Demand Control-Flow Analysis

Kimball Germane[1], Jay McCarthy[2], Michael D. Adams[3], and Matthew Might[4]

[1] Brigham Young University
kimball@cs.byu.edu
[2] University of Massachusetts Lowell
[3] University of Utah
[4] University of Alabama

Abstract. Points-to analysis manifests in a functional setting as control-flow analysis. Despite the ubiquity of *demand* points-to analyses, there are no analogous demand control-flow analyses for functional languages in general. We present demand 0CFA, a demand control-flow analysis that offers clients in a functional setting the same pricing model that demand points-to analysis clients enjoy in an imperative setting. We establish demand 0CFA's correctness via an intermediary exact semantics, demand evaluation, that can potentially support demand variants of more-precise analyses.

1 Introduction

Points-to analysis is a fundamental program analysis over languages that exhibit imperative or object-oriented features. A particular points-to analyses is specified as *exhaustive* or *demand*. An exhaustive points-to analysis calculates points-to facts for *every* variable reference in the program or component. In constrast, a demand points-to analysis calculates points-to facts for a client-specified set of variable references. A demand analysis that obtains points-to facts about a specified set and avoids analysis work that doesn't contribute thereto presents a pricing model to clients distinct from that of exhaustive analyses. As we discuss in the next section, this pricing model offers advantages to clients such as compilers and IDEs.

Control-flow analysis (CFA) is the analogue of points-to analysis in languages that offer first-class functions [12]. Unlike those of points-to analysis, however, the specifications of essentially all modern CFAs define *exhaustive* analyses that produce a comprehensive account of control flow for a target program or component. However, a demand CFA would offer clients in the functional setting many of the same advantages that a demand points-to analysis offers its clients in an imperative or object-oriented setting. This paper introduces *demand 0CFA*, a specification of a demand CFA. As a demand analysis, demand 0CFA resolves the (potentially higher-order) control flows of arbitrary client-specified subexpressions while avoiding analysis work that doesn't pertain to them. Previous demand analyses for functional languages offer limited demand behavior in higher-order settings [16] or apply only in limited higher-order settings [8];

in contrast, demand 0CFA offers full demand behavior in a general higher-order setting.

To a first approximation, demand 0CFA is achieved by extending exhaustive 0CFA with another mode of operation. Rather than (abstractly) evaluating every expression to its values as exhaustive 0CFA does, demand 0CFA in one mode evaluates some expressions to their values and in another traces some values to the expressions that take them on. At various points in analysis, the operation of each of these modes is informed by results of the other. To provide context for the additional tracing mode, we review exhaustive 0CFA in Section 4 before formally introducing demand 0CFA in Section 5; we briefly review the call-by-value λ calculus (their common language) in Section 3. We report on an evaluation of the efficiency and precision of demand 0CFA relative to 0CFA in Section 6. The connection between demand 0CFA and a ground-truth exact semantics is not as direct as that of exhaustive 0CFA; in Section 7, we bridge this connection with *demand evaluation*, a demand specification of *exact* evaluation. We discuss related work in Section 8 and future work in Section 9. In the next section, we overview the utility and operation of demand 0CFA.

2 Overview

Palsberg characterizes higher-order program analysis as the combination of first-order program analysis and control-flow analysis (under the name *closure analysis*) [17]. That is, in order to apply a first-order analysis to a higher-order program, one must be able to contend with higher-order control flow.

However, control-flow analysis is expensive. Even the least-expensive "full-precision" CFA—0CFA—has time complexity cubic in program size and this bound is unlikely to be decreased [21]. And, for some clients, control-flow information may be quickly obsoleted. For instance, both the transformations that optimizing compilers perform and the user edits made within a client IDE can invalidate analysis results [4]. This dynamic discourages potential clients of CFA, such as compilers and IDEs, even when the program insight it offers would be useful.

However, this dynamic is not rooted solely in the raw cost of CFA, but also in the pricing model it offers clients. Under this model, clients purchase an (exhaustive) conglomeration of control-flow facts for a large sum up front. For potential clients that forego CFA, the average utility (to the client) of a constituent control-flow fact must not outweigh the average cost (to the client). However, the fact that neither utility nor cost is constant across facts suggests that these clients could be better-served by an alternative pricing model. Before we discuss this pricing model in more detail, we will briefly discuss why neither the (1) utility nor (2) cost would be constant:

1. For clients that don't need *all* control-flow information to be effective, some control-flow facts are more valuable than others. For instance, optimizing compilers likely value facts regarding a critical path in the program at a premium over run-once code. Similarly, an IDE attempting to provide the

user with insight into a particular program part values information about that part higher than other parts.
2. The control flow at a particular program point has a kind of locality and not all program points exhibit the same locality. For instance, the target of f in the fragment $(\lambda f.(f\,x)\,\lambda y.e)$ has higher locality than it does in $(\lambda f.(f\,x)\,g)$. This locality can translate into the amount of analysis required to resolve the control flow [1]. This variation makes little difference to an exhaustive analysis, however, since such an analysis cannot selectively omit expensive facts.

2.1 The Demand Pricing Model

The demand pricing model allows clients to purchase analysis facts selectively. To illustrate some of the advantages of the demand pricing model for a functional language, let's consider super-β inlining [18]. Super-β inlining is the higher-order analogue of procedure inlining. Super-β inlining syntactically replaces the operator f in the call $(f\,x)$ with the target code (in terms of λ). For instance, if f always evaluates to a closure over $\lambda y.e$, the super-β inline of the call is $(\lambda y.e\,x)$, a form susceptible to further optimizations.

For the purposes of super-β inlining, the demand pricing model has several distinct advantages over the exhaustive pricing model; we discuss two:

1. Super-β inlining is not an essential optimization but also cannot be performed without flow information. Under an exhaustive pricing model, one inlining opportunity is revealed only if all inlining opportunities are revealed. In contrast, a demand pricing model allows clients to obtain control-flow information about individual program expressions without necessarily analyzing all other expressions.
2. Super-β inlining unlocks a potential cascade of optimizations as an inlined call site is simplified. In transforming the program, these optimizations can invalidate the CFA results of the original program. Under an exhaustive pricing model, these results were likely both comprehensive and expensive, resulting in a significant loss. In constrant, under the demand pricing model, the set of results invalidated is both small and relatively inexpensive.

In Section 6, we evaluate the fitness of demand 0CFA to super-β inlining in terms of its precision relative to exhaustive 0CFA.

2.2 Alternative Sources of the Demand Pricing Model

Demand analyses are confined to limited higher-order settings. For instance, demand points-to analysis for object-oriented languages has a rich literature (e.g. [20, 19]) but, especially as the same specification can have strikingly different manifestations in the object-oriented and functional settings [13], it is not clear that current techniques could port directly. The few demand analyses that have targeted functional languages directly each suffer from their own limitations. For

instance, Demand-Driven Program Analysis [16] constructs a call graph rooted at program entry and the subtransitive CFA of Heintze and McAllester [8] applies to typed programs with bounded types. In order to enable the whole host of demand first-order program analyses in a higher-order setting, general higher-order control flow must be directly addressed.

While not offering pure demand analysis, the CFA community has recognized the utility of and offered more-selective CFAs. Both Shivers' escape technique [18, Ch. 3] and Ashley and Dybvig's sub-0CFA [1] allow the client to delimit a region of the program to be analyzed. This option comes with its own difficulties: because each analysis is (willfully) blind to the actual control flow outside the region, selecting an appropriate region is critical but not straightforward. For instance, a region that is too small could omit some or even all of the sources of dependent value and control flow; on the other hand, a region that is too large wastes analysis effort obtaining irrelevant (to a particular question) control-flow facts. Exasperatingly, approaching optimal region selection in general likely requires control-flow analysis itself.

In a demand analysis, however, the region-selection problem is non-existent because the analyzer will traverse as much of the program as necessary to resolve the desired control flow. Because the reason the client initially was going to delimit a region was to minimize the analysis time, clients of demand analyses may impose time limits on the demand analyzer. Imposing a time limit rather than a region limit is a much better fit for the client as it was selecting the region to optimize for time, whereas in this arrangement it can optimize for time directly.

2.3 How Demand CFA Operates

An exhaustive CFA, regardless of whether it is based on abstract interpreters or flow constraints, proceeds in a kind of evaluation mode. To analyze a program, it starts at the top level and, like an evaluator, dispatches on the type of expression under focus. The analysis of each expression shadows its concrete evaluation: variable references bring the environment and result value into accord; λ-terms produce values themselves; and applications cause the analyzer to descend on the operator and argument and then operator body, once it's known. As we'll see, an evaluation-centric mode is inadequate to achieve a demand CFA. The key idea behind demand CFA is to introduce an additional *tracing* mode to the analyzer which performs the dual function of evaluation: where evaluation seeks the values to which an expression can evaluate, tracing seeks the expressions which evaluate to a particular value.

Clients specify a particular fact of interest to an analyzer by issuing a query. In the setting of an applicative functional language, queries take the form of subexpressions for which the client would like control-flow information. Issues arise, however, because a query may be an arbitrary subexpression and, in particular, depend on the resolution of a free reference to a variable x. In an exhaustive analysis, the flows to the binding of x are established before the evaluation of

any reference to it (which we discuss further in Section 4.1). A demand analysis has no such guarantee and must be prepared to establish the flows from this point. To do so, it first considers the way in which x is bound, which, in a lexically-scoped language, is syntactically apparent. For this example, suppose that it is bound by application of a closure over $\lambda x.e$. Next, it traces the flow of the closure over $\lambda x.e$ to each call site which applies it. Since x is bound by its application, the values of the arguments at those call sites constitute its value flows. The analyzer obtains these values by issuing evaluation subqueries for each argument expression.

Let's consider the resolution of the query `y` in the context of the program $(\lambda \texttt{f}\,\texttt{x}.(\texttt{f}\,\texttt{x})\,\lambda \texttt{y}.\texttt{y}\,\lambda \texttt{z}.\texttt{z})$. (In other words, we'll look at a demand approach to determining the values that the reference `y` can take on.) In order to resolve an evaluation query, the analyzer assumes evaluation mode and, accordingly, attempts to evaluate `y`. Since `y` is a variable reference and the analyzer doesn't have its binding available to it, it inspects the program to discover that it is the binder of λ`y.y`, bound when a closure over λ`y.y` is applied. At this point, the analyzer enters tracing mode, following the value of λ`y.y` to all of the places where it is applied. The way it traces a value flow manifests its duality to evaluation once again: evaluation dispatches on the type of expression but tracing dispatches on the type of syntactic context of the expression under focus. The analyzer observes that λ`y.y` is in an argument context and its value will be bound to an operator parameter. Its next task, then, is to obtain the operator value by shifting back into evaluation mode. Once it discovers the operator value to be a closure over $\lambda \texttt{f}\,\texttt{x}.(\texttt{f}\,\texttt{x})$, it can continue tracing the value of λ`y.y` through references to `f`. The only reference to `f` is in operator position in $(\texttt{f}\,\texttt{x})$ and $(\texttt{f}\,\texttt{x})$ constitutes a call site of the value of λ`y.y`. From here, the value bound to `y` can be obtained as the value of `x`. To resolve the value bound to `x`, the analyzer follows the same process as it did to resolve `y`, discovering the entire expression as a call site for the binding λ-term of `x` and λ`z.z` in the corresponding argument position. With λ`z.z`, the analyzer has determined that each reference to `y` evaluates to a closure over λ`z.z`.

In this example, the demand approach appears especially indirect since an exhaustive analysis that "evaluated" the call would shortly discover the value of `y`. Exhaustive analyses typically enjoy an economy of scale: they perform less work to obtain an analysis fact, on average, than a demand analysis [9]. Nevertheless, the selective nature of demand analyses can more than compensate for this overhead.

3 The call-by-value λ-calculus

In this section, we formally present the language that both 0CFA and demand 0CFA operate on. Its semantics serve as the ground truth for the correctness theorems of demand 0CFA in Section 7.

Following Nielson *et al.*, expressions e in our language are labelled terms t^ℓ and terms take the form of variable references x, λ-abstractions $\lambda x.e$, and applications $(e_0\, e_1)$. Formally, we have

$$\begin{array}{cc} x \in \mathbf{Var} & \ell \in \mathbf{Lab} \\ e \in \mathbf{Exp} & t \in \mathbf{Term} \end{array}$$

$$e ::= t^\ell \qquad t ::= x \mid \lambda x.e \mid (e_0\, e_1)$$

Variables and labels are drawn from the disjoint infinite sets **Var** and **Lab**, respectively. Labels are unique and therefore distinguish otherwise identical terms; we omit them when unnecessary.

We define an environment-based call-by-value semantics in big-step style which relates *configurations* (ρ, e, c) to values v when an expression e evaluates to a value v under an environment ρ and calling context c. We denote this relationship by the judgment $\rho, c \vdash e \Downarrow v$. In this simple language, the only form of value is that of a closure $(\lambda x.e, \rho)$, a pair of a λ-abstraction and its closing environment, where environments are a finite map from variables x to values v. Calling contexts are finite sequences of labels. Formally, values, environments, and calling contexts are given as

$$v ::= (\lambda x.e, \rho) \qquad \rho ::= \bot \mid \rho[x \mapsto v] \qquad c ::= \langle\rangle \mid \ell :: c$$

The semantic relation over configurations and values is defined by three rules, one for each type of syntactic expression.

$$\textsc{Ref}\ \frac{}{\rho, c \vdash x^\ell \Downarrow \rho(x)} \qquad \textsc{Lam}\ \frac{}{\rho, c \vdash (\lambda x.e)^\ell \Downarrow (\lambda x.e, \rho)}$$

$$\textsc{App}\ \frac{\rho, c \vdash e_0 \Downarrow (\lambda x.e, \rho') \quad \rho, c \vdash e_1 \Downarrow v' \quad \rho'[x \mapsto v'], \ell :: c \vdash e \Downarrow v}{\rho, c \vdash (e_0\, e_1)^\ell \Downarrow v}$$

These rules are standard: the Ref rule states that a variable reference evaluates to the value bound to that variable in the environment; the Lam rule states that a λ-abstraction evaluates to a closure, pairing it with its environment; and the App rule states that an application evaluates to the body of the operator value under the operator environment extended to bind the operator parameter to the argument value. We also assume that premises are established from the left to right.

4 Background: 0CFA

In this section, we review the core of the constraint-based formulation of 0CFA presented by Nielson *et al.* [15]. We consider the analysis over only the unary λ-calculus presented in the previous section but it is straightforward to extend it to a richer language (see Nielson *et al.* [15]).

A 0CFA analysis is a pair $(\hat{\mathcal{C}}, \hat{\rho})$ where $\hat{\mathcal{C}}$ is an *abstract cache* that associates to each expression e an abstract value and $\hat{\rho}$ is an *abstract environment* that

associates to each variable x an abstract value. The intent is that $\hat{\mathcal{C}}$ associates to each expression e an abstraction of those values to which e can evaluate and that $\hat{\rho}$ associates to each variable x an abstraction of those values to which x can be bound during evaluation. An abstract value $\hat{v}$ is a set of λ-terms which yields the following functionalities

$$\begin{array}{lll} \hat{v} \in \widehat{\mathbf{Val}} & = \mathcal{P}(\mathbf{Lam}) & \text{abstract values} \\ \hat{\rho} \in \widehat{\mathbf{Env}} & = \mathbf{Var} \to \widehat{\mathbf{Val}} & \text{abstract environments} \\ \hat{\mathcal{C}} \in \widehat{\mathbf{Cache}} & = \mathbf{Lab} \to \widehat{\mathbf{Val}} & \text{abstract caches} \end{array}$$

where **Var**, **Lab**, and **Lam** are the variables, labels, and λ-terms of the analyzed program and, hence, are finite.

[Ref]	$(\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} x^{\ell}$	*iff* $\hat{\rho}(x) \subseteq \hat{\mathcal{C}}(\ell)$
[Lam]	$(\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} (\lambda x.e)^{\ell}$	*iff* $\{\lambda x.e\} \subseteq \hat{\mathcal{C}}(\ell)$
[App]	$(\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} (t_0^{\ell_0}\ t_1^{\ell_1})^{\ell}$	*iff* $(\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} t_0^{\ell_0} \wedge (\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} t_1^{\ell_1} \wedge$ $\forall \lambda x.t^{\ell_2} \in \hat{\mathcal{C}}(\ell_0), (\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} t^{\ell_2} \wedge \hat{\mathcal{C}}(\ell_2) \subseteq \hat{\mathcal{C}}(\ell) \wedge \hat{\mathcal{C}}(\ell_1) \subseteq \hat{\rho}(x)$

Fig. 1. The $\models_{fs}$ relation

What constitutes a 0CFA analysis of a program is defined by a relation over analyses and programs; Figure 1 defines one such relation by means of a set of clauses, one for each category of expression:

The Ref *clause* The Ref clause relates $(\hat{\mathcal{C}}, \hat{\rho})$ to a reference x^{ℓ} if $\hat{\rho}(x) \subseteq \hat{\mathcal{C}}(\ell)$. For a closed program, the $\models_{fs}$ relation only considers a reference x^{ℓ} after it has ensured that $\hat{\rho}(x)$ abstracts all of the values to which x could be bound.

The Lam *clause* The Lam clause relates $(\hat{\mathcal{C}}, \hat{\rho})$ to a λ-term $(\lambda x.e)^{\ell}$ if $\{\lambda x.e\} \subseteq \hat{\mathcal{C}}(\ell)$. This clause leaves implicit the fact that, prior to ensuring that this constraint holds, the $\models_{fs}$ specification ensures that $\hat{\rho}(x)$ is populated appropriately for every variable y with a free reference in $\lambda x.e$.

The App *clause* The work of $\models_{fs}$ is done by the App clause, which relates $(\hat{\mathcal{C}}, \hat{\rho})$ to an application $(t_0^{\ell_0}\ t_1^{\ell_1})^{\ell}$ if several conditions hold. First, $\models_{fs}$ must relate $(\hat{\mathcal{C}}, \hat{\rho})$ to both the operator $t_0^{\ell_0}$ and argument $t_1^{\ell_1}$. Second, for every $\lambda x.e$ in the operator cache $\hat{\mathcal{C}}(\ell_0)$, the constraint $\hat{\mathcal{C}}(\ell_2) \subseteq \hat{\mathcal{C}}(\ell)$, ensuring that values of the call include those of the function body, and the constraint $\hat{\mathcal{C}}(\ell_1) \subseteq \hat{\rho}(x)$, ensuring that the values bound to x include those of the argument, must both hold.

When $(\hat{\mathcal{C}}, \hat{\rho}) \models_{fs} pr$ holds, we say that $(\hat{\mathcal{C}}, \hat{\rho})$ is *acceptable* with respect to pr. Acceptability implies soundness, so, for an acceptable analysis $(\hat{\mathcal{C}}, \hat{\rho})$, for every label ℓ of a term t, $\hat{\mathcal{C}}(\ell)$ abstracts every value to which t evaluates and, for every variable x, $\hat{\rho}(x)$ abstracts every value to which x becomes bound. We will not review how to arrive at an acceptable analysis, but the interested reader may consult Nielson *et al.* [15].

4.1 An Inherently-Exhaustive Specification

The $\models_{fs}$ relation inherently specifies an exhaustive analysis. The crux is essentially that the notion of acceptability it defines (which entails soundness) holds only for closed programs; it cannot make guarantees about an analysis related to an open expression. For instance, what analyses are related to the lone free variable $\mathtt{x}^2$? According to the specification, $(\bot, \bot) \models_{fs} \mathtt{x}^2$ holds since $\bot(\mathtt{x}) = \emptyset \subseteq \emptyset = \bot(2)$. But the analysis $(\bot, \bot)$ doesn't capture the flow behavior of $\mathtt{x}^2$ as it appears in $((\lambda \mathtt{x}.\mathtt{x}^2)^1\,(\lambda \mathtt{y}.\mathtt{y}^4)^3)^0$. The $\models_{fs}$ relation relies on previously-imposed constraints to populate the environment mapping of $\mathtt{x}$ and thus appropriately constrain $\mathtt{x}^2$. Without special provision, this occurs only if the binding term of $\mathtt{x}$ is processed before the reference $\mathtt{x}^2$ is encountered. Since $\models_{fs}$ relies on this for every variable, it accurately defines acceptability for only closed programs in general.

5 Demand 0CFA

Demand 0CFA specifies what it means for an analysis to be acceptable with respect to a (possibly open) program subexpression. In other words, this specification ensures that an analysis accounts for all values to which a subexpression may evaluate, even if that subexpression has free variables. Demand 0CFA strives to analyze only those parts of the program needed to obtain a sound result for the target subexpression, though additional expressions are often implicated by control or value dependencies.

A *demand 0CFA analysis* is a pair $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ where $\hat{\mathcal{C}}$ has the same form as in 0CFA and $\hat{\mathcal{E}}$ is an *abstract callers* relation which associates to λ-term body expressions e a set of call sites $(e_0\, e_1)$. In demand 0CFA, $\hat{\mathcal{C}}$ does not necessarily (nor typically) associate every expression to an abstract value but only those necessary to determine the control flow of a externally-selected expression. The intent is that $\hat{\mathcal{E}}$ associates to each of certain λ-terms (also determined by an externally-selected expression) the set of call sites that apply (closures over) it. As in Nielson *et al.* [15], an abstract value $\hat{v}$ is a set of λ-terms, yielding the following functionalities:

$$\begin{aligned}
\hat{v} \in \widehat{\mathbf{Val}} &= \mathcal{P}(\mathbf{Lam}) \\
\hat{c} \in \widehat{\mathbf{App}} &= \mathcal{P}(\mathbf{App}) \\
\hat{\mathcal{C}} \in \widehat{\mathbf{Cache}} &= \mathbf{Lab} \to \widehat{\mathbf{Val}} \\
\hat{\mathcal{E}} \in \widehat{\mathbf{Calls}} &= \mathbf{Exp} \to \widehat{\mathbf{App}}
\end{aligned}$$

Just as with exhaustive 0CFA, there is the notion of *acceptability* for a demand 0CFA analysis. Rather than being acceptable w.r.t. a program, however, a demand 0CFA analysis is acceptable w.r.t. a query. The relation $\models_{fs\,eval}$ relates an analysis $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ to an expression e when $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ is acceptable for the evaluation of e. This means that $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ entails the evaluation and tracing necessary to evaluate e. Similarly, the relation $\models_{fs\,call}$ relates an analysis $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ to an occurrence $(\lambda x.e_0, e)$ when $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ is acceptable for the trace of $\lambda x.e_0$ from e. This means

that $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ entails the evaluation and tracing necessary to trace $\lambda x.e_0$ from e. We discuss each relation in more detail below.

The demand 0CFA specification makes use of the *syntactic context* of expressions provided by a function $\mathbb{K}_{pr} : \mathbf{Exp} \to \mathbf{Ctx}$ where

$$\mathbf{Ctx} \ni ctx ::= \Box \mid (\Box\, e)^\ell \mid (e\, \Box)^\ell \mid (\lambda x.\Box)^\ell$$

That is, the syntactic context ctx of an expression e within a program pr is either the top-level context $\Box$, an operator context $(\Box\, e_1)^\ell$, an argument context $(e_0\, \Box)^\ell$, or an abstraction body context $(\lambda x.\Box)^\ell$. (From now on, we will leave pr implicit.) The syntactic context can also be seen as an inherited attribute at the node of an expression e within a program's abstract syntax tree.

5.1 The $\models_{fseval}$ relation

[Ref]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} x^\ell$	*iff*	$(\lambda x.e)^{\ell_b} = \mathrm{bind}_{fs}(x, x^\ell) \wedge (\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, (\lambda x.e)^{\ell_b}) \wedge$ $\forall (t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2} \in \hat{\mathcal{E}}(e), (\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} t_1^{\ell_1} \wedge \hat{\mathcal{C}}(\ell_1) \subseteq \hat{\mathcal{C}}(\ell)$
[Lam]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} (\lambda x.e)^\ell$	*iff*	$\{\lambda x.e\} \subseteq \hat{\mathcal{C}}(\ell)$
[App]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} (t_0^{\ell_0}\, t_1^{\ell_1})^\ell$	*iff*	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} t_0^{\ell_0} \wedge$ $\forall \lambda x.t_2^{\ell_2} \in \hat{\mathcal{C}}(\ell_0), (\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} t_2^{\ell_2} \wedge \hat{\mathcal{C}}(\ell_2) \subseteq \hat{\mathcal{C}}(\ell)$

Fig. 2. The $\models_{fseval}$ relation

The relation $\models_{fseval}$ relates an analysis $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ to an expression t^ℓ. Its purpose is to ensure that $\hat{\mathcal{C}}(\ell)$ contains an abstraction of all the values to which t^ℓ can evaluate; in this sense, it corresponds to the $\models_{fs}$ relation of exhaustive 0CFA. The definition of $\models_{fseval}$ can be seen in Figure 2 and, like $\models_{fs}$, includes a clause for each syntactic category of the λ-calculus.

The Ref *clause* To determine the values to which a reference x may evaluate, the Ref clause uses the bind_{fs} metafunction, defined in Figure 3, to reconstruct the binding λ-term $\lambda x.e$. It then relies on the $\models_{fscall}$ relation to ensure that each call site of (closures over) $\lambda x.e$ is known. For each such call site, the Ref clause constrains the reference x to evaluate to each value to which the argument may evaluate.

The Lam *clause* A λ-term $\lambda x.e$ evaluates to only closures over itself, so the Lam clause of $\models_{fseval}$ is the same as the Lam clause of $\models_{fs}$. However, unlike $\models_{fs}$, the $\models_{fseval}$ relation does not assume at this point that the free variables of $\lambda x.e$ are subject to any constraints to ensure they're bound.

The App *clause* The App clause ensures that the operator is evaluated, that the body of each of its values is evaluated, and that the application itself takes on each of the body values. Unlike the App clause of $\models_{fs}$, the App clause of $\models_{fseval}$ doesn't evaluate the argument nor bind its values in the operator's. If the argument value is needed, the Ref clause will obtain it.

$$
\begin{array}{ll}
\mathrm{bind}_{fs}(x, e) = \mathrm{bind}_{fs}(x, (e\, e')^{\ell}) & \text{if } \mathbb{K}(e) = (\square\, e')^{\ell} \\
\mathrm{bind}_{fs}(x, e) = \mathrm{bind}_{fs}(x, (e'\, e)^{\ell}) & \text{if } \mathbb{K}(e) = (e'\, \square)^{\ell} \\
\mathrm{bind}_{fs}(x, e) = (\lambda x.e)^{\ell} & \text{if } \mathbb{K}(e) = (\lambda x.\square)^{\ell} \\
\mathrm{bind}_{fs}(x, e) = \mathrm{bind}_{fs}(x, (\lambda y.e)^{\ell}) & \text{if } \mathbb{K}(e) = (\lambda y.\square)^{\ell} \text{ where } x \neq y
\end{array}
$$

Fig. 3. Given a variable x and an expression e in which x appears free, the bind_{fs} metafunction reconstructs the binding λ-term of x (of which e is a subexpression) by walking upward on the program syntax tree until it encounters the binder of x. Because whole programs are closed (and, we assume, demand 0CFA has access to them), bind_{fs} will always encounter the binder of x before it reaches the program top level. To perform demand 0CFA over components with free variables, bind_{fs} could be altered to signal the occurrence of one to the analyzer, which might apply, e.g., Shivers' escape technique [18].

5.2 The $\models_{fscall}$ relation

[Rator]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, t_0^{\ell_0})$ for $\mathbb{K}(t_0^{\ell_0}) = (\square\, t_1^{\ell_1})^{\ell_2}$	*iff*	$\{(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}\} \subseteq \hat{\mathcal{E}}(e)$
[Rand]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, t_1^{\ell_1})$ for $\mathbb{K}(t_1^{\ell_1}) = (t_0^{\ell_0}\, \square)^{\ell_2}$	*iff*	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fseval} t_0^{\ell_0} \wedge$ $\forall \lambda y.e' \in \hat{\mathcal{C}}(\ell_0), \forall \ell \in \mathrm{find}_{fs}(y, e'), (\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, y^{\ell})$
[Body]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, t^{\ell})$ for $\mathbb{K}(t^{\ell}) = (\lambda y.\square)^{\ell_y}$	*iff*	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda y.t^{\ell}, (\lambda y.t^{\ell})^{\ell_y}) \wedge$ $\forall (e_0\, e_1)^{\ell_2} \in \hat{\mathcal{E}}(t^{\ell}), (\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, (e_0\, e_1)^{\ell_2})$
[Top]	$(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fscall} (\lambda x.e, t^{\ell})$ for $\mathbb{K}(t^{\ell}) = \square$	*iff*	always

Fig. 4. The $\models_{fscall}$ relation

The relation $\models_{fscall}$ relates an analysis $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ to an *occurrence* $(\lambda x.e, t^{\ell})$ which denotes that t^{ℓ} evaluates (in some context) to a closure over $\lambda x.e$. Its purpose is to ensure that $\hat{\mathcal{E}}(e)$ contains every call site $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}$ which may apply a closure over $\lambda x.e$, as it flowed from t^{ℓ}. In order to trace the value flow from t^{ℓ}, $\models_{fscall}$ considers the syntactic context of t^{ℓ}, which reveals its next occurrence. Accordingly, the definition of $\models_{fscall}$, seen in Figure 4, includes a clause for each syntactic (expression) context: operator, argument, λ-term body, and top-level.

The Rator *clause* When $\lambda x.e$ occurs at the operator $t_0^{\ell_0}$ of the application $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}$, $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}$ is a caller of a closure over $\lambda x.e$. The Rator clause ensures that, in such cases, $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2} \in \hat{\mathcal{E}}(e)$.

The Rand *clause* When $\lambda x.e$ occurs at the argument $t_1^{\ell_1}$ of the application $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}$, it will be bound to x_2 for each closure over $\lambda y.e'$ to which $t_0^{\ell_0}$ evaluates. The Rand clause ensures that $t_0^{\ell_0}$ is evaluated and, for each closure over $\lambda y.e'$ to which it evaluates, uses the find_{fs} metafunction, defined in Figure 5, to locate

references to y in e'. The RAND clause then ensures that $\lambda x.e$ occurs at each such reference.

$$\begin{array}{lcll}
\mathrm{find}_{fs}(x, x^{\ell}) & = & \{\ell\} & \\
\mathrm{find}_{fs}(x, y^{\ell}) & = & \emptyset & \text{if } x \neq y \\
\mathrm{find}_{fs}(x, (\lambda x.e)^{\ell}) & = & \emptyset & \\
\mathrm{find}_{fs}(x, (\lambda y.e)^{\ell}) & = & \mathrm{find}_{fs}(x, e) & \text{if } x \neq y \\
\mathrm{find}_{fs}(x, (f\ e)^{\ell}) & = & \mathrm{find}_{fs}(x, f) \cup \mathrm{find}_{fs}(x, e) &
\end{array}$$

Fig. 5. Given a variable x and an expression e in its scope, the find_{fs} metafunction gathers the references to x within e by descending downward on the program syntax tree.

The BODY *clause* When $\lambda x.e$ occurs at the body t^{ℓ} of a λ-term $(\lambda y.t^{\ell})^{\ell_y}$, a closure over $\lambda x.e$ will be the result of a call to a closure over $\lambda y.t^{\ell}$. In other words, $\lambda x.e$ will also occur at each caller $(e_0\, e_1)^{\ell_2}$ of $\lambda y.t^{\ell}$. The BODY clause ensures that the callers of $\lambda y.t^{\ell}$ are known and that $\lambda x.e$ occurs at each of them.

The TOP *clause* When $\lambda x.e$ occurs at an expression t^{ℓ} with context $\Box$, it has reached the top level of the program or component. If this top level is of the entire program, such an occurrence means that the result of the program is a closure over $\lambda x.e$ and that it is not applied along this flow. If this top level is of only a component, such an occurrence means that a closure over $\lambda x.e$ escapes the component and can signal the analyzer to respond appropriately.

5.3 Constraint Generation

The constraint generation process of demand 0CFA is very similar to that of exhaustive 0CFA: it proceeds by recursion over the definitions of $\models_{fseval}$ and $\models_{fscall}$ using memoization to avoid revisiting any particular relation. The constraints themselves, however, differ substantially. Evaluation of x^{ℓ} with x bound as $\lambda x.e$ generates the constraint $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2} \in \hat{\mathcal{E}}(e) \implies \hat{\mathcal{C}}(\ell_1) \subseteq \hat{\mathcal{C}}(\ell)$. Like exhaustive 0CFA, evaluation of $(\lambda x.e)^{\ell}$ generates the constraint $\{\lambda x.e\} \subseteq \hat{\mathcal{C}}(\ell)$. Evaluation of $(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}$ generates the constraint $\lambda x.t^{\ell} \in \hat{\mathcal{C}}(\ell_0) \implies \hat{\mathcal{C}}(\ell) \subseteq \hat{\mathcal{C}}(\ell_2)$. Evaluation of $t_0^{\ell_0}$ to a closure over $\lambda x.e$ in syntactic context $(\Box\, t_1^{\ell_1})^{\ell_2}$ generates the constraint $\{(t_0^{\ell_0}\, t_1^{\ell_1})^{\ell_2}\} \subseteq \hat{\mathcal{E}}(e)$.

6 Evaluation

In this section, we evaluate whether

1. demand 0CFA is essentially as precise as exhaustive 0CFA, and

2. a non-trivial fraction of control-flow information is available for relatively low cost.

In each evaluation, we use the same corpus of 30 programs. The corpus was obtained by using a random program generator and filtering to include only those programs (1) consist of between 2,500 and 10,000 expressions and (2) take (individually) over five seconds for exhaustive 0CFA to analyze. The corpus consists of the first 30 programs encountered by this technique. While this corpus may not be representative of of real-world programs, the second criterion ensures that all programs within it exhibit non-trivial control flow (at least from the perspective of 0CFA). The time that exhaustive 0CFA takes to analyze each program is used as that program's baseline in proceeding evaluations.

6.1 The Relative Precision of Demand 0CFA

Control-flow analysis is necessary to justify valuable optimizations such as super-β inlining [18]. This particular optimization, when applied at a call site $(f\, e)$, requires that f evaluates to closures over only a single λ-term. A CFA demonstrates this condition when it calculates a singleton flow set for f.

Demand 0CFA sometimes considers unreachable code and therefore calculates a larger control-flow relation than exhaustive 0CFA does. (We discuss this further in Section 7.) This could undermine its ability to justify optimizations relative to exhaustive 0CFA if exhaustive 0CFA calculates a singleton flow set for an expression but demand 0CFA fails to. We compare, for each program, the number of reachable non-λ-term expressions for which exhaustive 0CFA calculates a singleton flow set to the number of those for which demand 0CFA calculates a singleton flow set. We omit an (uninteresting) table as data shows that, for our corpus, there are very few cases in which demand 0CFA doesn't calculate a singleton flow set when exhaustive 0CFA does: for two programs, it does so in about 98% of cases; for the remaining 28 programs, it does so in at least 99% of cases; and for six programs, it does so for 100% of cases. If λ-term expressions were included in these counts, theses percentages would uniformly increase since both exhaustive 0CFA and demand 0CFA always calculate a singleton set for them. These results demonstrate the demand 0CFA is essentially as precise as exhaustive 0CFA.

6.2 The Existence of Cheap Control-Flow Information

With an exhaustive CFA, it doesn't make sense to talk about the cost to obtain any given piece of control-flow information since, by design, exhaustive CFA bundles all control-flow information together. We can however talk about the *MCE*, the mean cost per expression as the quotient of the cost of the bundle and the number of program expressions whose control-flow information it includes. For example, Program 5 has 5,338 expressions and takes exhaustive 0CFA 5.84 seconds to analyze, so its MCE in terms of time is $5.84s/5338 \approx 1.09ms$.

The intuition presented in Section 2 suggests that the locality of control flow varies across expressions. If the locality of control flow is a proxy for the cost of obtaining it, then this cost varies across expressions as well. In turn, a varying cost means that the control-flow information for some expressions may be obtainable at sub-MCE cost. In real terms, if these assumptions hold, we would expect that the control-flow information of some of the expressions in Program 5 would be obtainable for less than $1.09ms$.

For this evaluation, we limit the running time of demand 0CFA to a fraction α of the MCE for each program. We then dispatch demand 0CFA on each program expression in succession with analysis of each expression subject to this limit. The analysis of an expression either succeeds, yielding a sound account of its control-flow information, or reaches the limit, yielding no information, before we dispatch demand 0CFA on a successive expression. Figure 6 shows the percentage of expressions for which demand 0CFA succeeds in this manner under time limits determined by various fractions α.

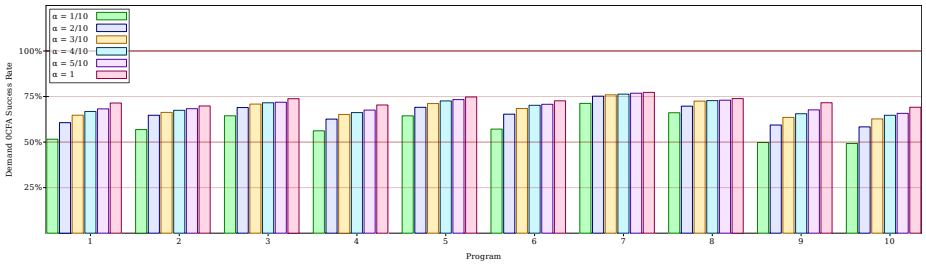

Fig. 6. This graph shows the demand 0CFA success rate for fractions $\alpha = 1/10, 2/10, 3/10, 4/10, 5/10, 1$ of the MCE for 10 programs randomly selected from our corpus. These data show that demand 0CFA can obtain the control flow for a significant fraction of expressions—on average over 50%—when time-limited to 1/10 of MCE. As expected, this fraction increases as the fraction of MCE increases, nearing 75% at when demand 0CFA is time-limited MCE itself. These results imply that nearly 75% of a program's control flow is obtainable for an order-of-magnitude less time than taken by exhaustive 0CFA. (See text.)

We report the median percentage of three runs for 10 programs programs randomly selected from our corpus and for $\alpha = 1/10, 2/10, 3/10, 4/10, 5/10, 1$. As the graph shows, demand 0CFA can obtain the control flow for a significant fraction of expressions—on average over 50%—when time-limited to 1/10 of MCE. As expected, this fraction increases as the fraction of MCE increases, nearing 75% at when demand 0CFA is time-limited MCE itself.

A first-order upper bound to demand 0CFA's relative running time is its fraction α. If demand 0CFA is time-limited to 1/10 of MCE, then, even if it is dispatched on every program expression, it will not take more than 1/10 of the time of exhaustive 0CFA. However, as the data show, the vast majority of expressions analyzable in 2/10 of MCE are analyzable in 1/10 of MCE, and

similarly for 3/10 relative to 2/10, etc. Using this fact, we can obtain a second-order upper bound A to demand 0CFA's relative running time via the formula

$$A = \sum_{i=1}^{n} \alpha_i (f_i - f_{i-1})$$

given a sequence of fractions $\alpha_1, \alpha_2, \ldots, \alpha_n$ of MCE and corresponding fractions $f_1, f_2, \ldots, f_n$ of demand 0CFA success rates, where $\alpha_0 = 0$ and $f_0 = 0$. By this estimate, when $\alpha = 1$, demand 0CFA on average obtains nearly 75% of a program's control flow in approximately 11% of the time taken by exhaustive 0CFA.

In practice, compilers can't limit demand 0CFA to a fraction of MCE, because the MCE is determined only by running an exhaustive 0CFA analysis. For instance, a compiler of Program 5 would not know that its MCE was $1.01ms$ since it would not know the time taken by exhaustive 0CFA was $5.84s$. Instead, it might know simply that it has $0.5s$ to budget to demand 0CFA. To increase effectiveness, it might allocate this budget non-uniformly across the program, using program knowledge to concentrate it on performance-critical program parts.

These results demonstrate not only that some pieces of control-flow information indeed cost less than MCE to obtain but also that

1. a significant fraction cost an order of magnitude less than MCE to obtain and
2. demand 0CFA can efficiently obtain them taking an order of magnitude less than MCE.

7 Demand 0CFA Correctness

The purpose of this section is to establish that demand 0CFA is sound w.r.t. an exact forward semantics. To do so, we will establish that demand 0CFA is sound w.r.t. an exact *demand* semantics and that this demand semantics has a formal correspondence to an exact forward semantics.

We term the exact demand semantics we define *demand evaluation*. To a first approximation, demand evaluation computes a subderivation of a full derivation of a program's evaluation, where the particular subderivation computed is determined in part by the program subexpression. This is only an approximate description of the action of demand evaluation for a few reasons: first, one may successfully apply demand evaluation to unreachable program subexpressions, computing derivations that don't appear in the derivation of the whole program's evaluation; second, the product of demand evaluation isn't necessarily a single contiguous subderivation but may instead be a set of related subderivations obtained (conceptually) by removing irrelevant judgments from a larger derivation.

Although our intuition is rooted in derivations, we formalize demand evaluation as exact demand analyses related to program configurations, analogous to how we formalized demand 0CFA as approximate demand analyses related

to program expressions. Doing so decreases the conceptual distance between demand 0CFA and demand evaluation, making soundness easier to establish. In order to connect demand evaluation to forward evaluation, we reify derivations from exact demand analyses and formally establish a correspondence between those derivations and forward derivations in a technical report [6].

7.1 Demand Evaluation

We define demand evaluation in the same way that we define demand 0CFA: namely, we define an *exact analysis* $(C, \mathcal{E})$ that records exact evaluation facts and two (undecidable) relations, $\models_{eval}$ and $\models_{call}$, over exact analyses and configurations (ρ^d, e, n_c). Both $\models_{eval}$ and $\models_{call}$ relate analyses and configurations in an analogous way to $\models_{fseval}$ and $\models_{fscall}$.

Figure 7 contains formal definitions for the domains of demand evaluation. An exact analysis $(C, \mathcal{E})$ consists of a *cache* C and a caller relation $\mathcal{E}$. In the exact semantics, a caller relation $\mathcal{E}$ is actually a function from called contexts to caller contexts. A cache C is itself a triple $(\$, \sigma, \nu)$ of three functions: $\$$ associates configurations with *names*, σ associates names with values, and ν associates names with calling contexts. A name n serves the function of an address that can be known and transmitted before anything is bound to it; it can be used to obtain the value that may eventually be bound to it in σ. The domain **Name** can be any countably-infinite set; when we must be concrete, it will assume the set of natural numbers $\mathbb{N}$. Demand evaluation environments map variables to names (and not to values contra environments in the exact, big-step semantics presented in Section 2) which are resolved in the store σ. Similarly, calling contexts are indirected by names which are resolved in the context store ν. (Names within environments and names in contexts come from different address spaces and never interact.) Closures remain the only type of value and remain a pair $(\lambda x.e, \rho^d)$ of a λ-term $\lambda x.e$ and an enclosing environment ρ^d. Configurations are a triple (ρ^d, e, n) of an environment ρ^d closing an expression e in a calling context named by n (and resolved through ν). In the exact semantics, an occurrence is merely a configuration, but we ensure that every configuration treated as such has a value in C.

An exact demand analysis encapsulates the evaluation of a given configuration. However, because configuration environments and calling contexts are threaded through stores, all configurations for a given expression have the same shape. In consequence, a configuration does not uniquely identify a particular evaluation for an expression—that is, the evaluation of a particular instance of the expression in evaluation. Instead, we will define our acceptability relations $\models_{eval}$ and $\models_{call}$ to admit analyses that encapsulate *some* evaluation of the given configuration.

7.2 The $\models_{eval}$ relation

The $\models_{eval}$ relation relates an analysis $(C, \mathcal{E})$ to a judgment $\rho^d, n_c \vdash e \Downarrow^d v^d$ when $(C, \mathcal{E})$ entails the evaluation of the configuration (ρ^d, e, n_c) to the value v^d. Its

$$
\begin{array}{rcrc}
n & \in & \mathbf{Name} & \text{a countably-infinite set} \\
\rho^d & \in & \mathbf{Env}_d & ::= \bot \mid \rho^d[x \mapsto n] \\
(\lambda x.e, \rho^d) & \in & \mathbf{Value} & = \mathbf{Lam} \times \mathbf{Env}_d \\
c^d & \in & \mathbf{CCtx}_d & ::= \mathrm{mt} \mid \ell :: n \\
(\rho^d, e, n) & \in & \mathbf{Config} & = \mathbf{Env}_d \times \mathbf{Exp} \times \mathbf{Name} \\
 & & \mathbf{Occur} & = \mathbf{Config} \\
\$ & \in & \mathbf{Names} & = \mathbf{Config} \to \mathbf{Name} \\
\sigma & \in & \mathbf{Store} & = \mathbf{Name} \to \mathbf{Value} \\
\nu & \in & \mathbf{CStore} & = \mathbf{Name} \to \mathbf{CCtx}_d \\
C & \in & \mathbf{Cache} & = \mathbf{Names} \times \mathbf{Store} \times \mathbf{Store} \\
\mathcal{E} & \in & \mathbf{Calls} & = \mathbf{Config} \to \mathbf{Config}
\end{array}
$$

Fig. 7. Demand evaluation domains

definition, seen in Figure 8, contains a clause for each type of expression. Each of these clauses functions essentially as its counterpart does in $\models_{fseval}$.

The Ref *clause* The Ref clause specifies that an analysis $(C, \mathcal{E})$ entails the evaluation of a variable reference configuration (ρ^d, x^ℓ, n_c). It ensures that such a configuration evaluates to a value v^d when (1) the name of the configuration reflects the name of the environment binding, (2) the closure that created that binding when applied (furnished by bind) is called at a call site, (3) the name of the argument configuration at that call site is consistent with the new environment binding, and (4) the argument configuration evaluates to v^d.

The bind metafunction (defined in Figure 9) reconstructs not simply the binding λ-term $\lambda x.e$ of x but the configuration at which the particular closure over $\lambda x.e$ first appears.

The Lam *clause* The Lam clause of $\models_{eval}$ specifies that an analysis $(C, \mathcal{E})$ entails the evaluation of a λ-term configuration $(\rho^d, (\lambda x.e)^\ell, n_c)$ if $C(\rho^d, (\lambda x.e)^\ell, n_c) = (\lambda x.e, \rho^d)$ meaning that $C_\$(\rho^d, (\lambda x.e)^\ell, n_c) = n$ and $\sigma(n) = (\lambda x.e, \rho^d)$ for some n.

The App *clause* The App clause applies to configurations focused on an application expression $(e_0\, e_1)^\ell$. It ensures that such a configuration evaluates to a value v^d when (1) the operator e_0 is evaluated (within its configuration) to some value $(\lambda x.e, \rho_0^d)$, (2) the environment ρ_0^d and calling context n_c are defined appropriately in the configuration of e, (3) the caller of that configuration is defined in $\mathcal{E}$, and (4) that configuration evaluates to v^d.

7.3 The $\models_{call}$ relation

The $\models_{call}$ relation relates an analysis $(C, \mathcal{E})$ to a judgment $(\rho^d, e, n_c) \Rightarrow_d (\rho_0^d, (e_0\, e_1)^\ell, n_{c'})$ when $(C, \mathcal{E})$ entails that the value of the configuration (ρ^d, e, n_c) is applied at the configuration $(\rho_0^d, (e_0\, e_1)^\ell, n_{c'})$. Its definition, seen in Figure 10, contains a clause

$$(C, \mathcal{E}) \models_{eval} \rho^d, n_c \vdash x^\ell \Downarrow^d v^d$$
$$\textit{iff}$$

[Ref]
$$\begin{array}{l} C_\$(\rho^d, x^\ell, n_c) = \rho^d(x) \\ (\rho_0^d[x \mapsto n], e, n_{c'}) = \mathrm{bind}(x, \rho^d, x^\ell, n_c) \\ C(\rho_0^d, \lambda x.e, n_{c''}) = (\lambda x.e, \rho_0^d) \\ (C, \mathcal{E}) \models_{call} (\rho_0^d, \lambda x.e, n_{c''}) \Rightarrow_d (\rho_1^d, (e_0\, e_1)^{\ell_0}, n_{c'''}) \implies \\ \quad C_\$(\rho_1^d, e_1, n_{c'''}) = n \\ \quad \mathcal{E}(\rho_0^d[x \mapsto n], e, n_{c'}) = (\rho_1^d, (e_0\, e_1)^{\ell_0}, n_{c'''}) \\ \quad \rho^d(x) = C_\$(\rho_1^d, e_1, n_{c'''}) \\ \quad (C, \mathcal{E}) \models_{eval} \rho_1^d, n_{c'''} \vdash e_1 \Downarrow^d v^d \end{array}$$

[Lam]
$$\begin{array}{c} (C, \mathcal{E}) \models_{eval} \rho^d, n_c \vdash (\lambda x.e)^\ell \Downarrow^d (\lambda x.e, \rho^d) \\ \textit{iff} \\ C(\rho^d, (\lambda x.e)^\ell, n_c) = (\lambda x.e, \rho^d) \end{array}$$

[App]
$$\begin{array}{c} (C, \mathcal{E}) \models_{eval} \rho^d, n_c \vdash (e_0\, e_1)^\ell \Downarrow^d v^d \\ \textit{iff} \\ (C, \mathcal{E}) \models_{eval} \rho^d, n_c \vdash e_0 \Downarrow^d (\lambda x.e, \rho_0^d) \implies \\ \quad \rho_1^d = \rho_0^d[x \mapsto C_\$(\rho^d, e_1, n_c)] \\ \quad C_\nu(n_{c'}) = \ell :: n_c \\ \quad \mathcal{E}(\rho_1^d, e, n_{c'}) = (\rho^d, (e_0\, e_1)^\ell, n_c) \\ \quad C_\$(\rho_1^d, e, n_{c'}) = C_\$(\rho^d, (e_0\, e_1)^\ell, n_c) \\ \quad (C, \mathcal{E}) \models_{eval} \rho_1^d, n_{c'} \vdash e \Downarrow^d v^d \end{array}$$

Fig. 8. The $\models_{eval}$ relation

$$\begin{array}{lll} \mathrm{bind}(x, \rho^d, e, n) = & (\rho_0^d, (\lambda x.e)^\ell, n_0) & \text{where } \mathbb{K}(e) = (\lambda x.\Box)^\ell \text{ and } \rho^d = \rho_0^d[x \mapsto n] \\ \mathrm{bind}(x, \rho^d, e, n) = \mathrm{bind}(x, \rho_0^d, (\lambda x.y)^\ell, n_0) & & \text{where } \mathbb{K}(e) = (\lambda y.\Box)^\ell,\ x \neq y, \text{ and } \rho^d = \rho_0^d[y \mapsto n] \\ \mathrm{bind}(x, \rho^d, e, n) = & \mathrm{bind}(x, \rho^d, (e\, e_1)^\ell, n) & \text{where } \mathbb{K}(e) = (\Box\, e_1)^\ell \\ \mathrm{bind}(x, \rho^d, e, n) = & \mathrm{bind}(x, \rho^d, (e_0\, e)^\ell, n) & \text{where } \mathbb{K}(e) = (e_0\, \Box)^\ell \end{array}$$

Fig. 9. Given a variable x and an expression e in which x appears free, along with its closing environment and calling context, the bind metafunction reconstructs the "birth" context of the closure which yields this binding of x when applied. The resultant name of the calling context must be consistent with (and is uniquely identified by) the calling context discovered for this closure.

for each type of expression context. Each of these clauses functions essentially as its counterpart does in $\models_{fscall}$.

$$
\begin{array}{|l|c|}
\hline
 & (C,\mathcal{E}) \models_{call} (\rho^d, e_0, n_c) \Rightarrow_d (\rho^d, (e_0\, e_1)^\ell, n_c) \text{ for } \mathbb{K}(e_0) = (\Box\, e_1)^\ell \\
\text{[RATOR]} & \textit{iff} \\
 & \text{always} \\
\hline
 & (C,\mathcal{E}) \models_{call} (\rho^d, e_1, n_c) \Rightarrow_d (\rho_0^d, (e_2\, e_3)^{\ell_0}, n_{c'}) \text{ for } \mathbb{K}(e_1) = (e_0\, \Box)^\ell \\
 & \textit{iff} \\
 & (C,\mathcal{E}) \models_{eval} \rho^d, n_c \vdash e_0 \Downarrow^d (\lambda x.e, \rho_1^d) \implies \\
\text{[RAND]} & \quad \rho_2^d = \rho_1^d[x \mapsto C_\$(\rho^d, e_1, n_c)] \\
 & \quad C_\nu(n_{c''}) = \ell :: n_c \\
 & \quad (\rho_3^d, x^{\ell_1}, n_{c'''}) \in \text{find}(x, \rho_2^d, e, n_{c''}) \implies \\
 & \quad\quad (C,\mathcal{E}) \models_{call} (\rho_3^d, x^{\ell_1}, n_{c'''}) \Rightarrow_d (\rho_0^d, (e_2\, e_3)^{\ell_0}, n_{c'}) \\
\hline
 & (C,\mathcal{E}) \models_{call} (\rho^d[x \mapsto n], e, n_c) \Rightarrow_d (\rho_0^d, (e_0\, e_1)^{\ell_0}, n_{c'}) \text{ for } \mathbb{K}(e) = (\lambda x.\Box)^\ell \\
 & \textit{iff} \\
\text{[BODY]} & (C,\mathcal{E}) \models_{call} (\rho^d, (\lambda x.e)^\ell, n_{c''}) \Rightarrow_d (\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''}) \implies \\
 & (C,\mathcal{E}) \models_{call} (\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''}) \Rightarrow_d (\rho_0^d, (e_0\, e_1)^{\ell_0}, n_{c'}) \\
\hline
 & (C,\mathcal{E}) \models_{call} (\rho^d, e, n_c) \Rightarrow_d (\rho_0^d, (e_0\, e_1)^\ell, n_{c'}) \text{ for } \mathbb{K}(e) = \Box \\
\text{[TOP]} & \textit{iff} \\
 & \text{never} \\
\hline
\end{array}
$$

Fig. 10. The $\models_{call}$ relation

The RATOR *clause* The resultant value of a configuration (ρ^d, e_0, n_c) where e_0 has context $(\Box\, e_1)^\ell$ is applied at $(e_0\, e_1)^\ell$ (assuming the convergence of evaluation of e_1) so its caller configuration is $(\rho^d, (e_0\, e_1)^\ell, n_c)$.

The RAND *clause* The resultant value of a configuration (ρ^d, e_1, n_c) where e_1 has context $(e_0\, \Box)^\ell$ is bound to the parameter x of the value $(\lambda x.e, \rho_0^d)$ of e_0 and appears at every reference to x in e. The RAND clause ensures that the argument value is called at configuration $(\rho_0^d, (e_2\, e_3)^{\ell_0}, n_{c'})$ when (1) the operator expression is evaluated to a value, (2) the environment of that value is extended with the name of the argument and the calling context is extended with the call-site label, and (3) the find metarelation furnishes a configuration whose value is called at $(\rho_0^d, (e_2\, e_3)^{\ell_0}, n_{c'})$.

The find metarelation, defined in Figure 11, relates references to x in a configuration (ρ^d, e, n_c) to configurations $(\rho_0^d, x^\ell, n_{c'})$ wich constitute references to x.

The BODY *clause* If a configuration is in a body context, its result becomes the result of the caller of the closure over its enclosing λ-term. The BODY clause ensures that the resultant value of a configuration $(\rho^d[x \mapsto n], e, n_c)$ such that e has syntactic context $(\lambda x.\Box)^\ell$ is called at a configuration $(\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''})$ when (1) the enclosing value is called at configuration $(\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''})$, and (2)

$$
\begin{array}{rcl}
(\rho^d, x^\ell, n_c) \in \text{find}(x, \rho^d, x^\ell, n_c) & \textit{iff} & \text{always} \\
(\rho_0^d, x^\ell, n_{c'}) \in \text{find}(x, \rho^d, \lambda y.e^\ell, n_c) & \textit{iff} & (\rho_0^d, x^\ell, n_{c'}) \in \text{find}(x, \rho^d[y \mapsto n], e, n_{c''}) \\
 & & \text{where } x \neq y \text{ and for some } n,\ n_{c''} \\
(\rho_0^d, x^\ell, n_c) \in \text{find}(x, \rho^d, (e_0\, e_1)^\ell, n_c) & \textit{iff} & (\rho_0^d, x^\ell, n_c) \in \text{find}(x, \rho^d, e_0, n_c) \\
(\rho_0^d, x^\ell, n_c) \in \text{find}(x, \rho^d, (e_0\, e_1)^\ell, n_c) & \textit{iff} & (\rho_0^d, x^\ell, n_c) \in \text{find}(x, \rho^d, e_1, n_c)
\end{array}
$$

Fig. 11. The find relation

the resultant value of $(\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''})$—the value of the initial configuration over e—is called at configuration $(\rho_1^d, (e_2\, e_3)^{\ell_1}, n_{c'''})$.

The Top *clause* A closure that reaches the top level of the program is not called at any configuration within evaluation.

7.4 Soundness

We can now formally state the correctness of demand 0CFA relative to demand evaluation. Correctness is expressed by two lemmas which each relate a demand evaluation relation to its demand 0CFA counterpart.

Lemma 1. *If* $(C, \mathcal{E}) \models_{eval} \rho^d, n_c \vdash t^\ell \Downarrow^d (\lambda x.e, \rho_0^d)$ *then, if* $(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fs\,eval} t^\ell$, $\lambda x.e \in \hat{\mathcal{C}}(\ell)$.

Lemma 2. *If* $(C, \mathcal{E}) \models_{call} (\rho^d, t^\ell, n_c) \Rightarrow_d (\rho_0^d, (e_0\, e_1)^{\ell_0}, n_{c'})$ *where* $C(\rho^d, t^\ell, n_c) = (\lambda x.e, \rho_0^d)$ *then, if* $(\hat{\mathcal{C}}, \hat{\mathcal{E}}) \models_{fs\,call} (\lambda x.e, t^\ell)$, $(e_0\, e_1)^{\ell_0} \in \hat{\mathcal{E}}(e)$.

Lemma 1 states that a demand 0CFA analysis $(\hat{\mathcal{C}}, \hat{\mathcal{E}})$ acceptable by $\models_{fs\,eval}$ for an expression e contains an abstraction of every value for which there is an acceptable (by $\models_{eval}$) exact analysis $(C, \mathcal{E})$. Lemma 2 says that the demand 0CFA specification $\models_{fs\,eval}$ will always include abstractions of calling configurations discovered by the demand evaluation specification $\models_{call}$. Because we took great pains to keep exact demand evaluation close to approximate demand evaluation, the proofs of these lemmas proceed straightforwardly by mutual induction on the definitions of $\models_{eval}$ and $\models_{call}$. The coinductive step proceeds by cases over expressions, in the case of Lemma 1, and syntactic contexts, in the case of Lemma 2. The corresponding clauses in the exact and approximate relations themselves tightly correspond, so each case proceeds without impediment.

8 Related Work

Palmer and Smith's Demand-Driven Program Analysis (DDPA) [16] is most-closely related to this work, being both demand-driven and a control-flow analysis. DDPA differs from demand 0CFA in that it must construct a call graph from the program entry point, using its demand lookup facilities to resolve targets along the way. In contrast, demand 0CFA is able to construct the call graph on demand from an arbitrary control point.

There are three nominal higher-order demand-driven analyses that use the term *demand* in a different sense than we do. The first is a "demand-driven 0-CFA" derived by using a calculational approach to abstract interpretation [11]. The derived analysis is not demand in our sense in that one cannot specify an arbitrary program expression to be analyzed but instead refers to an analyzer that attempts to analyze only those parts of the program that influence the final result. In this very loose sense, demand 0CFA is a generalization of demand-driven 0CFA. The authors relate their work to the second nominally demand analysis, Biswas [2] which uses the term *demand* in a similar way for first-order functional programs. Heintze and McAllester's [8] "subtransitive CFA" computes an underapproximation of control-flow in linear time which can be transitively closed at quadratic cost (for cubic total cost) and is described by the authors as "demand-driven". Their analysis operates over typed programs with bounded type; in contrast, demand 0CFA operates over untyped programs.

The CFA aspect of this work is related to the myriad exhaustive specifications of CFA [18, 14, 22, 5, 23, 10, 7, 3]. The most significant difference of this work is its demand-driven nature. However, other differences remain: modern conceptions of CFA are based on small-step abstract machines [22] or big-step definitional interpreters [3, 24] which offer flow, context, and path sensitivity; we have presented demand 0CFA as a constraint-based analysis that is flow-, context-, and path-*insensitive*.

Control-flow analysis is the higher-order analogue of points-to analysis. Even object-oriented programs exhibit higher-order control flow in that the destination of a method call depends on the class of the dynamic target. While earlier work [20] leveraged only the class hierarchy to approximate the call graph, later work used it merely as a foothold to a more-precise construction of it [19].

9 Conclusion and Future Work

In this paper, we introduced demand 0CFA, a monovariant, context-insensitive, constraint-based, demand-driven control-flow analysis, and discussed how it is well-suited to many CFA clients. Future work includes enhancing demand 0CFA with both polyvariance and context-sensitivity, arriving at a demand-driven k-CFA hierarchy [18]. While flow insensitivity is fundamental to our formalism, context- and even path-insensitivity are not. However, the constraint-based framework which underlies demand 0CFA is likely not essential to it and it may be possible to port the approach to a small-step abstract machine-based framework (e.g. [22]) to achieve flow sensitivity. Applying the insight of Gilray *et al.* [7], introducing an environment may provide the leverage needed to obtain a pushdown abstraction of control flow.

This material is partially based on research sponsored by DARPA under agreement number AFRL FA8750-15-2-0092 and by NSF under CAREER grant 1350344. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

1. Ashley, J.M., Dybvig, R.K.: A practical and flexible flow analysis for higher-order languages. ACM Trans. Program. Lang. Syst. (1998)
2. Biswas, S.K.: A demand-driven set-based analysis. In: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. ACM (1997)
3. Darais, D., Labich, N., Nguyen, P.C., Van Horn, D.: Abstracting definitional interpreters (functional pearl). Proceedings of the ACM on Programming Languages **1**(ICFP), 12 (2017)
4. Duesterwald, E., Gupta, R., Soffa, M.L.: A practical framework for demand-driven interprocedural data flow analysis. ACM Transactions on Programming Languages and Systems (TOPLAS) **19**(6), 992–1030 (1997)
5. Earl, C., Might, M., Van Horn, D.: Pushdown control-flow analysis of higher-order programs. In: Workshop on Scheme and Functional Programming (2010)
6. Germane, K., Might, M.: Demand control-flow analysis. Tech. rep. (January 2019), http://kimball.germane.net/germane-dcfa-techreport.pdf
7. Gilray, T., Lyde, S., Adams, M.D., Might, M., Van Horn, D.: Pushdown control-flow analysis for free. In: Proceedings of the 43rd Annual ACM SIGPLAN Symposium on Principles of Programming Languages. POPL '16, ACM (2016)
8. Heintze, N., McAllester, D.: Linear-time subtransitive control flow analysis. In: PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation. ACM Press (1997)
9. Heintze, N., Tardieu, O.: Demand-driven pointer analysis. In: ACM SIGPLAN Notices. vol. 36, pp. 24–34. ACM (2001)
10. Johnson, J.I., Van Horn, D.: Abstracting abstract control. In: Proceedings of the 10th ACM Symposium on Dynamic Languages. ACM (2014)
11. Midtgaard, J., Jensen, T.: A calculational approach to control-flow analysis by abstract interpretation. In: International Static Analysis Symposium. pp. 347–362. Springer (2008)
12. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: Illuminating functional vs. object-oriented program analysis. In: PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '10, ACM Press (2010)
13. Might, M., Smaragdakis, Y., Van Horn, D.: Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In: PLDI '10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM (2010)
14. Nielson, F., Nielson, H.R.: Infinitary control flow analysis: a collecting semantics for closure analysis. In: POPL '97: Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages. ACM Press (1997)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag (1999)
16. Palmer, Z., Smith, S.F.: Higher-order demand-driven program analysis. In: 30th European Conference on Object-Oriented Programming (2016)
17. Palsberg, J.: Closure analysis in constraint form. ACM Transactions Programming Languages and Systems **17**(1), 47–62 (Jan 1995)
18. Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University (1991)

19. Sridharan, M., Bodík, R.: Refinement-based context-sensitive points-to analysis for java. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 387–400. PLDI '06, ACM, New York, NY, USA (2006)
20. Sridharan, M., Gopan, D., Shan, L., Bodík, R.: Demand-driven points-to analysis for java. In: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 59–76. OOPSLA '05, ACM, New York, NY, USA (2005)
21. Van Horn, D., Mairson, H.G.: Flow analysis, linearity, and PTIME. In: Alpuente, M., Vidal, G. (eds.) Static Analysis, chap. 17. Lecture Notes in Computer Science, Springer Berlin / Heidelberg (2008)
22. Van Horn, D., Might, M.: Abstracting abstract machines. In: Proceedings of the 15th ACM International Conference on Functional Programming. pp. 51–62. ICFP '10, ACM, New York, NY, USA (2010)
23. Vardoulakis, D., Shivers, O.: CFA2: a context-free approach to control-flow analysis. Logical Methods in Computer Science (2011)
24. Wei, G., Decker, J., Rompf, T.: Refunctionalization of abstract abstract machines: Bridging the gap between abstract abstract machines and abstract definitional interpreters (functional pearl). Proc. ACM Program. Lang. **2**(ICFP), 105:1–105:28 (Jul 2018)