HMCFA: A Precise and Practical Big-Step Control Flow Analysis for Effect Handlers

TIM WHITING, Brigham Young University, USA
KIMBALL GERMANE, Brigham Young University, USA

Effect handlers enable powerful control flow patterns by capturing and resuming continuations, but no control flow analysis exists for programs using them. Applying existing approaches for other delimited control operators would either lose precision through CPS translation or be computationally intractable. We present HMCFA, the first practical control flow analysis for effect handlers based on big-step semantics. Our key technical contributions are: (1) a big-step semantics for effect handlers that allocates denotables and continuations in an explicit store to avoid unbounded syntactic growth, enabling systematic abstraction in the style of Abstracting Definitional Interpreters (ADI); we prove this semantics equivalent to Bauer and Pretnar's substitution-based big-step semantics. (2) A two-component timestamp that tracks both the current call context and the context in which each handler was installed, so that when an operation is dispatched to a handler, its analysis reflects the invocation context that installed it. We prove our concrete semantics equivalent to Bauer and Pretnar's and establish soundness of our abstraction via address freshness. An evaluation on Koka benchmarks shows that our address space has good precision even without context sensitivity, but that our timestamp can close most of the remaining gap in precision while still being tractable on our benchmark suite.

CCS Concepts: • **Theory of computation → Program analysis**; **Operational semantics**; • **Software and its engineering → Control structures**.

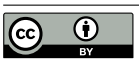
Additional Key Words and Phrases: Effect Handlers, Static Analysis, Control-Flow Analysis, Delimited Control, Abstract Interpretation, Big-Step Semantics

ACM Reference Format:
Tim Whiting and Kimball Germane. 2026. HMCFA: A Precise and Practical Big-Step Control Flow Analysis for Effect Handlers. *Proc. ACM Program. Lang.* 10, ICFP, Article 305 (August 2026), 29 pages. https://doi.org/10.1145/3828703

1 Introduction

Control flow analysis (CFA) is a static analysis that determines which functions can be called at each call site and, more generally, which values flow to each program point. The standard strategy, pioneered by Abstracting Abstract Machines [Van Horn and Might 2010] and refined by Abstracting Definitional Interpreters [Darais et al. 2017], is to take a concrete operational semantics, replace its unbounded components (addresses, environments, continuations) with finite abstractions, and run the resulting interpreter to a fixed point — yielding a sound over-approximation of all reachable program behaviors. For languages with effect handlers, no practical CFA currently exists. Beyond obtaining an abstractable semantics, the central challenge is choosing the right context sensitivity (how much calling context to track) and address space (how values are named and merged), which govern both the precision and cost of the analysis [Gilray et al. 2016]. Effect handlers, with their labeled delimiters and dynamic operation dispatch, pose novel demands on both fronts.

Authors' Contact Information: Tim Whiting, Brigham Young University, Computer Science, Provo, USA, whitim@byu.edu; Kimball Germane, Brigham Young University, Computer Science, Provo, USA, kimball@cs.byu.edu.



ACM 2475-1421/2026/8-ART305
https://doi.org/10.1145/3828703

A control flow analysis would enable static reasoning currently unavailable for effect handler programs: which handler handles a given operation call, which continuations are reachable, and whether distinct operation invocations may alias — information that could guide compiler optimizations and program understanding.

Existing control flow analyses for first-class control operators (call/cc, shift/reset) do not straightforwardly apply. CPS-based analyses [Vardoulakis and Shivers 2011] can analyze delimited control indirectly, but the CPS translation discards delimiter structure and expresses results in CPS terms rather than source constructs, limiting their use for optimization. Increasing k in a CPS-based k-CFA improves call-string precision in the usual way. However, at any k the delimiter and operator structure has been mostly erased by the translation, so k does not recover sensitivity directly related to the first-class control flow of the original program. The one direct approach to analysis of first-class control, Abstracting Abstract Control (AAC) [Glaze and Van Horn 2014], analyzes shift/reset via small-step reduction semantics but requires super-exponential time even without context-sensitivity and does not address effect handlers, which add operation dispatch and labeled delimiters.

We take a different approach based on big-step semantics. Abstracting Definitional Interpreters (ADI) [Darais et al. 2017] has shown how to systematically derive sound analyses from interpreters: write a concrete interpreter, then abstract its state space, memoizing calls to iterate to a fixpoint. However, the standard big-step semantics for effect handlers [Bauer and Pretnar 2014] uses syntactic substitution, leading to unbounded continuation growth incompatible with abstraction. We develop a big-step semantics that allocates both denotables and continuation segments in an explicit store, prove it equivalent to Bauer and Pretnar's semantics, and then derive a finite abstraction from it. Even with an abstractable semantics, continuation restoration poses a challenge for context sensitivity: standard mechanisms gradually lose the ability to distinguish different resumptions of the same continuation as the restored computation proceeds.

1.1 Contributions and Organization

Our paper makes the following contributions:

- **A store-based big-step semantics for effect handlers** (Section 2) that uses environments and stores rather than syntactic substitution, making it amenable to abstraction. While big-step semantics with environments for effect handlers are not by themselves new [Baelde and Schmitt 2025], our explicit store-allocated representation of denotables and continuations is, and we prove this semantics equivalent to Bauer and Pretnar's (B&P) [Bauer and Pretnar 2014] substitution-based semantics (Section 2.1).
- **A two-component timestamp** (Section 3) that tracks both the call stack and delimited control meta-stacks. We prove that this timestamp ensures fresh allocation, establishing simulation of the timestamped semantics with respect to the concrete semantics (3.2).
- **A finite abstract semantics** (Section 4) derived by truncating timestamps and introducing non-determinism. We prove this abstraction sound with respect to the timestamped concrete semantics (Section 4.1).
- **An empirical evaluation** (Section 5) on a suite of effect handler benchmarks integrated into the Koka compiler, demonstrating good precision and completing all benchmarks within the time budget.

```
effect ctl sample(x : int) : int
// Sum two samples
fun f(n)
  sample(n) + sample(n + 1)
```

(a) Shared definitions

```
// Adjust sample by +-10, branching
// Sum the results of both branches
fun main()
  with handler
    ctl sample(x)
      resume(x + 10) + resume(x - 10)
  f(0)
```

(b) Multi-shot, `f(0) = 4`

```
fun main()
  with handler
    ctl sample(x)
      x
  f(0)
```

(c) Zero-shot, `f(0) = 0`

```
fun main()
  with handler
    ctl sample(x)
      resume(x + 10)
  f(0)
```

(d) Single-shot, `f(0) = 21`

Figure. 1. Three different handlers for the same effect and function (top left). The code after `with handler` is executed underneath the handler. The `ctl` clause handles the effectful operation `sample` and binds the special identifier `resume` to the suspended computation.

1.2 Intuition and Running Example

Figure 1 shows three handlers for the same effect and function. Effect handlers are a user-defined delimited control mechanism: invoking `sample(n)` suspends the surrounding computation and transfers control to the matching `ctl` clause of the handler, which binds `resume` to that suspended continuation. How the clause uses `resume` determines the semantics. The zero-shot handler returns `x` immediately without calling `resume`, exiting the handler scope with a value — covering patterns such as exceptional return and early termination (in this case returning `0`). The single-shot handler calls `resume(x+10)` once, delivering $x + 10$ as the result of `sample(n)`. Because the handler is *deep* — reinstalled around every resumption — the second `sample(n+1)` is also intercepted, giving $f(0) = 10 + 11 = 21$. This single-resumption pattern covers a broad class of idioms: environment configuration, scoped state, dependency injection, cooperative concurrency, and async/await. The multi-shot handler calls `resume` twice per invocation, resuming the rest of the computation with different values. This pattern enables backtracking, checkpointing, and probabilistic sampling; and in this case the four resulting paths sum to 4.

Figure 2a traces execution of the example in Figure 2 (the same multi-shot example but where each intermediate value is bound). The `return` clause allows the programmer to transform the result on normal return to match the type of the final result from the operation clauses and is unimportant for this example. When `f(0)` invokes `sample(0)`, control transfers to the handler, which captures κ_1: the rest of `f`'s body awaiting s_1 (`val` s_1=□; ...). Calling the resumption with `hi`= 10 replaces □ in κ_1 with `10`, which then invokes `sample(`n_1`)` capturing κ_2 (`val` s_2=□;s_1+s_2 with s_1= 10). The handler resumes κ_2 first with `hi`= 11 (giving 21), then with `lo`= −9 (giving 1); r_1= 21 and r_2= 1 sum to 22.

The handler then resumes κ_1 a second time with `lo`= −10, resuming the same continuation with s_1= −10. This captures κ_3: syntactically identical to κ_2 but carrying s_1= −10. Resuming κ_3 twice yields 1 and −19, summed to −18. The original handler sums 22 and −18 to produce 4.

Throughout the paper we will use the following terminology: operations are *invoked*; the handler *captures* the delimited computation as a continuation; a continuation is *resumed*; and its individual stack frames are *restored* one at a time.

An analysis that naively handles multiple resumptions of the same continuation conflates paths under the handler which are identical modulo the resumption argument. Additionally, since

Point	Notes
`f(0)`	n=0
`  sample(n)` $\rightarrow \kappa_1$	n=0
`  |`	κ_1 = `val s1=□;...`
`  resume(hi)`	x=0, hi=10
`  | sample(n1)` $\rightarrow \kappa_2$	$s1$=10, $n1$=1
`  |  |`	κ_2 = `val s2=□;...`
`  |  resume(hi)`	x=1, hi=11
`  |  | s1 + s2` $\rightarrow 21$	$s1$=10, $s2$=11
`  |  resume(lo)`	$r1$=21, x=1, lo=−9
`  |  | s1 + s2` $\rightarrow 1$	$s1$=10, $s2$=−9
`  | r1 + r2` $\rightarrow 22$	$r1$=21, $r2$=1
`  resume(lo)`	x=0, lo=−10
`  | sample(n1)` $\rightarrow \kappa_3$	$s1$=−10, $n1$=1
`  |  |`	κ_3 = `val s2=□;...`
`  |  resume(hi)`	x=1, hi=11
`  |  | s1 + s2` $\rightarrow 1$	$s1$=−10, $s2$=11
`  |  resume(lo)`	$r1$=1, x=1, lo=−9
`  |  | s1 + s2` $\rightarrow -19$	$s1$=−10, $s2$=−9
`  | r1 + r2` $\rightarrow -18$	$r1$=1, $r2$=−19
`r1 + r2` $\rightarrow 4$	$r1$=22, $r2$=−18

(a) Execution Trace

```
fun f(n)
  val s1 = sample(n)
  val n1 = n + 1
  val s2 = sample(n1)
  s1 + s2
//
fun main()
  with handler
    return(z) z
    ctl sample(x)
      val hi = x + 10
      val r1 = resume(hi)
      val lo = x - 10
      val r2 = resume(lo)
      r1 + r2
  f(0)
```

(b) Example using ANF syntax

Figure. 2. Running Example: multi-shot continuation. Intermediate variable names: `n1` is the bumped form of `n`; `s1`, `s2` are the results of the two `sample` invocations; `hi` and `lo` are the two altered samples passed to `resume`; `r1`, `r2` are the results of those resumptions. Each `sample` invocation captures a continuation: κ_1 is `val s1 = □; ...`, while κ_2 and κ_3 are `val s2 = □; s1 + s2` in different environments; □ marks the hole for the resumption's value.

operations capture different continuations at different points in evaluation there is potential for conflating entirely different paths. We identify three challenges for analysis precision:

- **Delimiter forgetfulness.** The analysis loses track of the context in which a delimiter (handler or resumption) was introduced, so different resumption paths become indistinguishable as restored computation proceeds. In the example, the handler resumes κ_1 twice with different values. Without delimiter sensitivity, both resumptions execute under the same analysis state: the two bindings of `s1` merge, and the four distinct paths to the return clause collapse.
- **Operator forgetfulness.** The analysis loses track of which operation invocation captured which continuation, so when continuations are resumed, results flow to the wrong return sites. In the example, the two invocations of `sample(n1)` occur under different bindings of `s1`, capturing different continuations κ_2 and κ_3. Without operator sensitivity, the analysis cannot distinguish which resumption result should return to which operation call, conflating κ_2 and κ_3 with each other and potentially with κ_1 which was captured at a different call site!
- **Continuation fragmentation.** Even when the analysis distinguishes which operation captured a continuation, the individual frames may not stay together as a unit in the abstract address space. In the example, κ_2 and κ_3 share the same syntactic structure (`val s2=□;s1+s2`) but carry different environments (different bindings of `s1`). If the address space does not maintain cohesion within each captured continuation, the analysis could construct a spurious continuation combining frames from κ_2 with frames from κ_3.

Our approach addresses these challenges through two mechanisms. First, *timestamps* with two components — a call-stack component and a meta-continuation component — provide *delimiter sensitivity* and *operator sensitivity*: each function call, handler entry, and continuation resumption extends the timestamp, ensuring different execution paths receive different timestamps. The

meta-continuation component specifically records which handler or resumption introduced each delimited scope, addressing delimiter forgetfulness even as the restored computation's own calls evolve the call-stack component. Together, both components address operator forgetfulness by distinguishing operation invocations from different contexts. Second, we augment continuation addresses with information about which operation invocation they originated from, assisting with both *operator sensitivity* and *continuation cohesion*: this common context links individual frames from the same continuation, preventing them from merging with frames from other continuations that happen to share some structure. While both mechanisms remain finite and do not completely eliminate these challenges, they strike a practical balance between precision and tractability.

1.3 Concrete Syntax and Reduction Semantics

1.3.1 Syntax. We consider a call-by-value λ-calculus with effect handlers, denoted λ^h. Figure 3 presents the syntax in surface form and in administrative normal form (ANF) [Flanagan et al. 1993].

$c \in \mathsf{ConLabel}$
$x, y, f, g, \mathit{resume} \in \mathsf{Var}$
$op \in \mathsf{OpName}$

$b \in \mathsf{Branches} ::= \epsilon \mid c(\overline{x}) \to e;\ b \quad c \notin b$
$h \in \mathsf{Clauses} ::= \mathsf{return}(x)\ e \mid op(\overline{x})\ e;\ h \quad op \notin h$

(a) Common syntax

$æ \in \mathsf{AExp} ::= x \mid \mathsf{fn}(\overline{x})\ e$
$e \in \mathsf{Exp} ::= æ \mid \mathsf{fun}\ g(\overline{x})\ e \mid e_f(\overline{e}_{args}) \mid c(\overline{e}_{args}) \mid op(\overline{e}_{args}) \mid \mathsf{match}(e)\{b\} \mid \mathsf{handle}\{h\}(e) \mid \mathsf{val}\ y = e_{\mathsf{bind}};\ e_{let}$

(b) Surface Syntax

$l \in \mathsf{Label}$
$æ \in \mathsf{AExp} ::= x \mid \mathsf{fn}(\overline{x})\ e$
$ce \in \mathsf{CExp} ::= æ \mid \mathsf{fun}\ g(\overline{x})\ e \mid f(\overline{æ}) \mid c(\overline{x}) \mid op(\overline{æ}) \mid \mathsf{match}(x)\{b\} \mid \mathsf{handle}\{h\}(e)$
$e \in \mathsf{Exp} ::= ce \mid \mathsf{val}\ y = ce^l;\ e$

(c) ANF Syntax

Figure. 3. Syntax for λ^h

Beyond the standard lambda calculus constructs, the key additions are:

- **Operations** $op(\overline{e})$ *invoke* an effect, suspending the surrounding computation and propagating outward through enclosing contexts until a matching handler is found.
- **Handlers** $\mathsf{handle}\{h\}(e)$ delimit a computation e and specify how each operation is handled via clauses `{op(x) e; ..., return(x) e}`. When an operation is invoked within the delimiter, the handler *captures* the delimited computation as a continuation, binds the operation's argument and the continuation (initially named `resume`), and runs the clause. The return clause runs when evaluation under the delimiter completes with an ordinary value.
- **Continuation applications** $\mathsf{resume}(e)$ *resume*s a captured continuation: the delimited computation that was suspended by the operation. A continuation may be resumed any number of times: zero (the clause discards it), once (single-shot), or more (multi-shot).

Anonymous functions are written $\mathsf{fn}(\overline{x})\ e$ and recursive named functions $\mathsf{fun}\ g(\overline{x})\ e$ (where g is in scope inside the body for self-reference). We work primarily with the ANF variant (Figure 3c), which splits expressions into atomic expressions $æ$ (variables and anonymous lambdas) and complex expressions ce (applications, operations, matches). Complex expressions may only appear in let bindings or tail position. Each expression carries a label (omitted when unnecessary) for tracking program points. We restrict the scrutinee of match to a variable: the only other atomic expressions are lambdas, which are not scrutinizable.

1.3.2 Notation. As in Figure 2, semicolons can be replaced with newlines and braces with indented blocks. We use the overline convention $\overline{x}$ to denote sequences, written $x_1, \ldots, x_n$ in examples; a metavariable like $\overline{\text{æ}}$ is the sequence form of æ. When an overline contains a single metavariable we leave it plain; when it contains multiple, we **bold** the *subscriptable* ones — those indexed in lock-step across the sequence — leaving non-bold metavariables (typically environments, stores, labels, and times) shared across all positions. For instance, $[\overline{\boldsymbol{x} \mapsto \mathcal{A}(\boldsymbol{\text{æ}}, \rho, \sigma)}]$ expands to $[x_1 \mapsto \mathcal{A}(\text{æ}_1, \rho, \sigma),\ \ldots,\ x_n \mapsto \mathcal{A}(\text{æ}_n, \rho, \sigma)]$.

1.3.3 Reduction Semantics. Before developing our big-step semantics, we briefly present a standard reduction semantics for reference.

$$
\begin{array}{lll}
\mathsf{E} & ::= & [] \mid \mathsf{E}(\overline{e}) \mid v(\ldots \overline{v}, \mathsf{E}, \ldots \overline{e}) \\
 & & \mid \mathsf{val}\ y = \mathsf{E};\ e \mid \mathsf{match}(\mathsf{E})\{b\} \mid \mathsf{handle}\{h\}(\mathsf{E}) \\
X_{op} & ::= & [] \mid X_{op}(\overline{e}) \mid v(\ldots \overline{v}, X_{op}, \ldots \overline{e}) \\
 & & \mid \mathsf{val}\ y = X_{op};\ e \mid \mathsf{match}(X_{op})\{b\} \mid \mathsf{handle}\{h\}(X_{op}) \qquad op \notin h
\end{array}
$$

(a) Surface Syntax

$$
\begin{array}{lll}
\mathsf{E} & ::= & [] \mid \mathsf{val}\ y = \mathsf{E};\ e \mid \mathsf{handle}\{h\}(\mathsf{E}) \\
X_{op} & ::= & [] \mid \mathsf{val}\ y = X_{op};\ e \mid \mathsf{handle}\{h\}(X_{op}) \qquad op \notin h
\end{array}
$$

(b) ANF Syntax

Figure. 4. Reduction Contexts

$$
\begin{array}{llll}
(\beta) & (\mathsf{fn}(\overline{x})\ e)(\overline{v}) & \longrightarrow & e[\overline{x} \mapsto \overline{v}] \\
(\mathit{fun}) & \mathsf{fun}\ g(\overline{x})\ e & \longrightarrow & (\mathsf{fn}(\overline{x})\ e[g \mapsto \mathsf{fn}(\overline{x})\ e]) \\
(\mathit{match}) & \mathsf{match}\{b\}(c(\overline{v})) & \longrightarrow & e[\overline{x} \mapsto \overline{v}] \\
 & & & \text{where} \quad c(\overline{x})\ e \in b \\
(\mathit{let}) & \mathsf{val}\ y = v;\ e & \longrightarrow & e[y \mapsto v] \\
(\mathit{return}) & \mathsf{handle}\{h\}(v) & \longrightarrow & e[x \mapsto v] \\
 & & & \text{where} \quad \mathsf{return}(x)\ e \in h \\
(\mathit{handle}) & \mathsf{handle}\{h\}(X_{op}[op(\overline{v})]) & \longrightarrow & e[\overline{x} \mapsto \overline{v},\ \mathit{resume} \mapsto \mathsf{fn}(y)\ \mathsf{handle}\{h\}(X_{op}[y])] \\
 & & & \text{where} \quad op(\overline{x})\ e \in h
\end{array}
$$

Figure. 5. Reduction Rules

The reduction contexts in Figure 4 make evaluation order explicit. E denotes any evaluation context, whereas X_{op} is a context delimited by the nearest handler for *op* — spanning only handlers that do not handle *op*. ANF simplifies these considerably (compare 4b to 4a).

Beyond the standard β, match, and let rules, the reduction semantics includes two handler-specific rules. The return rule applies the return clause when the handler body reduces to a value. The handle rule fires when an operation *op* is invoked inside a matching handler: it binds the operation's argument to the handler clause's parameter, and `resume` to a lambda that plugs the hole in the captured context X_{op} with a fresh parameter and reinstalls the handler around the context.

2 An Abstractable Big-Step Semantics for Effect Handlers

Our new big-step semantics follows Bauer and Pretnar's (B&P) big-step structure, where continuations are built lazily when an operation is invoked and accumulate as it bubbles up to its handler. However, rather than building a lambda term through syntactic substitution, we represent continuations as explicit lists of frames and restore them via special application rules. This representation enables standard abstraction techniques, as we show in Sections 3 and 4.

$$
\begin{array}{llll}
a_v \in \mathsf{VAddr} & \text{an infinite set} & \sigma \in \mathsf{Store} & ::= \mathsf{VAddr} \to \mathsf{Denotable} \\
\rho \in \mathsf{Env} & ::= \mathsf{Var} \to \mathsf{VAddr} & \psi \in \mathsf{Frame} & ::= \mathsf{LetFrame} \mid \mathsf{Handler} \times \mathsf{Env} \\
clo \in \mathsf{Closure} & ::= \overline{\mathsf{Var}} \times \mathsf{Exp} \times \mathsf{Env} & k \in \mathsf{Kont} & ::= \mathsf{Frame}^* \\
\psi_{let} \in \mathsf{LetFrame} & ::= \mathsf{Var} \times \mathsf{Exp} \times \mathsf{Env} & & \\
d \in \mathsf{Denotable} & ::= \mathsf{Closure} \mid \mathsf{Handler} \times \mathsf{Env} \times \mathsf{Kont} \mid \mathsf{ConLabel} \times \overline{\mathsf{VAddr}} & & \\
v \in \mathsf{Value} & ::= \mathsf{OpName} \times \overline{\mathsf{VAddr}} \times \mathsf{Kont} \mid \mathsf{Denotable} & &
\end{array}
$$

Figure. 6. Semantic Components for Concrete Big-Step Machine

We develop this semantics in stages: first the semantic components, then the core lambda calculus with stores and environments, and finally extensions for effect handlers and continuation capture.

Figure 6 presents the semantic components used in the evaluation rules for our concrete machine. Stores map addresses to denotables, environments map variables to addresses, and closures pair a lambda with its environment. Continuations (Kont) are lists of frames: either let-bound closures or handler frames (containing the handler and its environment). A *denotable* (metavariable d) is a program value — anything that may be stored at an address and bound to a variable. In our system, denotables comprise closures, captured continuations ($\mathsf{Handler} \times \mathsf{Env} \times \mathsf{Kont}$), and constructed values ($\mathsf{ConLabel} \times \overline{\mathsf{VAddr}}$). Return *values* (metavariable v) are a purely semantic artifact, describing what an evaluation derivation produces: either a denotable, or a suspended operation ($\mathsf{OpName} \times \overline{\mathsf{VAddr}} \times \mathsf{Kont}$) bubbling outward toward a handler. Suspended operations never appear in stores or environments.

$$
\begin{array}{ll}
\mathcal{A} & : \mathsf{AExp} \times \mathsf{Env} \times \mathsf{Store} \to \mathsf{Denotable} \\
\mathcal{A}(x, \rho, \sigma_v) & = \sigma_v(\rho(x)) \\
\mathcal{A}(c(\overline{x}), \rho, \sigma_v) & = (c, \overline{\rho(x)}) \\
\mathcal{A}(\mathsf{fn}(\overline{x})\ e_{body}, \rho_\lambda, \sigma_v) & = (\overline{x}, e_{body}, \rho_\lambda)
\end{array}
$$

Atomic expressions are evaluated via the meta-function $\mathcal{A}$: variables are looked up in the environment, constructor applications pair the constructor label with the addresses of its field expressions, and closures capture the lambda's binder, body, and environment.

We split evaluation into $\Downarrow_{eval}$ (regular) and $\Downarrow_{evalc}$ (complex). Tail evaluation [EVAL-TAIL] defers to the complex expression, while atomic evaluation [EVAL-ATOMIC] uses meta-function $\mathcal{A}$. Let expressions [EVAL-LET] evaluate their binding, then delegate continuation processing to relation $\Rightarrow_{cont}$. This separation prepares for effects by isolating the continuation handling. For pure lambda calculus, the only $\Rightarrow_{cont}$ rule is [CONTINUE-LET] which binds the result (always a denotable) to a fresh address and evaluates the body in the extended environment.

Recursive definitions [EVAL-FUN] extend the closure's environment with a self-reference before extending the store. Application [EVAL-APP-CLOS] evaluates arguments, binds them to fresh addresses, then evaluates the body. We treat branch lists as partial maps from constructor labels to clauses, writing $b(c) = (\overline{x},\ e)$ for branch lookup. Match statements [EVAL-MATCH] evaluate the scrutinee, look up the matching branch in b, allocate fresh addresses $\overline{a'_v}$ seeded with the scrutinee's field values $\overline{\sigma(\boldsymbol{a_v})}$, and evaluate the branch body. Reallocating fresh addresses on match is unnecessary in the concrete semantics — we could just as well reuse the scrutinee's addresses — but the timestamped and abstract semantics (Sections 3–4) require that all bindings introduced at a given timestamp share that timestamp, so we adopt a uniform allocation discipline here for direct comparison.

Effects require two additional relations: $\Downarrow_{\mathsf{handle}}$ (dispatches on effectful results) and $\Downarrow_{apply}$ (restores captured continuations).

$\Downarrow_{eval} : (\text{Exp} \times \text{Env} \times \text{Store}) \times (\text{Value} \times \text{Store})$
$\Downarrow_{evalc} : (\text{CExp} \times \text{Env} \times \text{Store}) \times (\text{Value} \times \text{Store})$
$\Rightarrow_{cont} : (\text{LetFrame} \times \text{Value} \times \text{Store}) \times (\text{Value} \times \text{Store})$

EVAL-LET

$$\frac{(ce, \rho, \sigma) \Downarrow_{evalc} (v, \sigma') \quad ((y, e, \rho), v, \sigma') \Rightarrow_{cont} (v', \sigma'')}{(\text{let } y = ce;\ e, \rho, \sigma) \Downarrow_{eval} (v', \sigma'')}$$

EVAL-TAIL

$$\frac{(ce, \rho, \sigma) \Downarrow_{evalc} (v, \sigma')}{(ce, \rho, \sigma) \Downarrow_{eval} (v, \sigma')}$$

EVAL-ATOMIC

$$\frac{}{(æ, \rho, \sigma) \Downarrow_{evalc} (\mathcal{A}(æ, \rho, \sigma), \sigma)}$$

CONTINUE-LET

$$\frac{\sigma_{let} = \sigma[a_v \mapsto d] \quad a_v \text{ is fresh} \quad (e, \rho[y \mapsto a_v], \sigma_{let}) \Downarrow_{eval} (v', \sigma')}{((y, e, \rho), d, \sigma) \Rightarrow_{cont} (v', \sigma')}$$

EVAL-FUN

$$\frac{v_f = (\overline{x}, e_1, \rho[f \mapsto a_v]) \quad a_v \text{ is fresh}}{(\text{fun } f(\overline{x})\ e_1, \rho, \sigma) \Downarrow_{evalc} (v_f, \sigma[a_v \mapsto v_f])}$$

EVAL-APP-CLOS

$$\frac{(\overline{x}, e_{body}, \rho_\lambda) = \mathcal{A}(f, \rho, \sigma) \quad \sigma_\lambda = \sigma[\overline{\boldsymbol{a_v} \mapsto \mathcal{A}(\boldsymbol{æ}, \rho, \sigma)}] \quad \overline{a}_v \text{ are fresh} \quad (e_{body}, \rho_\lambda[\overline{x \mapsto \boldsymbol{a_v}}], \sigma_\lambda) \Downarrow_{eval} (v', \sigma')}{(f(\overline{æ}), \rho, \sigma) \Downarrow_{evalc} (v', \sigma')}$$

EVAL-MATCH

$$\frac{c(\overline{a}_v) = \mathcal{A}(æ, \rho, \sigma) \quad \overline{a}'_v \text{ are fresh} \quad b(c) = (\overline{x}, e) \quad \rho_b = \rho[\overline{x \mapsto \boldsymbol{a'_v}}] \quad \sigma_b = \sigma[\overline{\boldsymbol{a'_v} \mapsto \sigma(\boldsymbol{a_v})}] \quad (e, \rho_b, \sigma_b) \Downarrow_{eval} (v', \sigma')}{(\text{match}(æ)\{b\}, \rho, \sigma) \Downarrow_{evalc} (v', \sigma')}$$

Figure. 7. Big-Step Lambda Rules

Operation application [EVAL-APP-OP] immediately returns a suspended operation $(op, \overline{\rho(x)}, [])$ with an empty continuation. As this value returns through enclosing contexts via [CONTINUE-OP], frames are prepended to the continuation.

Handler evaluation [EVAL-HANDLER] evaluates the body, while continuation application [EVAL-APP-KONT] first restores the continuation via $\Downarrow_{apply}$. Both pass the result to $\Downarrow_{\text{handle}}$, which dispatches on whether it is a denotable or a suspended operation. The metavariable f in [EVAL-APP-KONT] ranges over arbitrary atomic expressions: although `resume` is the initial binding of a captured continuation, the continuation is a first-class denotable that may flow through any binding, so f need not literally be `resume`.

We treat handlers as partial maps from operation names to clauses, writing $h(op) = (\overline{x},\ e_{op})$ for clause lookup. Rule [HANDLE-OP] handles a matching suspended operation: it looks up the clause for op, allocates $a_{v_k} \mapsto (h,\ \rho,\ \kappa)$ as the captured continuation, copies the argument values into fresh addresses $\overline{a'_v}$, and evaluates the clause body with *resume* bound to a_{v_k} and parameters bound to $\overline{a'_v}$. Rule [HANDLE-RETURN] similarly looks up the return clause and applies it. Rule [HANDLE-CAPTURE-OP] handles operations not in h by prepending a handler frame $(h,\ \rho)$ to the continuation, so that [APPLY-RESTORE-HANDLE] reinstalls the handler when the continuation is resumed.

The $\Downarrow_{apply}$ rules restore continuations. Rule [APPLY-RESTORE] recursively applies the rest of the continuation before processing the current let-closure (since continuations are captured innermost-first but restored outermost-first). Importantly, it uses $\Rightarrow_{cont}$ to evaluate the let-closure, allowing later operations to re-capture stack frames. Rule [APPLY-CONTINUE] returns the denotable when all frames are restored. Rule [APPLY-RESTORE-HANDLE] applies the continuation tail first, then passes the result to $\Downarrow_{\text{handle}}$, so the restored computation sees this handler frame as its enclosing delimiter.

Two invariants govern these rules: (1) continuations are captured innermost-to-outermost during unwinding, and (2) they are restored by recursively applying the tail before the head. We prepend frames rather than append (as in small-step semantics) so that when implemented as a definitional

$\Downarrow_{\text{handle}} : (\text{Handler} \times \text{Env} \times \text{Value} \times \text{Store}) \times (\text{Value} \times \text{Store})$
$\Downarrow_{apply} : (\text{Kont} \times \text{Denotable} \times \text{Store}) \times (\text{Value} \times \text{Store})$

EVAL-APP-OP

$$\frac{v = (op, \overline{\rho(x)}, [])}{(op(\overline{x}), \rho, \sigma) \Downarrow_{evalc} (v, \sigma)}$$

CONTINUE-OP

$$\frac{v = (op, \overline{a}_v, clo : \kappa)}{(clo, (op, \overline{a}_v, \kappa), \sigma) \Rightarrow_{cont} (v, \sigma)}$$

EVAL-HANDLER

$$\frac{(e_{body}, \rho, \sigma) \Downarrow_{eval} (v', \sigma') \quad (h, \rho, v', \sigma') \Downarrow_{\text{handle}} (v'', \sigma'')}{(h(e_{body}), \rho, \sigma) \Downarrow_{evalc} (v'', \sigma'')}$$

EVAL-APP-KONT

$$\frac{(\kappa, \mathcal{A}(æ, \rho, \sigma), \sigma) \Downarrow_{apply} (v', \sigma') \quad (h, \rho_h, \kappa) = \mathcal{A}(f, \rho, \sigma) \quad (h, \rho_h, v', \sigma') \Downarrow_{\text{handle}} (v'', \sigma'')}{(f(æ), \rho, \sigma) \Downarrow_{evalc} (v'', \sigma'')}$$

HANDLE-OP

$$\frac{h(op) = (\overline{x}, e_{op}) \quad \sigma_{op} = \sigma[a_{vk} \mapsto (h, \rho, \kappa), \overline{\boldsymbol{a'_v} \mapsto \sigma(\boldsymbol{a_v})}] \quad a_{vk}, \overline{a}'_v \text{ are fresh} \quad (e_{op}, \rho[resume \mapsto a_{vk}, \overline{\boldsymbol{x} \mapsto \boldsymbol{a'_v}}], \sigma_{op}) \Downarrow_{eval} (v, \sigma')}{(h, \rho, (op, \overline{a}_v, \kappa), \sigma) \Downarrow_{\text{handle}} (v, \sigma')}$$

APPLY-RESTORE

$$\frac{(\kappa, d, \sigma) \Downarrow_{apply} (v', \sigma') \quad (clo, v', \sigma') \Rightarrow_{cont} (v'', \sigma'')}{(clo : \kappa, d, \sigma) \Downarrow_{apply} (v'', \sigma'')}$$

HANDLE-RETURN

$$\frac{h(\text{return}) = (x_{ret}, e_{ret}) \quad \sigma_{ret} = \sigma[a_v \mapsto d] \quad a_v \text{ is fresh} \quad (e_{ret}, \rho[x_{ret} \mapsto a_v], \sigma_{ret}) \Downarrow_{eval} (v', \sigma')}{(h, \rho, d, \sigma) \Downarrow_{\text{handle}} (v', \sigma')}$$

APPLY-CONTINUE

$$\frac{}{([], d, \sigma) \Downarrow_{apply} (d, \sigma)}$$

HANDLE-CAPTURE-OP

$$\frac{op \notin h \quad v = (op, \overline{a}_v, (h, \rho) : \kappa)}{(h, \rho, (op, \overline{a}_v, \kappa), \sigma) \Downarrow_{\text{handle}} (v, \sigma)}$$

APPLY-RESTORE-HANDLE

$$\frac{(\kappa, d, \sigma) \Downarrow_{apply} (v', \sigma') \quad (h, \rho, v', \sigma') \Downarrow_{\text{handle}} (v'', \sigma'')}{((h, \rho) : \kappa, d, \sigma) \Downarrow_{apply} (v'', \sigma'')}$$

Figure. 8. Big-Step Handler Rules

interpreter, restoration reconstructs the original evaluation stack onto the host's stack. The resulting rules are deterministic and syntax-directed, directly implementable as a definitional interpreter.

2.1 Concrete Big-Step Equivalence

Although unimportant for the rest of the paper, our semantics is equivalent to the existing big-step substitution semantics of Bauer and Pretnar [2014]. To show this we overload the notation $\equiv$ for two equivalences: configuration equivalence $c \equiv (e, \rho, \sigma)$ relates a B&P computation c (with all variables substituted) to our configuration (e, ρ, σ) by applying the environment and store to reconstruct the substituted term; result equivalence $r \equiv (v, \sigma)$ similarly relates B&P's syntactic results to our store-based values. Their detailed inductive definitions are delegated to the Lean 4 formalization. We restrict both languages to single-parameter functions and omit first-class handler labels; extending to these is straightforward.

Theorem 2.1. (*Simulation Soundness*)
If $c \equiv (e, [], [])$, $c.fvs = []$, and $(e, \rho, \sigma) \Downarrow_{eval} (v, \sigma')$, then $\exists r,\ c \Downarrow r$ and $r \equiv (v, \sigma')$.

$$
\begin{array}{llll}
a_v \in \text{VAddr} & ::= \text{Var} \times \text{Time} & \rho \in \text{Env} & ::= \text{Var} \to \text{VAddr} \\
clo \in \text{Closure} & ::= \overline{\text{Var}} \times \text{Exp} \times \text{Env} & o \in \text{OpInfo} & ::= \text{OpName} \times \overline{\text{VAddr}} \times \text{TFrame} \\
\tau_\kappa \in \text{KTime} & ::= \text{Label}* & \tau \in \text{Time} & ::= \text{KTime} \times \text{MKTime} \\
\tau_{m\kappa} \in \text{MKTime} & ::= (\text{Label} \times \text{KTime})* & \psi_t \in \text{TFrame} & ::= \text{Frame} \times \text{Time} \\
\psi \in \text{Frame} & ::= \text{LetFrame} \mid \text{Label} \times \text{Env} \mid \text{Handler} \times \text{Env} & & \\
\psi_{let} \in \text{LetFrame} & ::= \text{Var} \times \text{Exp} \times \text{Env} & & \\
a_\kappa \in \text{KAddr} & ::= \text{TFrame} \times \text{OpInfo} \mid a_{\kappa end} & & \\
d \in \text{Denotable} & ::= \text{Closure} \mid \text{Handler} \times \text{Env} \times \text{KAddr} \mid \text{ConLabel} \times \overline{\text{VAddr}} & & \\
v \in \text{Value} & ::= \text{OpInfo} \times \text{TFrame} \times \text{KAddr} \mid \text{Denotable} & & \\
\sigma \in \text{Store} & ::= \text{KAddr} \to \text{KAddr} \bigcup \text{VAddr} \to \text{Denotable} & &
\end{array}
$$

Figure. 9. Timestamped Big-Step Components

Theorem 2.2. (*Simulation Completeness*)
If $c \equiv (e, [], [])$, $c.fvs = []$, and $c \Downarrow r$, then $\exists v, \sigma',\ (e, \rho, \sigma) \Downarrow_{eval} (v, \sigma')$ and $r \equiv (v, \sigma')$.

Both proofs are formalized in Lean 4 and proceed by height-indexed well-founded recursion on the evaluation derivation, simultaneously establishing soundness (respectively completeness) for both expression evaluation ($\Downarrow_{eval}$) and continuation application ($\Downarrow_{apply}$).

3 Timestamped Semantics with Store Allocated Continuations

To abstract the concrete semantics, we apply standard techniques of indirecting recursive structures through a store and finitizing the address space, making the analysis finite and computable. However, the presence of first-class continuations requires careful treatment. This section presents an intermediate timestamped concrete semantics that makes all allocations explicit and prepares for abstraction, deferring the actual abstraction to Section 4.

Recall the three precision challenges from Section 1.2: *delimiter forgetfulness* (losing delimiter context), *operator forgetfulness* (conflating continuations from different operation calls), and *continuation fragmentation* (frames from distinct continuations mixing in the address space). Our timestamped semantics addresses these through two mechanisms. First, a two-component timestamp $(\tau_\kappa,\ \tau_{m\kappa})$ tracks the call stack and the delimiter meta-stack, providing delimiter sensitivity and operator sensitivity. Second, a specialized continuation address space augments frame-based addressing with operation invocation context, maintaining continuation cohesion.

Figure 9 shows the timestamped components. We make two key changes: adding timestamps for context sensitivity, and store-allocating continuation segments.

Timestamps provide the context-sensitive distinguishing power needed for finite abstraction. Each timestamp has two components: the *continuation timestamp* τ_κ (a sequence of call-site labels tracking the call stack) and the *meta-continuation timestamp* $\tau_{m\kappa}$ (a sequence of delimiter labels paired with τ_κ values, tracking handler nesting and their local continuations). Together these components provide operator sensitivity (distinguishing operation invocations from different contexts) and delimiter sensitivity (distinguishing multiple resumptions of the same continuation). Value addresses become pairs of identifiers and timestamps, frames become timestamped frames (TFrame), and operation invocations are recorded as $o \in$ OpInfo. A key property of this timestamp design is that it ensures *fresh allocation*: distinct dynamic allocations (of values or continuation segments) receive distinct addresses. We prove this formally in Section 3.2.

While stores and environments already indirect their recursive structure, continuations now require store allocation. We separate continuation addresses from value addresses using a distinguished space KAddr. Each frame is addressed by a KAddr = (ψ_τ, o); the store maps it to the

next KAddr in the chain, with $a_{\kappa end}$ terminating the chain. The OpInfo component o records the suspended operation that caused capture. While o is not needed for fresh allocation, it is crucial for keeping continuations distinct when finitizing the address space, preventing frames from separate continuations from merging even when they share syntactic structure.

$\Downarrow^{\kappa\tau}_{eval}$: (Exp × Env × Store × Time) × (Value × Store)
$\Downarrow^{\kappa\tau}_{evalc}$: (CExp × Env × Store × Time) × (Value × Store)
$\Rightarrow^{\kappa\tau}_{cont}$: (LetFrame × Value × Store × Time) × (Value × Store)

TIME-EVAL-LET

$$\frac{(ce, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{evalc} (v, \sigma') \quad ((y, e, \rho), v, \tau) \Rightarrow^{\kappa\tau}_{cont} (v', \sigma'')}{(\text{let } y = ce;\ e, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{eval} (v', \sigma'')}$$

TIME-EVAL-TAIL

$$\frac{(ce, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{evalc} (v, \sigma')}{(ce, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{eval} (v, \sigma')}$$

TIME-EVAL-ATOMIC

$$\frac{v = \mathcal{A}(\text{æ}, \rho, \sigma)}{(\text{æ}, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{evalc} (v, \sigma')}$$

TIME-CONTINUE-LET

$$\frac{\sigma_{let} = \sigma[a_v \mapsto d] \quad a_v = (y, \tau) \quad (e, \rho[y \mapsto a_v], \sigma_{let}, \tau) \Downarrow^{\kappa\tau}_{eval} (v', \sigma')}{((y, e, \rho), d, \sigma, \tau) \Rightarrow^{\kappa\tau}_{cont} (v', \sigma')}$$

TIME-EVAL-FUN

$$\frac{v_f = (\overline{x}, e_1, \rho[f \mapsto a_v]) \quad a_v = (f, \tau)}{(\text{fun } f(\overline{x})\ e_1, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{evalc} (v_f, \sigma[a_v \mapsto v_f])}$$

TIME-EVAL-APP-CLOS

$$\frac{(\overline{x}, e_{body}, \rho_\lambda) = \mathcal{A}(f, \rho, \sigma) \quad \tau' = (l : \tau_\kappa, \tau_{m\kappa}) \quad \sigma_\lambda = \sigma[\overline{\boldsymbol{a_v} \mapsto \mathcal{A}(\text{æ}, \rho, \sigma)}] \quad \overline{\boldsymbol{a_v} = (x, \tau')} \quad (e_{body}, \rho_\lambda[\overline{x \mapsto \boldsymbol{a_v}}], \sigma_\lambda, \tau') \Downarrow^{\kappa\tau}_{eval} (v', \sigma')}{(f(\overline{\text{æ}})^l, \rho, \sigma, (\tau_\kappa, \tau_{m\kappa})) \Downarrow^{\kappa\tau}_{evalc} (v', \sigma')}$$

TIME-EVAL-MATCH

$$\frac{c(\overline{a}_v) = \mathcal{A}(\text{æ}, \rho, \sigma) \quad \overline{\boldsymbol{a'_v} = (x, \tau')} \quad b(c) = (\overline{x}, e) \quad \rho_b = \rho[\overline{x \mapsto \boldsymbol{a'_v}}] \quad \sigma_b = \sigma[\overline{\boldsymbol{a'_v} \mapsto \sigma(\boldsymbol{a_v})}] \quad (e, \rho_b, \sigma_b, \tau) \Downarrow^{\kappa\tau}_{eval} (v', \sigma')}{(\text{match}(\text{æ})\{b\}, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{evalc} (v', \sigma')}$$

Figure. 10. Timestamped Big-Step Lambda Rules

The semantics for the lambda calculus requires few changes. Changes related to timestamps are highlighted in green, and changes related to store allocation and the address space are in blue. Continuations remain implicit in the derivation tree structure, so no continuation addresses appear. The primary changes are: (1) timestamps added throughout, and (2) addresses allocated using timestamps to ensure freshness. The environment and store changes move in lockstep: whenever a binder introduces an address $a_v = (x, \tau)$, that address simultaneously enters the environment as $\rho[x \mapsto a_v]$ and the store as $\sigma[a_v \mapsto d]$. Since environments still map source variables to addresses, the atomic-evaluation meta-function $\mathcal{A}$ is unchanged. New timestamps are created only in the application rule, where call-site labels extend τ_κ. This will provide call-site sensitivity (like m-CFA) when abstracted.

The handler rules have more substantial changes. Operation application [TIME-EVAL-APP-OP] returns a suspended operation with context information, the initial continuation frame, and the end address $a_{\kappa end}$. As evaluation returns through enclosing contexts via [TIME-CONTINUE-OP] with the suspended operation, we allocate continuation segments using pushdown-for-free addressing (P4F) [Gilray et al. 2016]: each segment's address is constructed from the next frame's context. However, unlike standard P4F we augment addresses with the operation context so that frames from continuations capturing different holes in evaluation can remain distinct under abstraction.

Handler evaluation [TIME-EVAL-HANDLER] evaluates the body at a new timestamp by pushing the current τ_κ onto $\tau_{m\kappa}$ with the handler's label l: $([],\ (l, \tau_\kappa) : \tau_{m\kappa})$ (starting afresh in a new delimited scope). In contrast, the handler operations evaluate using the original timestamp $(\tau_\kappa,\ \tau_{m\kappa})$, since their evaluation occurs above the delimiter in the context where the handler was installed.

$\Downarrow^{K\tau}_{\text{handle}} : (\text{Handler} \times \text{Env} \times \text{Label} \times \text{Value} \times \text{Store} \times \text{Time}) \times (\text{Value} \times \text{Store})$
$\Downarrow^{K\tau}_{apply} : (\text{KAddr} \times \text{Denotable} \times \text{Store} \times \text{Time}) \times (\text{Value} \times \text{Store})$

TIME-EVAL-APP-OP

$$\frac{\psi_\tau = ((l, \rho), \tau) \quad o = (op, \rho(\boldsymbol{x}), \psi_\tau) \quad v = (o, \psi_\tau, a_{\kappa end})}{(op(\overline{x})^l, \rho, \sigma, \tau) \Downarrow^{K\tau}_{evalc} (v, \sigma)}$$

TIME-CONTINUE-OP

$$\frac{o = (op, \overline{a}_v, \psi_{\tau\kappa}) \quad a_\kappa = (\psi'_\tau, o) \quad \psi_\tau = (clo, \tau) \quad v = (o, \psi_\tau, a_\kappa) \quad \sigma' = \sigma[a_\kappa \mapsto a_\kappa{}']}{(clo, (o, \psi'_\tau, a_\kappa{}'), \sigma, \tau) \Rightarrow^{K\tau}_{cont} (v, \sigma')}$$

TIME-EVAL-HANDLER

$$\frac{(e_{body}, \rho, \sigma, ([], (l, \tau_\kappa) : \tau_{m\kappa})) \Downarrow^{K\tau}_{eval} (v', \sigma') \quad (h, \rho, l, v', \sigma', (\tau_\kappa, \tau_{m\kappa})) \Downarrow^{K\tau}_{\text{handle}} (v'', \sigma'')}{(h(e_{body})^l, \rho, \sigma, (\tau_\kappa, \tau_{m\kappa})) \Downarrow^{K\tau}_{evalc} (v'', \sigma''')}$$

TIME-EVAL-APP-KONT

$$\frac{v = \mathcal{A}(\text{æ}, \rho, \sigma) \quad (h, \rho_h, \_, a_\kappa) = \mathcal{A}(f, \rho, \sigma) \quad (a_\kappa, v, \sigma, (l, l : \tau_\kappa) : \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (v', \sigma') \quad (h, \rho_h, l, v', \sigma', (l : \tau_\kappa, \tau_{m\kappa})) \Downarrow^{K\tau}_{\text{handle}} (v'', \sigma'')}{(f(\text{æ})^l, \rho, \sigma, (\tau_\kappa, \tau_{m\kappa})) \Downarrow^{K\tau}_{evalc} (v'', \sigma'')}$$

TIME-HANDLE-OP

$$\frac{h(op) = (\overline{x}, e_{op}) \quad a_{vk} = (resume, \tau) \quad \boldsymbol{a'_v} = (\boldsymbol{x}, \tau) \quad \sigma_{op} = \sigma[a_{vk} \mapsto (h, \rho, l, a_\kappa), \boldsymbol{a'_v} \mapsto \sigma(\boldsymbol{a_v})] \quad (e_{op}, \rho[resume \mapsto a_{vk}, \boldsymbol{x} \mapsto \boldsymbol{a'_v}], \sigma_{op}, \tau) \Downarrow^{K\tau}_{eval} (v, \sigma')}{(h, \rho, l, (op, \overline{a}_v, a_\kappa), \sigma, \tau) \Downarrow^{K\tau}_{\text{handle}} (v, \sigma')}$$

TIME-APPLY-RESTORE

$$\frac{(clo, (\tau_\kappa, \_)) = a_\kappa \quad a_\kappa{}' = \sigma(a_\kappa) \quad (a_\kappa{}', d, \sigma, \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (v', \sigma') \quad (clo, v', \sigma', (\tau_\kappa, \tau_{m\kappa})) \Rightarrow^{K\tau}_{cont} (v'', \sigma'')}{(a_\kappa, d, \sigma, \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (v'', \sigma'')}$$

TIME-HANDLE-RETURN

$$\frac{h(\text{return}) = (x_{ret}, e_{ret}) \quad \sigma_{ret} = \sigma[a_v \mapsto d] \quad a_v = (x_{ret}, \tau) \quad (e_{ret}, \rho[x_{ret} \mapsto a_v], \sigma_{ret}, \tau) \Downarrow^{K\tau}_{eval} (v', \sigma')}{(h, \rho, l, d, \sigma, \tau) \Downarrow^{K\tau}_{\text{handle}} (v', \sigma')}$$

TIME-APPLY-CONTINUE

$$\frac{((l, \rho), \tau) = a_\kappa \quad a_{\kappa end} = \sigma(a_\kappa)}{(a_\kappa, d, \sigma, \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (d, \sigma)}$$

TIME-HANDLE-CAPTURE-OP

$$\frac{op \notin h \quad o = (op, \overline{a}_v, \psi_{\tau\kappa}) \quad v = (o, \psi_\tau, a_\kappa) \quad \psi_\tau = ((h, \rho, l), \tau) \quad a_\kappa = (\psi'_\tau, o) \quad \sigma' = \sigma[a_\kappa \mapsto a_\kappa{}']}{(h, \rho, l, (o, \psi'_\tau, a_\kappa{}'), \sigma, \tau) \Downarrow^{K\tau}_{\text{handle}} (v, \sigma')}$$

TIME-APPLY-RESTORE-HANDLE

$$\frac{((h, \rho, l), (\tau_\kappa, \_)) = a_\kappa \quad a_\kappa{}' = \sigma(a_\kappa) \quad (a_\kappa{}', d, \sigma, (l, \tau_\kappa) : \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (v', \sigma') \quad (h, \rho, l, v', \sigma', (\tau_\kappa, \tau_{m\kappa})) \Downarrow^{K\tau}_{\text{handle}} (v'', \sigma'')}{(a_\kappa, d, \sigma, \tau_{m\kappa}) \Downarrow^{K\tau}_{apply} (v'', \sigma'')}$$

Figure. 11. Timestamped Big-Step Handler Rules

Continuation application [TIME-EVAL-APP-KONT] at timestamp $(\tau_\kappa,\ \tau_{m\kappa})$ invokes the apply relation at $((l,\ l : \tau_\kappa) : \tau_{m\kappa})$ and the handle premise at $(l : \tau_\kappa,\ \tau_{m\kappa})$. The label l serves two purposes: it identifies the resumption call site (extending τ_κ to $l : \tau_\kappa$), which is important for separating environment bindings from later sequential operation invocations, and it marks the delimiter introduction point (extending $\tau_{m\kappa}$). The apply relation restores the appropriate τ_κ from stored segments using only $\tau_{m\kappa}$ from the new context.

Rules [TIME-HANDLE-OP] and [TIME-HANDLE-RETURN] use timestamped addresses for bindings. Rule [TIME-HANDLE-CAPTURE-OP] allocates a continuation address and adds a handler frame when the operation doesn't match.

For restoration, [TIME-APPLY-RESTORE] applies the continuation tail (at the same $\tau_{m\kappa}$), then continues with the current frame at $(\tau_\kappa,\ \tau_{m\kappa})$ where τ_κ is the τ_κ stored in the closure frame. Rule [TIME-APPLY-RESTORE-HANDLE] applies the tail at $(l,\ \tau_\kappa) : \tau_{m\kappa}$, then handles the result at $(\tau_\kappa,\ \tau_{m\kappa})$, where τ_κ is the τ_κ stored in the handler frame. Recursion ends at [TIME-APPLY-CONTINUE], which returns the denotable.

3.1 Running Example: Timestamp Evolution

Point	(t_k, t_{mk})	Notes
`f(0)`	$([l_f], [(l_h, [])])$	n=0
`sample(n)` $\to \kappa_1$	$([l_f], [(l_h, [])])$	n=0
`resume(hi)`$^{l_{r1}}$	$([], [])$	x=0, hi=10
`| sample(n1)` $\to \kappa_2$	$([l_f], [(l_{r1}, [l_{r1}])])$	$s1$=10, $n1$=1
`| resume(hi)`$^{l_{r1}}$	$([], [(l_{r1}, [l_{r1}])])$	$s1$=10, x=1, hi=11
`| | s1 + s2` $\to 21$	$([l_f], [(l_{r1}, [l_{r1}, l_{r1}])])$	$s1$=10, $s2$=11
`| resume(lo)`$^{l_{r2}}$	$([], [(l_{r1}, [l_{r1}])])$	$s1$=10, x=1, lo=−9
`| | s1 + s2` $\to 1$	$([l_f], [(l_{r1}, [l_{r1}, l_{r2}])])$	$s1$=10, $s2$=−9
`| r1 + r2` $\to 22$	$([l_{r1}], [])$	$r1$=21, $r2$=1
`resume(lo)`$^{l_{r2}}$	$([], [])$	x=0, lo=−10
`| sample(n1)` $\to \kappa_3$	$([l_f], [(l_{r2}, [l_{r2}])])$	$s1$=−10, $n1$=1
`| resume(hi)`$^{l_{r1}}$	$([], [(l_{r2}, [l_{r2}])])$	$s1$=−10, x=1, hi=11
`| | s1 + s2` $\to 1$	$([l_f], [(l_{r2}, [l_{r2}, l_{r1}])])$	$s1$=−10, $s2$=11
`| resume(lo)`$^{l_{r2}}$	$([], [(l_{r2}, [l_{r2}])])$	$s1$=−10, x=1, lo=−9
`| | s1 + s2` $\to -19$	$([l_f], [(l_{r2}, [l_{r2}, l_{r2}])])$	$s1$=−10, $s2$=−9
`| r1 + r2` $\to -18$	$([l_{r2}], [])$	$r1$=1, $r2$=−19
`r1 + r2` $\to 4$	$([], [])$	$r1$=22, $r2$=−18

Figure. 12. Timestamp evolution in the running example. The four s_1 + s_2 evaluations have distinct $\tau_{m\kappa}$: $(l_{r_1},\ [l_{r_1},\ l_{r_1}])$, $(l_{r_1},\ [l_{r_1},\ l_{r_2}])$, $(l_{r_2},\ [l_{r_2},\ l_{r_1}])$, $(l_{r_2},\ [l_{r_2},\ l_{r_2}])$. This distinguishes the bindings for s_1 and s_2.

Returning to our running example (Figure 2), Figure 12 shows how timestamps evolve during evaluation, demonstrating how our two-component design distinguishes all four execution paths.

When the handler resumes κ_1 twice, each resumption receives a different call-site label (l_{r_1} vs l_{r_2}), extending τ_κ differently. Both resumptions also extend $\tau_{m\kappa}$ with the handler's delimiter information and the current τ_κ. This creates two distinct timestamp contexts for evaluating the resumed continuation.

When each resumed continuation reaches `sample(`n_1`)`, it executes under a unique timestamp. This second operation is then also resumed twice, with each resumption further extending the timestamp. The cascading effect is that all four paths through the program — corresponding to the four leaf results (21, 1, 1, −19) — have distinct timestamps at every program point.

This timestamp distinctness is crucial: it ensures that each allocation during execution receives a fresh address, as formalized in the correctness theorems below.

3.2 Timestamped Correctness

We establish that the timestamped semantics is equivalent to the concrete semantics.

Theorem 3.3. (*Timestamp Soundness*)
$(e, [], [], \tau_0) \Downarrow^{K\tau}_{eval} (v_t, \sigma_t) \implies \exists v\ \sigma.\ (e, [], []) \Downarrow_{eval} (v, \sigma)$

Theorem 3.4. (*Timestamp Completeness*)
$(e, [], []) \Downarrow_{eval} (v, \sigma) \implies \exists v_t\ \sigma_t.\ (e, [], [], \tau_0) \Downarrow^{\kappa\tau}_{eval} (v_t, \sigma_t)$

Both theorems factor through a *fresh-guarded* timestamped semantics ($\Downarrow^{\kappa\tau f}_{eval}$) — identical to the ordinary timestamped semantics $\Downarrow^{\kappa\tau}_{eval}$ except that each allocation rule additionally requires its target address to be absent from the store. The key property, *address freshness*, is that every $\Downarrow^{\kappa\tau}_{eval}$ derivation from an initially empty store already meets these guards. It rests on how timestamps evolve: each allocation happens at a timestamp that extends, and is therefore a *descendant* of, the current one. By mutual induction over all five timestamped relations ($\Downarrow^{\kappa\tau}_{eval}$, $\Downarrow^{\kappa\tau}_{evalc}$, $\Rightarrow^{\kappa\tau}_{cont}$, $\Downarrow^{\kappa\tau}_{\mathsf{handle}}$, $\Downarrow^{\kappa\tau}_{apply}$), we maintain descendant freshness as a pair of store invariants — an input precondition that the store holds no address with a descendant timestamp, and an output guarantee (a *write bound*) that a sub-derivation only allocates at its own descendants. Together with segment disjointness (Section 3.2.1) and a well-formedness invariant on continuation chains (Section 3.2.3), these ensure sibling sub-derivations never collide. Allocations are thus always fresh, so $\Downarrow^{\kappa\tau}_{eval}$ never overwrites a live entry, $\Downarrow^{\kappa\tau}_{eval}$ and $\Downarrow^{\kappa\tau f}_{eval}$ coincide from an empty store, and a store correspondence with the concrete semantics can be maintained throughout evaluation.

3.2.1 Segment Disjointness. A *segment* is the set of derivation nodes sharing a common timestamp. Within a segment, evaluation follows an acyclic ANF let-chain, so each labeled expression is reached at most once; the rules that re-enter code — function calls, handler entries, and resumptions — all extend the timestamp, opening a new segment. Allocations within a segment therefore land at distinct labels, and distinct segments carry distinct timestamps.

3.2.2 Freshness by Induction. Formally, the input precondition $\mathit{descFresh}(\sigma,\ \tau,\ l)$ holds when no address in σ has a timestamp descending from τ extended with l. An allocating rule such as [TIME-EVAL-APP-CLOS] allocates at exactly this descendant timestamp, so the precondition immediately makes the new address fresh.

The output write bound keeps this compositional: each sub-derivation writes only within its own descendant segments. Since segment disjointness makes a sibling's segments disjoint from these, descendant freshness carries over to siblings — after a let-binding's value is evaluated, the continuation still allocates freshly in the segments it goes on to evaluate.

3.2.3 Store Well-Formedness. The hardest case is continuation restoration ([TIME-EVAL-APP-KONT]), which re-inserts frames from a previously captured continuation into the current store. These frames carry old timestamps, so we must know they collide neither with current store contents nor with each other. Store well-formedness invariants provide this: continuation chains are acyclic, and their frames are kept mutually distinct by the timestamps and labels they carry. Freshness then keeps these frames clear of addresses from later evaluation, and the main induction preserves them.

3.2.4 From Freshness to Simulation. With freshness established, it remains to relate the fresh-guarded semantics to the concrete one — a simulation in which the allocation guards can now be taken for granted. Factoring the proof this way cleanly separates the freshness argument from the correspondence argument.

Both directions are proved for arbitrary intermediate configurations, maintaining a *store correspondence*: a map $\alpha : \mathsf{VAddr} \to \mathsf{TVAddr}$ that is a bijection between the live portions of the two stores — every live concrete address a pairs with a live timestamped address $\alpha(a)$ holding a structurally matching value, and vice versa. Environments correspond pointwise, and α extends monotonically at each allocation.

Completeness (concrete ⇒ fresh-guarded) is a mutual induction on the concrete derivation height, constructing a $\Downarrow_{eval}^{\kappa\tau f}$ derivation alongside it. At each allocation, descendant freshness makes the new timestamped address unused, so extending α preserves the correspondence; the induction also threads the store invariants from Section 3.2.3 and the write bounds that continuation restoration needs.

Soundness (fresh-guarded ⇒ concrete) is a mutual induction on the $\Downarrow_{eval}^{\kappa\tau f}$ derivation height. At each allocation, the fresh-guarded premise supplies an unused timestamped address; the proof pairs it with a fresh concrete address (always available, as the concrete address space is $\mathbb{N}$) and extends both α and the concrete store without colliding with existing entries. This direction needs no store invariants, since the concrete semantics allocates independently.

Both theorems then follow: starting configurations with empty stores and environments trivially satisfy the correspondence, so the generalized statements specialize to the simulation.

4 Abstract Big Step Semantics

$$
\begin{array}{llll}
\hat{a}_v \in \widehat{\mathsf{VAddr}} & ::= \mathsf{Var} \times \widehat{\mathsf{Time}} & \hat{\rho} \in \widehat{\mathsf{Env}} & ::= \mathsf{Var} \to \widehat{\mathsf{VAddr}} \\
clo \in \widehat{\mathsf{Closure}} & ::= \overline{\mathsf{Var}} \times \mathsf{Exp} \times \widehat{\mathsf{Env}} & o \in \widehat{\mathsf{OpInfo}} & ::= \mathsf{OpName} \times \overline{\widehat{\mathsf{VAddr}}} \times \widehat{\mathsf{TFrame}} \\
\hat{\tau}_k \in \widehat{\mathsf{KTime}} & ::= \mathsf{Label}^{m} & \hat{\tau} \in \widehat{\mathsf{Time}} & ::= \widehat{\mathsf{KTime}} \times \widehat{\mathsf{MKTime}} \\
\hat{\tau}_{m\kappa} \in \widehat{\mathsf{MKTime}} & ::= (\mathsf{Label} \times \widehat{\mathsf{KTime}})^{h} & \psi_t \in \widehat{\mathsf{TFrame}} & ::= \widehat{\mathsf{Frame}} \times \widehat{\mathsf{Time}} \\
\psi \in \widehat{\mathsf{Frame}} & ::= \widehat{\mathsf{LetFrame}} \mid \mathsf{Label} \times \widehat{\mathsf{Env}} \mid \widehat{\mathsf{Handler}} \times \widehat{\mathsf{Env}} & & \\
\psi_{let} \in \widehat{\mathsf{LetFrame}} & ::= \mathsf{Var} \times \mathsf{Exp} \times \widehat{\mathsf{Env}} & & \\
\hat{a}_\kappa \in \widehat{\mathsf{KAddr}} & ::= \widehat{\mathsf{TFrame}} \times \widehat{\mathsf{OpInfo}} \mid a_{\kappa end} & & \\
d \in \widehat{\mathsf{Denotable}} & ::= \widehat{\mathsf{Closure}} \mid \widehat{\mathsf{Handler}} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{KAddr}} \mid \mathsf{ConLabel} \times \overline{\widehat{\mathsf{VAddr}}} & & \\
\hat{v} \in \widehat{\mathsf{Value}} & ::= \widehat{\mathsf{OpInfo}} \times \widehat{\mathsf{TFrame}} \times \widehat{\mathsf{KAddr}} \mid \widehat{\mathsf{Denotable}} & & \\
\hat{\sigma} \in \widehat{\mathsf{Store}} & ::= \widehat{\mathsf{KAddr}} \to \mathcal{P}(\widehat{\mathsf{KAddr}}) \bigcup \widehat{\mathsf{VAddr}} \to \mathcal{P}(\widehat{\mathsf{Denotable}}) & &
\end{array}
$$

Figure. 13. Abstract Big-Step Components.

For the abstract semantics we finitize the state space by truncating timestamps (in green) and adding non-determinism (in red), yielding a computable analysis; the abstraction is mechanical, as the key design decisions were already made in the timestamped semantics. Even under truncation, the $\widehat{\mathsf{OpInfo}}$ component of $\widehat{\mathsf{KAddr}}$s preserves continuation cohesion: continuations from distinct operation invocations stay structurally separate, independent of context sensitivity.

The abstract timestamp components $\hat{t}_k \in \widehat{\mathsf{KTime}}$ and $\hat{t}_{m\kappa} \in \widehat{\mathsf{MKTime}}$ are truncated to lengths m and h respectively (using $\lfloor\, l \,\rfloor_n$ to denote truncation to the first n elements). These two lengths name our analysis, **HMCFA** (handler m-CFA): we write $\mathsf{H}(h, m)$ for the instance with handler sensitivity h and call-site sensitivity m. All other components replace unbounded components with their bounded counterparts. Since the address space is now finite, the store must map addresses to powersets of values.

Store operations use join ($\hat{\sigma} \sqcup \hat{\sigma}' = \lambda a.\ \hat{\sigma}(a) \cup \hat{\sigma}'(a)$) to combine stores and non-deterministic lookup ($d \in \hat{\sigma}(a)$) to select values.

$$
\begin{array}{ll}
\hat{\mathcal{A}} & : \mathsf{AExp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \to \mathcal{P}(\widehat{\mathsf{Denotable}}) \\
\hat{\mathcal{A}}(x, \hat{\rho}, \hat{\sigma}_v) & = \hat{\sigma}_v(\hat{\rho}(x)) \\
\hat{\mathcal{A}}(c(\overline{x}), \hat{\rho}, \hat{\sigma}_v) & = \{(c, \overline{\hat{\rho}(x)})\} \\
\hat{\mathcal{A}}(\mathsf{fn}(\overline{x})\ e_{body}, \hat{\rho}_\lambda, \hat{\sigma}_v) & = \{(\overline{x}, e_{body}, \hat{\rho}_\lambda)\}
\end{array}
$$

$\hat{\mathcal{A}}$ evaluates primitives to abstract values (powersets) via store lookup or set injection.

$$\widehat{\Downarrow^{\kappa\tau}_{eval}} : (\mathsf{Exp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{Time}}) \times (\widehat{\mathsf{Value}} \times \widehat{\mathsf{Store}})$$
$$\widehat{\Downarrow^{\kappa\tau}_{evalc}} : (\mathsf{CExp} \times \widehat{\mathsf{Env}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{Time}}) \times (\widehat{\mathsf{Value}} \times \widehat{\mathsf{Store}})$$
$$\widehat{\Rightarrow^{\kappa\tau}_{cont}} : (\widehat{\mathsf{LetFrame}} \times \widehat{\mathsf{Value}} \times \widehat{\mathsf{Store}} \times \widehat{\mathsf{Time}}) \times (\widehat{\mathsf{Value}} \times \widehat{\mathsf{Store}})$$

ABS-EVAL-LET
$$\frac{(ce, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (\hat{v}, \hat{\sigma}') \qquad ((y, e, \hat{\rho}), \hat{v}, \hat{\tau}) \; \widehat{\Rightarrow^{\kappa\tau}_{cont}} \; (\hat{v}', \hat{\sigma}'')}{(\mathsf{let}\ y = ce;\ e, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{eval}} \; (\hat{v}', \hat{\sigma}'')}$$

ABS-EVAL-TAIL
$$\frac{(ce, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (ce, \hat{\sigma}')}{(ce, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{eval}} \; (ce, \hat{\sigma}')}$$

ABS-EVAL-ATOMIC
$$\frac{\hat{v} = \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma})}{(æ, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (\hat{v}, \hat{\sigma}')}$$

ABS-CONTINUE-LET
$$\frac{\hat{\sigma}_{let} = \hat{\sigma} \sqcup [\hat{a}_v \mapsto d] \qquad \hat{a}_v = (y, \hat{\tau}) \qquad (e, \hat{\rho}[y \mapsto \hat{a}_v], \hat{\sigma}_{let}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{eval}} \; (\hat{v}', \hat{\sigma}')}{((y, e, \hat{\rho}), d, \hat{\sigma}, \hat{\tau}) \; \widehat{\Rightarrow^{\kappa\tau}_{cont}} \; (\hat{v}', \hat{\sigma}')}$$

ABS-EVAL-FUN
$$\frac{\hat{v}_f = (\overline{x}, e_1, \hat{\rho}[f \mapsto \hat{a}_v]) \qquad \hat{a}_v = (f, \hat{\tau})}{(\mathsf{fun}\ f(\overline{x})\ e_1, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (\hat{v}_f, \hat{\sigma} \sqcup [\hat{a}_v \mapsto \hat{v}_f])}$$

ABS-EVAL-APP-CLOS
$$\frac{\begin{array}{c}(\overline{x}, e_{body}, \hat{\rho}_{clo}) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \qquad \overline{\hat{\rho}_\lambda = \hat{\rho}_{clo}[\boldsymbol{x} \mapsto \hat{\boldsymbol{a}}_{\boldsymbol{v}}]} \\ \hat{\sigma}_\lambda = \hat{\sigma} \sqcup [\hat{\boldsymbol{a}}_{\boldsymbol{v}} \mapsto \hat{\mathcal{A}}(\boldsymbol{æ}, \hat{\rho}, \hat{\sigma})] \qquad \overline{\hat{\boldsymbol{a}}_{\boldsymbol{v}} = (\boldsymbol{x}, \hat{\tau}')} \\ (e_{body}, \hat{\rho}_\lambda, \hat{\sigma}_\lambda, (\lfloor\, l : \hat{\tau}_k \,\rfloor_m, \hat{\tau}_{m\kappa})) \; \widehat{\Downarrow^{\kappa\tau}_{eval}} \; (\hat{v}', \hat{\sigma}')\end{array}}{(f(\overline{æ})^l, \hat{\rho}, \hat{\sigma}, (\hat{\tau}_k, \hat{\tau}_{m\kappa})) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (\hat{v}'', \hat{\sigma}'')}$$

ABS-EVAL-MATCH
$$\frac{\begin{array}{c}c(\overline{\hat{a}}_v) \in \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma}) \qquad b(c) = (\overline{x}, e) \qquad \overline{\hat{\boldsymbol{a}}'_{\boldsymbol{v}} = (\boldsymbol{x}, \hat{\tau})} \\ \hat{\sigma}_b = \hat{\sigma} \sqcup [\overline{\hat{\boldsymbol{a}}'_{\boldsymbol{v}} \mapsto \hat{\sigma}(\hat{\boldsymbol{a}}_{\boldsymbol{v}})}] \qquad \hat{\rho}_b = \hat{\rho}[\overline{\boldsymbol{x} \mapsto \hat{\boldsymbol{a}}'_{\boldsymbol{v}}}] \\ (e, \hat{\rho}_b, \hat{\sigma}_b, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{eval}} \; (\hat{v}', \hat{\sigma}')\end{array}}{(\mathsf{match}(æ)\{b\}, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \; \widehat{\Downarrow^{\kappa\tau}_{evalc}} \; (\hat{v}', \hat{\sigma}')}$$

Figure. 14. Abstract Big-Step Lambda Rules

The abstract rules otherwise differ from the timestamped rules only in truncating timestamps, joining stores, and non-deterministic store access.

Why a single-component timestamp is insufficient. A naive application of m-CFA or k-CFA to effect handlers would not store timestamps in continuation frames, relying instead on the current evaluation context to provide sensitivity. Consider two resumptions of the same continuation from different program points. With m-CFA (stack-based), each restored frame pushes its call-site label onto τ_κ. The first few restored frames remain distinct because the resumption label is still in τ_κ, but as deeper frames are restored, τ_κ fills with the continuation's own labels, pushing the resumption label off. The deeper frames execute under identical timestamps regardless of which resumption invoked the continuation. With k-CFA (threaded like the store), restoring frames does not change the timestamp — only new calls do. So the resumption label persists through restoration itself. But once the restored frames begin executing and making calls, calls to the same resumption encounter the same code paths and call sites. After k such calls, the resumption label is pushed out and the paths merge. The $\tau_{m\kappa}$ component solves both problems by recording *which resumption introduced each delimiter* in a separate component unaffected by the restored computation's calls. Even as τ_κ evolves during restored evaluation, the surrounding $\tau_{m\kappa}$ preserves the resumption context, keeping distinct resumption paths separated throughout.

4.1 Soundness

To prove soundness of the abstract semantics with respect to the concrete, we first define an abstraction function α 16. Timestamps are truncated to lengths m and h, other components abstract

$\widehat{\Downarrow^{K\tau}_{\text{handle}}} : (\widehat{\text{Handler}} \times \widehat{\text{Env}} \times \text{Label} \times \widehat{\text{Value}} \times \widehat{\text{Store}} \times \widehat{\text{Time}}) \times (\widehat{\text{Value}} \times \widehat{\text{Store}})$

$\widehat{\Downarrow^{K\tau}_{\text{apply}}} : (\widehat{\text{KAddr}} \times \widehat{\text{Denotable}} \times \widehat{\text{Store}} \times \widehat{\text{Time}}) \times (\widehat{\text{Value}} \times \widehat{\text{Store}})$

ABS-EVAL-APP-OP

$$\frac{\hat{\psi}_\tau = ((l, \hat{\rho}), \hat{\tau}) \quad o = (op, \overline{\hat{\rho}(\boldsymbol{x})}, \hat{\psi}_\tau) \quad \hat{v} = (o, \hat{\psi}_\tau, a_{\kappa end})}{(op(\overline{x})^l, \hat{\rho}, \hat{\sigma}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{evalc}} \ (\hat{v}, \hat{\sigma})}$$

ABS-CONTINUE-OP

$$\frac{o = (op, \overline{\hat{a}}_v, \hat{\psi}_{\tau\kappa}) \quad \hat{a}_\kappa = (\hat{\psi}'_\tau, o) \quad \hat{\psi}_\tau = (clo, \hat{\tau}) \quad \hat{v} = (o, \hat{\psi}_\tau, \hat{a}_\kappa) \quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_\kappa \mapsto \{\hat{a}'_\kappa\}]}{(clo, (o, \hat{\psi}'_\tau, \hat{a}'_\kappa), \hat{\sigma}, \hat{\tau}) \ \widehat{\Rightarrow^{K\tau}_{cont}} \ (\hat{v}, \hat{\sigma}')}$$

ABS-EVAL-HANDLER

$$\frac{(e_{body}, \hat{\rho}, \hat{\sigma}, (\lfloor\, \rfloor, \lfloor\, (l, \hat{\tau}_k) : \hat{\tau}_{m\kappa} \,\rfloor_h)) \ \widehat{\Downarrow^{K\tau}_{evalc}} \ (\hat{v}', \hat{\sigma}') \quad (h, \hat{\rho}, l, \hat{v}', \hat{\sigma}', (\hat{\tau}_k, \hat{\tau}_{m\kappa})) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}'', \hat{\sigma}'')}{(h(e_{body})^l, \hat{\rho}, \hat{\sigma}, (\hat{\tau}_k, \hat{\tau}_{m\kappa})) \ \widehat{\Downarrow^{K\tau}_{evalc}} \ (\hat{v}'', \hat{\sigma}'')}$$

ABS-EVAL-APP-KONT

$$\frac{\hat{v} \in \hat{\mathcal{A}}(æ, \hat{\rho}, \hat{\sigma}) \quad (h, \hat{\rho}_h, \_, \hat{a}_\kappa) \in \hat{\mathcal{A}}(f, \hat{\rho}, \hat{\sigma}) \quad (\hat{a}_\kappa, \hat{v}, \hat{\sigma}, \lfloor\, (l, \lfloor\, l : \hat{\tau}_k \,\rfloor_m) : \hat{\tau}_{m\kappa} \,\rfloor_h) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (\hat{v}', \hat{\sigma}') \quad (h, \hat{\rho}_h, l, \hat{v}', \hat{\sigma}', (\lfloor\, l : \hat{\tau}_k \,\rfloor_m, \hat{\tau}_{m\kappa})) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}'', \hat{\sigma}'')}{(f(æ)^l, \hat{\rho}, \hat{\sigma}, (\hat{\tau}_k, \hat{\tau}_{m\kappa})) \ \widehat{\Downarrow^{K\tau}_{evalc}} \ (\hat{v}'', \hat{\sigma}'')}$$

ABS-HANDLE-OP

$$\frac{h(op) = (\overline{x}, e_{op}) \quad \hat{a}_{vk} = (resume, \hat{\tau}) \quad \overline{\hat{\boldsymbol{a}}'_{\boldsymbol{v}} = (\boldsymbol{x}, \hat{\tau})} \quad \hat{\sigma}_{op} = \hat{\sigma} \sqcup [\hat{a}_{vk} \mapsto \{(h, \hat{\rho}, l, \hat{a}_\kappa)\}, \overline{\hat{\boldsymbol{a}}'_{\boldsymbol{v}} \mapsto \hat{\sigma}(\hat{\boldsymbol{a}}_{\boldsymbol{v}})}] \quad (e_{op}, \hat{\rho}[resume \mapsto \hat{a}_{vk}, \overline{\boldsymbol{x} \mapsto \hat{\boldsymbol{a}}'_{\boldsymbol{v}}}], \hat{\sigma}_{op}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{eval}} \ (\hat{v}, \hat{\sigma}')}{(h, \hat{\rho}, l, ((op, \overline{\hat{a}}_v, \hat{\psi}_{\tau\kappa}), \hat{\psi}'_\tau, \hat{a}_\kappa), \hat{\sigma}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}, \hat{\sigma}')}$$

ABS-APPLY-RESTORE

$$\frac{(clo, (\hat{\tau}_k, \_)) = \hat{a}_\kappa \quad \hat{a}'_\kappa \in \hat{\sigma}(\hat{a}_\kappa) \quad (\hat{a}'_\kappa, d, \hat{\sigma}, \hat{\tau}_{m\kappa}) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (\hat{v}', \hat{\sigma}') \quad (clo, \hat{v}', \hat{\sigma}', (\hat{\tau}_k, \hat{\tau}_{m\kappa})) \ \widehat{\Rightarrow^{K\tau}_{cont}} \ (\hat{v}'', \hat{\sigma}'')}{(\hat{a}_\kappa, d, \hat{\sigma}, \hat{\tau}_{m\kappa}) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (\hat{v}'', \hat{\sigma}'')}$$

ABS-HANDLE-RETURN

$$\frac{h(\text{return}) = (x_{ret}, e_{ret}) \quad \hat{\sigma}_{ret} = \hat{\sigma} \sqcup [\hat{a}_v \mapsto d] \quad \hat{a}_v = (x_{ret}, \hat{\tau}) \quad (e_{ret}, \hat{\rho}[x_{ret} \mapsto \hat{a}_v], \hat{\sigma}_{ret}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{eval}} \ (\hat{v}, \hat{\sigma}')}{(h, \hat{\rho}, l, d, \hat{\sigma}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}, \hat{\sigma}')}$$

ABS-APPLY-CONTINUE

$$\frac{((l, \hat{\rho}), \hat{\tau}) = \hat{a}_\kappa \quad \hat{a}_{\kappa end} \in \hat{\sigma}(\hat{a}_\kappa)}{(\hat{a}_\kappa, d, \hat{\sigma}, \hat{\tau}_{m\kappa}) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (d, \hat{\sigma})}$$

ABS-HANDLE-CAPTURE-OP

$$\frac{op \notin h \quad o = (op, \overline{\hat{a}}_v, \hat{\psi}_{\tau\kappa}) \quad \hat{v} = (o, \hat{\psi}_\tau, \hat{a}_\kappa) \quad \hat{\psi}_\tau = ((h, \hat{\rho}, l), \hat{\tau}) \quad \hat{a}_\kappa = (\hat{\psi}'_\tau, o) \quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_\kappa \mapsto \{\hat{a}'_\kappa\}]}{(h, \hat{\rho}, l, (o, \hat{\psi}'_\tau, \hat{a}'_\kappa), \hat{\sigma}, \hat{\tau}) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}, \hat{\sigma}')}$$

ABS-APPLY-RESTORE-HANDLE

$$\frac{((h, \hat{\rho}, l), (\hat{\tau}_\kappa, \_)) = \hat{a}_\kappa \quad \hat{a}'_\kappa \in \hat{\sigma}(\hat{a}_\kappa) \quad (\hat{a}'_\kappa, d, \hat{\sigma}, \lfloor\, (l, \hat{\tau}_\kappa) : \hat{\tau}_{m\kappa} \,\rfloor_h) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (\hat{v}', \hat{\sigma}') \quad (h, \hat{\rho}, l, \hat{v}', \hat{\sigma}', (\hat{\tau}_\kappa, \hat{\tau}_{m\kappa})) \ \widehat{\Downarrow^{K\tau}_{\text{handle}}} \ (\hat{v}'', \hat{\sigma}'')}{(\hat{a}_\kappa, d, \hat{\sigma}, \hat{\tau}_{m\kappa}) \ \widehat{\Downarrow^{K\tau}_{apply}} \ (\hat{v}'', \hat{\sigma}'')}$$

Figure. 15. Abstract Big-Step Handler Rules

$$\alpha(a_{\kappa end}) = \hat{a}_{\kappa end} \qquad \alpha(\tau_\kappa) = \lfloor\, \tau_\kappa \,\rfloor_m \qquad \alpha((l, \tau_\kappa) : \tau_{m\kappa}) = \lfloor\, (l, \alpha(\tau_\kappa)) : \alpha(\tau_{m\kappa}) \,\rfloor_h$$
$$\alpha(\rho) = \lambda x.\, \alpha(\rho(x)) \qquad \alpha(\sigma) = \lambda \hat{a}.\, \{\alpha(d) \mid a \in dom(\sigma),\ \alpha(a) = \hat{a},\ d \in \sigma(a)\}$$

Figure. 16. Abstraction of Components: unmentioned components abstract componentwise.

componentwise: applying α to embedded components. Since truncation forces multiple concrete addresses to abstract to the same abstract address, such addresses map to sets of values.

Theorem 4.5. (*Abstract Soundness*)
If $(e, \rho, \sigma, \tau) \Downarrow^{\kappa\tau}_{eval} (v, \sigma')$ and $\alpha(\sigma) \sqsubseteq \hat{\sigma}$, then $(e, \alpha(\rho), \hat{\sigma}, \alpha(\tau)) \widehat{\Downarrow^{\kappa\tau}_{eval}} (\hat{v}, \hat{\sigma}')$ where $\alpha(v) \in \hat{v}$ and $\alpha(\sigma') \sqsubseteq \hat{\sigma}'$.

Proof. The Lean 4 proof is by well-founded recursion on derivation height, simultaneously over the five timestamped relations ($\Downarrow^{\kappa\tau}_{eval}$, $\Downarrow^{\kappa\tau}_{evalc}$, $\Rightarrow^{\kappa\tau}_{cont}$, $\Downarrow^{\kappa\tau}_{\mathsf{handle}}$, $\Downarrow^{\kappa\tau}_{apply}$). Three properties give soundness:

(1) **Timestamp abstraction commutes with the rule operations.** α distributes over extension, truncation, and restoration of both timestamp components, so every abstract rule matches the shape of its timestamped counterpart under α.
(2) **Store abstraction is preserved by allocation.** If $\alpha(\sigma) \sqsubseteq \hat{\sigma}$, then $\alpha(\sigma[a \mapsto d]) \sqsubseteq \hat{\sigma} \sqcup [\alpha(a) \mapsto \{\alpha(d)\}]$, and likewise for continuation-frame allocation; environments and frames abstract componentwise.
(3) **Atomic evaluation abstracts soundly.** If $\mathcal{A}(æ, \rho, \sigma) = d$ and $\alpha(\sigma) \sqsubseteq \hat{\sigma}$, then $\alpha(d) \in \hat{\mathcal{A}}(æ, \alpha(\rho), \hat{\sigma})$. This justifies the $\in$-based non-determinism in the abstract rules.

For each rule, the inductive hypothesis on the premises yields abstract sub-derivations, which properties (1)–(3) re-align to the abstract rule's shape. Simple cases (atomic, tail, and let) apply each property once or twice; function application folds property (2) over the argument bindings; and the handler and continuation rules add case analysis on whether the operation matches and whether the result is a value or a suspended operation.

4.2 Taming Computational Complexity via Rebinding

$$
\begin{array}{lll}
\tilde{a}_v \in \widetilde{\mathsf{VAddr}} & ::= \mathsf{Var} \times \widetilde{\mathsf{Time}} & o \in \widetilde{\mathsf{OpInfo}} ::= \mathsf{OpName} \times \widetilde{\overline{\mathsf{VAddr}}} \times \widetilde{\mathsf{TFrame}} \\
clo \in \widetilde{\mathsf{Closure}} & ::= \overline{\mathsf{Var}} \times \mathsf{Exp} \times \widetilde{\mathsf{Time}} & \tilde{\tau} \in \widetilde{\mathsf{Time}} ::= \widetilde{\mathsf{KTime}} \times \widetilde{\mathsf{MKTime}} \\
\tilde{\tau}_k \in \widetilde{\mathsf{KTime}} & ::= \mathsf{Label}^m & \psi_t \in \widetilde{\mathsf{TFrame}} ::= \widetilde{\mathsf{Frame}} \times \widetilde{\mathsf{Time}} \\
\tilde{\tau}_{m\kappa} \in \widetilde{\mathsf{MKTime}} & ::= (\mathsf{Label} \times \widetilde{\mathsf{KTime}})^h & \\
\psi \in \widetilde{\mathsf{Frame}} & ::= \widetilde{\mathsf{LetFrame}} \mid \mathsf{Label} \mid \widetilde{\mathsf{Handler}} \times \widetilde{\mathsf{Time}} & \\
\psi_{let} \in \widetilde{\mathsf{LetFrame}} & ::= \mathsf{Var} \times \mathsf{Exp} \times \widetilde{\mathsf{Time}} & \\
\tilde{a}_\kappa \in \widetilde{\mathsf{KAddr}} & ::= \widetilde{\mathsf{TFrame}} \times \widetilde{\mathsf{OpInfo}} \mid a_{\kappa end} & \\
d \in \widetilde{\mathsf{Denotable}} & ::= \widetilde{\mathsf{Closure}} \mid \widetilde{\mathsf{Handler}} \times \widetilde{\mathsf{Time}} \times \widetilde{\mathsf{KAddr}} \mid \mathsf{ConLabel} \times \widetilde{\overline{\mathsf{VAddr}}} & \\
\tilde{v} \in \widetilde{\mathsf{Value}} & ::= \widetilde{\mathsf{OpInfo}} \times \widetilde{\mathsf{TFrame}} \times \widetilde{\mathsf{KAddr}} \mid \widetilde{\mathsf{Denotable}} & \\
\tilde{\sigma} \in \widetilde{\mathsf{Store}} & ::= \widetilde{\mathsf{KAddr}} \to \mathcal{P}(\widetilde{\mathsf{KAddr}}) \bigcup \widetilde{\mathsf{VAddr}} \to \mathcal{P}(\widetilde{\mathsf{Denotable}}) &
\end{array}
$$

Figure. 17. Abstract Big-Step Rebinding Components

The abstract semantics as presented remains computationally expensive. The core issue is that environments map free variables to addresses with potentially different timestamps—each from a different call history. This creates an exponential state space: the analysis must explore all combinations of timestamps that free variables could be bound under.

Rebinding, introduced by Might et al. [Might et al. 2010] for m-CFA, keeps environments out of the abstract state entirely. Whenever evaluation enters a new timestamp τ' (e.g., a function call extending the call-site stack), the analysis copies every free variable's address to the canonical address (x, τ'), so a variable x in scope at τ' is *always* found at (x, τ'). Environments are thus eliminated: closures and handler values carry timestamps instead, removing the exponential branching factor while preserving context sensitivity.

$$
\begin{array}{ll}
\tilde{\mathcal{A}} & : \mathsf{AExp} \times \widetilde{\mathsf{Time}} \times \widetilde{\mathsf{Store}} \to \mathcal{P}(\widetilde{\mathsf{Denotable}}) \\
\tilde{\mathcal{A}}(x, \tilde{\tau}, \tilde{\sigma}_v) & = \tilde{\sigma}_v((x, \tilde{\tau})) \\
\tilde{\mathcal{A}}(c(\overline{x}), \tilde{\tau}, \tilde{\sigma}_v) & = \{(c, \overline{(\boldsymbol{x}, \tilde{\tau})})\} \\
\tilde{\mathcal{A}}(\mathsf{fn}(\overline{x})\ e_{body}, \tilde{\tau}_\lambda, \tilde{\sigma}_v) & = \{(\overline{x}, e_{body}, \tilde{\tau}_\lambda)\}
\end{array}
$$

$\tilde{\mathcal{A}}$ constructs variable addresses as canonical pairs $(x,\ \tau)$, using the current timestamp in place of an environment lookup.

ABSR-EVAL-LET

$$\frac{(ce, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}, \tilde{\sigma}') \qquad ((y, e, \tilde{\tau}), \tilde{v}, \tilde{\tau})\ \widetilde{\Rightarrow^{K\tau}_{cont}}\ (\tilde{v}', \tilde{\sigma}'')}{(\mathsf{let}\ y = ce;\ e, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{eval}}\ (\tilde{v}', \tilde{\sigma}'')}$$

ABSR-EVAL-TAIL

$$\frac{(ce, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}, \tilde{\sigma}')}{(ce, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{eval}}\ (\tilde{v}, \tilde{\sigma}')}$$

ABSR-EVAL-ATOMIC

$$\frac{\tilde{v} = \tilde{\mathcal{A}}(\text{æ}, \tilde{\tau}, \tilde{\sigma})}{(\text{æ}, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}, \tilde{\sigma}')}$$

ABSR-CONTINUE-LET

$$\frac{\tilde{\sigma}_d = \tilde{\sigma} \sqcup [\tilde{a}_v \mapsto d] \qquad \tilde{a}_v = (y, \tilde{\tau}') \qquad \tilde{\sigma}_{let} = \tilde{\sigma}_d \sqcup \mathsf{rebind}(fvs(e) \setminus \{y\}, \tilde{\tau}, \tilde{\tau}') \qquad (e, \tilde{\sigma}_{let}, \tilde{\tau}')\ \widetilde{\Downarrow^{K\tau}_{eval}}\ (\tilde{v}', \tilde{\sigma}')}{((y, e, \tilde{\tau}), d, \tilde{\sigma}, \tilde{\tau}')\ \widetilde{\Rightarrow^{K\tau}_{cont}}\ (\tilde{v}', \tilde{\sigma}')}$$

ABSR-EVAL-FUN

$$\frac{\tilde{v}_f = (\overline{x}, e_1, \tilde{\tau}) \qquad \tilde{a}_v = (f, \tilde{\tau})}{(\mathsf{fun}\ f(\overline{x})\ e_1, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}_f, \tilde{\sigma} \sqcup [\tilde{a}_v \mapsto \tilde{v}_f])}$$

ABSR-EVAL-APP-CLOS

$$\frac{\tilde{\tau}_\lambda = (\lfloor\ l : \tilde{\tau}_k\ \rfloor_m, \tilde{\tau}_{m\kappa}) \qquad (\overline{x}, e_{body}, \tilde{\tau}_{clo}) \in \tilde{\mathcal{A}}(f, \tilde{\tau}, \tilde{\sigma}) \qquad \overline{\tilde{\boldsymbol{a}}_{\boldsymbol{v}} = (\boldsymbol{x}, \tilde{\tau}')} \qquad \tilde{\sigma}_b = \tilde{\sigma} \sqcup [\overline{\tilde{\boldsymbol{a}}_{\boldsymbol{v}} \mapsto \tilde{\mathcal{A}}(\boldsymbol{æ}, \tilde{\tau}, \tilde{\sigma})}] \qquad \tilde{\sigma}_\lambda = \tilde{\sigma}_b \sqcup \mathsf{rebind}(fvs(e_{body}) \setminus \{\ldots\overline{x}\}, \tilde{\tau}_{clo}, \tilde{\tau}_\lambda) \qquad (e_{body}, \tilde{\sigma}_\lambda, \tilde{\tau}_\lambda)\ \widetilde{\Downarrow^{K\tau}_{eval}}\ (\tilde{v}', \tilde{\sigma}')}{(f(\overline{\text{æ}})^l, \tilde{\sigma}, (\tilde{\tau}_k, \tilde{\tau}_{m\kappa}))\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}'', \tilde{\sigma}'')}$$

ABSR-EVAL-MATCH

$$\frac{c(\overline{\tilde{a}}_v) \in \tilde{\mathcal{A}}(\text{æ}, \tilde{\tau}, \tilde{\sigma}) \qquad b(c) = (\overline{x}, e) \qquad \overline{\tilde{\boldsymbol{a}}'_{\boldsymbol{v}} = (\boldsymbol{x}, \tilde{\tau})} \qquad \tilde{\sigma}_{body} = \tilde{\sigma} \sqcup [\overline{\tilde{\boldsymbol{a}}'_{\boldsymbol{v}} \mapsto \tilde{\sigma}(\tilde{\boldsymbol{a}}_{\boldsymbol{v}})}] \qquad (e, \tilde{\sigma}_{body}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{eval}}\ (\tilde{v}', \tilde{\sigma}')}{(\mathsf{match}(\text{æ})\{b\}, \tilde{\sigma}, \tilde{\tau})\ \widetilde{\Downarrow^{K\tau}_{evalc}}\ (\tilde{v}', \tilde{\sigma}')}$$

Figure. 18. Abstract Big-Step Rebinding Lambda Rules

Effect handlers introduce additional timestamp transitions beyond function calls, requiring rebinding at more points than standard m-CFA. Rebinding occurs wherever evaluation crosses timestamp boundaries. Function calls rebind free variables to the new call-site timestamp ([ABSR-EVAL-APP-CLOS]). Handler entries rebind the handler body's free variables to the new timestamp ([ABSR-EVAL-HANDLER]). Operation handling and handler returns rebind free variables from the handler closure to the latest timestamp ([ABSR-HANDLE-OP], [ABSR-HANDLE-RETURN]). Continuation applications, however, cannot rebind all of the continuation's free variables at once; this responsibility falls to frame restoration. Frame restoration presents a unique challenge: it combines timestamps from the captured frame with the current context, requiring careful rebinding strategy.

For frame restoration, we delay rebinding until frames are actually evaluated rather than eagerly rebinding at restoration time. Eager rebinding would simplify the formalism but risks spurious merging when frames are restored and then recaptured before evaluation. Delaying rebinding causes a small increase in state space (the closure retains its original timestamp while the enclosing timestamped frame carries the restored timestamp), but avoids premature loss of precision. Concretely, we omit rebinding in [ABSR-APPLY-RESTORE] and instead rebind in [CONTINUE-LET] when frames are evaluated.

ABSR-EVAL-APP-OP

$$\frac{\tilde{\psi}_\tau = (l, \tilde{\tau}) \quad o = (op, \overline{(x, \tilde{\tau})}, \tilde{\psi}_\tau) \quad \tilde{\nu} = (o, \tilde{\psi}_\tau, a_{\kappa end})}{(op(\overline{x})^l, \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{evalc}} \; (\tilde{\nu}, \tilde{\sigma}')}$$

ABSR-CONTINUE-OP

$$\frac{o = (op, \overline{\tilde{a}}_v, \tilde{\psi}_{\tau\kappa}) \quad \tilde{a}_\kappa = (\tilde{\psi}'_\tau, o) \quad \tilde{\psi}_\tau = (clo, \tilde{\tau}) \quad \tilde{\nu} = (o, \tilde{\psi}_\tau, \tilde{a}_\kappa) \quad \tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_\kappa \mapsto \{\tilde{a}'_\kappa\}]}{(clo, (o, \tilde{\psi}'_\tau, \tilde{a}'_\kappa), \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Rightarrow^{\kappa\tau}_{cont}} \; (\tilde{\nu}, \tilde{\sigma}')}$$

ABSR-EVAL-HANDLER

$$\frac{\tilde{\tau}' = ([], \lfloor\, (l, \tilde{\tau}_k) : \tilde{\tau}_{m\kappa} \,\rfloor_h) \quad \tilde{\sigma}_{body} = \tilde{\sigma} \sqcup \text{rebind}(fvs(e_{body}), \tilde{\tau}, \tilde{\tau}') \quad (e_{body}, \tilde{\sigma}_{body}, \tilde{\tau}') \; \widetilde{\Downarrow^{\kappa\tau}_{evalc}} \; (\tilde{\nu}', \tilde{\sigma}') \quad (h, \tilde{\tau}, l, \tilde{\nu}', \tilde{\sigma}', (\tilde{\tau}_k, \tilde{\tau}_{m\kappa})) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}'', \tilde{\sigma}'')}{(h(e_{body})^l, \tilde{\sigma}, (\tilde{\tau}_k, \tilde{\tau}_{m\kappa})) \; \widetilde{\Downarrow^{\kappa\tau}_{evalc}} \; (\tilde{\nu}'', \tilde{\sigma}'')}$$

ABSR-EVAL-APP-KONT

$$\frac{\tilde{\tau} = (\tilde{\tau}_k, \tilde{\tau}_{m\kappa}) \quad \tilde{\nu} \in \tilde{\mathcal{A}}(æ, \tilde{\tau}, \tilde{\sigma}) \quad (h, \tilde{\tau}_h, \_, \tilde{a}_\kappa) \in \tilde{\mathcal{A}}(f, \tilde{\tau}, \tilde{\sigma}) \quad (\tilde{a}_\kappa, \tilde{\nu}, \tilde{\sigma}, \lfloor\, (l, \lfloor\, l : \tilde{\tau}_k \,\rfloor_m) : \tilde{\tau}_{m\kappa} \,\rfloor_h) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (\tilde{\nu}', \tilde{\sigma}') \quad (h, \tilde{\tau}_h, l, \tilde{\nu}', \tilde{\sigma}', (\lfloor\, l : \tilde{\tau}_k \,\rfloor_m, \tilde{\tau}_{m\kappa})) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}'', \tilde{\sigma}'')}{(f(æ)^l, \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{evalc}} \; (\tilde{\nu}'', \tilde{\sigma}'')}$$

ABSR-HANDLE-OP

$$\frac{h(op) = (\overline{x}, e_{op}) \quad \tilde{a}_{vk} = (resume, \tilde{\tau}) \quad \overline{\tilde{\boldsymbol{a}}'_{\boldsymbol{v}} = (\boldsymbol{x}, \tilde{\tau})} \quad \tilde{\sigma}_r = \tilde{\sigma} \sqcup [\tilde{a}_{vk} \mapsto \{(h, \tilde{\tau}_h, l, \tilde{a}_\kappa)\}] \quad \tilde{\sigma}_{xs} = \tilde{\sigma}_r \sqcup [\tilde{\boldsymbol{a}}'_{\boldsymbol{v}} \mapsto \tilde{\sigma}(\tilde{\boldsymbol{a}}_{\boldsymbol{v}})] \quad \tilde{\sigma}_{op} = \tilde{\sigma}_{xs} \sqcup \text{rebind}(fvs(e_{op}) \setminus \{resume, \ldots \overline{x}\}, \tilde{\tau}_h, \tilde{\tau}) \quad (e_{op}, \tilde{\sigma}_{op}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{eval}} \; (\tilde{\nu}, \tilde{\sigma}')}{(h, \tilde{\tau}_h, l, ((op, \overline{\tilde{a}}_v, (\tilde{\psi}_\kappa, \tilde{\tau}_\kappa)), \tilde{\psi}'_\tau, \tilde{a}_\kappa), \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}, \tilde{\sigma}')}$$

ABSR-APPLY-RESTORE

$$\frac{(clo, (\tilde{\tau}_k, \_)) = \tilde{a}_\kappa \quad \tilde{a}'_\kappa \in \tilde{\sigma}(\tilde{a}_\kappa) \quad (\tilde{a}'_\kappa, d, \tilde{\sigma}, \tilde{\tau}_{m\kappa}) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (\tilde{\nu}', \tilde{\sigma}') \quad (clo, \tilde{\nu}', \tilde{\sigma}', (\tilde{\tau}_k, \tilde{\tau}_{m\kappa})) \; \widetilde{\Rightarrow^{\kappa\tau}_{cont}} \; (\tilde{\nu}'', \tilde{\sigma}'')}{(\tilde{a}_\kappa, d, \tilde{\sigma}, \tilde{\tau}_{m\kappa}) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (\tilde{\nu}'', \tilde{\sigma}'')}$$

ABSR-HANDLE-RETURN

$$\frac{h(\text{return}) = (x_{ret}, e_{ret}) \quad \tilde{\sigma}_d = \tilde{\sigma} \sqcup [\tilde{a}_v \mapsto d] \quad \tilde{a}_v = (x_{ret}, \tilde{\tau}) \quad \tilde{\sigma}_{ret} = \tilde{\sigma}_d \sqcup \text{rebind}(fvs(e_{ret}) \setminus \{x_{ret}\}, \tilde{\tau}_h, \tilde{\tau}) \quad (e_{ret}, \tilde{\sigma}_{ret}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{eval}} \; (\tilde{\nu}, \tilde{\sigma}')}{(h, \tilde{\tau}_h, l, d, \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}, \tilde{\sigma}')}$$

ABSR-APPLY-CONTINUE

$$\frac{(l, \tilde{\tau}) = \tilde{a}_\kappa \quad \tilde{a}_{\kappa end} \in \tilde{\sigma}(\tilde{a}_\kappa)}{(\tilde{a}_\kappa, d, \tilde{\sigma}, \tilde{\tau}_{m\kappa}) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (d, \tilde{\sigma})}$$

ABSR-HANDLE-CAPTURE-OP

$$\frac{op \notin h \quad o = (op, \overline{\tilde{a}}_v, \tilde{\psi}_{\tau\kappa}) \quad \tilde{\nu} = (o, \tilde{\psi}_\tau, \tilde{a}_\kappa) \quad \tilde{\psi}_\tau = ((h, \tilde{\tau}_h, l), \tilde{\tau}) \quad \tilde{a}_\kappa = (\tilde{\psi}'_\tau, o) \quad \tilde{\sigma}' = \tilde{\sigma} \sqcup [\tilde{a}_\kappa \mapsto \{\tilde{a}'_\kappa\}]}{(h, \tilde{\tau}_h, l, (o, \tilde{\psi}'_\tau, \tilde{a}_\kappa), \tilde{\sigma}, \tilde{\tau}) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}, \tilde{\sigma}')}$$

ABSR-APPLY-RESTORE-HANDLE

$$\frac{((h, \tilde{\tau}_h, l), (\tilde{\tau}_\kappa, \_)) = \tilde{a}_\kappa \quad \tilde{a}'_\kappa \in \tilde{\sigma}(\tilde{a}_\kappa) \quad (\tilde{a}'_\kappa, d, \tilde{\sigma}, \lfloor\, (l, \tilde{\tau}_\kappa) : \tilde{\tau}_{m\kappa} \,\rfloor_h) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (\tilde{\nu}', \tilde{\sigma}') \quad (h, \tilde{\tau}_h, l, \tilde{\nu}', \tilde{\sigma}', (\tilde{\tau}_k, \tilde{\tau}_{m\kappa})) \; \widetilde{\Downarrow^{\kappa\tau}_{\text{handle}}} \; (\tilde{\nu}'', \tilde{\sigma}'')}{(\tilde{a}_\kappa, d, \tilde{\sigma}, \tilde{\tau}_{m\kappa}) \; \widetilde{\Downarrow^{\kappa\tau}_{apply}} \; (\tilde{\nu}'', \tilde{\sigma}'')}$$

Figure. 19. Abstract Big-Step Rebinding Handler Rules

Our implementation also applies standard *store widening* [Might et al. 2010; Van Horn and Might 2010], replacing the threaded store with a monotonically growing global store. This sacrifices precision (every lookup sees the latest store) but is essential for tractability: without it, distinct store snapshots multiply configurations. The bounds below assume widening.

4.2.1 Complexity. We now characterize the computational complexity of the rebinding analysis. Let $n = |\mathsf{Expr}|$ be the number of program expressions and $c = |\mathsf{Call}|$ the number of call-site and handler labels appearing in timestamps ($c \leq n$). The timestamp space is $|\mathsf{Time}| = |\mathsf{KTime}| \times |\mathsf{MKTime}| = c^m \cdot c^{h(m+1)} = c^T$ where $T = m + mh + h$.

Table. 1. A subset of state space component sizes (upper) and explorable relation spaces (lower, input × output domain) for the rebinding analysis at $(m, h) \in \{(0,0), (1,0), (1,1), (2,2)\}$, where $T = m + mh + h$

Component	General	$(0,0)$	$(1,0)$	$(1,1)$	$(2,2)$
$\widehat{\mathsf{Time}}$	c^T	1	c	c^3	c^8
$\widehat{\mathsf{VAddr}}$	nc^T	n	nc	nc^3	nc^8
$\widehat{\mathsf{TFrame}}$	nc^{2T}	n	nc^2	nc^6	nc^{16}
$\widehat{\mathsf{OpInfo}}$	c^{T+1}	c	c^2	c^4	c^9
$\widehat{\mathsf{KAddr}}$	nc^{3T+1}	nc	nc^4	nc^{10}	nc^{25}
$\widehat{\mathsf{Denotable}}$	n^2c^{4T+1}	n^2c	n^2c^5	n^2c^{13}	n^2c^{33}
$\widehat{\mathsf{Value}}$	n^2c^{6T+2}	n^2c^2	n^2c^8	n^2c^{20}	n^2c^{50}
$\widehat{\mathsf{VAddr}}$ store	n^3c^{5T+1}	n^3c	n^3c^6	n^3c^{16}	n^3c^{41}
$\widehat{\mathsf{KAddr}}$ store	n^2c^{6T+2}	n^2c^2	n^2c^8	n^2c^{20}	n^2c^{50}
$\Downarrow^{\widehat{\kappa\tau}}_{apply}$	$n^5c^{14T+4-m}$	n^5c^4	n^5c^{17}	n^5c^{45}	n^5c^{114}
$\Downarrow^{\widehat{\kappa\tau}}_{handle}$	n^5c^{14T+4}	n^5c^4	n^5c^{18}	n^5c^{46}	n^5c^{116}
$\Rightarrow^{\widehat{\kappa\tau}}_{cont}$	n^5c^{15T+4}	n^5c^4	n^5c^{19}	n^5c^{49}	n^5c^{124}

$\widehat{\mathsf{OpInfo}}$ is compact: operation name and argument variables are syntactically fixed per call site and rebinding ensures all addresses at a call site share a timestamp, giving $|\widehat{\mathsf{OpInfo}}| = c^{T+1}$ independent of arity.

The explorable space of a relation is its input × output domain (Table 1). The dominant term is $\widehat{\Rightarrow^{\kappa\tau}_{cont}}$ at n^5c^{15T+4}, which is polynomial in n for fixed (m, h) but still large. In practice we expect these spaces to be sparsely populated: most input combinations are unreachable, and much of the state space that is reached serves to rule out spurious control-flow paths rather than enumerate them. This makes the analysis tractable for small yet complex programs at small (m, h), which our evaluation on small Koka benchmarks confirms (Section 5). Evaluating how it scales to larger programs remains future work.

Without rebinding, closures carry environments $\widehat{\mathsf{Env}} = \mathsf{Var} \to \widehat{\mathsf{OpInfo}}$, making the state space exponential in the number of free variables: $|\widehat{\mathsf{Env}}| = (n \cdot c^T)^{|fvs|}$. Rebinding collapses environments to timestamps: c^T making the analysis polynomial for fixed m and h.

5 Evaluation

We evaluate our two-component analysis on a suite of Koka programs to answer three questions:

(1) **Effectiveness**: Does our handler-sensitive timestamp (h, m) discover more precise control-flow information than standard k-CFA?
(2) **Efficiency**: What is the cost of this precision in terms of analysis time and state-space size?
(3) **Robustness**: How consistent are these improvements across different benchmark categories?

Table. 2. Continuation precision (singleton fraction) by category and selected benchmarks. H(h, m) is HMCFA; k-CFA is standard last k call-site sensitivity. Timeouts use a 10-minute budget. Full per-benchmark results appear in the supplementary material.

	0-CFA	**1-kCFA**	**2-kCFA**	**H**(1,0)	**H**(1,1)	**H**(1,2)
Category averages						
Koka-Gen (N=27)	60.5%	83.4%	86.1%	73.9%	85.9%	**87.8%**
Koka-Samples (N=24)	84.6%	93.1%	**93.4%**	93.1%	**93.4%**	**93.4%**
Micro-Suite (N=50)	94.3%	95.7%	95.9%	94.3%	95.9%	**97.3%**
Selected benchmarks						
`build/mymakefile-example3`	75.0%	75.0%	75.0%	**100.0%**	**100.0%**	**100.0%**
`interp`$_2$`/err3`	15.6%	93.8%	93.8%	53.1%	**100.0%**	**100.0%**
`scoped/example4`	51.9%	92.3%	**100.0%**	92.3%	**100.0%**	**100.0%**
`complex-flow/complex-layers`	80.0%	80.0%	**88.0%**	80.0%	**88.0%**	**88.0%**
Timeouts (/101)	0	0	4	0	0	0

5.1 Benchmark Suite and Methodology

We evaluate on **101 benchmarks** in three categories:

- **Micro-Suite** ($N = 50$): ~5–20 line programs specifically designed to exercise the precision challenges from Section 1.2 (delimiter forgetfulness, operator forgetfulness, and continuation fragmentation) as well as other patterns: operations from different handlers invoked in different orders or within other operations, different resumption patterns (single-shot, multi-shot, discard), tail and non-tail calls to functions, tail and non-tail resumptions of continuations, and recursive operation invocations.
- **Koka-Samples** ($N = 24$): Standard existing Koka examples including game trees, iterators, parsers, ambient state, and scoped effects (~50–100 LOC), as well as a few samples from Rosetta-Code that use handlers.
- **Koka-Gen** ($N = 27$): Larger programs (~100–300 LOC) including lambda calculus interpreters, incremental build systems with dependency tracking, probabilistic programming, μKanren relational programming, and cooperative schedulers with channel communication.

Configurations. We compare the following analyses, all using a global abstract store:

- **0-CFA**: The flow-insensitive baseline ($h = m = 0$)[1].
- **H**(h, m): The rebinding semantics with handler sensitivity h and call-site sensitivity m, varying $h,\ m\ \in \{0, 1, 2\}$.
- **k-CFA** ($k\ \in \{1, 2\}$): The non-rebinding abstract semantics but with a standard single-component timestamp threaded through evaluation as in classical k-CFA. This has a larger state space without rebinding so naively one might expect it to take much longer but be more precise.

Metrics. To compare across analyses with different address spaces, we first project each analysis result to the 0-CFA address space by joining all values whose addresses share the same 0-CFA address (i.e., the same variable or frame, ignoring timestamps and environments). We then measure the *singleton fraction*: the proportion of these projected addresses whose value set has a unique top-level structure — the same lambda (regardless of environment/timestamp), the same constructor (regardless of field values), or a precise literal. We call such an address *precise* and report the

[1]Due to the non-ANF representation of Koka's Core IR we have a slightly different address space for intermediate values not bound to a variable, these are rebound at constructors in the baseline (0-CFA times out on 3 examples otherwise), but we cannot use rebinding for k-CFA due to definite precision loss when applying constructors to parameters from recursive calls. The precision comparison is based on the 0-CFA address space that the analyses share in common.

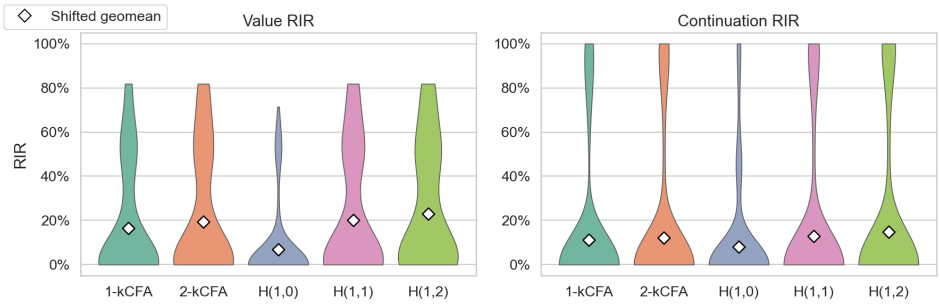


Figure. 20. Distribution of RIR values for $H(1,1)$ across all benchmarks. The shifted geometric mean reflects the overall mass distribution.

singleton fraction separately for continuation addresses (*continuation precision*) and value addresses (*value precision*). A precise continuation address means the analysis identifies a unique next frame at that program point, but the frame's environment (timestamp) may still be imprecise. Consequently, value precision can benefit from increased context sensitivity even when continuation precision is already high. To compare across benchmarks with different baseline difficulties, we use *Relative Imprecision Recovery* (RIR):

$$\mathrm{RIR} = \frac{\mathrm{Prec}_{\mathrm{new}} - \mathrm{Prec}_{0\text{-CFA}}}{1 - \mathrm{Prec}_{0\text{-CFA}}} \tag{1}$$

which measures what fraction of 0-CFA's remaining imprecision is resolved by a given configuration. We aggregate RIR across benchmarks using the *shifted geometric mean* $\exp(\frac{1}{N}\sum \ln(1+x)) - 1$, which handles zero values without destroying the average. For cost we measure analysis time and total abstract states (the number of entries across all analysis relations and the abstract store).

Table 2 summarizes continuation precision across configurations and benchmark categories. Figure 20 shows the distribution of RIR values across all benchmarks. The shifted geometric mean handles zero-RIR benchmarks without collapsing the aggregate, and it reflects the overall mass of the distribution rather than being skewed by outliers.

5.2 Precision Results

Figure 21 shows RIR by category, measuring the fraction of 0-CFA's imprecision resolved by each configuration. Table 2 provides the underlying absolute precision numbers.

Continuation precision. The headline result is that $H(1,1)$ closely matches 2-kCFA continuation precision while completing all 101 benchmarks (2-kCFA times out on 4). On Koka-Gen — the hardest category — $H(1,1)$ achieves 86% precision, matching 2-kCFA and well above 83% for 1-kCFA (Table 2). With higher parameters, $H(1,2)$ reaches 88%. The gains are visible in the RIR breakdown by category (Figure 21, right), where $H(1,1)$ and $H(1,2)$ consistently lead.

On Koka-Samples, 0-CFA already achieves 85% continuation precision and the others plateau at 93%. On the Micro-Suite, continuation precision is already 94% under 0-CFA, so continuation RIR is near zero for all configurations (Figure 21). The strong 0-CFA baseline across all categories reflects the continuation cohesion provided by the $\widehat{\mathsf{OpInfo}}$-augmented address space (Section 4).

Value precision. $H(1,1)$ also improves value precision over k-CFA (Figure 21, left). The gains are consistent across categories, with $H(1,1)$ outperforming 1-kCFA on Koka-Gen, Koka-Samples, and Micro-Suite. On the Micro-Suite, although continuation precision is already high under 0-CFA, the environments captured in those continuations and the store itself can still be refined — value precision is the interesting dimension. RIR measures improvement relative to 0-CFA, so it cannot

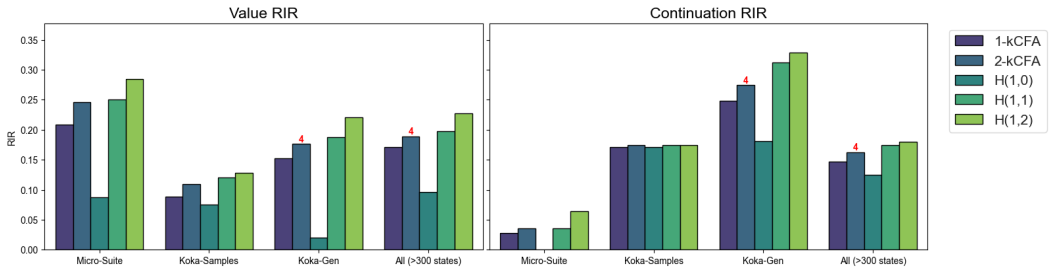


Figure. 21. Relative Imprecision Recovery (shifted geometric mean) by category. The "All" aggregate includes only non-trivial benchmarks (>300 0-CFA states). Red numbers indicate timeouts excluded from the geometric mean. H(h, m) denotes our analysis; k-CFA denotes standard call-site–only timestamps.

capture precision gains that are incomparable across analysis kinds. The store may become more precise in ways that do not change whether a projected address is precise. The RIR improvements therefore represent a lower bound on the actual precision benefit of context sensitivity.

Per-benchmark highlights. The selected benchmarks in Table 2 illustrate where the two components interact. On `build/mymakefile-example3`, 0-CFA and 1-kCFA both achieve 75%, while H(1, 0) and H(1, 1) reach 100% — showing the importance of handler sensitivity. On `interp`$_2$`/err3`, 1-kCFA improves 0-CFA from 16% to 94%, and H(1, 1) closes the remaining gap to 100%. On `scoped/example4`, 1-kCFA achieves 92% while both 2-kCFA and H(1, 1) reach 100%. On `complex-flow/complex-layers`, 0-CFA and 1-kCFA stall at 80%, while 2-kCFA, H(1, 1), and H(1, 2) reach 88% — handler sensitivity alone at H(1, 0) is insufficient; both components together are needed. Full per-benchmark results appear in the supplementary material (Tables A1–A6).

5.3 Parameter Sensitivity

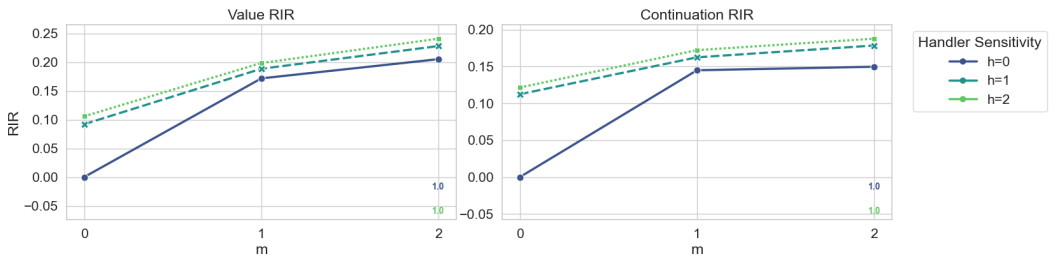


Figure. 22. RIR as h and m vary over non-trivial benchmarks (>300 0-CFA states) that complete in all nine H(h, m) configurations. Each line fixes h (handler sensitivity) and sweeps m (call-site sensitivity).

Figure 22 shows how precision varies with the two timestamp components. For continuation precision, **both components contribute, with m providing the largest single-step gain**: going from H(0, 0) to H(0, 1) by adding call-site sensitivity provides the biggest jump, while adding handler sensitivity ($h = 0 \rightarrow h = 1$ at $m = 0$) provides a complementary boost. The combination H(1, 1) captures most of the achievable precision, with H(2, 2) providing only marginal further improvement. For value precision, **both components contribute meaningfully**. At $m = 0$, handler sensitivity alone ($h \geq 1$) provides approximately 10–12% RIR, while at $h = 0$, call-site sensitivity provides the primary gains. The combination H(1, 1) achieves value RIR close to H(2, 2), suggesting that (1, 1) captures most of the benefit at lower cost.

5.4 Cost and Tradeoffs

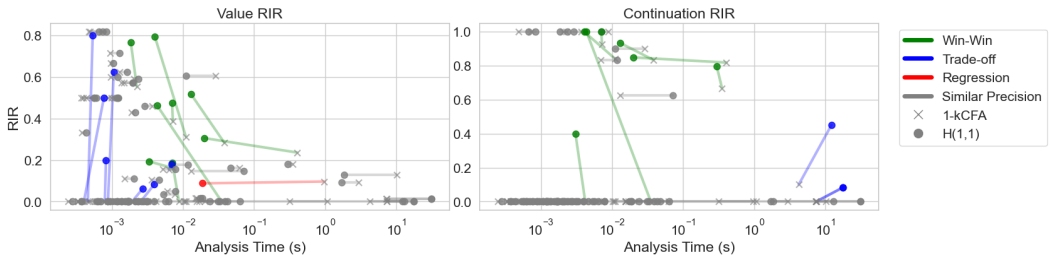


Figure. 23. Precision-cost tradeoff: 1-kCFA (small dot) vs. H(1, 1) (large dot) per benchmark. Green: win-win (better precision, lower cost); blue: tradeoff (better precision, higher cost); red: regression; gray: same precision.

Figure 23 shows the precision-cost tradeoff between H(1, 1) and 1-kCFA for individual benchmarks.

Precision can reduce cost. Several benchmarks appear as green (win-win) points in Figure 23, where H(1, 1) is both more precise *and* faster than 1-kCFA. More precise control flow avoids exploring spurious continuation paths, reducing the reachable state space. Additionally, there are many blue points, reflecting a small cost overhead for increased precision. The figure shows that H(1, 1) dominates 1-kCFA in precision for almost every program, and the one example it does worse on it reduces the analysis cost by almost two orders of magnitude — suggesting a higher h or m might be needed for a particular place where imprecision is causing extra work.

Tractability on benchmarks. H(1, 1) solves all 101 benchmarks within the 10-minute budget, matching 1-kCFA. By contrast, 2-kCFA times out on 4 benchmarks (incremental `build` programs with deep dependency chains). Within this benchmark suite, (1, 1) represents a practical sweet spot: it captures the main precision benefits of 2-kCFA without the timeouts.

Koka-Gen efficiency. In the hardest category, H(1, 1) achieves 86% continuation precision versus 83% for 1-kCFA (Table 2). Several Koka-Gen benchmarks that time out under 2-kCFA (e.g., `build/mymakefile-example`$_1$, `build/mymakefile-example4`) complete under H(1, 1), because the two-component design resolves ambiguity that k-CFA sensitivity cannot, preventing state explosion from repeated exploration of indistinguishable continuation paths.

5.5 Threats to Validity

Implementation-formalization gap. Our implementation is in Haskell on a more complex non-ANF representation, while the formalism uses ANF. While the implementation differs (e.g., number and type of frames and addresses for intermediate non-named values), it follows the timestamping approach exactly. We tested the Micro-Suite benchmarks on a second implementation using a simplified ANF language to validate correctness. Primitive lattice operations and handling of local state are bespoke but applied identically across all analyses.

AI-generated code bias. The Koka-Gen set is AI-generated, which may not reflect human coding patterns. We mitigate this by including hand-written Koka standard library examples, requesting diverse effect handler usage patterns, and manually reviewing generated code for correctness.

Parameter selection. Our choice of $h,\ m \in \{0, 1, 2\}$ is sufficient to achieve concrete precision on many smaller examples but may be inadequate for programs with deeper nesting. Future work should evaluate parameter sensitivity on larger codebases.

Precision metrics. Singleton ratio is a standard precision metric but may not directly correlate with optimization potential. We focus on it because it is objective and interpretable, but acknowledge that downstream impact is the ultimate measure of analysis effectiveness.

6 Related Work

6.1 Context Sensitivity and Pushdown Precision

Gilray et al. [Gilray et al. 2016] showed that allocation characterizes polyvariance: call-strings, object-sensitivity, and others can be seen as different timestamp and allocation strategies. Our work follows this philosophy, designing an allocation strategy for effect handlers.

For the lambda calculus, P4F [Gilray et al. 2016] showed that a simple continuation address space suffices for pushdown precision in the regular lambda calculus (precisely matching calls and returns), and m-CFA was later shown to achieve the same precision by a different route [Germane 2023]. Our approach draws on both strategies: continuation addresses use P4F-style addresses extended with operation context (for continuation cohesion), and timestamps respect stack adjustments. Our high empirical continuation precision suggests we may be close to pushdown-precise, but we leave formal characterization and improvements to future work. The closely related question of CFL-reachability decidability for higher-order effectful programs has seen recent progress which could inform future work [Dal Lago and Ghyselen 2024; Endo and Terauchi 2026; Kobayashi 2025].

6.2 Analyzing First-Class Control

AAC [Glaze and Van Horn 2014] directly analyzes shift/reset but includes stores in the continuation address, causing recursion. AAC breaks this recursion by approximating stores at capture points, but the resulting analysis has super-exponential complexity. Our big-step approach sidesteps this by keeping continuations implicit in the derivation structure and using P4F style addresses.

Vardoulakis and Shivers [Vardoulakis and Shivers 2011] analyze first-class control via summarization in CPS-translated programs. Their local semantics keeps local variables precise, only losing precision when a continuation variable is captured under a lambda. In CPS, continuations are created eagerly, whereas in our direct-style semantics continuation variables are bound late, potentially avoiding some precision losses from early commitment to continuation allocation.

6.3 Semantics for Effect Handlers

Plotkin and Pretnar [Plotkin and Pretnar 2009] introduced algebraic effect handlers. Bauer and Pretnar [Bauer and Pretnar 2014] gave a substitution-based big-step semantics, against which we prove equivalence. In the same paper they give small step operational semantics, and later give denotational semantics: [Bauer and Pretnar 2015]. Hillerström et al. developed CPS translations for handlers [Hillerström et al. 2017], and Xie et al give an evidence passing semantics for Koka's effect handlers in terms of an underlying multi-prompt delimited control monad [Xie and Leijen 2021].

These different semantic formulations present different tradeoffs for static analysis. Substitution-based big-step semantics [Bauer and Pretnar 2014] resist finite abstraction due to unbounded term growth. CPS translations enable reuse of existing analyses but lose structural information about delimiters and express results in CPS terms. Other translations rely on semantics for multi-prompt delimited control which are not supported by existing analyses. Small-step reduction semantics are not commonly used as implementation strategies for effect handlers, and as AAC demonstrates, precisely abstracting evaluation contexts eagerly can be costly. Our big-step formulation preserves direct-style reasoning while making all recursive components (environments, stores, continuations) amenable to standard abstraction and optimization strategies via ADI [Darais et al. 2017].

Forster et al. [Forster et al. 2019] established expressiveness equivalences between effect handlers, monadic reflection, and delimited control. Our techniques extend naturally to other forms of higher order control including multi-prompt delimited control operations.

6.4 Effect Handler Implementations

Effect handler implementations span a wide design space: evidence-passing with continuation monads [Xie and Leijen 2021], capability-passing compilation [Brachthäuser and Schuster 2017; Schuster et al. 2020], CPS via generalised continuations [Hillerström et al. 2020], runtime fibers [Sivaramakrishnan et al. 2021], stack copying [Leijen 2018; Pham and Odersky 2024], stack switching [Muhcu et al. 2025; Phipps-Costin et al. 2023], and delimited control [Moreno and Pham 2015]. Control flow information from our analysis — such as which continuations are actually captured and resumed and how many times — could inform implementation tradeoffs across this design space per program.

7 Future Work

Our evaluation measures analysis precision in isolation, but the information it produces can enable compiler optimizations. Koka's existing static passes are local: they turn dynamic handler dispatch into evidence vector lookup and optimize operations based on local and global operator restrictions. Our analysis supplies full-program flow information such as which handler handles a given operation, whether a continuation is ever captured, and how many times it is resumed. This enables opportunities existing optimizations cannot derive: replacing evidence lookup with direct calls when only one handler is reachable, dead continuation elimination, and fusing frames that always execute together. We have a prototype that uses our CFA output to inline continuation captures and eliminate join points, confirming the analysis produces actionable information.

Our big-step approach also extends naturally to related control operators: the two-component timestamp and store-allocated continuation segments transfer to any stack-based abstract machine that can lazily build captured stacks, including shift/reset and multi-prompt delimited control.

8 Conclusion

Effect handlers pose three challenges for control flow analysis: delimiter forgetfulness (losing delimited context), operator forgetfulness (conflating operation invocations and their captured continuations), and continuation fragmentation (mixing continuation frames). Our continuation address space addresses fragmentation for any context-sensitivity, but standard k-CFA timestamps cannot meet the first two challenges. Our key insight is that a two-component timestamp — providing handler and operator sensitivity — captures the remaining structure.

We developed this insight in three stages: (1) a big-step semantics for effect handlers using environments and stores (proved equivalent to B&P's substitution-based semantics), (2) a timestamped semantics with fresh allocation, and (3) a sound finite abstraction. On 101 Koka benchmarks, $\mathsf{H}(1,1)$ closely matches 2-kCFA precision while completing all benchmarks (2-kCFA times out on 4), often at both higher precision and lower cost by avoiding spurious path exploration. $\mathsf{H}(1,1)$ appears to be a robust default; deeply nested multi-shot handlers may need larger h. HMCFA is polynomial and thus more practical than other context-sensitivities, though whether it scales to even larger programs remains an open question.

Acknowledgments

AI assisted in refining presentation and implementing proofs in Lean 4 under the authors' direction; the formalism, implementation, and evaluation are the authors' own work.

References

Baelde, and Schmitt. Dec. 2025. Introduction à La Théorie Des Langages de Programmation. https://people.irisa.fr/David.Baelde/prog/prog.pdf. Course notes, L3 SIF, ENS Rennes & Université Rennes.

Bauer, and Pretnar. Dec. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science* Volume 10, Issue 4 (December): 1153. doi:https://doi.org/10.2168/LMCS-10(4:9)2014.

Bauer, and Pretnar. 2015. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming* 84 (1). Elsevier: 108–123. doi:https://doi.org/10.1016/j.jlamp.2014.02.001.

Brachthäuser, and Schuster. Oct. 2017. Effekt: Extensible Algebraic Effects in Scala (short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, 67–72. ACM, Vancouver BC Canada. doi:https://doi.org/10.1145/3136000.3136007.

Dal Lago, and Ghyselen. Jan. 2024. On Model-Checking Higher-Order Effectful Programs. *Proc. ACM Program. Lang.* 8 (POPL). Association for Computing Machinery, New York, NY, USA. doi:https://doi.org/10.1145/3632929.

Darais, Labich, Nguyen, and Van Horn. 2017. Abstracting Definitional Interpreters (Functional Pearl). *Proceedings of the ACM on Programming Languages* 1 (ICFP). ACM New York, NY, USA: 1–25. doi:https://doi.org/10.1145/3110256.

Endo, and Terauchi. 2026. Reachability Is Decidable for ATM-Typable Finitary PCF with Effect Handlers. In *Programming Languages and Systems*, edited by Alex Potanin, 67–87. Springer Nature Singapore, Singapore. doi:https://doi.org/10.1007/978-981-95-3585-9_4.

Flanagan, Sabry, Duba, and Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 237–247. doi:https://doi.org/10.1145/155090.155113.

Forster, Kammar, Lindley, and Pretnar. 2019. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Journal of Functional Programming* 29. Cambridge University Press: e15. doi:https://doi.org/10.1017/S0956796819000121.

Germane. 2023. M-CFA Exhibits Perfect Stack Precision. In *Programming Languages and Systems*, edited by Chung-Kil Hur, 14405:290–309. Springer Nature Singapore, Singapore. doi:https://doi.org/10.1007/978-981-99-8311-7_14. Series Title: Lecture Notes in Computer Science.

Gilray, Adams, and Might. 2016. Allocation Characterizes Polyvariance: A Unified Methodology for Polyvariant Control-Flow Analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 407–420. doi:https://doi.org/10.1145/2951913.2951936.

Gilray, Lyde, Adams, Might, and Van Horn. 2016. Pushdown Control-Flow Analysis for Free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 691–704. doi:https://doi.org/10.1145/2837614.2837631.

Glaze, and Van Horn. 2014. Abstracting Abstract Control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, 11–22. doi:https://doi.org/10.1145/2661088.2661098.

Hillerström, Lindley, and Atkey. 2020. Effect Handlers via Generalised Continuations. *Journal of Functional Programming* 30: e5. doi:https://doi.org/10.1017/S0956796820000040.

Hillerström, Lindley, Atkey, and Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. *LIPIcs, Volume 84, FSCD 2017* 84. Schloss Dagstuhl – Leibniz-Zentrum für Informatik: 18:1–18:19. doi:https://doi.org/10.4230/LIPICS.FSCD.2017.18. Artwork Size: 19 pages, 774356 bytes ISBN: 9783959770477 Medium: application/pdf.

Kobayashi. Jan. 2025. On Decidable and Undecidable Extensions of Simply Typed Lambda Calculus. *Proc. ACM Program. Lang.* 9 (POPL). Association for Computing Machinery, New York, NY, USA. doi:https://doi.org/10.1145/3704875.

Leijen. 2018. *Algebraic Effect Handlers with Resources and Deep Finalization*. Technical Report MSR-TR-2018-10. Microsoft Research.

Might, Smaragdakis, and Van Horn. 2010. Resolving and Exploiting the K-CFA Paradox: Illuminating Functional vs. Object-Oriented Program Analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 305–315. doi:https://doi.org/10.1145/1806596.1806631.

Moreno, and Pham. Jan. 2015. Lexical Delimited Continuations for Scala 3, January.

Muhcu, Schuster, Steuwer, and Brachthäuser. Aug. 2025. Multiple Resumptions and Local Mutable State, Directly. *Proceedings of the ACM on Programming Languages* 9 (ICFP): 704–733. doi:https://doi.org/10.1145/3747529.

Pham, and Odersky. Sep. 2024. Stack-Copying Delimited Continuations for Scala Native. In *Proceedings of the 19th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems*, 2–13. ACM, Vienna Austria. doi:https://doi.org/10.1145/3679005.3685979.

Phipps-Costin, Rossberg, Guha, Leijen, Hillerström, Sivaramakrishnan, Pretnar, and Lindley. Oct. 2023. Continuing WebAssembly with Effect Handlers. *Proceedings of the ACM on Programming Languages* 7 (OOPSLA2): 460–485. doi:https://doi.org/10.1145/3622814.

Plotkin, and Pretnar. 2009. Handlers of Algebraic Effects. In *European Symposium on Programming*, 80–94. Springer. doi:https://doi.org/10.1007/978-3-642-00590-9_7.

Schuster, Brachthäuser, and Ostermann. Aug. 2020. Compiling Effect Handlers in Capability-Passing Style. *Proceedings of the ACM on Programming Languages* 4 (ICFP): 1–28. doi:https://doi.org/10.1145/3408975.

Sivaramakrishnan, Dolan, White, Kelly, Jaffer, and Madhavapeddy. Jun. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 206–221. ACM, Virtual Canada. doi:https://doi.org/10.1145/3453483.3454039.

Van Horn, and Might. 2010. Abstracting Abstract Machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, 51–62. doi:https://doi.org/10.1145/1863543.1863553.

Vardoulakis, and Shivers. 2011. Pushdown Flow Analysis of First-Class Control. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, 69–80. ICFP '11. Association for Computing Machinery, New York, NY, USA. doi:https://doi.org/10.1145/2034773.2034785.

Xie, and Leijen. 2021. Generalized Evidence Passing for Effect Handlers: Efficient Compilation of Effect Handlers to C. *Proceedings of the ACM on Programming Languages* 5 (ICFP). ACM New York, NY, USA: 1–30. doi:https://doi.org/10.1145/3473576.

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