enclosing λ of a caller query is reachable.

Figure 7 presents an extension of $\Uparrow^m_{reach}$ which propagates instantiations to evaluation and trace 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 ∞-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 ∞-CFA, in which context lengths are unbounded. From here, we bridge Demand ∞-CFA to a CBV semantics with a concrete form of demand analysis called *Demand Evaluation*. Our strategy will be to show that the Demand ∞-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 σ 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.

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

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

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

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

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

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

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

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

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

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

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

Fig. 8. Demand Evaluation

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* $\hat{\rho}_1\ \hat{\rho} \Leftrightarrow_\rho \rho_1, \sigma_1$.

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 EVALUATION

We implemented Demand m-CFA for a subset of R6RS Scheme [Sperber et al. 2010] including `let`, `let*`, `letrec` binding forms, mutually-recursive definitions and a few dozen primitives. We also implemented support for constructors, numbers, symbols, strings, and characters.

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

(1) How does the implementation cost compare to a typical CFA?
(2) What is the distribution of required analysis effort to resolve queries?
(3) How does the precision compare to a typical CFA?

To answer performance questions, we evaluate Demand m-CFA on the set of R6RS programs used by Johnson et al. [2014], which is standard within the literature.

8.1 Implementation cost

We used the *Abstracting Definitional Interpreters* approach [Darais et al. 2017] to implement m-CFA and Demand m-CFA analyses for the Pure Scheme language. The amount of code needed to implement each analysis is on the same order of magnitude. Demand m-CFA requires 630 lines of code while m-CFA uses 450 lines of code, without counting the supporting functions shared between the two. Demand m-CFA requires additional lines of code due to the fact that in addition to evaluation, it also traces the flow of values.

We also implemented Demand m-CFA for the Koka language compiler and language server to provide analyzer interaction within the editor (e.g. providing control flow information on hover). Many queries resolve at interactive latencies, providing the user with near-real-time control flow information. We did not implement a corresponding exhaustive analysis for Koka which would require whole program transformations and many more primitives.

Our implementation for Koka is about 652 lines of Haskell code for the core analysis, with an additional 2364 lines of supporting code for primitives, the fixpoint ADI framework, and mapping the core syntax to the user syntax for showing results.

Our experience is that integrating Demand m-CFA into a compiler or language server is in some cases *more* tractable than an exhaustive analysis, since not all language features or primitives need to be implemented to get at least some utility.

8.2 Query complexity distribution

Like an exhaustive analysis, a demand analysis is subject to client-imposed resource constraints. However, unlike with exhaustive analysis, failure in a demand analysis is not catastrophic: because analysis information is sought selectively and independently through queries, the client can increase the budget and reissue the query or proceed without control flow information.

We evaluate the proportion of queries that Demand m-CFA is able to resolve for different effort limits and at different context sensitivity levels. Effort is measured in gas, where each subquery consumes one unit. Figure 9 displays the percent of queries answered depending on the gas allotted. The graphs trend upward and to the right, regardless of our choice of m. Notable exceptions include `regex`, and `scheme2java` which both contain highly connected control flow graphs. Additionally `scheme2java` uses `set!` which we do not support. Both significantly improve when explored at higher m which exemplifies the known paradox of precision from exhaustive CFA, in which increasing precision can sometimes decrease the state space due to fewer spurious flows. Our results show that this phenomenon holds even for subportions of programs. We hypothesize that this paradox might be more useful in the demand context due to not exploring undemanded or unrelevant portions of the program with higher sensitivity.

8.3 Demand m-CFA precision

To measure the precision of Demand m-CFA, we focus on the singleton flow sets that it is able to identify. Singleton flow sets represent actionable flow information; for instance, they enable compilers to proceed with constant propagation and procedure inlining. Figure 10, shows the number of results that contain singleton flow sets. Exhaustive m-CFA results are dashed and are represented as a straight line, whereas Demand m-CFA results are graphed as a function of effort. `scheme2java` doesn't have results for exhaustive analysis beyond $m = 1$ due to timing out after 10 minutes.

In some cases, Demand m-CFA finds singleton flow sets that exhaustive m-CFA doesn't, explained by the fact that some of the code on which Demand m-CFA is dispatched is dead.

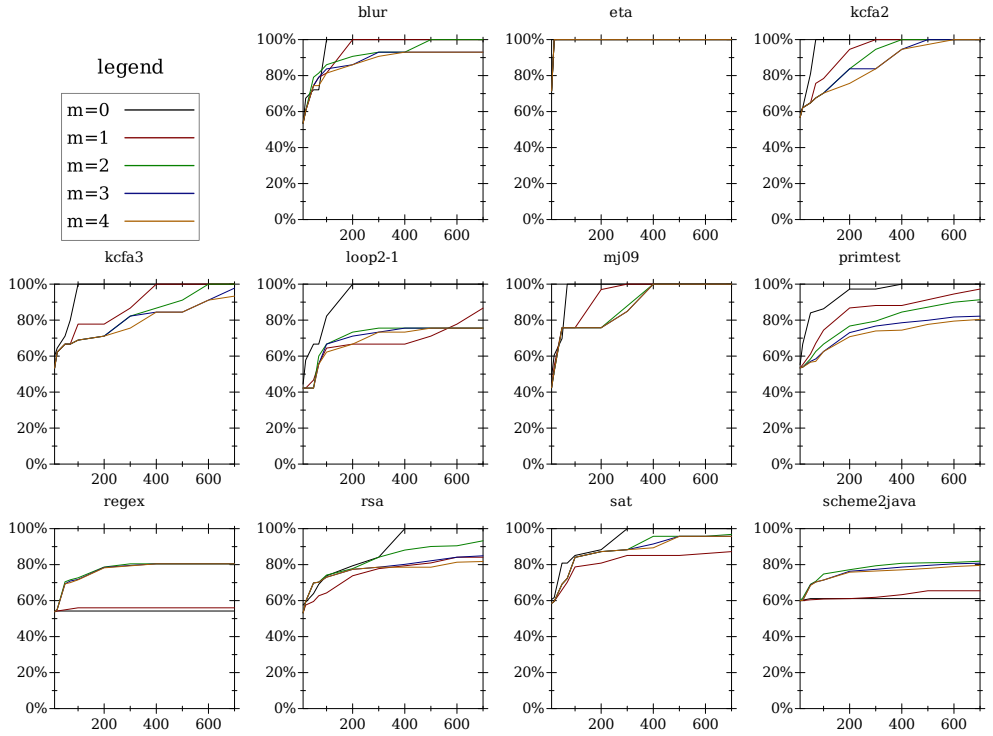

Fig. 9. The percent of queries answered (y-axis) given the effort allocated (x-axis) to Demand m-CFA per query. Each figure has a line for each setting of m. Many trivial queries (40%-60%) are answered given very little effort. In order to answer more queries sometimes it is enough to give the analysis more gas, while larger examples like scheme2java and regex require more context sensitivity to appropriately distinguish unrelated flows to avoid exploring the full control flow graph.

In some cases, Demand m-CFA doesn't find singleton flow sets that exhaustive m-CFA does. These cases are explained by reachability assumptions Demand m-CFA makes when resolving bindings. For instance, if Demand m-CFA is finding the callers of `f` in the expression `(λ (g) (f 42))` to discover arguments to `f`, 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 as imprecision.

The results show that Demand m-CFA generally finds a similar amount of singleton value flows as exhaustive m-CFA, and does it with a low effort bound.

9 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 refers to a class of 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 a context abstraction similar to that of m-CFA [Might et al. 2010].

The most-closely related work is that of Schoepe et al. [2023] which utilizes first-order solvers to construct call graphs on demand to effect higher-order analysis. Although it treats control with

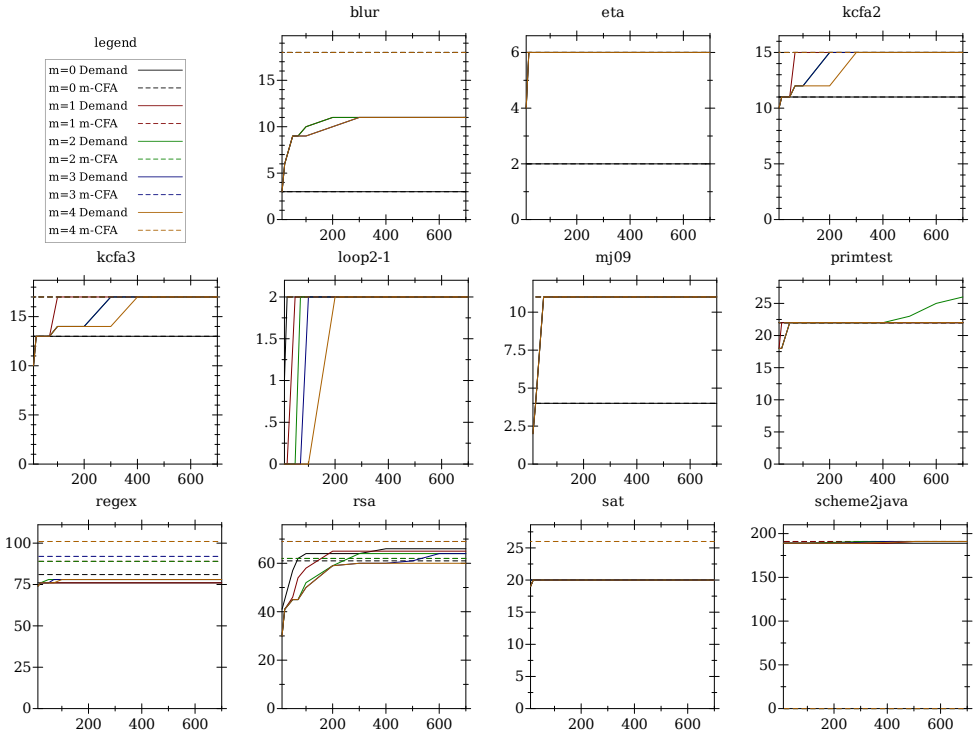

Fig. 10. The number of singleton flow sets (y-axis) found by a Demand m-CFA analysis given limited effort (x-axis). Dashed lines represent the baseline number of singleton flow sets found by an exhaustive exponential m-CFA analysis with a 10 minute timeout, regardless of effort. `scheme2java` doesn't have results for $m >= 1$ exhaustive m-CFA due to timing out.

context sensitivity, it does not model or track the environment structure of higher-order objects. Our work has a similar goal—to achieve demand-driven analysis of higher-order programs—but it has sought to provide a concise analysis which models both the control and environment structure of evaluation with equal facility.

DDPA [Palmer and Smith 2016] is nominally a context-sensitive, demand-driven analysis for higher-order programs. 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 expected of a demand analysis.

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.

Nicolay et al. [2019] produce a flow analysis where new information is propagated between interprocedural and intraprocedural parts of the program via the store. As such they incrementally build a reactive control flow graph on demand. In addition to read and write dependency tracking

their analysis explicitly tracks call effects which triggers a demand for functions stored at that point in the store to be evaluated interprocedurally with new arguments. However, while they present the theory for a context sensitive analysis, they do not present results for context sensitivity, and while their approach could be adapted to a demand based setting, their focus is on full program analysis.

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.

10 FUTURE WORK

Having established context sensitivity for Demand CFA, its primary limitation is its inability to reason about higher-order effectful computation. To address this limitation, we intend to develop a general operational framework in which a variety of effects can be expressed. We also intend to (1) investigate the tradeoff between precision and the nondetection of dead code within the analysis (occurring during binding resolution); (2) determine how to reuse fixpoint caches between queries

while keeping environments as indeterminate as possible to create an incremental analysis [Stein et al. 2021; Van der Plas et al. 2020], (3) consider how selective context sensitivity [Li et al. 2020] could be realized given the indeterminate environments of our approach.

REFERENCES

2019. Koka programming language. https://github.com/koka-lang/koka. Accessed: 2024-05-30.

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

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.

J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. 2014. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming* 24, 2–3 (2014), 218–283. https://doi.org/10.1017/S0956796814000100

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

Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. 2019. Effect-driven flow analysis. In *Verification, Model Checking, and Abstract Interpretation: 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13–15, 2019, Proceedings 20*. Springer, 247–274.

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.

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.

Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 282–295.

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

Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. 2020. Incremental Flow Analysis through Computational Dependency Reification. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 25–36. https://doi.org/10.1109/SCAM51674.2020.00008