Demand-on-Demand Control-Flow Analysis

CHAHYUN KANG, Brigham Young University, USA
KIMBALL GERMANE, Brigham Young University, USA

Understanding program behaviors requires reasoning about control flow. Functional programs complicate this reasoning since call targets are computed in general. Control-flow analysis (CFA) can effectively reason about control flow (and much more) but is costly. Demand-driven CFA is less expensive but less versatile, and it is difficult to make it do what classical CFA does. This is unfortunate because many fundamental optimizations rely on control flow reasoning. In this paper, we present Demand-on-Demand Control-Flow Analysis (DoDCFA), a hybrid exhaustive–demand-driven CFA that recovers some of the versatility/applicability of classical CFA while improving on the speed of Demand CFA. The result is an analysis that can inexpensively and effectively analyze control flow and environment behavior sufficient to justify inlining.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Additional Key Words and Phrases: Control-flow Analysis, Static Analysis, Abstract Interpretation, Demand-driven Analysis, Higher-Order Languages, Environment Analysis, Must-alias Analysis

ACM Reference Format:
Chahyun Kang and Kimball Germane. 2026. Demand-on-Demand Control-Flow Analysis. *Proc. ACM Program. Lang.* 10, ICFP, Article 289 (August 2026), 34 pages. https://doi.org/10.1145/3828687

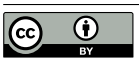
1 Introduction

Determining the target of a call in a higher-order program is not trivial. Because this target is computed, it can be approximated by a data flow analysis operating over a program control-flow graph (CFG). But the particular target of the call *determines* the CFG, so we are left in a situation where we need a call target to determine a CFG to use data-flow analysis to compute that same call target. Shivers referred to this as the *control-flow problem* [20].

Shivers also offered a solution in the form of *control-flow analysis* (CFA). His presented solution, k-CFA, operates in the same way as effectively all succeeding forms of CFA: compute the data flow and the CFG in tandem, each providing the other with the information it needs just-in-time. This tandem computation is achieved by devising a computable *abstract semantics*—a sound over-approximation of the *concrete semantics* of the language. Using this abstract semantics, CFA explores the program just as the concrete semantics does: it starts from the entry point and proceeds through forward evaluation. The CFA concludes when it has explored every possible (approximated) path through the program, and the analysis it produces is an account of that exploration.

The closeness of the abstract semantics to the concrete semantics has allowed for the development of several systematic frameworks that transform the latter to the former. However, likely unintentionally, it has also induced a particular pricing model: The soundness of the CFA's control flow account is guaranteed only after the program is exhaustively analyzed, at which point the entire control flow account is delivered. If the CFA is unable to complete program analysis—whether

Authors' Contact Information: Chahyun Kang, Brigham Young University, Provo, USA, chi2eut@byu.edu; Kimball Germane, Brigham Young University, Provo, USA, kimball@cs.byu.edu.



ACM 2475-1421/2026/8-ART289
https://doi.org/10.1145/3828687

due to resource exhaustion, an internal analyzer defect, or an unhandled object language feature or primitive—it cannot deliver any sound information.

To address this failure mode, Demand CFA [8] was developed as an alternative formulation of CFA, which offers a different pricing model. Here, "demand" refers to a query-driven model of computation: analysis is performed only in response to a specific request, rather than exhaustively over the entire program. In this model, one poses a control flow query at a specific (but arbitrary) program point in the context of a containing program. The CFA resolves this query as parsimoniously and locally as possible—often issuing subqueries—to resolve the control flow of just that point. While a Demand CFA analysis may fail for just the same reasons as an exhaustive CFA, such failure only directly implicates the specified program point, and similar queries to nearby program points may succeed.

Of course, Demand CFA has its own shortcomings, three of which are relevant to this work:

(1) Demand CFA has an evaluation model relatively distant from both the concrete and abstract semantics, which makes it nontrivial to apply advances made in exhaustive CFA to Demand CFA. For example, many abstract semantics use a store to model the heap, which is utilized to ensure termination [11] and reason about variable bindings [13, 14, 16]. Demand CFA does not use a store to model the heap, which prevents the direct application of such techniques.
(2) Rather than using a store to model the heap, Demand CFA models it globally, which induces flow and path insensitivity in the analysis. Without flow or path sensitivity, Demand CFA cannot respect temporal ordering within programs, nor distinguish between different paths through the same code.
(3) Despite explicit design goals for parsimony, Demand CFA ignores information about arguments it encounters during analysis, which causes it to later expend analysis effort retrieving that information. In fact, Demand CFA ignores it because it does not have a convenient place to record such information, such as a store.

To address these shortcomings, we present **Demand-on-Demand CFA (DoDCFA)**, a hybrid exhaustive—demand-driven analysis. Like exhaustive CFA, DoDCFA utilizes a store—only DoDCFA uses it to record arguments as it encounters them. If an evaluated reference points to a known argument, it is evaluated immediately; if not, a demand analysis is dispatched at that point—on demand—to determine it. This approach yields an analysis with none of the presented shortcomings:

(1) The addition of a store moves the semantics much closer to the abstract semantics of exhaustive CFA. We witness a material benefit of this move by demonstrating how to integrate abstract counting, a way to reason about variable bindings, into DoDCFA straightforwardly but also in a way that leverages the parsimonious nature of Demand CFA (§6).
(2) Situating the store in the semantics as we do has the effect of inducing flow and path sensitivity (§8.3).
(3) Finally, the introduction of the store allows DoDCFA to record (but not eagerly evaluate) the arguments of a call, which avoids the need to resolve callers when DoDCFA itself entered the call.

Altogether, DoDCFA reliably obtains control flow information more cheaply than exhaustive CFA and can perform richer reasoning than Demand CFA. In fact, applying abstract counting to DoDCFA provides more-precise reasoning about variable binding than it does when applied to exhaustive CFA. Our evaluation shows that DoDCFA explores 86–91% fewer configurations than k-CFA while discovering 13–28% more inlinable call sites.

The remainder of the paper is as follows. We review exhaustive CFA and Demand CFA in §2 and provide an informal overview of DoDCFA's two-phase design in §3. §4 formalizes DoDCFA; its soundness with respect to call-by-value evaluation is established via a series of intermediate

semantics in §5. We extend DoDCFA with abstract counting that enables must-alias reasoning—a capability lacked by pure demand-driven frameworks (§6). We then empirically evaluate DoDCFA on benchmark programs (§7). We discuss noteworthy characteristics of DoDCFA in §8 and survey related work in §9.

2 Background

2.1 Control-Flow Analysis (Exhaustive)

Control-flow analysis (CFA) determines the set of possible call targets in higher-order languages [20]. Since higher-order languages treat functions as first-class values, a call site's operator variable may be bound to multiple distinct closures at runtime. CFA computes which closures flow to these call sites to approximate the program's dynamic control-flow graph.

Abstract interpretation provides a principled framework for designing such analyses [3]. The Abstracting Abstract Machine (AAM) framework [11] systematically derives abstract interpreters by finitizing small-step operational semantics, such as the CESK or Krivine machines. Building on this methodology, the Abstracting Definitional Interpreter (ADI) framework [4] applies similar abstractions to big-step definitional interpreters. ADI currently represents the state-of-the-art for implementing straightforward yet expressive abstract interpreters for higher-order languages.

However, CFAs derived through these frameworks are exhaustive, which inherently limits their scalability. Because they are derived from concrete interpreters, they must traverse all reachable expressions and program states. Many analysis clients regularly require information about specific program points, making a full exhaustive sweep redundant and computationally expensive.

Consider the example program in Figure 1. From a global perspective, the bindings and computations for variables `y` and `z` are irrelevant to the final value returned by `f`, assuming there are no effects on `x` in applications of `g` and `h`. An exhaustive CFA nevertheless evaluates all reachable program points, consuming resources to analyze irrelevant call sites despite their lack of influence on the result. Furthermore, exploring unnecessary program paths can lead to over-approximation when disparate control flows are merged in the abstract store. Consequently, exhaustive analysis can compromise both the performance and the precision of the resulting CFA.

```
(define (f x g h)
  (let ([y (g x)])
    (let ([z (h y)])
      x)))
```

Fig. 1. Program with Some Calls in Scheme

2.2 Demand Control-Flow Analysis

Demand CFA [8] provides an effective demand-driven alternative to exhaustive CFA by limiting the analysis search space to a specific query. The objective of Demand CFA is to answer a localized query by traversing only the relevant program paths rather than performing an exhaustive global search. Consider the example program again in Figure 1. Demand CFA recognizes that the variable `x` is the only expression contributing to the return value of the function `f`. Since `x` is bound by the application of `f`, the analysis identifies call sites `(g x)` and `(h y)` as irrelevant to the query result. By ignoring these potentially expensive computations and focusing solely on the binding of `x`, the analyzer directly evaluates the argument of the call sites that apply `f`.

```
(let ([id (λ (x) x)])
  (let ([y (id 42)])
    y))
```

Fig. 2. Program with Application to id in Scheme

The mechanism of Demand CFA relies on two mutually recursive query models: evaluation and trace. An evaluation query searches for the set of values that flow into a specific expression. Conversely, a trace query searches for all call sites where a given closure is applied as a function. We demonstrate the synergy of these models using the program in Figure 2. If a user queries the result of the entire program, the analyzer first identifies that the final value depends on the variable

`y`. To resolve `y`, the analyzer locates its binding site in the second `let` expression, which triggers the evaluation of the application `(id 42)`. The analyzer then evaluates the operator `id` to determine which closure flows into the call site. By resolving `id` back to its binding in the first `let` expression, it retrieves the function `(λ (x) x)`. The analyzer then enters the function body to evaluate `x` and initiates a trace query on the closure `(λ (x) x)` to find its call sites. Upon locating the call site `(id 42)`, the analyzer evaluates the argument `42` and successfully answers the original query.

Conceptually, Demand CFA partially mimics the behavior of an abstract interpreter modeled after lazy evaluation (call-by-need). It delays the evaluation of call-site arguments until the resolution of a query specifically necessitates their values. By utilizing trace queries to recover argument expressions, Demand CFA demonstrates that lazy abstract interpretation can provide a demand-driven pricing model for CFA. Furthermore, this framework can initiate evaluation at any arbitrary program point. If a sub-expression contains no free variables, the analysis can resolve the query locally without searching the outer scope, offering significant flexibility. When external information is required, the analyzer efficiently navigates the AST to retrieve only the necessary context.

Despite its efficiency in localized closure-flow resolution, the structural design of Demand CFA introduces several fundamental limitations. Because the analysis is designed to be environment- and store-agnostic to preserve its backward-traversal performance, it lacks a persistent representation of a threaded store. The absence of a store means the analysis is inherently flow-insensitive, forcing it to rely entirely on the immediate calling context for precision and leaving it unable to distinguish between different binding instances of the same variable across multiple paths. Furthermore, Demand CFA suffers from a lack of memoization; because it does not remember previously encountered arguments, it frequently engages in "computational amnesia," redundantly re-triggering potentially expensive backward searches for the same variable across disparate calling contexts. These shortcomings—the inability to persist bindings and the lack of memoization—create a precision ceiling that prevents Demand CFA from supporting more sophisticated reasoning, such as environment analysis.

3 Informal Overview on DoDCFA

In this section, we informally describe the approach of DoDCFA, derived via a systematic refactoring of exhaustive and Demand CFA into a single framework. As established, Exhaustive CFA is effective in processing the "known" (context within the current scope) with a memoization of known facts in the environment and store, but is inefficient due to its exhaustiveness. In contrast, Demand CFA effectively searches the "unknown" (context outside the current evaluation scope) but is inefficient when processing the "known."

DoDCFA's strategy is to adopt Demand CFA's mechanism as a backward analysis for traversing the unknown via Trace Queries while modeling forward evaluation after abstracted call-by-need semantics. This enables the analysis to start the evaluation at any point of the program by making the context unknown and allocating all variables in scope unknown. This provides a good division between a "known" and an "unknown" side, where the forward analysis focuses on the "known," and the backward analysis focuses on the "unknown." We memoize any information with a store that is threaded through both forward and backward analysis.

Essentially, DoDCFA refactors exhaustive and Demand CFA by synthesizing forward and backward analysis techniques: forward analysis is used for standard abstract call-by-need semantics, while backward analysis handles the trace-based discovery of arguments. We note that call-by-need semantics models the forward analysis because it is a better fit with the demand-driven pricing model, since it delays evaluation of the argument until demanded by the semantics.

3.1 Forward Analysis Side: Call-by-Need Semantics

We model the forward analysis, the evaluation query, after the call-by-need semantics. Whenever the analysis encounters an expression that requires extending the lexical context, it records the binding between the parameter and its corresponding argument expression. By caching these relationships, the analysis eliminates the need for redundant searches to re-locate previously encountered arguments. This model follows the configuration and structure of the standard operational semantics, where the environment functions as a stack, and the store acts as threaded memory throughout the analysis. If an expression contains no free variables relative to the current environment, the analysis does not require a backward analysis.

For example, consider the program again in Figure 2. Suppose the analysis begins at the variable `y`, which is the body of the nested `let` expression that binds `y`. Because the argument for the binding of `y` is determined by a `let` expression, it remains consistent throughout the scope. Thus, the analysis can pre-allocate the address and argument expression for `y` as it moves the cursor to the body of the expression. Once the cursor reaches the body `y`, the analyzer already has a binding for `y` allocated in both the environment and the store. This allows the analysis to perform a direct lookup to retrieve the argument expression `(id 42)`, which then proceeds to evaluate that sub-expression.

When the forward analysis encounters a variable that is "unknown," it starts the backward analysis to resolve the unknown call context to obtain the argument for evaluation. Because the store is threaded, every information found in the backward analysis is memoized, which eliminates the need for another search for the same query later on. If the forward analysis does not encounter any "unknown" variables in its analysis, it doesn't require the backward analysis. In other words, forward analysis only attempts to resolve call contexts that are required for the evaluation. This refactoring uses memoization to significantly reduce the number of required AST traversals.

3.2 Backward Analysis Side: Demand CFA Trace Query

We model the backward analysis after the Demand CFA's method to resolve unknown call contexts. In Demand CFA, evaluating a variable reference always triggers a combination of a bind/find operator and a trace query to locate the potentially correct call site that binds the variable in query. The bind operator traverses the AST upward to locate the lambda abstraction that binds the variable, while the trace query, aided by the find operator, searches for all call sites where that lambda is applied to find the associated argument. While this combination is effective for discovering arguments, Demand CFA applies it indiscriminately to every reference, regardless of whether the binding is already known. Our framework instead reserves this expensive backward search specifically for "unknown" references outside the current memoized environment.

We first refactor the bind operator to account for the environment and store during its upward traversal of the AST. As the analysis moves upward, it pops variables from the environment to accurately reflect the restoration of the stack at the upper program point. The store remains threaded, ensuring that previous allocations are preserved throughout the traversal.

The trace query remains conceptually identical to the original Demand CFA's trace query, but is updated to maintain the environment and store. Whenever a trace query passes through a binding site, it extends the environment and stores to memoize the discovered argument expressions. For instance, when tracing an argument into the body of a lambda, the analysis allocates a fresh heap address for that argument. The find operator, which identifies all usage sites of a variable, is similarly refactored to allocate addresses for unknown variables discovered during the search.

Consider the program again in Figure 2, assuming the analysis begins at the reference `x` within the body of the `id` function. Because `x` is bound via a function application rather than a direct `let` binding, its value remains unknown when the analysis cursor enters the lambda expression.

$$
\begin{array}{lll}
e \in Exp ::= (e\ e)\ [App] & E \in EvalCtx ::= (E\ e)\ [Fun] & x \in Var \\
\quad \mid \lambda x.e\ [Lam] & \quad \mid (e\ E)\ [Arg] & \\
\quad \mid x\ [Ref] & \quad \mid \lambda x.E\ [Bod] & \\
 & \quad \mid \square\ [Hole] &
\end{array}
$$

Fig. 3. Lambda Calculus Syntax With Evaluation Context

Consequently, the store allocates an entry for `x` that is explicitly marked as an unknown value. When the analyzer attempts to evaluate `x`, the lookup reveals this unknown status, which triggers the backward analysis phase. Following the methodology of Demand CFA, the analyzer locates the binding site of `x`—the `id` function—and initiates a trace query. This query directs the analysis to the call site `(id 42)`, at which point the analyzer evaluates the argument to obtain the value `42`. Crucially, the analysis then memoizes this result in the store, ensuring that all recovered information is persisted for future lookups.

The backward analysis now holds the environment and store in its configuration, which boosts its efficiency. Because Demand CFA is environment- and store-agnostic, it doesn't memoize any information it found previously. This makes Demand CFA do the same search again when the analysis requires it, which decreases its efficiency. This refactoring likewise uses memoization to improve efficiency.

4 DoDCFA Formalism

In this section, we provide a formal definition of the DoDCFA. We begin by defining the syntactic domains of our target language to establish a foundation for the subsequent development. We then present a series of big-step semantics, demonstrating how DoDCFA is systematically constructed from a definitional interpreter. The development starts with a standard big-step call-by-value semantics, which we then extend to a big-step call-by-need semantics. Subsequently, we derive an abstract call-by-need semantics by applying the ADI framework to our big-step model. Finally, we formalize the DoDCFA by incorporating demand-driven search operators into the abstract call-by-need semantics.

4.1 Language

We employ a standard lambda calculus as the base language for demonstrating our framework, with the syntax defined in Figure 3. The expression grammar consists of function applications, lambda abstractions, and variable references. For simplicity and clarity, we assume that all variables are uniquely named (alpha-renamed) to prevent variable shadowing throughout the analysis. Fundamental to our semantics are evaluation contexts, denoted by E, which represent a program fragment with a single "hole" (□). We use the notation $E[e]$ to represent the expression formed by filling the hole in context E with the expression e. Crucially, in our framework, this notation signifies that the analysis cursor is currently positioned at e. While E provides the surrounding program structure and lexical history, e is the specific sub-expression targeted for the evaluation. Contexts can be composed to represent deeper nesting. For example, the notation $E[(e_0\ e_1)]$ describes an application within an arbitrary context E. When we specifically write $E[(\square\ e_1)]$, we are indicating that the focus of the analysis is the operator position of an application. By extension, $E[([e_0]\ e_1)]$

$$
\begin{aligned}
cfg \in Config &= Cursor \times Env \times Store \times Time \\
csr \in Cursor &= EvalCtx \times Exp \\
\rho \in Env &= Var \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup D \\
call \in Call &= EvalCtx \times App
\end{aligned}
\qquad
\begin{aligned}
\alpha \in Addr &= Var \times Time \\
d \in D &= Val \\
v \in Val &= Clo \\
t \in Time &= Call^* \\
r \in Result &= Val \times Store \\
clo \in Clo &= (EvalCtx \times Lam) \times Env
\end{aligned}
$$

Fig. 4. State Space for the Call-by-value Operational Semantics

$$
\textsc{Lam}\ \frac{}{E[\lambda x.e],\ \rho,\ \sigma,\ t\ \Downarrow_v\ ((E[\lambda x.e],\ \rho),\ \sigma)}
\qquad
\textsc{Ref}\ \frac{}{E[x],\ \rho,\ \sigma,\ t\ \Downarrow_v\ (\sigma(\rho(x)),\ \sigma)}
$$

$$
\textsc{App}\ \frac{\begin{array}{c} E[([e_0]\ e_1)],\ \rho,\ \sigma,\ t\ \Downarrow_v\ ((E'[\lambda x.e],\ \rho_0),\ \sigma_1) \\ E[(e_0\ [e_1])],\ \rho,\ \sigma_1,\ t\ \Downarrow_v\ (v_1,\ \sigma_2) \\ t_1 = E[(e_0\ e_1)] :: t \qquad \alpha = (x,\ t_1) \\ E'[\lambda x.[e]],\ \rho_0[x \rightarrow \alpha],\ \sigma_2[\alpha \rightarrow v_1],\ t_1\ \Downarrow_v\ r \end{array}}{E[(e_0\ e_1)],\ \rho,\ \sigma,\ t\ \Downarrow_v\ r}
$$

$$
\Downarrow_v\ \subseteq\ Config\ \times\ Result
$$

Fig. 5. Call-by-value Operational Semantics

signifies that the analysis is currently targeting e_0 while maintaining the knowledge that it is the operator of an application within the broader context E.

4.2 Call-by-Value Operational Semantics

We begin by defining the state space for the call-by-value operational semantics, as shown in Figure 4. A configuration *cfg* is a quadruple consisting of a cursor, an environment, a store, and a timestamp. The cursor *csr* pairs an evaluation context E with an expression e, providing a structured interface for navigating the abstract syntax tree. The environment ρ maps variables to addresses, while the store σ maps these addresses to values in the store domain D. In this semantics, a value v is a closure that pairs a lambda expression (and its context) with its captured environment. The time t is represented as a sequence of call sites, portraying the call stack of a program point. Finally, a result r is a pair consisting of a value and a store, ensuring that store allocations are properly threaded through the big-step evaluation.

The operational rules for this semantics are detailed in Figure 5. We define the evaluation relation $\Downarrow_v$ as a big-step mapping from configurations to results. The [Lam] rule specifies that a lambda expression evaluates directly to a closure containing the current context and environment. The [Ref] rule handles variable lookups by retrieving the value associated with the address found in the environment. The [App] rule governs function application, which manages memory allocation and the binding of arguments. This rule first evaluates the operator to obtain a closure and then

$$
\begin{aligned}
cfg \in Config &= Cursor \times Env \times Store \times Time \\
csr \in Cursor &= EvalCtx \times Exp \\
\rho \in Env &= Var \rightharpoonup Addr \\
\sigma \in Store &= Addr \rightharpoonup D \\
call \in Call &= EvalCtx \times App \\
th \in Thunk &= (Cursor \times Env \times Time)
\end{aligned}
\qquad
\begin{aligned}
\alpha \in Addr &= Var \times Time \\
d \in D &= Storable \\
v \in Val &= Clo \\
t \in Time &= Call^* \\
r \in Result &= Val \times Store \\
clo \in Clo &= (EvalCtx \times Lam) \times Env \\
s \in Storable &= Val + Thunk
\end{aligned}
$$

Fig. 6. State Space for the Call-by-need Operational Semantics

$$
\textsc{Lam}\ \frac{}{E[\lambda x.e],\ \rho,\ \sigma,\ t\ \Downarrow_n\ ((E[\lambda x.e],\ \rho),\ \sigma)}
\qquad
\textsc{Memoized Ref}\ \frac{v = \sigma(\rho(x))}{E[x],\ \rho,\ \sigma,\ t\ \Downarrow_n\ (v,\ \sigma)}
$$

$$
\textsc{UnMemoized Ref}\ \frac{(E'[e],\ \rho',\ t') = \sigma(\rho(x)) \qquad E'[e],\ \rho',\ \sigma,\ t'\ \Downarrow_n\ (v,\ \sigma_1)}{E[x],\ \rho,\ \sigma,\ t\ \Downarrow_n\ (v,\ \sigma_1[\rho(x) \rightarrow v])}
$$

$$
\textsc{App}\ \frac{\begin{array}{c} E[([e_0]\ e_1)],\ \rho,\ \sigma,\ t\ \Downarrow_n\ ((E'[\lambda x.e],\ \rho_0),\ \sigma_1) \\ t_1 = E[(e_0\ e_1)] :: t \qquad \alpha = (x,\ t_1) \\ E'[\lambda x.[e]],\ \rho_0[x \rightarrow \alpha],\ \sigma_2[\alpha \rightarrow (E[(e_0\ [e_1])],\ \rho,\ t)],\ t_1\ \Downarrow_n\ r \end{array}}{E[(e_0\ e_1)],\ \rho,\ \sigma,\ t\ \Downarrow_n\ r}
$$

$$
\Downarrow_n\ \subseteq\ Config\ \times\ Result
$$

Fig. 7. Call-by-need Operational Semantics

evaluates the operand to obtain the argument value. The analysis then generates a new timestamp t_1 by prepending the current call site to the existing time. A fresh address α is allocated using the variable name and the new timestamp to ensure uniqueness. Finally, the function body is evaluated within an environment and store extended by this new binding.

4.3 Call-by-Need Operational Semantics

We now define a big-step call-by-need operational semantics, drawing inspiration from the lazy Krivine machine as presented in the AAM framework [11]. The state space remains largely identical to the call-by-value model, with the primary distinction residing in the domain of the store. In this semantics, the store maps addresses to storables, which are either fully evaluated values or thunks, i.e., unevaluated expressions paired with their lexical environments and timestamps. The domains are extended as in Figure 6.

The call-by-need big-step relation, $\Downarrow_n$, is defined in Figure 7. This semantics delays the evaluation of function arguments, only reducing them to values when a variable reference necessitates their result. The [App] rule illustrates this delay: rather than evaluating the operand e_1, the analysis wraps it in a thunk and binds the resulting storable to the parameter in the store.

Variable reference is split into two mutually exclusive rules to handle memoization. The [Un-Memoized Ref] rule applies when a lookup retrieves a thunk; the analyzer evaluates the delayed expression, returns the resulting value, and updates the store (memoization) to ensure future references do not re-trigger the computation. The [Memoized Ref] rule applies when the store already contains an evaluated value, permitting a direct lookup.

Call-by-need semantics is sound w.r.t. call-by-value semantics in the sense that the any result produced by call-by-value semantics will be produced by call-by-need semantics as well. In the case call-by-value semantics evaluation diverges, this soundness relationship holds vacuously. This property ensures that our overall soundness result holds for all programs, including those nonterminating.

As a preparataion to formalizing the soundness between call-by-value and call-by-need semantics, we define a relation $\sim$ between the stores and storables of the two semantics:

$$
\begin{aligned}
(\rho_v, \sigma_v) \sim (\rho_n, \sigma_n) &\iff \rho_v = \rho_n \text{ and } \forall \alpha \in range(\rho_v), (\sigma_v(\alpha), \sigma_v) \sim (\sigma_n(\alpha), \sigma_n) \\
((E[\lambda x.e], \rho_v), \sigma_v) \sim ((E[\lambda x.e], \rho_n), \sigma_n) &\implies (\rho_v, \sigma_v) \sim (\rho_n, \sigma_n) \\
((E[\lambda x.e], \rho_v), \sigma_v) \sim ((csr, \rho_n, t_n), \sigma_n) &\iff (\rho_v, \sigma_v) \sim (\rho'_n, \sigma'_n) \\
&\text{where } (csr, \rho, t_n), \sigma_n \Downarrow_n (E[\lambda x.e], \rho'_n), \sigma'_n
\end{aligned}
$$

A call-by-value store is related to a call-by-need store if their environments are identical and each corresponding pair of storables is itself related—closures are related directly when they are syntactically identical under equivalent environments, while a closure on the call-by-value side is related to an unevaluated call-by-need thunk precisely when forcing that thunk yields a closure related to the original. In other words, the relation $\sim$ identifies a call-by-value value with whatever a call-by-need computation eventually produces once forced, allowing the two stores to diverge in their intermediate representations—one eager, one lazy—while still agreeing on the values they ultimately denote. This leads to the following theorem regarding the semantic equivalence of our models:

Theorem 4.1. *If* $E[e], \rho_v, \sigma_v, t_v \Downarrow_v ((E[\lambda x.e], \rho'_v), \sigma'_v)$ *and* $(\rho_v, \sigma_v) \sim (\rho_n, \sigma_n), t_v = t_n$ *then* $E[e], \rho_n, \sigma_n, t_n \Downarrow_n ((E[\lambda x.e], \rho'_n), \sigma'_n)$ *where* $(\rho'_v, \sigma'_v) \sim (\rho'_n, \sigma'_n)$

This theorem can be trivially proven by induction on the structure of the evaluation derivation.

4.4 Abstract Call-by-Need Semantics

We now define the abstract semantics for call-by-need, which serves as the first stage of abstraction in the DoDCFA. Figure 8 details the abstract state space. A configuration $\widehat{cfg}$ consists of a cursor, an abstract environment $\widehat{\rho}$, an abstract store $\widehat{\sigma}$, and an abstract timestamp $\widehat{t}$. The abstract environment $\widehat{\rho}$ maps variables to abstract addresses $\widehat{\alpha}$, while the abstract store $\widehat{\sigma}$ maps these addresses to the store domain $\widehat{d}$, which consists of a set of abstract storables $\widehat{s}$. An abstract storable is either an abstract closure or a thunk. An abstract closure $\widehat{clo}$ is a cursor that contains a lambda expression and an abstract environment. An abstract thunk $\widehat{th}$ is a cursor, an abstract environment, and an abstract time. We achieve a finite state space by bounding the timestamp $\widehat{t}$ to a maximum length m (representing last m call sites), following the standard m-CFA technique [17]. This finitization ensures that the analysis always reaches a fixed point, guaranteeing termination.

$$\widehat{cfg} \in \widehat{Config} = Cursor \times \widehat{Env} \times \widehat{Store} \times \widehat{Time}$$
$$\widehat{r} \in \widehat{Result} = \widehat{Val} \times \widehat{Store}$$
$$\widehat{\rho} \in \widehat{Env} = Var \rightharpoonup \widehat{Addr}$$
$$\widehat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{D}$$
$$\widehat{s} \in \widehat{Storable} = \widehat{Thunk} + \widehat{Clo}$$
$$\widehat{th} \in \widehat{Thunk} = (Cursor \times \widehat{Env} \times \widehat{Time})$$
$$\widehat{\alpha} \in \widehat{Addr} = Var \times \widehat{Time}$$
$$\widehat{d} \in \widehat{D} = \mathcal{P}(\widehat{Storable})$$
$$\widehat{v} \in \widehat{Val} = \mathcal{P}(\widehat{Clo})$$
$$\widehat{t} \in \widehat{Time} = Call^m$$
$$\widehat{clo} \in \widehat{Clo} = (EvalCtx \times Lam) \times \widehat{Env}$$

Fig. 8. State Space for the Abstract Call-by-need Semantics

$$\textsc{Lam}\ \frac{}{E[\lambda x.e],\ \widehat{\rho},\ \widehat{\sigma},\ \widehat{t} \Downarrow_n (\{(E[\lambda x.e],\ \widehat{\rho})\},\ \widehat{\sigma})}$$

$$\textsc{Memoized Ref}\ \frac{\widehat{v} = \widehat{\sigma}(\widehat{\rho}(x)) \cap \widehat{Val}}{E[x],\ \widehat{\rho},\ \widehat{\sigma},\ \widehat{t} \Downarrow_n (\widehat{v},\ \widehat{\sigma})}$$

$$\textsc{UnMemoized Ref}\ \frac{\begin{array}{c}\widehat{\alpha} = \widehat{\rho}(x) \\ (E'[e],\ \widehat{\rho}',\ \widehat{t}') \in \widehat{\sigma}(\widehat{\alpha}) \cap \widehat{Thunk} \\ E'[e],\ \widehat{\rho}',\ \widehat{\sigma},\ \widehat{t}' \Downarrow_n (\widehat{v},\ \widehat{\sigma}_v)\end{array}}{E[x],\ \widehat{\rho},\ \widehat{\sigma},\ \widehat{t} \Downarrow_n (\widehat{v},\ \widehat{ext}(\widehat{\sigma}_v,\ \widehat{\alpha},\ \widehat{v}))}$$

$$\textsc{App}\ \frac{\begin{array}{c}E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{\sigma},\ \widehat{t} \Downarrow_n (\widehat{v},\ \widehat{\sigma}_1) \\ (E'[\lambda x.e],\ \widehat{\rho}_0) \in \widehat{v} \\ \widehat{t}_1 = \lfloor E[(e_0\ e_1)] :: \widehat{t} \rfloor_m \qquad \widehat{\alpha} = (x,\ \widehat{t}_1) \\ E'[\lambda x.[e]],\ \widehat{\rho}_0[x \rightarrow \widehat{\alpha}],\ \widehat{ext}(\widehat{\sigma}_1,\ \widehat{\alpha},\ \{(E[([e_0]\ [e_1])],\ \widehat{\rho},\ \widehat{t})\}),\ \widehat{t}_1 \Downarrow_n \widehat{r}\end{array}}{E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{\sigma},\ \widehat{t} \Downarrow_n \widehat{r}}$$

$$\widehat{\Downarrow}_n\ \subseteq\ \widehat{Config} \times \widehat{Result}$$

Fig. 9. Abstract Call-by-Need Semantics

The abstract evaluation relation, $\widehat{\Downarrow}_n$, is defined in Figure 9. The abstract rules mirror the operational call-by-need semantics but operate over the power-set domains required for soundness. In the [Memoized Ref] rule, we simply filter the values from the storable set of the store entry and return them in the result. In the [UnMemoized Ref] rule, when multiple thunks reside at a single abstract address, the analysis non-deterministically explores them, joining the results into the abstract store. Unlike the concrete semantics, extending the store in the abstract setting utilizes a join operator $\widehat{ext}$, which merges new information with existing values rather than overwriting them:

$$\widehat{cfg} \in \widehat{Config} = \widehat{Cursor} \times \widehat{Store}$$
$$\widehat{csr} \in \widehat{Cursor} = EvalCtx \times Exp \times \widehat{Env} \times \widehat{Time}$$
$$\widehat{r} \in \widehat{Result} = \widehat{Val} \times \widehat{Store}$$
$$\widehat{\rho} \in \widehat{Env} = Var \rightharpoonup \widehat{Addr}$$
$$\widehat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{D}$$
$$\widehat{clo} \in \widehat{Clo} = EvalCtx \times Lam \times \widehat{Env} \times \widehat{Time}$$

$$\widehat{\alpha} \in \widehat{Addr} = Var \times \widehat{Time}$$
$$\widehat{d} \in \widehat{D} = \mathcal{P}(\widehat{Storable})$$
$$\widehat{v} \in \widehat{Val} = \mathcal{P}(\widehat{Clo})$$
$$\widehat{t} \in \widehat{Time} = \widehat{CallCtx}^*$$
$$\widehat{cc}^m \in \widehat{CallCtx}^m = (Call \times CallCtx^{m-1}) + Var + \epsilon$$
$$\widehat{s} \in \widehat{Storable} = \widehat{Thunk} + \widehat{Clo}$$
$$\widehat{th} \in \widehat{Thunk} = \widehat{Cursor}$$

Fig. 10. State Space for the DoDCFA Semantics

$$\widehat{ext}(\widehat{\sigma},\ \widehat{\alpha},\ \widehat{d}_0) = \widehat{\sigma}[\widehat{\alpha} \rightarrow \widehat{d} \cup \widehat{d}_0] \quad \text{where } \widehat{d} = \widehat{\sigma}(\widehat{\alpha})$$

$$\widehat{\sigma}(\widehat{\alpha}) = \begin{cases} \widehat{\sigma}(\widehat{\alpha}) & \widehat{\alpha} \in dom(\widehat{\sigma}) \\ \emptyset & \widehat{\alpha} \notin dom(\widehat{\sigma}) \end{cases}$$

To establish the correctness of this abstraction, we define an abstraction function $|\cdot|$ that maps concrete configurations to their abstract counterparts:

$$|\rho| = \lambda x.|\rho(x)| \qquad |t| = \lfloor t \rfloor_m \qquad |(\lambda x.e, \rho)| = \{(\lambda x.e, |\rho|)\} \qquad |(x,t)| = (x, |t|)$$
$$|(E[e], \rho, t)| = \{(E[e], |\rho|, |t|)\} \qquad |\sigma| = \lambda\widehat{\alpha}.\ \bigcup_{|\alpha| \models \widehat{\alpha}} |\sigma(\alpha)|$$
$$\widehat{\sigma}_0 \sqsubseteq \widehat{\sigma}_1 \iff \forall \widehat{\alpha} \in \widehat{Addr},\ \widehat{\sigma}_0(\widehat{\alpha}) \subseteq \widehat{\sigma}_1(\widehat{\alpha}) \qquad \widehat{v}_0 \sqsubseteq \widehat{v}_1 \iff \widehat{v}_0 \subseteq \widehat{v}_1$$

Abstract environments are formed by abstracting their constituent addresses, while timestamps are truncated to length m. Concrete closures and thunks are mapped to singleton sets containing their abstracted versions. The abstract store is constructed by partitioning the concrete store according to the address abstraction and taking the union of the values in each partition. We define a partial ordering $\sqsubseteq$ over abstract stores and values based on the subset relation, which allows us to formalize the soundness of our analysis:

Theorem 4.2. *If* $|\sigma| \sqsubseteq \widehat{\sigma}$ *and* $|\rho| \sqsubseteq \widehat{\rho}$ *and* $|t| \sqsubseteq \widehat{t}$ *and* $E[e], \rho, \sigma, t \Downarrow_n (v, \sigma_v)$ *then* $E[e], \widehat{\rho}, \widehat{\sigma}, \widehat{t} \widehat{\Downarrow}_n (\widehat{v}, \widehat{\sigma}_v)$ *where* $|v| \sqsubseteq \widehat{v}$ *and* $|\sigma_v| \sqsubseteq \widehat{\sigma}_v$

This theorem can be trivially proven by induction on the derivation of the abstract call-by-need evaluation $\widehat{\Downarrow}_n$.

4.5 DoDCFA Semantics

In this subsection, we define the DoDCFA semantics, which integrates forward abstract call-by-need evaluation with backward demand-driven search. Figure 10 defines the state space for DoDCFA semantics. The cursor is redefined to include the evaluation context, expression, environment, and

$$\textsc{Time Empty}\ \frac{}{[] \sim []} \qquad \textsc{Time Non-Empty}\ \frac{call :: \widehat{t}_n \sqsubseteq \widehat{cc}}{\widehat{cc} :: \widehat{t}_d \sim call :: \widehat{t}_n}$$

$$\textsc{Call Context Empty}\ \frac{}{[] \sqsubseteq []} \qquad \textsc{Call Context Non-Empty}\ \frac{\widehat{t}_n \sqsubseteq \widehat{cc}}{call :: \widehat{t}_n \sqsubseteq call :: \widehat{cc}} \qquad \textsc{Unknown Context}\ \frac{}{call :: \widehat{t}_n \sqsubseteq (?\ x)}$$

Fig. 11. Time equivalence between de bruijn Time and call sequence Time

timestamp, allowing for flexible navigation of the abstract syntax tree. To accurately model stack frames during both forward and backward traversal, we utilize a de Bruijn-indexed timestamp, represented as a list of call contexts. A call context is either a linked list of a call site to another call context, an unknown variable placeholder $(?\ x)$ representing an unresolved stack frame, or an empty list ϵ. This structure, inspired by Demand m-CFA [26], allows the analysis to "pop" timestamps to restore the calling context during backward search. We define a $\widehat{succ}$ and $\lfloor\cdot\rfloor_m$ operators to evolve the timestamp during function application:

$$\begin{aligned}
\widehat{succ}(call, []) &= \lfloor call :: [] \rfloor_m \\
\widehat{succ}(call, \widehat{cc} :: \widehat{t}) &= \lfloor call :: \widehat{cc} \rfloor_m \\
\lfloor \widehat{cc} \rfloor_0 &= \epsilon \\
\lfloor (?\ x) \rfloor_m &= (?\ x) \\
\lfloor E[(e_0\ e_1)] :: \widehat{cc} \rfloor_m &= E[(e_0\ e_1)] :: \lfloor \widehat{cc} \rfloor_{m-1}
\end{aligned}$$

The operator $\widehat{succ}$ takes two arguments: a call and a time. If the given time is not empty, it simply adds the call to the first call context of the given time and bound it to the length m if the entire call context is concrete. If the given time is empty, it simply returns a list with the given $call$. The operator $\lfloor\cdot\rfloor_m$ bounds the list to the length m, unless the call context is unknown.

The de Bruijn timestamp is related to the standard call-sequence timestamp via the equivalence relation defined in Figure 11. In this relation, the first call context in a de Bruijn list corresponds to the current stack trace, while the remainder of the list represents contexts inherited from closure evaluations. This dual representation allows DoDCFA semantics to leave a local scope (e.g., a lambda body) and precisely restore the environment and timestamp of the caller.

The DoDCFA semantics configuration consists of a cursor and a threaded store, while the result pairs an abstract value with the final store. To take account of the "unknown" reference in the store, we set an empty set to represent the placeholder, i.e., if an entry of a store is an empty set, it denotes an unknown storable that needs to be resolved.

The forward evaluation rules for DoDCFA semantics, defined in Figure 12, merge abstract call-by-need with Demand CFA mechanisms. The [Ref-Unknown-Eval] rule is the centerpiece of this integration: when a reference points to an "unknown" placeholder in the store, the analysis invokes the backward bind ($\widehat{\Uparrow}^b_d$) and trace ($\widehat{\Rightarrow}_d$) operators. These operators locate the appropriate call sites that could have bound the variable, allowing the analysis to memoize and evaluate the discovered argument expression.

The backward rules (Trace, Bind, and Find) are detailed in Figure 13. The Trace rules identify call sites by propagating backward through the program's control flow. When tracing through

$$\textsc{Lam-Eval}\quad \frac{}{(E[\lambda x.e],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\{(E[\lambda x.e],\ \widehat{\rho},\ \widehat{t})\},\ \widehat{\sigma})}$$

$$\textsc{Ref-Memoized-Eval}\quad \frac{\widehat{v} = \widehat{\sigma}(\widehat{\rho}(x)) \cap \widehat{Val}}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{\sigma})}$$

$$\textsc{Ref-Unmemoized-Eval}\quad \frac{\widehat{\alpha} = \widehat{\rho}(x) \qquad \widehat{th} \in \widehat{\sigma}(\alpha) \cap \widehat{Thunk} \qquad \widehat{th},\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{\sigma}_v)}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{ext}(\widehat{\sigma}_v,\ \widehat{\alpha},\ \widehat{v}))}$$

$$\textsc{Ref-Unknown-Eval}\quad \frac{\begin{array}{c}\widehat{\alpha} = \widehat{\rho}(x) \qquad \widehat{\sigma}(\widehat{\alpha}) = \emptyset \\ ((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x\ \widehat{\Uparrow}^b_d\ ((E'[\lambda x.[e']],\ \widehat{\rho}_0,\ cc :: \widehat{t}_1),\ \widehat{\sigma}_1) \\ (E'[\lambda x.e'],\ (\widehat{\rho}_0 \setminus x),\ \widehat{t}_1),\ \widehat{\sigma}_1 \mathrel{\widehat{\Rightarrow}}_d\ ((E_x[(e_0\ e_1)],\ \widehat{\rho}_x,\ \widehat{t}_x),\ \widehat{\sigma}_2) \\ (E_x[(e_0\ [e_1])],\ \widehat{\rho}_x,\ \widehat{t}_x),\ \widehat{ext}(\widehat{\sigma}_2,\ \widehat{\alpha}, \{(E_x[(e_0\ [e_1])],\ \widehat{\rho}_x,\ \widehat{t}_x)\})\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{\sigma}_v)\end{array}}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{ext}(\widehat{\sigma}_v,\ \widehat{\alpha},\ \widehat{v}))}$$

$$\textsc{App-Eval}\quad \frac{\begin{array}{c}(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ (\widehat{v},\ \widehat{\sigma}_1) \\ (E'[\lambda x.e],\ \widehat{\rho}',\ \widehat{t}_1) \in \widehat{v} \\ \widehat{t}_2 = \widehat{succ}(E[(e_0\ e_1)],\ \widehat{t}) :: \widehat{t}_1 \qquad \widehat{\alpha} = (x,\ \widehat{t}_2) \\ (E'[\lambda x.[e]],\ \widehat{\rho}'[x \to \widehat{\alpha}],\ \widehat{t}_2),\ \widehat{ext}(\widehat{\sigma}_1,\ \widehat{\alpha},\ \{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t})\})\ \widehat{\Downarrow}_d\ \widehat{r}\end{array}}{(E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_d\ \widehat{r}}$$

$$\widehat{\Downarrow}_d\ \subseteq\ \widehat{Config} \times \widehat{Result}$$

Fig. 12. DoDCFA Semantics Evaluation Rules (Forward)

a function argument, the analysis allocates a fresh address and wraps the argument in a thunk, ensuring that subsequent lookups are memoized. The Bind and Find operators manage scope transitions: the Bind operator "pops" the environment and timestamp when leaving a lambda body, while the Find operator "pushes" unknown frames when entering a scope with unresolved context.

To prove the soundness of DoDCFA semantics relative to the abstract call-by-need semantics, we first establish a lemma for the trace relation. This lemma shows that trace query soundly retrieves every correct call sites that apply the lambda expression that binds its parameter reference with a refined cursor.

Lemma 4.3. *If* $(E[\lambda x.e], \widehat{\rho}, \widehat{t}), \widehat{\sigma} \widehat{\Rightarrow}_d (E'[(e_0\ e_1)], \widehat{\rho}', \widehat{t}'), \widehat{\sigma}'$
Then $(E'[([e_0]\ e_1)], \widehat{\rho}', \widehat{t}'), \widehat{\sigma}' \widehat{\Downarrow}_d (E[\lambda x.e], \widehat{\rho}'', \widehat{t}''), \widehat{\sigma}''$
s.t. $\widehat{\rho}'' \sqsubseteq \widehat{\rho}$ *and* $\widehat{t}'' \sqsubseteq \widehat{t}$ *and* $\widehat{\sigma} \sqsubseteq \widehat{\sigma}''$

Lemma 4.3 can be proven by the soundness of the trace relation follows from existing proofs in Demand CFA literature [8, 26], adapted here to the big-step call-by-need setting with nontrivial extensions. Rather than prove it directly, we generalize it twice and recover it as an instance. First we generalize the *source*: the demanded expression need not be a λ but any expression, the transferred λ being whatever closure that expression evaluates to — a one-directional *trace-to-eval* transfer, of which the original lemma is the λ-instance (a λ evaluates to itself). Then we generalize the *direction*: this transfer is one direction of a *bidirectional* theorem relating $\widehat{\Rightarrow}_d$ and $\widehat{\Downarrow}_d$, from a

$$\textsc{Fun-Trace}\quad \frac{}{(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma} \mathrel{\widehat{\Rightarrow}_d} ((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Arg-Trace}\quad \frac{\begin{array}{c}(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma} \Downarrow_d\ (\widehat{v},\ \widehat{\sigma}_1) \qquad (E'[\lambda x.e],\ \widehat{\rho}_0,\ \widehat{t}') \in \widehat{v} \\ \widehat{t}_1 = \widehat{succ}(E[(e_0\ e_1)],\ \widehat{t}) :: \widehat{t}' \qquad \widehat{\alpha} = (x,\ \widehat{t}_1) \\ (E'[\lambda x.[e]],\ \widehat{\rho}'[x \to \alpha],\ \widehat{t}_1),\ \widehat{ext}(\widehat{\sigma}_1,\ \widehat{\alpha}, \{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t})\}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}_x \\ \widehat{cfg}_x \mathrel{\widehat{\Rightarrow}_d} \widehat{cfg}\end{array}}{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma} \mathrel{\widehat{\Rightarrow}_d} \widehat{cfg}}$$

$$\textsc{Bod-Trace}\quad \frac{\begin{array}{c}(E[\lambda x.e],\ (\widehat{\rho} \setminus x),\ \widehat{t}),\ \widehat{\sigma} \Rightarrow_d\ (E'[(e_0\ e_1)],\ \widehat{\rho}',\ \widehat{t}'),\ \widehat{\sigma}_1 \\ \widehat{cc}_1 = \widehat{succ}E'[(e_0\ e_1)],\ \widehat{t}' \qquad \widehat{cc}_1 \sqsubseteq \widehat{cc} \qquad (E'[(e_0\ e_1)],\ \widehat{\rho}',\ \widehat{t}'),\ \widehat{\sigma}_1 \Rightarrow_d\ \widehat{cfg}\end{array}}{(E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{cc} :: \widehat{t}),\ \widehat{\sigma} \Rightarrow_d\ \widehat{cfg}}$$

$$\textsc{Bind-Fun}\quad \frac{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}{((E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}$$

$$\textsc{Bind-Arg}\quad \frac{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}{((E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}$$

$$\textsc{Bind-Bod-Zero}\quad \frac{}{((E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} ((E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Bind-Bod-Succ}\quad \frac{x \neq y \qquad ((E[\lambda y.e],\ (\widehat{\rho} \setminus y),\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}{((E[\lambda y.[e]],\ \widehat{\rho},\ \widehat{cc} :: \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_d} \widehat{cfg}}$$

$$\textsc{Find-Ref}\quad \frac{}{((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} ((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Find-Fun}\quad \frac{((E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}$$

$$\textsc{Find-Lam}\quad \frac{x \neq y \qquad \widehat{t}_1 = (?\ y) :: \widehat{t} \qquad \widehat{\alpha} = (y,\ \widehat{t}_1) \qquad ((E[\lambda\ y.[e]],\ \widehat{\rho}[y \to \widehat{\alpha}],\ \widehat{t}_1),\ \widehat{ext}(\widehat{\sigma},\ \widehat{\alpha},\ \emptyset)),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}{((E[\lambda\ y.e],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}$$

$$\textsc{Find-Arg}\quad \frac{((E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_d} \widehat{cfg}}$$

$$\widehat{\Rightarrow}_d \subseteq \widehat{Config} \times \widehat{Config}$$
$$\widehat{\Uparrow}^f_d,\ \widehat{\Uparrow}^b_d \subseteq \widehat{Config} \times Var \times \widehat{Config}$$

Fig. 13. DoDCFA Semantics Trace/Bind/Find Rules (Backward)

trace to a call site recovering an evaluation of the function-demanded call site to the same λ, and dually from an evaluation to a λ recovering a trace of the λ-config to the call site, each with a refined environment and time and a grown store. Because the two directions are mutually dependent

through the cross-recursion of $\widehat{\Downarrow}_d$, $\widehat{\Rightarrow}_d$, and the auxiliary binder and use-site searches, they are established together.

The bidirectional theorem is proved by a single simultaneous induction over the four mutually inductive relations, taking the two transfer directions as the non-trivial motives (the binder and use-site relations carry the trivial motive). Each constructor case reduces to an application of the induction hypotheses together with a small set of supporting facts: *store monotonicity* (the store only grows, so a result lifts across each intervening write), *source-refinement monotonicity* (a more precise input env/time/store yields a correspondingly more precise output), a *callee-descent* step for application (after the operator's hypothesis places the chosen closure's trace at the call site, one pops out of the callee body and composes), and a few well-formedness facts (no shadowing of a closure's own paramete,cov and memoisation consistency). The environment, time, and store refinements compose by transitivity, giving the refined witnesses in the conclusion. The extensive proof can be found in the appendix in the Section A.

We then define a refinement relation between DoDCFA semantics stores and abstract call-by-need semantics stores, accounting for the unknown placeholder:

$$\widehat{\sigma}_n \sqsubseteq \widehat{\sigma}_d \iff \forall \alpha \in Addr,\ \widehat{\sigma}_n(\alpha) \sqsubseteq \widehat{\sigma}_d(\alpha)$$
$$\widehat{d}_d = \emptyset \rightarrow \widehat{d}_n \sqsubseteq \widehat{d}_d \qquad \widehat{d}_n \sqsubseteq \widehat{ss} \iff \widehat{d}_n \subseteq \widehat{ss}$$

If the DoDCFA semantics' store entry is an empty set, it is a placeholder that approximate all call-by-need storables. Otherwise, each store entry is compared in a subset relation. We now define the soundness theorem of DoDCFA semantics to abstract call-by-need semantics.

Theorem 4.4. *If* $E[e], \widehat{\rho}_n, \widehat{\sigma}_n, \widehat{t}_n \widehat{\Downarrow}_n (v_n, \widehat{\sigma}_{vn})$ *and* $\widehat{\rho}_n \sqsubseteq \widehat{\rho}_d, \widehat{\sigma}_n \sqsubseteq \widehat{\sigma}_d, \widehat{t}_n \sim \widehat{t}_d$ *then* $(E[e], \widehat{\rho}_d, \widehat{t}_d), \widehat{\sigma}_d \widehat{\Downarrow}_d (v_d, \widehat{\sigma}_{vd})$ *where* $v_n \sqsubseteq v_d$ *and* $\widehat{\sigma}_{vn} \sqsubseteq \widehat{\sigma}_{vd}$

This theorem can be trivially proven by induction on the derivation of $\widehat{\Downarrow}_d$, using the trace soundness lemma to handle the resolution of unknown references.

5 Soundness and Correctness of DoDCFA

In this section, we establish the global soundness and correctness of the DoDCFA. We demonstrate our proof strategy by constructing a "soundness bridge" that links DoDCFA semantics back to the fundamental call-by-value operational semantics. This bridge is composed of several semantic correspondences derived in the preceding sections.

The first segment of the bridge establishes the equivalence between the call-by-value and call-by-need operational semantics. We have previously shown that our call-by-need model is a conservative representation of the call-by-value model, ensuring that both evaluation strategies yield consistent results for a given expression. The second segment consists of the soundness relation between the concrete call-by-need operational semantics and its abstract counterpart. Through systematic abstraction, we ensure that the abstract call-by-need semantics serves as a sound over-approximation of all possible concrete executions. The final segment of the bridge links the abstract call-by-need semantics to the DoDCFA semantics. This link is justified by our lemma that the backward trace analysis for unknown variables soundly approximates the set of all possible arguments that could be bound to those variables at runtime. By the property of transitivity across these three segments, DoDCFA semantics is a sound and correct approximation of the call-by-value operational semantics.

We formally state the global soundness theorem for the DoDCFA semantics as follows:

Theorem 5.1. *If* $E[e], \rho, \sigma, t \Downarrow_v (v, \sigma_v)$ *where* $|\rho| \sqsubseteq \widehat{\rho}, |\sigma| \sqsubseteq \widehat{\sigma}, |t| \sqsubseteq \widehat{t}$

then $(E[e], \widehat{\rho}, \widehat{t}), \widehat{\sigma}\widehat{\Downarrow}_d(\widehat{v}, \widehat{\sigma}_v)$ *where* $|v| \sqsubseteq \widehat{v}, |\sigma_v| \sqsubseteq \widehat{\sigma}_v$

This theorem can be proven by transitivity over the established soundness relations between the intermediate semantics, ultimately relating the DoDCFA semantics to the foundational call-by-value operational semantics.

6 Application: Environment Analysis for Compiler Optimizations

6.1 Environment Analysis

Environment analysis is essential for justifying compiler optimizations in higher-order languages, such as procedural inlining and copy propagation. In these languages, a closure captures its lexical environment when a lambda expression is evaluated. This captured environment may contain bindings that differ significantly from the environment present when the closure is eventually applied. Reasoning about these differences requires formal methods to ensure environment equivalence between disparate program points.

Consider the program in Figure 14, a classic example of Shivers' environment problem [21]. Suppose a compiler must determine if it is safe to inline the call site labeled A. A standard CFA reveals that the only closure flowing to `h` is `(λ () x)`. Superficially, it might appear that the call to `h` could be replaced by the variable `x`. However, the closure flowing to `h` captures an environment where `x` is bound to `42`, whereas the environment at call site A binds `x` to `0`. Because `x` denotes different values in these two environments, inlining is unsafe; doing so would incorrectly change the program's output from `42` to `0`.

```
(let ([f (λ (x h)
           (if (= x 0)
               (h)^A
               (λ () x)))])
  (f 0 (f 42 #f)))
```

Fig. 14. Program Unsafe to Inline in Scheme

Several theories have been developed to reason about bindings within these environments. However, existing environment analyses rely heavily on an underlying CFA, either by being integrated into the analysis engine or by post-processing its output. Consequently, environment analysis inherits the scalability issues of the underlying CFA. Since traditional CFAs are exhaustive and often fail to scale, the environment information they produce is frequently unavailable to the tools that need it. Because environment analysis extends the reach of CFA, it remains computationally expensive and often impractical for use in production compilers.

As a demonstration of its utility, we extend the DoDCFA to perform environment analysis for the language defined in Section 3. Because DoDCFA significantly reduces the state space explored during evaluation, it can perform environment analysis with an efficiency that often exceeds exhaustive techniques while maintaining or improving precision. We first informally describe our approach to enabling environment analysis within DoDCFA and then formally extend the semantics with abstract counting. For this application case, we focus specifically on resolving the environment problem to verify the safety of procedural inlining for program call sites.

6.2 Adding Abstract Counting to DoDCFA

Abstract counting integrates naturally with both the forward and backward components of the DoDCFA semantics. The threaded store maintains an abstract count for each address, which is incremented whenever the forward evaluation or the backward trace query performs a fresh allocation. For the forward phase, this functions similarly to exhaustive analyses by tracking addresses during call-by-need evaluation, incrementing a count of an address that is freshly allocated. For the backward phase, the analysis increments counts as it allocates fresh addresses for unknown references encountered while traversing the AST toward an arbitrary entry point. If the analysis never rebinds these addresses—leaving the abstract count at exactly one—the reference is identified

as a "must-alias." By leveraging this counting technique, we can detect whether a variable of interest remains unchanged throughout the analysis, effectively performing an "unchanged variable analysis" within a demand-driven scope.

We demonstrate the mechanism of DoDCFA with abstract counting using two examples: one where inlining is safe, and another where it is not. Figure 15 shows a "Triangle" program in Scheme where a compiler seeks to determine if call site A is safe for inlining. Instead of analyzing the entire program, DoDCFA initiates evaluation specifically at the program point $E[f]$, targeting the operator `f` of the call site `A`. To establish the initial configuration, the analysis allocates abstract addresses for variables `xs`, `y`, `ys`, `f`, `zs`, `w`, and `ws`, initializing them as unknown references. Since `f` is an unknown reference, the analyzer invokes the bind and trace operators to discover that the closure `(λ (u) (+ y u))` flows to this site. The only free variable in this closure is `y`, which requires a must-alias relationship between the environment at call site A and the closure's captured environment. Because the evaluation traversal does not exit the scope where `y` is bound (the match expression), `y` resolves to the same address in both environments. Furthermore, as the analysis of `f` does not encounter a rebinding of `y`, the abstract count for that address remains one. The analysis, therefore, concludes that call site A is safe to inline.

```
(define (tri xs)
  (match xs
    [(list)
     (list)]
    [(cons y ys)
     (letrec ([map
               (λ (f zs)
                 (match zs
                   [(list)
                    (list)]
                   [(cons w ws)
                    (cons (f w)^A
                          (map f ws))]))])
       (cons y (map (λ (u) (+ y u)) (tri ys))))]))
```

Fig. 15. Triangle program in Scheme

Conversely, we reconsider the unsafe program from Figure 14. As before, the compiler queries the safety of inlining call site A, and the analysis starts at the operator `h`. The initial configuration allocates fresh addresses, pointing to unknown values, for `x` and `h` within the local scope. Evaluating `h` triggers a bind operator to the enclosing lambda and a trace query that identifies the call `(f 42 #f)` as the source of the closure (we omit a trace to `#f` since it is not an applicable closure). Evaluating `(f 42 #f)` rebinds the variable `x`, which `x` can potentially point to a different address or a value from the initial address allocated for `x`. In a context-insensitive analysis, the address for `x` is reused, causing its abstract count to increment beyond one and failing the must-alias check. In a context-sensitive analysis, the address for `x` in the current environment will be distinct from the address in the captured closure environment. In both cases, the analysis cannot guarantee that `x` points to the same value in both contexts, correctly identifying the inlining as unsafe.

6.3 Extending the Formalism with Abstract Counting

By leveraging a threaded store, we can extend the DoDCFA semantics with abstract counting to support must-alias analysis. Figure 16 defines the extended state space from the original DoDCFA semantics. The store is now a mapping from abstract addresses to a tuple consisting of a set of storables and an abstract count $\widehat{c}$. The abstract count is drawn from the lattice $\{0, 1, \infty\}$, representing the number of times an address has been allocated, where ∞ denotes any count greater than one. The remainder of the state space remains consistent with the standard DoDCFA semantics.

To track these counts accurately, we redefine store extension into two distinct operations: fresh extension ($\widehat{ext}^F$) and update extension ($\widehat{ext}$). The fresh extension, $\widehat{ext}^F$, is invoked during new allocations and increments the abstract count of the target address. It accepts a store, an address,

$$\widehat{cfg} \in \widehat{Config} = \widehat{Cursor} \times \widehat{Store}$$
$$\widehat{csr} \in \widehat{Cursor} = EvalCtx \times Exp \times \widehat{Env} \times \widehat{Time}$$
$$\widehat{\rho} \in \widehat{Env} = Var \rightharpoonup \widehat{Addr}$$
$$\widehat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup (\widehat{D} \times \widehat{Count})$$
$$\widehat{c} \in \widehat{Count} = \{0,\ 1,\ \infty\}$$
$$\widehat{clo} \in \widehat{Clo} = EvalCtx \times Lam \times \widehat{Env} \times \widehat{Time}$$
$$\widehat{th} \in \widehat{Thunk} = \widehat{Cursor}$$

$$\widehat{r} \in \widehat{Result} = \widehat{Val} \times \widehat{Store}$$
$$\widehat{\alpha} \in \widehat{Addr} = Var \times \widehat{Time}$$
$$\widehat{d} \in \widehat{D} = \mathcal{P}(\widehat{Storable})$$
$$\widehat{t} \in \widehat{Time} = \widehat{CallCtx}^{*}$$
$$\widehat{cc}^{m} \in \widehat{CallCtx}^{m} = (Call \times CallCtx^{m-1}) + Var + \epsilon$$
$$\widehat{s} \in \widehat{Storable} = \widehat{Thunk} + \widehat{Clo}$$
$$\widehat{v} \in \widehat{Val} = \mathcal{P}(\widehat{Clo})$$

Fig. 16. State Space for the DoDCFA Semantics with Abstract Counting

$$\widehat{ext}^{F}(\widehat{\sigma},\ \widehat{\alpha},\ \widehat{d}) = \widehat{\sigma}[\widehat{\alpha} \rightarrow (\widehat{d} \cup \widehat{d}_0,\ \widehat{incr}(\widehat{c}))] \quad \text{where } (\widehat{d}_0,\ \widehat{c}) = \widehat{\sigma}(\widehat{\alpha})$$

$$\widehat{ext}(\widehat{\sigma},\ \widehat{\alpha},\ \widehat{d}) = \widehat{\sigma}[\widehat{\alpha} \rightarrow (\widehat{d} \cup \widehat{d}_0,\ \widehat{c})] \quad \text{where } (\widehat{d}_0,\ \widehat{c}) = \widehat{\sigma}(\widehat{\alpha})$$

$$\widehat{\sigma}(\widehat{\alpha}) = \begin{cases} \widehat{\sigma}(\widehat{\alpha}) & \widehat{\alpha} \in dom(\widehat{\sigma}) \\ (\emptyset,\ 0) & \widehat{\alpha} \notin dom(\widehat{\sigma}) \end{cases}$$

$$\widehat{incr}(0) = 1$$
$$\widehat{incr}(1) = \infty$$
$$\widehat{incr}(\infty) = \infty$$

Fig. 17. Store operators for DoDCFA Semantics with abstract counting

and a domain set, returning a store where the count is evolved via the $\widehat{incr}$ operator. Conversely, the update extension, $\widehat{ext}$, is used when the analysis retrieves a pre-allocated address to memoize a result. This operator updates the set of storables at the given address but leaves the abstract count unchanged, preserving the precision of the must-alias information. These store operators are defined in Figure 17.

The forward evaluation rules for DoDCFA semantics with abstract counting are detailed in Figure 18. These rules follow the structure of the original DoDCFA semantics but apply the specialized store operators at critical junctions. Specifically, the [App-Eval] rule utilizes fresh extension to account for the allocation of new argument bindings. In contrast, the [Ref-Unmemoized-Eval] and [Ref-Unknown-Eval] rules use update extensions to memoize discovered values without falsely signaling a new allocation.

$$\textsc{Lam-Eval}\quad \frac{}{(E[\lambda x.e],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ (\{(E[\lambda x.e],\ \widehat{\rho},\ \widehat{t})\}, \widehat{\sigma})}$$

$$\textsc{Ref-Memoized-Eval}\quad \frac{(\widehat{d},\ \widehat{c}) = \widehat{\sigma}(\widehat{\rho}(x)) \qquad \widehat{v} = \widehat{d} \cap \widehat{Val}}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ (\widehat{v},\ \widehat{\sigma})}$$

$$\textsc{Ref-Unmemoized-Eval}\quad \frac{\widehat{\alpha} = \widehat{\rho}(x) \qquad (\widehat{d},\ \widehat{c}) = \widehat{\sigma}(\alpha) \qquad \widehat{th} \in \widehat{d} \cap \widehat{Thunk} \qquad \widehat{th},\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ (\widehat{v},\ \widehat{\sigma}_v)}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \Downarrow_{dc}\ (\widehat{v},\ \widehat{ext}(\widehat{\sigma}_v,\ \widehat{\alpha},\ \widehat{v}))}$$

$$\textsc{Ref-Unknown-Eval}\quad \frac{\begin{array}{c}\widehat{\alpha} = \widehat{\rho}(x) \qquad \widehat{\sigma}(\widehat{\alpha}) = (\emptyset,\ \widehat{c}) \\ ((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x\ \widehat{\Uparrow}^{b}_{dc}\ ((E'[\lambda x.[e']],\ \widehat{\rho}_0,\ \widehat{cc} :: \widehat{t}_1),\ \widehat{\sigma}_1) \\ (E'[\lambda x.e'],\ (\widehat{\rho}_0 \setminus x),\ \widehat{t}_1),\ \widehat{\sigma}_1\ \widehat{\Rightarrow}_{dc}\ ((E_x[(e_0\ e_1)],\ \widehat{\rho}_x,\ \widehat{t}_x),\ \widehat{\sigma}_2) \\ (E_x[(e_0\ [e_1])],\ \widehat{\rho}_x,\ \widehat{t}_x),\ \widehat{ext}(\widehat{\sigma}_2,\ \widehat{\alpha},\ \{(E_x[(e_0\ [e_1])],\ \widehat{\rho}_x,\ \widehat{t}_x)\})\ \widehat{\Downarrow}_{dc}\ (\widehat{v},\ \widehat{\sigma}_v)\end{array}}{(E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ (\widehat{v},\ \widehat{ext}(\widehat{\sigma}_v,\ \widehat{\alpha},\ \widehat{v}))}$$

$$\textsc{App-Eval}\quad \frac{\begin{array}{c}(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ (\widehat{v},\ \widehat{\sigma}_1) \\ (E'[\lambda x.e],\ \widehat{\rho'},\ \widehat{t}_1) \in \widehat{v} \\ \widehat{t}_2 = \widehat{succ}(E[(e_0\ e_1)],\ \widehat{t}) :: \widehat{t}_1 \qquad \alpha = (x,\ \widehat{t}_2) \\ (E'[\lambda x.[e]],\ \widehat{\rho'}[x \to \widehat{\alpha}],\ \widehat{t}),\ \widehat{ext}^{F}(\widehat{\sigma}_1,\ \widehat{\alpha},\ \{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t})\})\ \widehat{\Downarrow}_{dc}\ \widehat{r}\end{array}}{(E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \widehat{\Downarrow}_{dc}\ \widehat{r}}$$

$$\widehat{\Downarrow}_{dc}\ \subseteq\ \widehat{Config} \times \widehat{Result}$$

Fig. 18. DoDCFA Semantics with Abstract Counting Evaluation Rules (Forward)

The backward analysis rules, including Trace, Bind, and Find, are similarly updated in Figure 19. The [Arg-Trace] rule employs fresh extension to allocate addresses for discovered argument expressions before proceeding with the find operator. The Find operator also uses fresh extension when descending into a lambda body, extending the store entry with an empty set and an incremented count to represent an unknown binding. Because the Bind operator only traverses existing lexical structures without modifying the store, its rules remain identical to those in the original DoDCFA semantics. As a relation, we define the $\sqsubseteq$ between original DoDCFA smenatics store $\widehat{\sigma}_d$ and DoDCFA semantics with abstract counting store $\widehat{\sigma}_{dc}$.

$$\widehat{\sigma}_d \sqsubseteq \widehat{\sigma}_{dc} \iff \forall \widehat{\alpha} \in Addr,\ \widehat{\sigma}_d(\widehat{\alpha}) \sqsubseteq \widehat{\sigma}_{dc}(\widehat{\alpha})$$
$$\widehat{d}_d \sqsubseteq (\widehat{d}_{dc}, \widehat{c}) \iff \widehat{d}_d \sqsubseteq \widehat{d}_{dc}$$

With the store relation, the soundness of this extension follows naturally, as abstract counting only refines the information tracked within the store without altering the reachability of the underlying analysis. We formalize the theorem as the following:

Theorem 6.1. *If* $(E[e], \widehat{\rho}, \widehat{t}), \widehat{\sigma}_d \widehat{\Downarrow}_d (v, \widehat{\sigma}_{vd})$ *and* $\widehat{\sigma}_d \sqsubseteq \widehat{\sigma}_{dc}$
then $(E[e], \widehat{\rho}, \widehat{t}), \widehat{\sigma}_{dc} \widehat{\Downarrow}_{dc} (v, \widehat{\sigma}_{vdc})$ *where* $\widehat{\sigma}_{vd} \sqsubseteq \widehat{\sigma}_{vdc}$

$$\textsc{Fun-Trace}\quad \frac{}{(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma} \mathrel{\widehat{\Rightarrow}_{dc}} ((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Arg-Trace}\quad \frac{\begin{array}{c}(E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}\ \Downarrow_{dc}\ (\widehat{v},\ \widehat{\sigma}_1) \qquad (E'[\lambda x.e],\ \widehat{\rho}_0,\ \widehat{t}') \in \widehat{v} \\ \widehat{t}_1 = \widehat{succ}(E[(e_0\ e_1)],\ \widehat{t}) :: \widehat{t}' \qquad \widehat{\alpha} = (x,\ \widehat{t}_1) \\ ((E'[\lambda x.[e]],\ \widehat{\rho}'[x \to \widehat{\alpha}],\ \widehat{t}_1),\ \widehat{ext}^F(\widehat{\sigma}_1,\ \widehat{\alpha},\ \{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t})\})),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}_x \\ \widehat{cfg}_x \mathrel{\widehat{\Rightarrow}_{dc}} \widehat{cfg}\end{array}}{(E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma} \mathrel{\widehat{\Rightarrow}_{dc}} \widehat{cfg}}$$

$$\textsc{Bod-Trace}\quad \frac{\begin{array}{c}(E[\lambda x.e],\ (\widehat{\rho} \setminus x),\ \widehat{t}),\ \widehat{\sigma} \Rightarrow_{dc}\ (E'[(e_0\ e_1)],\ \widehat{\rho}',\ \widehat{t}'),\ \widehat{\sigma}_1 \\ \widehat{cc}_1 = \widehat{succ}E'[(e_0\ e_1)],\ \widehat{t}' \qquad \widehat{cc}_1 \sqsubseteq \widehat{cc} \qquad (E'[(e_0\ e_1)],\ \widehat{\rho}',\ \widehat{t}'),\ \widehat{\sigma}_1 \Rightarrow_{dc} \widehat{cfg}\end{array}}{(E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{cc} :: \widehat{t}),\ \widehat{\sigma} \Rightarrow_{dc} \widehat{cfg}}$$

$$\textsc{Bind-Fun}\quad \frac{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}{((E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}$$

$$\textsc{Bind-Arg}\quad \frac{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}{((E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}$$

$$\textsc{Bind-Bod-Zero}\quad \frac{}{((E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} ((E[\lambda x.[e]],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Bind-Bod-Succ}\quad \frac{x \neq y \qquad ((E[\lambda y.e],\ (\widehat{\rho} \setminus y),\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}{((E[\lambda y.[e]],\ \widehat{\rho},\ \widehat{cc} :: \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^b_{dc}} \widehat{cfg}}$$

$$\textsc{Find-Ref}\quad \frac{}{((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} ((E[x],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma})}$$

$$\textsc{Find-Fun}\quad \frac{((E[([e_0]\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}$$

$$\textsc{Find-Lam}\quad \frac{x \neq y \qquad \widehat{t}_1 = (?\ y) :: \widehat{t} \qquad \widehat{\alpha} = (y,\ \widehat{t}_1) \qquad ((E[\lambda y.[e]],\ \widehat{\rho}[y \to \widehat{\alpha}],\ \widehat{t}_1),\ \widehat{ext}^F(\widehat{\sigma},\ \widehat{\alpha},\ \emptyset)),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}{((E[\lambda y.e],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}$$

$$\textsc{Find-Arg}\quad \frac{((E[(e_0\ [e_1])],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}{((E[(e_0\ e_1)],\ \widehat{\rho},\ \widehat{t}),\ \widehat{\sigma}),\ x \mathrel{\widehat{\Uparrow}^f_{dc}} \widehat{cfg}}$$

$$\widehat{\Rightarrow}_{dc} \subseteq \widehat{Config} \times \widehat{Config}$$
$$\widehat{\Uparrow}^f_{dc},\ \widehat{\Uparrow}^b_{dc} \subseteq \widehat{Config} \times Var \times \widehat{Config}$$

Fig. 19. DoDCFA Semantics with abstract counting Trace/Bind/Find Rules (Backward Analysis)

This theorem can be trivially proven by induction on the structure of the evaluation derivation, demonstrating that the count-augmented store remains a sound over-approximation of the standard abstract store. Consequently, this theorem can be added to the transitivity theorem previously discussed to build soundness to the standard call-by-value operational semantics.

7 DoDCFA Implementation and Evaluation

In this section, we evaluate the empirical effectiveness of the DoDCFA. We have implemented DoDCFA using the Abstracting Definitional Interpreters (ADI) framework [4] to construct a high-level, big-step abstract interpreter. To establish a baseline for comparison, we also implemented a standard k-CFA augmented with abstract counting within the same ADI environment. We compare DoDCFA relative to k-CFA rather than Demand m-CFA because Demand m-CFA does not admit abstract counting: it lacks a persistent store in which bindings are made, precluding environment analysis entirely. Moreover, Demand m-CFA's flow-insensitivity would make it imprecise for the must-alias reasoning central to our evaluation. Our evaluation compares DoDCFA relative to this k-CFA baseline across two primary axes: performance and precision. Performance is quantified by measuring both the total execution time and the cardinality of the state space explored during the analysis of each benchmark. Precision is evaluated by measuring the number of uniquely identifiable, inlinable call sites discovered by each analysis.

Before presenting our results, we briefly describe the necessary extensions to the core DoDCFA formalism. While our formal model targets a pure lambda calculus, our implementation extends this language to support the primitives and data structures required to execute standard benchmark programs. Following this description, we present an evaluation using to compare the scalability and accuracy of DoDCFA relative to traditional k-CFA using the following two category of benchmark:

(1) **R6RS Benchmark Suite** [24]: We selected a subset of the R6RS benchmark suite, excluding programs that contain side effects such as mutations and exceptions, as DoDCFA does not support their analysis. The selected programs range from 1–75 lines and are organized into three groups by purpose. The first group—`eta`, `kcfa2`, `kcfa3`, `mj09`, `blur`, `facehugger`, `map-pattern`, and `kcfa10`—stress-tests the precision of CFAs. The second group—`sat`, `rsa`, `regex`, and `solovay-strassen`—evaluates performance on real-world algorithms. The third group—`ack`, `fib`, `tak`, `cpstak`, and `sum`—targets recursive algorithms.
(2) **Larger Programs**: - We evaluated the performance of both DoDCFA and k-CFA on a small suite of programs (written with the aid of LLM agent): a Huffman encoder, a csp solver, a ray tracer, llrb tree implementation, metacircular scheme interpreter. These programs range from 256–341 lines.

7.1 Extending the Language

To support the analysis of real-world benchmarks, we extend the core DoDCFA formalism with several common programming constructs. First, we introduce `let` and `letrec` expressions, which are straightforward to integrate into the DoDCFA. Unlike lambda abstractions, `let` bindings are deterministic, allowing the analyzer to bind argument expressions in the store as it traverses the binding site. For `letrec`, the analyzer performs this store extension when entering the binding definitions themselves to support recursive references, rather than waiting to enter the body expression.

Second, we extend the language to support multi-parameter functions and n-ary call sites. We update the evaluation contexts to include each argument position in a call site's argument list, enabling the cursor to navigate into any arbitrary operand. Upon entering a function body, the analyzer allocates distinct addresses in the environment and store for each corresponding parameter variable.

Third, we implement primitive operations by following a pattern similar to function applications. The analyzer utilizes specific evaluation contexts for primitive argument lists and resolves the operation based on the semantics of the primitive operator.

Fourth, we incorporate conditional branching via `if` expressions, which serve as the target for desugaring all other conditional constructs. The analyzer evaluates the condition expression and, based on the resulting value, non-deterministically or deterministically selects the appropriate branch to continue the evaluation.

Finally, we add support for variant data types, which are essential for representing structures like lists in our benchmark programs. We introduce a new value type for variants that includes a type identifier, a variant constructor ID, and a list of addresses pointing to the field data in the store. This representation allows the DoDCFA to precisely track the flow of structured data throughout the analysis of the R6RS benchmarks.

7.2 Evaluation Setup

We evaluate the empirical performance of DoDCFA against k-CFA in both context-insensitive (0CFA) and context-sensitive (1CFA) configurations. While k-CFA executes as an exhaustive, top-down analysis, DoDCFA leverages its demand-driven nature to initiate a separate, independent analysis for each call site's function. For k-CFA, we measure the time from the start to the end of the single top-down analysis, from which inline-safety conclusions for all call sites are derived. For DoDCFA, we measure the time of each per-call-site analysis independently and sum them to obtain the total execution time for a given program; each such analysis determines the inline-safety of exactly one call site. This comparison is fair because both approaches perform precisely the work needed to determine the inline-safety of all call sites in a given program. For example, if a program contains 10 call sites, k-CFA runs a single analysis and derives inline-safety conclusions for all 10 call sites from its result, whereas DoDCFA runs 10 independent analyses and sums their execution times.

The benchmarks were conducted on an Ubuntu machine equipped with an Intel® Core™ i7-6700 CPU and 32 GB of RAM. Figure 20 and Figure 23 presents the comprehensive results, where time is recorded in milliseconds and ϵ denotes a duration below 1ms. The "Cfgs" column tracks the total number of configurations explored, serving as a proxy for state-space cardinality, while "Inls" represents the count of identified inline-safe call sites.

7.3 R6RS Benchmark Suite Evaluation Result

The table of the comprehensive result is presented in Figure 20. The visualization of the evaluation result in double bar graphs is presented in Figure 21. Overall for R6RS benchmark suite, DoDCFA exhibits remarkably low execution times, frequently remaining within the single-digit millisecond range even for complex benchmarks like `cpstak` and `map-pattern`. In contrast, k-CFA experiences a dramatic expansion in state space as program size and complexity increase. For larger programs or known k-CFA worst-case scenarios, the exhaustive approach frequently times out or experiences exponential growth in analysis time. Notably, the set of inline-safe call sites detected by DoDCFA is a strict superset of those detected by k-CFA across all benchmarks and sensitivity settings.

Figure 22 summarizes the average performance and precision gains of DoDCFA relative to the k-CFA baseline. We note that this summary does not account for programs that have timed out for the k-CFA. In a 0CFA setting, DoDCFA explores 86.55% fewer configurations on average, demonstrating its ability to eliminate redundant state exploration even without context sensitivity. Furthermore, DoDCFA identifies 28.09% more inlineable call sites than 0CFA, highlighting its superior reasoning capabilities for environment-sensitive optimizations.

	0CFA						1CFA					
	k-CFA-AC ($k = 0$)			DoDCFA-AC ($m = 0$)			k-CFA-AC ($k = 1$)			DoDCFA-AC ($m = 1$)		
Programs	Time	Cfgs	Inls	Time	Cfgs	Inls	Time	Cfgs	Inls	Time	Cfgs	Inls
eta	ϵ	15	2	ϵ	4	2	ϵ	15	2	ϵ	4	2
ack	1ms	62	4	ϵ	6	4	111ms	818	4	ϵ	6	4
fib	ϵ	32	3	ϵ	5	3	5ms	139	3	ϵ	5	3
tak	1ms	51	5	ϵ	7	5	180ms	1007	5	ϵ	7	5
cpstak	100ms	462	6	2ms	44	6	-	-	-	2ms	44	6
sum	ϵ	36	3	ϵ	6	3	1ms	49	3	ϵ	6	3
kcfa2	2ms	86	5	ϵ	21	9	39ms	235	5	ϵ	21	9
kcfa3	5ms	133	6	ϵ	25	11	24092ms	4740	6	ϵ	25	11
mj09	1ms	49	5	ϵ	12	6	1ms	57	6	ϵ	12	6
sat	31003ms	2667	6	ϵ	26	6	-	-	-	ϵ	26	6
blur	3ms	95	5	ϵ	22	8	4ms	110	8	ϵ	22	8
rsa	73ms	872	15	ϵ	27	15	96587ms	25394	15	ϵ	27	15
facehugger	1ms	71	5	ϵ	15	6	9ms	173	6	ϵ	15	6
map-pattern	35179ms	7917	5	2ms	57	7	14339ms	2883	7	ϵ	27	7
solovay-strassen	134ms	1124	9	ϵ	17	9	-	-	-	ϵ	17	9
kcfa10	409ms	1068	13	1ms	54	32	-	-	-	1ms	54	32

Fig. 20. R6RS Benchmark: 0CFA and 1CFA performance between k-CFA and DoDCFA. "-" denotes a timeout after 1 hour.

In the 1CFA setting, DoDCFA's efficiency advantage increases to a 91.32% reduction in explored configurations. This suggests that while k-CFA becomes prohibitively expensive as sensitivity increases, DoDCFA maintains a highly localized and efficient computation space. Interestingly, in some cases such as `map-pattern`, the workload under the 1CFA setting is actually reduced for both DoDCFA and k-CFA compared to 0CFA, as context-sensitivity eliminates spurious flows that would otherwise require exploration. Although the precision gap narrows to 13.61% as k-CFA becomes more accurate with context sensitivity, DoDCFA remains both more efficient and more precise across the board.

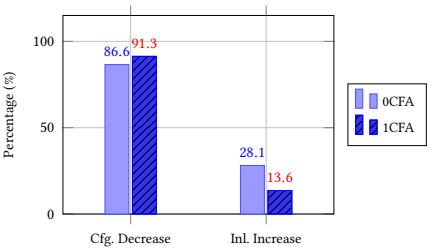


Fig. 22. Summary of average DoDCFA performance and precision gains

7.4 Larger Programs Evaluation Result

The comprehensive result for larger programs is presented in Figure 23, which we only present the result of 0CFA performance of DoDCFA. **The results for k-CFA were omitted since they timed out (1 hour) for all programs.** We also omitted the results for 1CFA of DoDCFA since there were no significance difference from 0CFA's result. For all programs in the suite, k-CFA exhausts the 1-hour time limit without completing its analysis in both the 0CFA and 1CFA settings. Meanwhile, DoDCFA keeps the analysis time well under one second and exhibits strong precision, with a notably high ratio of inline-safe call sites detected across all programs. Furthermore, the number of explored configurations remains identical between 0CFA and 1CFA settings for all programs, suggesting that DoDCFA's demand-driven nature keeps the analysis space stable even as context sensitivity increases. Our results show that DoDCFA makes control-flow analysis for large programs both scalable and precise.

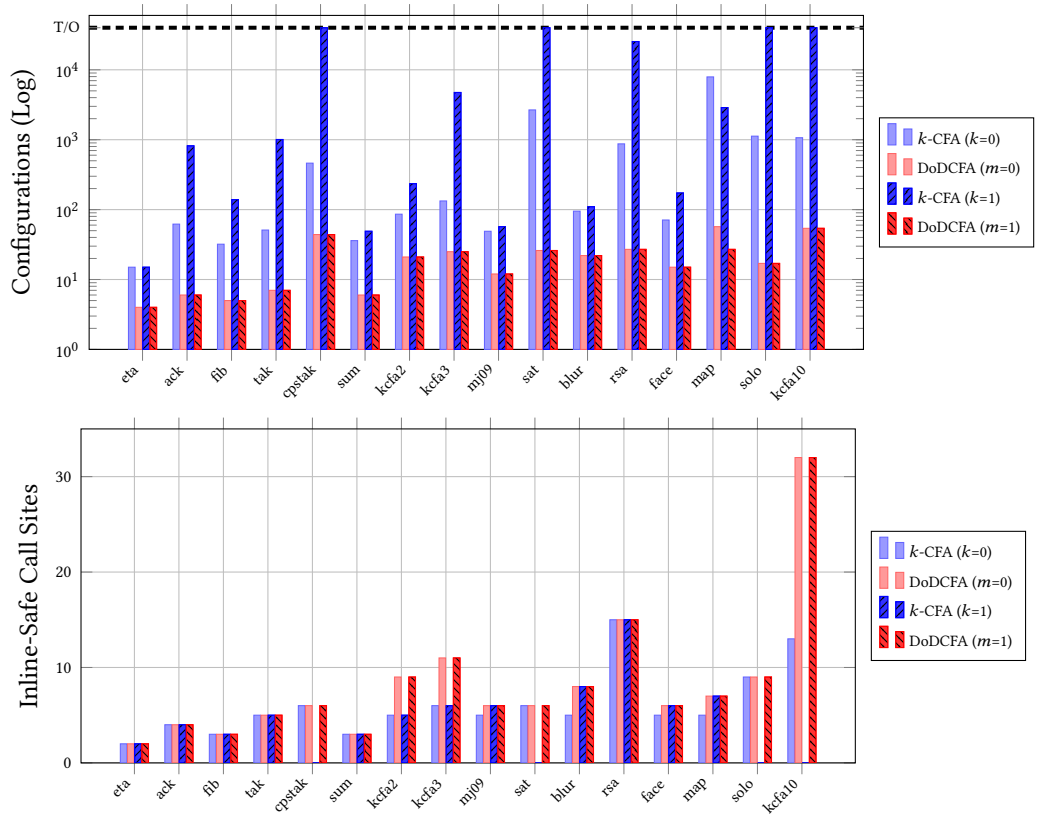


Fig. 21. Comparative evaluation of analysis cost and optimization benefit. The configuration count **(upper graph, lower is better, dashed line set for Timeout (T/O))** on a logarithmic scale illustrates that DoDCFA consistently reduces the state-space cardinality by orders of magnitude compared to k-CFA, effectively eliminating the state-space explosion typically associated with increasing context sensitivity. The inline-safe call site discovery **(lower graph, higher is better, absence of bar indicates timeout)** demonstrates that despite its significantly lower computational cost, DoDCFA (m=0) frequently recovers the same or greater precision than the more expensive k-CFA (k=1).

	0CFA		
	DoDCFA-AC (m = 0)		
Programs	Time	Cfgs	Inls
huffman	5ms	175	96
raytracer	54ms	425	221
llrb-tree	10ms	392	244
csp-solver	14ms	229	100
metacircular	12ms	275	216

Fig. 23. Larger Programs Benchmark Evaluation Result for DoDCFA in 0CFA Setting

8 Discussion

8.1 A Flexible Framework for Control-Flow Analysis

The DoDCFA provides a highly flexible approach to control-flow analysis, offering several distinct advantages over traditional models. By initiating analysis at arbitrary program points via demand-driven search, DoDCFA achieves superior precision and efficiency compared to exhaustive CFA. Our empirical results confirm that DoDCFA outcompetes k-CFA in both execution speed and the discovery of inline-safe call sites. This boost in efficiency is evidenced by the significantly lower number of configurations explored, allowing DoDCFA to analyze relatively large-scale benchmarks like `rsa` and `solovay-strassen` in single-digit milliseconds. To further demonstrate scalability, we evaluated DoDCFA on a suite of larger programs ranging from 256 to 341 lines, and DoDCFA completes all analyses well under one second, with execution times ranging from 5ms to 54ms, while maintaining high precision. Such flexibility enables developers to analyze critical fragments of large software systems without the computational burden of a full-program sweep.

A primary strength of the DoDCFA is its ability to facilitate the direct adoption of well-established exhaustive CFA techniques within a demand-driven setting. As demonstrated in our evaluation, we successfully integrated abstract counting into DoDCFA, yielding precision results that surpass those of traditional exhaustive CFA by enabling a more granular "must-alias" reasoning. Because DoDCFA maintains a structured store and environment, it serves as a hospitable host for a wide array of existing analysis extensions—such as abstract garbage collection or heap fragments—that were previously difficult to reconcile with pure demand-driven models. This modularity ensures that DoDCFA is flexible enough to accommodate these sophisticated techniques, allowing researchers to enhance both the performance and the insight-gathering capabilities of their analyzers without needing to develop entirely new formalisms to fit the demand-driven paradigm.

Furthermore, DoDCFA enables an incremental approach to analysis development. Since the framework avoids exhaustive exploration, the analysis designer is not required to implement semantic rules for every language feature present in a target program. If a specific query is independent of certain features, the analyzer can simply bypass them. This modularity is particularly beneficial for higher-order control-flow analysis, as complex language features are often orthogonal to the flow of closures. Consequently, DoDCFA allows for the construction of lightweight, specialized analyzers that focus exclusively on the properties relevant to the user's query.

8.2 Best from the Both Worlds

DoDCFA synthesizes the strengths of forward and backward analysis to mitigate their respective weaknesses. By employing backward search, the framework demands only the information necessary for a specific query, resolving the scalability and over-approximation issues inherent in exhaustive forward analysis. This targeted exploration ensures that DoDCFA reaches a sound result while exploring a fraction of the state space required by traditional methods.

Simultaneously, DoDCFA addresses the efficiency limitations of purely demand-driven frameworks like Demand CFA. Demand CFA suffers from an inability to memoize results during its recursive search, leading to redundant traversals. For example, Demand m-CFA [26] incurs significant overhead by repeatedly tracing all possible call sites to validate a timestamp. In contrast, DoDCFA maintains an explicit timestamp and store during its forward component, eliminating the need for the redundant search patterns found in purely backward models. By bridging these two paradigms, DoDCFA occupies a unique "sweet spot" in the spectrum of static analysis.

8.3 Path- and Flow-Sensitivity, Store Widening

The introduction of an explicit store in DoDCFA enables the implementation of various analysis sensitivities within a demand-driven pricing model. Full path- and flow-sensitivity [6] are typically achieved by altering the position and granularity of the fact base within the analysis state. Placing a store within each configuration enables path sensitivity, while mapping each program point to a specific store provides flow sensitivity. Conversely, a global store—a technique known as store widening—reduces cost at the expense of precision.

DoDCFA makes these high-precision sensitivities more affordable because it only explores divergent paths when the query necessitates it. While path-sensitivity is often prohibitively expensive in exhaustive CFA, DoDCFA's localized search makes such precision accessible for real-world applications. This allows analysis designers to tune the sensitivity of the analyzer to meet the specific precision requirements of the optimization being performed.

8.4 Limitations

As mentioned in our section discussing the benchmark selection, a primary limitation of the DoDCFA is its inability to analyze programs with side effects, such as mutable state or exceptions. Because our model is founded on call-by-need semantics, it cannot easily reason about the strict evaluation order required to soundly approximate side effects in a call-by-value setting. In call-by-need evaluation, the lack of a fixed execution sequence makes it difficult to track the state of a store affected by mutations.

Despite this, we believe DoDCFA remains highly valuable for analyzing the higher-order structural properties of programs. In many functional contexts, the flow of closures is orthogonal to the effects being performed, allowing DoDCFA to reason about control flow even in the presence of ignored side effects. However, extending DoDCFA to support effects is a key objective for our future work. One potential solution involves syntax labeling to manifest that an expression contains an effect—similar to Haskell's do notation—to simulate effectful evaluation within a lazy framework. Alternatively, we are exploring an alternate model that replaces the forward component with a call-by-value semantics to more naturally accommodate side-effect analysis in local regions. Such extensions would broaden DoDCFA's applicability to mainstream, multi-paradigm languages that increasingly incorporate higher-order features.

9 Related Work

Our work builds upon the foundational literature of control-flow analysis, including k-CFA [20], CFA2 [25], the Abstracting Abstract Machine (AAM) framework [11], P4F [10], and Abstracting Definitional Interpreters (ADI) [4]. Recent advancements in the field include trace-based control-flow analysis [18], which enhances both precision and efficiency by utilizing call-site sequences to detect and handle cycles within the analysis.

Demand-driven static analysis [12] has long served as a promising alternative to exhaustive methods by focusing computation on specific program properties. Prior work in this space includes Boomerang [23], which applies a demand-driven approach to pointer analysis for Java, and DDPA [5], which introduces demand-driven lookup for higher-order programs. Building on this lineage, Demand CFA [8, 26] brought demand-driven analysis to higher-order languages; though initially restricted to context-insensitive 0-CFA, it has since enabled practical applications such as on-demand call graph construction for JavaScript [19]. Pure Demand Operational Semantics [22] offers a complementary formulation for demand-driven reasoning in higher-order settings. Our work extends the Demand CFA framework by incorporating environment-sensitive reasoning, building directly on this body of work.

Environment analysis provides a rigorous framework for resolving aliasing problems by reasoning about the equivalence of bindings within higher-order lexical contexts. The earliest attempt at such reasoning was Shivers' reflow analysis [21], which performed a secondary pass over an existing CFA result to deduce environment properties. ΔCFA [9, 15] later refined this approach by utilizing frame strings to abstract the environment and resolve alias problems more directly. To address the inherent inefficiency of reflow analysis, Unchanged Variable Analysis (UVA) [1] tracks the invariance of variable content to verify inlining safety without a full re-analysis. More recently, the technique of environment-sharing [2] has enhanced precision by explicitly tracking when portions of an environment are shared across different execution contexts.

Abstract counting [13, 14, 16] is a widely used technique for must-alias analysis that tracks whether a heap address corresponds to a unique allocation. Must-alias information is particularly critical for analyzing mutations, as it enables "strong updates" where new values replace old ones rather than being merged. Modern frameworks continue to adapt abstract counting for high-precision tasks, such as Γ-CFA [16], which manages garbage collection for CFA and uses abstract counting, and heap-fragment settings [7] to achieve pushdown precision alongside garbage collection.

10 Conclusion and Future Work

We have presented Demand-on-Demand Control-flow Analysis (DoDCFA), a hybrid exhaustive–demand-driven control-flow analysis that integrates the backward search mechanisms of Demand CFA with an abstracted call-by-need semantics. DoDCFA replaces the redundant explorations of exhaustive CFA with targeted, query-driven evaluation while overcoming the efficiency bottlenecks of pure demand-driven models. Empirical evaluations demonstrate that DoDCFA consistently outperforms the standard k-CFA baseline in both performance and precision, identifying more inline-safe call sites while exploring a fraction of the state space.

The introduction of a store into the demand-driven setting opens several directions for future work. Our primary objective is to extend DoDCFA to support effectful operations, including mutable state and exception handling, which would broaden its applicability to languages such as JavaScript, TypeScript, and Python. Another direction is the integration of logic flow analysis, incorporating path conditions and branch predicates to refine precision for properties that depend on complex control-flow decisions. Finally, we intend to explore store optimization techniques such as demand-driven garbage collection and heap fragmentation, which could further enhance the precision of must-alias reasoning.

11 Acknowledgments

We acknowledge the use of AI assistants for refining the prose, writing, proofreading, and preparing technical visualizations for this paper.

A Extensive Proof for Lemma 4.3

A.1 Lemma to Prove: Lemma 4.3

Lemma 4.3. *If* $(E[\lambda x.e], \widehat{\rho}, \widehat{t}), \widehat{\sigma} \widehat{\Longrightarrow}_d (E'[(e_0\ e_1)], \widehat{\rho}', \widehat{t}'), \widehat{\sigma}'$
Then $(E'[([e_0]\ e_1)], \widehat{\rho}', \widehat{t}'), \widehat{\sigma}' \widehat{\Downarrow}_d (E[\lambda x.e], \widehat{\rho}'', \widehat{t}''), \widehat{\sigma}''$
s.t. $\widehat{\rho}'' \sqsubseteq \widehat{\rho}$ *and* $\widehat{t}'' \sqsubseteq \widehat{t}$ *and* $\widehat{\sigma} \sqsubseteq \widehat{\sigma}''$

It reads: if a λ-expression traces backward to a call site, then demanding the *function* of that call site evaluates to the same λ, with a refined environment and time and a grown store.

We do not prove it directly: the λ-config and the call site are linked through the entire demand chain, a link cleanest to establish in a more general form, so we generalize the lemma twice and

recover it as an instance. First we generalize the *source*: the demanded expression need not be a λ but any expression e, the transferred λ being whatever closure e evaluates to (Theorem A.1). Then we generalize the *direction*, proving the mutually dependent trace-to-eval and eval-to-trace transfers together as a single bidirectional theorem (Theorem A.2, proved in Section A.4). Lemma 4.3 is then the $e := \lambda x.e$ instance of Theorem A.1, in turn Part 1 of Theorem A.2.

A.2 First Generalization: An Arbitrary Source, One Direction

Theorem A.1 (Trace-to-Eval transfer). *Suppose the source both traces to a call site and evaluates to a λ:*

$$(E[e],\, \widehat{\rho},\, \widehat{t}) \Rrightarrow_d (E'[(e_0\ e_1)],\, \widehat{\rho}',\, \widehat{t}'),\, \widehat{\sigma}',$$
$$(E[e],\, \widehat{\rho},\, \widehat{t}),\, \widehat{\sigma}\ \widehat{\Downarrow}_d\ (E''[\lambda x.e''],\, \widehat{\rho}'',\, \widehat{t}''),\, \widehat{\sigma}''.$$

Then

$$(E'[([e_0]\ e_1)],\, \widehat{\rho}',\, \widehat{t}'),\, \widehat{\sigma}'\ \widehat{\Downarrow}_d\ (E''[\lambda x.e''],\, \widehat{\rho}''',\, \widehat{t}'''),\, \widehat{\sigma}'''$$

with $\widehat{\rho}''' \sqsubseteq \widehat{\rho}''$, $\widehat{t}''' \sqsubseteq \widehat{t}''$, and $\widehat{\sigma}'' \sqsubseteq \widehat{\sigma}'''$.

Here the source expression e is arbitrary; the λ that is transferred is no longer the source itself but *whatever closure the source evaluates to*. Instantiating $e := \lambda x.e$ recovers the lemma.

Proof (Lemma 4.3 from Theorem A.1). Lemma 4.3 is proved by instantiating Theorem A.1 at the source expression $e := \lambda x.e$. Evaluating a λ is immediate (the Lam-Eval rule): a λ at any cursor is already a value, so

$$(E[\lambda x.e], \widehat{\rho}, \widehat{t}), \widehat{\sigma}\ \widehat{\Downarrow}_d\ (E[\lambda x.e], \widehat{\rho}, \widehat{t}), \widehat{\sigma},$$

i.e. the evaluated closure $E''[\lambda x.e'']$ is exactly the source $E[\lambda x.e]$, with $\widehat{\rho}'' = \widehat{\rho}$, $\widehat{t}'' = \widehat{t}$, and the store left unchanged ($\widehat{\sigma}'' = \widehat{\sigma}$). Substituting these identifications into the conclusion of Theorem A.1 yields precisely Lemma 4.3, with the refinements $\widehat{\rho}''' \sqsubseteq \widehat{\rho}$, $\widehat{t}''' \sqsubseteq \widehat{t}$, and $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'''$. □

Theorem A.1 covers only the trace-to-eval direction; since it and its converse are mutually dependent through the cross-recursion of the underlying relations, the second generalization proves both at once, with Theorem A.1 falling out as Part 1.

A.3 Second Generalization: Both Directions

Theorem A.2 (Bidirectional transfer). *Write $\triangle$ for the demanded source cursor, $\circ$ for the traced-to call site $E_\circ[(e_0\ e_1)]$, and $\square$ for a resulting λ-closure $(E_\square[\lambda x.e_\square], \widehat{\rho}_\square, \widehat{t}_\square)$.*

Part 1 (trace $\Rightarrow$ eval). *Suppose the source both traces to the call site and evaluates to a value containing the $\square$-closure:*

$$(E_\triangle[e_\triangle],\, \widehat{\rho}_\triangle,\, \widehat{t}_\triangle),\, \widehat{\sigma}_\triangle \Rrightarrow_d (E_\circ[(e_0\ e_1)],\, \widehat{\rho}_\circ,\, \widehat{t}_\circ),\, \widehat{\sigma}_\circ,$$
$$(E_\triangle[e_\triangle],\, \widehat{\rho}_\triangle,\, \widehat{t}_\triangle),\, \widehat{\sigma}_\triangle\ \widehat{\Downarrow}_d\ \widehat{v},\, \widehat{\sigma}_\square,$$

where $(E_\square[\lambda x.e_\square], \widehat{\rho}_\square, \widehat{t}_\square) \in \widehat{v}$. Then

$$(E_\circ[([e_0]\ e_1)],\, \widehat{\rho}_\circ,\, \widehat{t}_\circ),\, \widehat{\sigma}_\circ\ \widehat{\Downarrow}_d\ \widehat{v}',\, \widehat{\sigma}'_\square,$$

where $(E_\square[\lambda x.e_\square], \widehat{\rho}'_\square, \widehat{t}'_\square) \in \widehat{v}'$, with $\widehat{\rho}'_\square \sqsubseteq \widehat{\rho}_\square$, $\widehat{t}'_\square \sqsubseteq \widehat{t}_\square$, and $\widehat{\sigma}_\square \sqsubseteq \widehat{\sigma}'_\square$.

Part 2 (eval $\Rightarrow$ trace). *Suppose the source both evaluates to a value containing the $\square$-closure and traces to the call site:*

$$(E_\triangle[e_\triangle],\, \widehat{\rho}_\triangle,\, \widehat{t}_\triangle),\, \widehat{\sigma}_\triangle\ \widehat{\Downarrow}_d\ \widehat{v},\, \widehat{\sigma}_\square,$$
$$(E_\triangle[e_\triangle],\, \widehat{\rho}_\triangle,\, \widehat{t}_\triangle),\, \widehat{\sigma}_\triangle \Rrightarrow_d (E_\circ[(e_0\ e_1)],\, \widehat{\rho}_\circ,\, \widehat{t}_\circ),\, \widehat{\sigma}_\circ,$$

where $(E_\square[\lambda x.e_\square], \widehat{\rho}_\square, \widehat{t}_\square) \in \widehat{v}$. *Then*

$$(E_\square[\lambda x.e_\square],\ \widehat{\rho}_\square,\ \widehat{t}_\square),\ \widehat{\sigma}_\square \mathrel{\widehat{\Rightarrow}_d} (E_\circ[(e_0\ e_1)],\ \widehat{\rho}'_\circ,\ \widehat{t}'_\circ),\ \widehat{\sigma}'_\circ,$$

with $\widehat{\rho}'_\circ \sqsubseteq \widehat{\rho}_\circ, \widehat{t}'_\circ \sqsubseteq \widehat{t}_\circ$, *and* $\widehat{\sigma}_\circ \sqsubseteq \widehat{\sigma}'_\circ$.

Figure 24 depicts the two directions as a cycle between the λ-config and the call site, with *trace* running one way and *eval* the other; the marked point on each arc records that the destination is independent of where along the chain one starts.

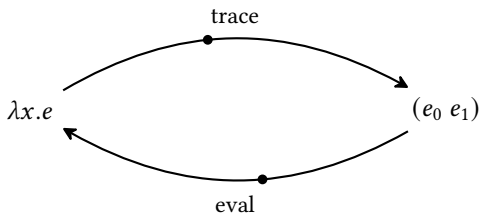


Fig. 24. The bidirectional relation between $\widehat{\Rightarrow}_d$ and $\widehat{\Downarrow}_d$. *Trace* carries the λ-config forward to the call site (Part 2); *eval* demands the function of that call site and returns the same λ (Part 1).

Part 1 and Theorem A.1 differ only presentationally: Theorem A.2 writes the evaluation result as a value set $\widehat{v}$ containing the transferred closure as a *member* ($(E_\square[\lambda x.e_\square], \cdot, \cdot) \in \widehat{v}$), the literal reading since the abstract evaluation relation returns a set of closures. Theorem A.1 is thus exactly Part 1, read through $\widehat{v} \ni (E''[\lambda x.e''], \widehat{\rho}'', \widehat{t}'')$ with $\circ = E'[(e_0\ e_1)]$ and $\triangle = E[e]$; no further argument is needed.

A.4 Proof of the Bidirectional Theorem

The two parts are mutually dependent and are established together, by a single simultaneous mutual induction over the four DoDCFA relations. The rest of this section fixes the refinement order, gives the two induction motives, walks through each constructor case at the surface level, and states the supporting monotonicity, descent, and well-formedness lemmas through which the cases are discharged.

A.4.1 Setting. We work with the four mutually inductive DoDCFA relations $\widehat{\Downarrow}_d$, $\widehat{\Rightarrow}_d$, $\widehat{\Uparrow}^b_d$, $\widehat{\Uparrow}^f_d$ of the main text — keyed by the k-CFA depth m, on configurations $\widehat{cfg} = \langle \widehat{csr}, \widehat{\sigma} \rangle$ (cursor $\widehat{csr}$, store $\widehat{\sigma}$) — whose definitions we do not repeat. The only structural fact we rely on is that the store grows monotonically (every write joins into a cell rather than overwriting); the refinement order below records this.

Definition 1 (Store refinement). $\widehat{\sigma}_0 \sqsubseteq \widehat{\sigma}_1$ iff $\mathrm{dom}(\widehat{\sigma}_0) \sqsubseteq \mathrm{dom}(\widehat{\sigma}_1)$ and $\forall \widehat{\alpha} \in \mathrm{dom}(\widehat{\sigma}_1),\ \widehat{\sigma}_0(\widehat{\alpha}) \sqsubseteq \widehat{\sigma}_1(\widehat{\alpha})$. The value order $\sqsubseteq$ on D is subset $\subseteq$ (the powerset domain order, matching the grow-only discipline), and the domain order is inclusion. We write $\sqsubseteq$ for the environment and time refinement orders (*EnvLeq* and *TimeLeq*, the same symbol as the store), and $\preceq$ for the remaining refinement orders (*CloLeq*, …).

Remark 1 (Scope). The four relations $\widehat{\Downarrow}_d$, $\widehat{\Rightarrow}_d$, $\widehat{\Uparrow}^b_d$, $\widehat{\Uparrow}^f_d$ are the analysis's *derivations*; the induction we run is the ordinary structural induction over a *given* derivation. Termination of the analysis itself (reaching a fixpoint) is guaranteed externally by the finite state space together with a caching algorithm, and is orthogonal to this argument.

A.4.2 Why Mutual Induction. Each constructor may cross-recurse into the other relations. For instance, App-Eval and Ref-Unknown recurse through $\widehat{\Rightarrow}_d$ and $\widehat{\Uparrow}^b_d$; Arg-Trace recurses through $\widehat{\Downarrow}_d$ and $\widehat{\Uparrow}^f_d$; and Bod-Trace through $\widehat{\Downarrow}_d$. A proof of either direction must therefore induct over all four relations *simultaneously*, supplying one motive per relation. We drive the simultaneous recursor, which leaves exactly one subgoal per constructor — sixteen in total. Concretely, both directions are obtained by

```
induction h using Eval.rec
    (motive_2 := fun c₁ c₂ _ => MotiveTrace m c₁ c₂)
    (motive_3 := fun c x c' _ => MotiveBind m c x c')
    (motive_4 := fun c x c' _ => MotiveFind m c x c') with
| lamEval => . . .   | refMemoized => . . .   | appEval => . . .   . . .
```

and symmetrically through `Trace.rec` for the other direction; passing the remaining motives in the `using` clause is what makes the single recursor discharge the whole family in one pass.

A.4.3 Motives. The two non-trivial motives are precisely the two parts of Theorem A.2, attached to an arbitrary sub-derivation:

- MotiveEval $(\widehat{csr}, \widehat{\sigma})$ $(\widehat{v}, \widehat{\sigma}')$ — the Part-2 obligation: if the result $\widehat{v}$ contains a λ-closure and the same source traces to a call site, then the λ-config traces to that call site, with $\widehat{\rho}'_\circ \sqsubseteq \widehat{\rho}_\circ$, $\widehat{t}'_\circ \sqsubseteq \widehat{t}_\circ$, and $\widehat{\sigma}_\circ \sqsubseteq \widehat{\sigma}'_\circ$.
- MotiveTrace $(\widehat{cfg}_1)$ $(\widehat{cfg}_2)$ — the Part-1 obligation: if $\widehat{cfg}_2$'s cursor is a call site and the source evaluates to a λ, then the function-demanded call site evaluates to the same λ, with $\widehat{\rho}'_\square \sqsubseteq \widehat{\rho}_\square$, $\widehat{t}'_\square \sqsubseteq \widehat{t}_\square$, and $\widehat{\sigma}_\square \sqsubseteq \widehat{\sigma}'_\square$.
- MotiveBind = MotiveFind = True. The theorem says nothing about $\widehat{\Uparrow}^b_d/\widehat{\Uparrow}^f_d$ on their own, but their eight constructors still appear as (trivially closed) subgoals because the family is mutual.

A.4.4 The Two Entry Points, Case by Case. Both entry points discharge the same sixteen constructors with identical case text; they differ only in which relation is the induction subject (and hence which motive is the *conclusion* versus a hypothesis). The eight $\widehat{\Uparrow}^b_d/\widehat{\Uparrow}^f_d$ cases are closed by `trivial` (True motive). We describe the eight $\widehat{\Downarrow}_d/\widehat{\Rightarrow}_d$ cases; each reduces to a clean application of the induction hypotheses and a named lemma.

Base Cases.

Lam-Eval. The result is already the singleton λ-closure, so the supplied trace *is* the witness; all refinements are reflexive.

Fun-Trace. The call site $E_\circ[(e_0\ e_1)]$ traces from exactly $E_\circ[([e_0]\ e_1)]$, the function-demanded config that is the very source for which we hold an $\widehat{\Downarrow}_d$ hypothesis; that hypothesis is the witness.

Reference Cases.

Ref-Memoized. x is bound to an address $\widehat{\alpha}$ already holding the λ-closure. Lemma A.8 recovers the closure's own trace to the call site.

Ref-Unmemoized. x is bound to a thunk. The reference transfers the call-site trace to the thunk (Lemma A.7); the inductive hypothesis on the forced thunk yields a λ-config trace at the thunk's result store, which is then lifted across the memoising write extVal by store monotonicity (Lemma A.3).

Ref-Unknown. The address-indexed read at x is empty, so the rule walks back to the binder ($\widehat{\Uparrow}^b_d$), traces it outward to a call site ($\widehat{\Rightarrow}_d$), and forces the argument there. Lemma A.9 transfers

the call-site trace onto that argument config; the argument's IH then applies, again lifted across the memoising extVal. (This is the mirror of Ref-Unmemoized.)

Application / Body Cases.

App-Eval. The crux. The operator e_0 evaluates to the chosen closure, and Fun-Trace traces the function position back to the application; so the *operator IH* gives that the closure — as a λ-config — traces to the application, at a refined env/time/store. The no-shadow fact (Lemma A.6) supplies $\widehat{\rho}'(x) = \text{none}$. The descent lemma (Lemma A.5) then pops out of the callee body to that application via Bod-Trace, and the *body IH* traces the λ-config to the final call site. The three refinement chains compose by transitivity of $\sqsubseteq$.

Arg-Trace. From the argument position, the rule evaluates the function, picks the applied closure, and finds x's use site $\widehat{cfg}_x$ inside the callee body ($\widehat{\Uparrow}^f_d$). That use site evaluates to the same value the argument does (Lemma A.10), so the onward-trace IH (for $\widehat{cfg}_x \widehat{\Rightarrow}_d \widehat{cfg}$) delivers the function-demanded evaluation.

Bod-Trace. Dual of App-Eval: the demanded body evaluates to the same value the enclosing application does (Lemma A.11); the onward-trace IH then delivers the goal.

The two entry points assemble directly into the two parts of Theorem A.2: the $\widehat{\Rightarrow}_d$-induction supplies Part 1, and the $\widehat{\Downarrow}_d$-induction supplies Part 2, each with its env, time, and store refinements. In particular Part 1 is Theorem A.1, and its $e := \lambda x.e$ instance is Lemma 4.3.

A.4.5 Supporting Lemmas. The case analysis is parametric in a few lemmas. The two monotonicity lemmas are each proved by a second simultaneous induction over the four relations; the descent lemma packages the application crux; the remaining lemmas are consequences of program and store well-formedness (no shadowing, memoisation consistency).

Monotonicity.

Lemma A.3 (Store monotonicity). *If $\widehat{\sigma} \sqsubseteq \widehat{\sigma}'$ and $(\widehat{csr}, \widehat{\sigma}) \widehat{\Rightarrow}_d (\widehat{csr}_\circ, \widehat{\sigma}_\circ)$, then*

$$(\widehat{csr}, \widehat{\sigma}') \;\widehat{\Rightarrow}_d\; (\widehat{csr}_\circ, \widehat{\sigma}'_\circ) \qquad \text{for some } \widehat{\sigma}'_\circ \text{ with } \widehat{\sigma}_\circ \sqsubseteq \widehat{\sigma}'_\circ.$$

(Dually for $\widehat{\Downarrow}_d$: a larger source store yields a superset result value and a larger result store.)

Structure. A combined induction with value- and store-growth motives; each constructor reconstructs under the larger store, using that $\widehat{\text{ext}}$ is monotone and that storeVal/storeThunk grow with the store. □

Lemma A.4 (Source-refinement monotonicity). *If $\widehat{\rho}' \sqsubseteq \widehat{\rho}, \widehat{t}' \sqsubseteq \widehat{t}, \widehat{\sigma} \sqsubseteq \widehat{\sigma}'$, and*

$$(E[e],\, \widehat{\rho},\, \widehat{t}),\, \widehat{\sigma} \;\widehat{\Rightarrow}_d\; (E_\circ[(f_0\ f_1)],\, \widehat{\rho}_\circ,\, \widehat{t}_\circ),\, \widehat{\sigma}_\circ,$$

then

$$(E[e],\, \widehat{\rho}',\, \widehat{t}'),\, \widehat{\sigma}' \;\widehat{\Rightarrow}_d\; (E_\circ[(f_0\ f_1)],\, \widehat{\rho}'_\circ,\, \widehat{t}'_\circ),\, \widehat{\sigma}'_\circ$$

for some $\widehat{\rho}'_\circ \sqsubseteq \widehat{\rho}_\circ, \widehat{t}'_\circ \sqsubseteq \widehat{t}_\circ$, and $\widehat{\sigma}_\circ \sqsubseteq \widehat{\sigma}'_\circ$.

Structure. The refinement-valued counterpart of Lemma A.3, again by combined induction: the rule's writes relocate along the refined time, and the read-through store order tracks them via the address-indexed storeRead together with closure/time refinement carried in the motive. □

Descent.

Lemma A.5 (Callee descent, residual form). *Suppose the enclosing λ-config (at the operator's result store $\widehat{\sigma}_1$) already traces to the application, at a refined env/time/store ($\widehat{\rho}_a \sqsubseteq \widehat{\rho}, \widehat{t}_a \sqsubseteq \widehat{t}, \widehat{\sigma} \sqsubseteq \widehat{\sigma}_a$), the application traces onward to a call site, and the closure's parameter is unbound in its captured env ($\widehat{\rho}'(x) = \text{none}$). Then evaluating into the callee body — with x bound to the argument thunk at $\widehat{\alpha} = \langle x, \widehat{t}_2 \rangle$ — traces out to a refined call site.*

Structure. This residual carries no induction hypothesis (the operator IH is used *inline* by App-Eval, since establishing "the chosen closure traces to the application" is exactly the operator sub-derivation's own eval⇒trace fact; a standalone lemma would have to re-invoke the induction and be circular). What remains is bookkeeping: lift the closure's trace across the thunk-extending write (extThunk) by Lemma A.3; refine the application's onward trace to the callee's env/time/store by Lemma A.4; pop out of the body by Bod-Trace, discharging its $\widehat{cc}_1 \sqsubseteq \widehat{cc}$ side-condition by monotonicity of $\widehat{\text{succ}}$; and reconcile the body environment via update/remove identities using $\widehat{\rho}'(x) = \text{none}$. □

Well-Formedness Lemmas. The following package the semantic content used by the reference and application/body cases; each follows from program/store well-formedness (no shadowing, memoisation consistency).

Lemma A.6 (Closure parameter unbound). *A closure $(E'[\lambda x.e], \widehat{\rho}', \widehat{t}')$ produced by evaluating an operator never has its own parameter already bound in its captured env: $\widehat{\rho}'(x) = \text{none}$ (no shadowing).*

Lemma A.7 (Reference demands its thunk). *If x is bound to an unmemoised address holding thunk $\widehat{th}$, any call-site trace from the reference $E[x]$ transfers to a trace from $(\widehat{th}, \widehat{\sigma})$.*

Lemma A.8 (Memoised closure traces to the call site). *A λ-closure already memoised at x's address traces (as a λ-config) to wherever the reference $E[x]$ traces, with the env/time/store refinements; the store records, for each memoised value, a demand path consistent with the current one.*

Lemma A.9 (Unknown-path argument traces to the call site). *When the read at x's address is empty, the binder is found and traced to its call site, and the argument is demanded there; that argument config traces to wherever the reference traces. (Mirror of Lemma A.7.)*

Lemma A.10 (Use site evaluates to the argument). *The use site $\widehat{cfg}_x$ located by $\widehat{\Uparrow}_d^f$ inside the callee body — where x is bound to the argument thunk — evaluates to the same value the argument does, by memoisation.*

Lemma A.11 (Body evaluates to the application's value). *When an enclosing λ traces to an application $E'[(e_0\ e_1)]$, evaluating the body produces the same value evaluating that application produces (dual of* App-Eval*).*

References

[1] Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. 2014. Practical and effective higher-order optimizations. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 81–93. doi:10.1145/2628136.2628153

[2] J. A. Carr, Benjamin Quiring, John Reppy, Olin Shivers, Skye Soss, and Byron Zhong. 2025. Environment-Sharing Analysis and Caller-Provided Environments for Higher-Order Languages. *Artifact for "Environment-Sharing Analysis and Caller-Provided Environments for Higher-Order Languages"* 9, ICFP (Aug. 2025), 247:341–247:370. doi:10.1145/3747516

[3] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California) *(POPL '77)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/512950.512973

[4] David Darais, Nicholas Labich, Phúc C. Nguyen, and David Van Horn. 2017. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.* 1, ICFP (Aug. 2017), 12:1–12:25. doi:10.1145/3110256

[5] Leandro Facchinetti, Zachary Palmer, and Scott Smith. 2019. Higher-order Demand-driven Program Analysis. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 14 (July 2019), 53 pages. doi:10.1145/3310340

[6] Kimball Germane. 2025. Full Control-Flow Sensitivity for Definitional Interpreters. In *Static Analysis: 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, Proceedings*. Springer-Verlag, Berlin, Heidelberg, 120–146. doi:10.1007/978-3-031-74776-2_5

[7] Kimball Germane and Jay McCarthy. 2021. Newly-single and loving it: improving higher-order must-alias analysis with heap fragments. *Proc. ACM Program. Lang.* 5, ICFP (Aug. 2021), 96:1–96:28. doi:10.1145/3473601

[8] Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. 2019. Demand Control-Flow Analysis. In *Verification, Model Checking, and Abstract Interpretation*, Constantin Enea and Ruzica Piskac (Eds.). Springer International Publishing, Cham, 226–246. doi:10.1007/978-3-030-11245-5_11

[9] Kimball Germane and Matthew Might. 2017. A posteriori environment analysis with Pushdown Delta CFA. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*. Association for Computing Machinery, New York, NY, USA, 19–31. doi:10.1145/3009837.3009899

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

[11] David Van Horn and Matthew Might. 2010. Abstracting Abstract Machines. doi:10.48550/arXiv.1007.4446 arXiv:1007.4446 [cs].

[12] Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand interprocedural dataflow analysis. *SIGSOFT Softw. Eng. Notes* 20, 4 (Oct. 1995), 104–115. doi:10.1145/222132.222146

[13] Paul Hudak. 1986. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM conference on LISP and functional programming (LFP '86)*. Association for Computing Machinery, New York, NY, USA, 351–363. doi:10.1145/319838.319876

[14] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. 1998. Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '98)*. Association for Computing Machinery, New York, NY, USA, 329–341. doi:10.1145/268946.268973

[15] Matthew Might and Olin Shivers. 2006. Environment analysis via ΔCFA. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '06)*. Association for Computing Machinery, New York, NY, USA, 127–140. doi:10.1145/1111037.1111049

[16] Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: abstract garbage collection and counting. *SIGPLAN Not.* 41, 9 (Sept. 2006), 13–25. doi:10.1145/1160074.1159807

[17] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. *SIGPLAN Not.* 45, 6 (June 2010), 305–315. doi:10.1145/1809028.1806631

[18] Benoît Montagu and Thomas Jensen. 2021. Trace-based control-flow analysis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 482–496. doi:10.1145/3453483.3454057

[19] 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 *Static Analysis*, Manuel V. Hermenegildo and José F. Morales (Eds.). Vol. 14284. Springer Nature Switzerland, Cham, 460–484. doi:10.1007/978-3-031-44245-2_20 Series Title: Lecture Notes in Computer Science.

[20] O. Shivers. 1988. Control flow analysis in scheme. *SIGPLAN Not.* 23, 7 (June 1988), 164–174. doi:10.1145/960116.54007

[21] Olin Grigsby Shivers. 1991. *Control-flow analysis of higher-order languages of taming lambda.* Ph. D. Dissertation. Carnegie Mellon University, USA. UMI Order No. GAX91-26964.

[22] Scott Smith and Robert Zhang. 2024. A Pure Demand Operational Semantics with Applications to Program Analysis. *Software Artifact for A Pure Demand Operational Semantics with Applications to Program Analysis* 8, OOPSLA1 (April 2024), 135:1154–135:1180. doi:10.1145/3649852

[23] 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) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:26. doi:10.4230/LIPIcs.ECOOP.2016.22

[24] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. 2009. Revised6 Report on the Algorithmic Language Scheme. *Journal of Functional Programming* 19, S1 (2009), 1–301.

doi:10.1017/S0956796809990074

[25] Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: a Context-Free Approach to Control-Flow Analysis. *Logical Methods in Computer Science* Volume 7, Issue 2 (May 2011), 684. doi:10.2168/LMCS-7(2:3)2011 arXiv:1102.3676 [cs].

[26] Tim Whiting and Kimball Germane. 2025. Context-Sensitive Demand-Driven Control-Flow Analysis. In *Programming Languages and Systems*, Viktor Vafeiadis (Ed.). Springer Nature Switzerland, Cham, 374–401. doi:10.1007/978-3-031-91121-7_15

Received 2026-02-19; accepted 2026-05-13