Call-Guarded Abstract Definitional Interpreters

KIMBALL GERMANE, Brigham Young University, USA

Over the last 15 years, several popular *systematic abstraction frameworks* have emerged—frameworks that allow a static analysis to be derived by systematically transforming a concrete semantics. These frameworks guarantee computability of the resulting artifact by the application of an *a priori* abstraction which induces a particular finitization in the execution space. While effective, this abstraction occurs without regard for program structure, subjecting each program point to the same fixed degree of context sensitivity. In this paper, we present CGADI, an enhancement to systematic abstraction frameworks based on definitional interpreters which defers abstraction until a parameterized safety property signals that it should be applied. We then examine this enhanced framework instantiated with two such safety properties: a simple reentrancy property which detects non-recursive portions of program execution, and a size change property which detects evaluation paths destined to converge by virtue of appropriately decreasing values along them. The result is that CGADI can operate in the fully-precise concrete space for portions of execution without forfeiting computability. Our evaluation demonstrates that CGADI is able to produce a higher number of precise results than a corresponding CFA at relatively low cost and that, with no special treatment, CGADI can handle many programming patterns targeted by specific analysis techniques.

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

Additional Key Words and Phrases: abstract interpreters, higher-order languages, size-change termination

ACM Reference Format:
Kimball Germane. 2025. Call-Guarded Abstract Definitional Interpreters. *Proc. ACM Program. Lang.* 9, ICFP, Article 270 (August 2025), 33 pages. https://doi.org/10.1145/3747539

1 Introduction

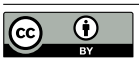
The *Abstracting Abstract Machines* (AAM) framework [32] provides a recipe to systematically transform a small-step abstract machine-based semantics into a sound and computable static analysis. Its spiritual successor, the *Abstracting Definitional Interpreters* (ADI) framework [4], improves on AAM in two ways:

(1) It allows the semantics to be expressed via a definitional interpreter [29] which provides a more natural setting for many implementations.
(2) It models the run-time stack using the stack of a pushdown automaton [33] which offers superior control precision.

Subsequent work has enhanced ADI with compositional soundness facilities [19], fine-grained caching strategies [18], and meta-level staging to accelerate the analyzer [35].

An essential characteristic of AAM—and inherited by ADI—is the *a priori* coarsening of the execution space by the systematic abstraction. This is not to say that the approach is not effective or versatile: Gilray et al. [10] show that it is able to encode a wide variety of existing and novel context sensitivities and Gilray et al. [11] show how to leverage it to recover pushdown stack precision under some conditions. However, in each of these cases, the fact remains that the execution space is coarsened without any necessary regard for the program under analysis.

Author's Contact Information: Kimball Germane, Brigham Young University, Provo, USA, kimball@cs.byu.edu.



ACM 2475-1421/2025/8-ART270
https://doi.org/10.1145/3747539

Outside of these systematic frameworks, researchers have developed more responsive context sensitivities. For example, both Montagu and Jensen [26] and Keidel et al. [18] offer strategies to move beyond the fixed k of k-CFA [30] until recursion is detected.

In this paper, we present an enhanced definitional interpreter framework that incorporates these and more sophisticated strategies to allow the analysis to respond to the program. Morever, this framework can be cleanly integrated into ADI's. Before describing this framework any more, we examine the problem more closely.

1.1 Too-Static Static Analysis

A flow analysis is a static analysis that seeks a sound account of the values that flow to each variable and expression. For instance, for the program on the right, a sound flow analysis will determine that both `add1` and `sub1` flow to `f1` and both 12 and 10 flow to `x1`. However, absent some way to correlate the arguments, a flow analysis will conclude that `add1` may be applied to 10 and `sub1` to 12, which does not in fact occur.

```
(define (apply1 f1 x1) (f1 x1))
(+ (apply1 add1 12) (apply1 sub1 10))
```

To remove such spurious conclusions, flow analyses use *context* at each control point to distinguish between distinct points in program execution. For example, the prolific k-CFA uses the most-recent-k call sites in execution history up to that point and m-CFA [25] uses the top-m stack frames (each represented as a call site). Each of these contexts endows the flow analysis with a form of call-site sensitivity. Instantiating $k = 1$ or $m = 1$ produces an analysis which is able to determine that one call to `(f1 x1)` is made in the context of the previous (but still active) call to `(apply1 add1 12)` and the other of the call to `(apply1 sub1 10)` and thereby correlate each callee and its argument.

Although the contexts of k-CFA and m-CFA allow their respective analyses to make new distinctions, they do not allow them to do so arbitrarily: each context space is finite (limited in length by k and m respectively) and so are the number of distinct contexts in which any given expression may be evaluated. For any particular k or m, it is straightforward to transform a program which enjoys precision to an extensionally-equivalent program which doesn't—a corollary to the result that non-trivial abstract interpretation (to which CFA may be cast) is necessarily sensitive to intensional program aspects [1]. The mechanism to thwart the context sensitivity depends on the particular kind of context used; a well-known disturber of call-site sensitivity is function indirection.[1] For instance, indirecting the `apply1` operator in the above program, as in the program on the right, confuses the previously-precise $[k = 1]/[m = 1]$-CFA, which now sees two calls to `apply1` but only one to `f2`. Under this transformation, $[k = 1]/[m = 1]$-CFA is as imprecise as 0CFA—a CFA which ignores context altogether—even though it resides in a bigger context and execution space. Increasing the degree of call-site sensitivity—instantiating the CFA at $k = 2$—restores clarity but the flow analysis remains vulnerable to yet another indirection.

```
(define (apply2 f2 x2) f2 x2)
(define (apply1 f1 x1) (apply2 f1 x1))
(+ (apply1 add1 12) (apply1 sub1 10))
```

1.2 More-Dynamic Static Analysis

Having diagnosed the problem as a finite execution space fixed before analysis time, we prescribe an analysis which operates in an infinite and unfixed execution space. To this end, we present CGADI, a definitional interpreter-based static analysis framework which incorporates concrete evaluation, allowing it to roam an infinite execution space. To maintain computability, CGADI is

[1] This argument is easily generalized to other forms of context sensitivity. We keep the discussion grounded in terms of call-site sensitivity until later.

parameterized by a convergence safety property which dictates when concrete evaluation should give way to abstract evaluation. We offer two such properties: one based on reentrancy to detect recursion and one based on size change termination. The effect of incorporating concrete evaluation is that precision is not *a priori* bounded. For instance, an inductive data structure can be built up and torn down with perfect precision using CGADI, as long as it upholds the convergence safety property.

We next walk through some examples to review AAM-style frameworks (§2) and demonstrate CGADI (§3). We then present a core language (§4) and a concrete (§5.1) and abstract (§5.2) semantics over it. We then present CGADI (§5.3) as a hybrid of the concrete and abstract semantics. As CGADI is parameterized by convergence safety property, we present a simple one based on reentrancy (§6) and a more sophisticated one based on size change (§7). We then compare and contrast the behavior of CGADI with a classical flow analysis (§8) and conclude with related and future work (§9).

2 Background: Abstracting Abstract Machines

This section provides background knowledge about the AAM framework. Readers familiar with this framework may skip this section but should acquaint themselves with our example programs and notation.

2.1 A Counter without Bound

The program to the right, written with Scheme syntax, uses the recursive `loop` to increment a count indefinitely. To examine its execution, we can define a CEK machine [8], each state of which consists a **c**ontrol expression, an **e**nvironment, and a **k**ontinuation. (The continuation behavior of this program is trivial, so we'll omit the continuation component. Thus, our machine more closely resembles the CE machine of Flanagan et al. [9].) Omitting trivial argument evaluation, the execution of this program passes through the states on the left.

```
(define (loop n) (loop (+ n 1)))
(loop 42)
```

1	$\langle$`(loop 42)` , $\bot$ $\rangle$
2	$\langle$`(loop (+ n 1))` ,[`n` $\mapsto$ 42]$\rangle$
3	$\langle$`(loop (+ n 1))` ,[`n` $\mapsto$ 43]$\rangle$
4	$\langle$`(loop (+ n 1))` ,[`n` $\mapsto$ 44]$\rangle$
	...

Evaluation begins in the initial execution state 1 consisting of the top-level expression and a base environment (omitting the `loop` binding for clarity). After trivial evaluation of the operator `loop` and argument `42`, `loop` is applied to `42`, execution arrives at state 2. After the addition, the machine applies `loop` to `43` and arrives at state 3. The control expression here is the same as in state 2 so the same evaluation recurs, but in state 3's environment, and execution arrives at state 4. This process continues indefinitely, visiting an infinite sequence of distinct states—an infinite journey through an infinite state space.

2.2 Abstracting the Interpretation

The responsibility of a sound static analysis is to provide a computable account of this program's behavior. The AAM framework offers computability through a systematic process which finitizes the program's execution space. Once the state space is finite, any journey through it—so long as it is careful not to retread ground—will be finite as well. The systematic process AAM prescribes consists of (1) removing direct recursion from the state space definition by redirecting recursive structure through a store component and (2) finitizing the address space, which finitizes the entire execution space downstream. We'll now walk through both of these steps.

2.2.1 Untie the Recursive Knot. Typical semantic state spaces contain recursively-defined structures: environments contain higher-order values which themselves contain environments, continuations contain a frame and another continuation, etc. AAM's first step is to introduce a **s**tore component to the CEK machine, thereby yielding a CESK machine [7], and redirect all recursive structure through it. Afterwards, environments don't map variables to values but to store references (addresses) which themselves locate values within the store; continuations are heap-allocated in a similar way. After applying this step to our CEK machine, execution proceeds through the states on the right. In contrast to the CEK machine, the CESK machine binds x in the environment not directly to a value but to an address x_n and binds each address to its corresponding value in the store. One effect of this binding discipline is that the store accumulates all the bindings made during execution and thus offers a comprehensive account of them. Although execution still proceeds without bound at this point, the machine state definition is poised for the AAM's next step.

1 | $\langle$`(loop 42)` , $\bot$, $\bot$ $\rangle$
2 | $\langle$`(loop (+ n 1))` ,$[\mathtt{x} \mapsto x_0]$,$[x_0 \mapsto 42]$ $\rangle$
3 | $\langle$`(loop (+ n 1))` ,$[\mathtt{x} \mapsto x_1]$,$[x_0 \mapsto 42, x_1 \mapsto 43]$ $\rangle$
4 | $\langle$`(loop (+ n 1))` ,$[\mathtt{x} \mapsto x_2]$,$[x_0 \mapsto 42, x_1 \mapsto 43, x_2 \mapsto 44]\rangle$
... |

2.2.2 Bound the Unbounded. With no more recursively-defined structures, the only remaining source of unboundedness is infinite domains—in our case, the address and integer domains. AAM's second step is to finitize the address domain and abstract the integer domain to prohibit any infinite ascending chains. Finitizing the address domain ensures that the store cannot indefinitely grow horizontally with entries at fresh addresses. Limiting store growth in this way all but ensures that allocation will occur at an already-allocated address, in which case the entry becomes the least upper bound of the previous resident and the new. Prohibiting infinite ascending chains in the domains that may be store-allocated ensures that the store cannot indefinitely grow vertically with new entries at old addresses. Together, these limitations ensure that the store space is finite and, with all recursive structure allocated therein, that the execution space is finite in turn.

To illustrate limiting horizontal growth, we finitize the address space to the set of program variables—a choice which induces 0CFA [28]—but abstract the integers with a powerset domain which *has* infinite ascending chains. Using this finitization, execution visits the states on the left (and more). On each transition, the store accumulates all values bound to n on that path. The result of an access to n then becomes nondeterministic, so that execution is no longer characterized by a trace but a graph. For example, when n is accessed in state 3, the result is nondeterministically 43 *and* 42, so that state 3's successors include both state 4, in the former case, and state 3 itself, in the latter.

1 | $\langle$`(loop 42)` , $\bot$, $\bot$ $\rangle$
2 | $\langle$`(loop (+ n 1))` ,$[\mathtt{n} \mapsto \mathtt{n}]$,$[\mathtt{n} \mapsto \{42\}]$ $\rangle$
3 | $\langle$`(loop (+ n 1))` ,$[\mathtt{n} \mapsto \mathtt{n}]$,$[\mathtt{n} \mapsto \{42, 43\}]$ $\rangle$
4 | $\langle$`(loop (+ n 1))` ,$[\mathtt{n} \mapsto \mathtt{n}]$,$[\mathtt{n} \mapsto \{42, 43, 44\}]\rangle$
... |

To illustrate limiting vertical growth as well, we abstract integers to a flat domain in which the least upper bound of distinct integers is $\top_{int}$. This abstraction induces the set of states to the right. As before, state 1's successor is state 2 and state 2's is state 3. In the transition from state 2 to state 3, the 42 at n is joined with $42 + 1 = 43$, yielding $\top_{int}$. Having reached the top of the store space, state 3's successor is state 3. In contrast to the executions under the previous abstract machines, these states constitute the entire abstract execution space of the program or, put differently, they *are* a 0CFA analysis of the program.

1 | $\langle$`(loop 42)` , $\bot$, $\bot$ $\rangle$
2 | $\langle$`(loop (+ n 1))` ,$[\mathtt{n} \mapsto \mathtt{n}]$,$[\mathtt{n} \mapsto 42]$ $\rangle$
3 | $\langle$`(loop (+ n 1))` ,$[\mathtt{n} \mapsto \mathtt{n}]$,$[\mathtt{n} \mapsto \top_{int}]\rangle$

2.3 Analyzing Factorial

The 0CFA just derived via the AAM recipe is general and can produce analyses of other programs too. Let's apply it to the factorial program to the right. This program is tail-recursive, which keeps continuation behavior trivial, so we once again omit the continuation component from machine states. We also omit the identity-map environments characteristic of 0CFA. Execution visits a state space consisting of the states just below.

```
(define (fact n a) (if (zero? n) a (fact (- n 1) (* n a))))
(fact 5 1)
```

Execution begins in state 1 and transitions to state 2. Because of our use of a flat numeric domain, `n` and `a` exhibit precise values in this state. With a precise value for `n`, the guard is evaluated precisely and state 2's sole successor is state 3. The arguments to the recursive call are computed and the flat numeric domain represents each as $\top_{int}$ in state 4. Now execution is once again at the guard, but `n` and `a` are abstract, so each outcome of the guard must be considered. Consequently, state 4's successors include both states 5a and 5b. State 5a is a terminal state, indicating that the result of the program is bounded by $\top_{int}$ (i.e. the result could be any number). State 5b performs the recursive call and transitions back to state 4.

1	$\langle$`(fact 5 1)`, $\bot$ $\rangle$
2	$\langle$`(zero? n)`, $[\mathtt{n} \mapsto 5, \mathtt{a} \mapsto 1]$ $\rangle$
3	$\langle$`(fact (- n 1) (* n a))`, $[\mathtt{n} \mapsto 5, \mathtt{a} \mapsto 1]$ $\rangle$
4	$\langle$`(zero? n)`, $[\mathtt{n} \mapsto \top_{int}, \mathtt{a} \mapsto \top_{int}]\rangle$
5*a*	$\langle$`a`, $[\mathtt{n} \mapsto \top_{int}, \mathtt{a} \mapsto \top_{int}]\rangle$
5*b*	$\langle$`(fact (- n 1) (* n a))`, $[\mathtt{n} \mapsto \top_{int}, \mathtt{a} \mapsto \top_{int}]\rangle$

3 CGADI: A Finite Journey through Infinite Space

Having seen a few examples of an AAM-induced finite execution space, we can now explain how CGADI is able to finitize its execution in an infinite space by combining concrete and abstract execution.

Classical analysis operates in a single mode—one which we might call *abstract*—in which all evaluation, allocation, and binding operations are carried out abstractly. A CGADI analysis by default operates in *concrete* mode, in which evaluation, allocation, and binding are carried out with complete precision, and, when necessary, shifts to the abstract mode of a classical analysis. The necessity of this shift is determined by a *call guard*, a parameter of the analysis.

The job of the call guard is to ensure that convergence of the current evaluation path is guaranteed. There are a wide variety of approaches a call guard can take to obtain such a guarantee; in this paper, we consider two.

(1) The *reentrancy* call guard permits concrete interpretation on any path that has not reentered an active function, an act which signals recursion in execution (§6). This call guard implements the strategies used by Montagu and Jensen [26] and Keidel et al. [18].
(2) The *size-change* call guard utilizes the size-change termination principle [20] to maintain concrete interpretation on paths that are decreasing on a well-founded metric (§7). This call guard is original to this work.

CGADI eagerly switches to abstract mode without appealing to the call guard when control (e.g. branching and function call) depends on an abstract or nondeterministic element. On the other side of the coin, CGADI proactively restores concrete mode on all non-tail (e.g. argument) evaluation, which allows it to retain concrete precision in the leaves of the computation tree, which can naturally lead to increased precision downstream.

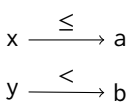
3.1 The Size-Change Principle for Program Termination

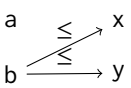
Before we take a look at CGADI in action, we offer a brief primer on the size-change termination (SCT) principle [20] that underlies the size-change call guard.

The SCT principle is based on the observation that any ostensibly-infinite path along which a well-founded metric monotonically decreases must in fact be finite. Lee et al. apply this principle to the argument values passed along call sequences. Specifically, the principle admits that, if some argument is decreasing across the subsequence of recursive calls to any particular function, then that call sequence is guaranteed to terminate.

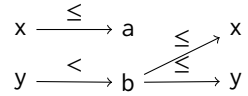
Lee et al. show how to detect SCT violations using a size-change graph (SCG), which summarizes the size change between argument values along a path. An SCG is bipartite with references to initial path values in one partition, references to terminal path values in the other, and labelled edges from initial to final value references expressing known size-change relationships. For example, given the definitions of `f` and `g` on the left, the relationship between `f`'s parameters and `g`'s parameters, bound to the arguments of the call to `g` within `f`'s body, are summarized by the size-change graph on the right. Perhaps counterintuitively, this graph does *not* state that `x` is less than or equal to `a`; instead, it states that, on the path from `f`'s call to `g`'s call, the value of `g`'s parameter `a` descends or is equal to ("does not ascend") relative to `f`'s parameter `x` and that the value of `g`'s parameter `b` strictly descends relative to `f`'s parameter `y`. An absent edge in the graph (such as, in this graph, between `x` and `b`) denotes that the values are not known to descend, strictly or not. Similarly, the relationship between `g`'s free variables and `f`'s free variables, bound to the call to `f`'s arguments within `g`'s body, are summarized by the graph to the left which states that the values of neither `x` nor `y` ascend relative to the value of `b`.

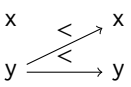
```
(define (f x y) (g x (- y 1)))
(define (g a b) (+ a (f b b)))
```





SCGs can be composed to combine the size-change information across two contiguous paths into an SCG for their concatenation. For example, the two SCGs above can be composed to obtain the size-change information between the free variables in a call to `f` and those of the subsequent recursive call to `f`. This composition is the tripartite graph to the right which can be flattened to the bipartite graph to the left in which the edge between, e.g., `y` and `x` is labelled with the most strict descent of a path from `y` to `x` in the tripartite graph.





Any well-founded relation on values can support the SCT principle, though different relations will vary in their effectiveness on any given program. In this paper, we employ an SCT-standard relation, exhibiting numeric magnitude for numbers, structural containment for data structures, and the closure ordering developed by Jones and Bohr [16]; we discuss this relation in detail in §7.3. Unlike Lee et al. [20] who infer size changes according to the metric based on the program text, we follow the approach of Nguyen et al. [27] who compare concrete values dynamically to detect SCT violations as the program runs. In our setting, however, this comparison is dynamic with respect to analysis time, not run time.

We review the technical criteria to detect SCT violations from SCGs in §7. Until then, we will look for self-descent on a variable when the domain and range of the SCG are the same, as occurs when comparing recursive calls of the same function, to establish SCT. Also, going forward, we will use a more compact notation in which the size change from `f` to `g` is denoted $\{(\mathtt{x}, \mathtt{a}) \mapsto \leq, (\mathtt{y}, \mathtt{b}) \mapsto <\}$.

3.2 CGADI in Action

Since it doesn't terminate, the unbounded counter of §2.1 must (1) be recursive, violating the reentrancy call guard, and (2) fail to satisfy the SCT principle, violating the size-change call guard.

$$
\begin{array}{lll}
1 & \langle (\texttt{loop 42}), & \bot, \quad \bot, \quad \text{con}, \emptyset \rangle \\
 & & \overbrace{\phantom{body_{\texttt{loop}}}}^{body_{\texttt{loop}}} \quad \overbrace{}^{\rho_0} \quad \overbrace{}^{\sigma_0} \\
2 & \langle \overbrace{(\texttt{loop (+ n 1))}}^{body_{\texttt{loop}}}, \overbrace{[\texttt{n} \mapsto a_0]}^{\rho_0}, \overbrace{\bot[a_0 \mapsto 42]}^{\sigma_0}, & \text{con}, \{body_{\texttt{loop}}\} \rangle \\
 & & \overbrace{}^{\sigma_1} \\
3 & \langle (\texttt{loop (+ n 1)}), [\texttt{n} \mapsto a_1], \overbrace{\sigma_0[a_1 \mapsto 43]}^{\sigma_1}, & \text{abs}, \{body_{\texttt{loop}}\} \rangle \\
4 & \langle (\texttt{loop (+ n 1)}), [\texttt{n} \mapsto \texttt{n}], \sigma_1[\texttt{n} \mapsto |44|_{num}], & \text{abs}, \{body_{\texttt{loop}}\} \rangle \\
5 & \langle (\texttt{loop (+ n 1)}), [\texttt{n} \mapsto \texttt{n}], \sigma_1[\texttt{n} \mapsto \top_{int}], & \text{abs}, \{body_{\texttt{loop}}\} \rangle
\end{array}
$$

Fig. 1. The state sequence of 0CGADI analysis of the `loop` program

Let's walk through a CGADI analysis of it to see how the reentrancy call guard detects this violation and how CGADI responds.

CGADI includes a parameter m which refers to the number of stack frames that qualify abstract evaluation (as in m-CFA [25]). For this walkthrough, we suppose $m = 0$ so that like the preceding 0CFA, a 0CGADI state includes the control expression, the environment, the store, and the continuation (which we continue to omit). To these components, 0CGADI adds a *mode signifier*, indicating whether that configuration's interpretation proceeds concretely or abstractly, and a *call cache* which the call guard may use to track auxiliary data regarding the current path. Here the call cache contains active entry states.

The five states which comprise the entirety of the 0CGADI analysis can be seen in Figure 1. Execution starts with state 1 which has an empty environment, store, and call cache, and a concrete evaluation mode. The transition to state 2 evaluates the argument concretely. No entry for the body of `loop` exists in the call cache, which means that this is the first call to `loop` on this evaluation path. Evaluation remains concrete and the concrete argument is bound to a concrete address a_0 in the store. Finally, the call cache is augmented with the call entry $body_{\texttt{loop}}$. The transition to state 3 proceeds concretely so its argument is bound to a fresh address a_1. At this call, however, a call entry for $body_{\texttt{loop}}$ exists in the call cache, causing the reentrancy call guard to signal that interpretation should continue in abstract mode. Accordingly, state 3's transition to state 4, another recursive call, is performed in abstract mode. In non-tail position, its argument is evaluated in concrete mode and yields the concrete value 44. Because the call itself is in abstract mode, this argument value is abstracted (via the abstraction function $|\cdot|_{num}$) and allocated in the store, as dictated by 0CFA. From state 4, the input to the argument calculation is abstract; nevertheless, the resulting argument value is kept precise as the abstract value $|45|$. State 4 transitions to state 5, whose store is calculated as $\sigma_1[\texttt{n} \mapsto |44|_{num}] \sqcup [\texttt{n} \mapsto |45|_{num}] = \sigma_1[\texttt{n} \mapsto |44|_{num} \sqcup |45|_{num}] = \sigma_1[\texttt{n} \mapsto \top_{int}]$. (This evaluation uses the "crush" store extension technique [4] which preserves as much precision as possible when calculating values and approximates only when joining in the store.) State 5's successor is itself, and the 0CGADI analysis is complete.

3.3 When the Size-Change Canary is Silent

The previous analysis showed how 0CGADI deployed with a reentrancy call guard transitions to abstract interpretation in the face of a nonterminating program by detecting recursive calls along a path. As a safety condition, this criteria is very broad, as it would also induce abstract interpretation for the factorial program of §2.3. But, this program, while recursive, obeys the SCT principle, and we will now examine how 0CGADI deployed with a size-change call guard justifies its concrete interpretation.

$$
\begin{array}{lll}
1 & \langle \texttt{(fact 5 1)}, & \bot, \qquad \bot, \qquad \bot \qquad \rangle \\
 & & \overbrace{\phantom{[\mathsf{n} \mapsto a_0, \mathsf{a} \mapsto a_1]}}^{\rho_0} \quad \overbrace{}^{\sigma_0} \quad \overbrace{\phantom{[body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto \leq, (\mathsf{a},\mathsf{a}) \mapsto \leq], \rho_0)]}}^{C_0} \\
2 & \langle \texttt{(zero? n)}, & [\mathsf{n} \mapsto a_0, \mathsf{a} \mapsto a_1], \quad \bot[a_0 \mapsto 5, a_1 \mapsto 1], \quad [body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto \leq, (\mathsf{a},\mathsf{a}) \mapsto \leq], \rho_0)] \rangle \\
3 & \langle call_{\texttt{fact}}, & \rho_0, \qquad \sigma_0, \qquad C_0 \qquad \rangle \\
 & & \overbrace{\phantom{[\mathsf{n} \mapsto a_2, \mathsf{a} \mapsto a_3]}}^{\rho_1} \quad \overbrace{}^{\sigma_1} \quad \overbrace{\phantom{[body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <, (\mathsf{n},\mathsf{a}) \mapsto \leq], \rho_1)]}}^{C_1} \\
4 & \langle \texttt{(zero? n)}, & [\mathsf{n} \mapsto a_2, \mathsf{a} \mapsto a_3], \quad \sigma_0[a_2 \mapsto 4, a_3 \mapsto 5], \quad [body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <, (\mathsf{n},\mathsf{a}) \mapsto \leq], \rho_1)] \rangle \\
5 & \langle call_{\texttt{fact}}, & \rho_1, \qquad \sigma_1, \qquad C_1 \qquad \rangle \\
 & & \overbrace{\phantom{[\mathsf{n} \mapsto a_4, \mathsf{a} \mapsto a_5]}}^{\rho_2} \quad \overbrace{}^{\sigma_2} \quad \overbrace{\phantom{[body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <], \rho_2)]}}^{C_2} \\
6 & \langle \texttt{(zero? n)}, & [\mathsf{n} \mapsto a_4, \mathsf{a} \mapsto a_5], \quad \sigma_1[a_4 \mapsto 3, a_5 \mapsto 20], \quad [body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <], \rho_2)] \qquad \rangle \\
7 & \langle call_{\texttt{fact}}, & \rho_2, \qquad \sigma_2, \qquad C_2 \qquad \rangle \\
 & \ldots & \\
 & & \overbrace{\phantom{[\mathsf{n} \mapsto a_{10}, \mathsf{a} \mapsto a_{11}]}}^{\rho_5} \quad \overbrace{\phantom{\sigma_4[a_{10} \mapsto 0, a_{11} \mapsto 120]}}^{\sigma_5} \quad \overbrace{\phantom{[body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <], \rho_5)]}}^{C_5} \\
12 & \langle \texttt{(zero? n)}, & [\mathsf{n} \mapsto a_{10}, \mathsf{a} \mapsto a_{11}], \sigma_4[a_{10} \mapsto 0, a_{11} \mapsto 120], [body_{\texttt{fact}} \mapsto ([(\mathsf{n},\mathsf{n}) \mapsto <], \rho_5)] \qquad \rangle \\
13 & \langle \mathsf{a}, & \rho_5, \qquad \sigma_5, \qquad C_5 \qquad \rangle
\end{array}
$$

Fig. 2. The state sequence of 0CGADI analysis of the `fact` program

Because 0CGADI is able to maintain concrete evaluation for the entire analysis, we will dispense with mode signifiers from states. Under the size-change call guard, the call cache contains an SCG over the free variables of the initial and terminal entry states on each path as well as the environment of the initial entry state by which values can be resolved to calculate size change. Within the analysis, $body_{\texttt{fact}}$ denotes the expression `(if (zero? n) a (fact (- n 1) (* n a)))` and $call_{\texttt{fact}}$ `(fact (- n 1) (* n a))`.

Figure 2 presents the states that arise in a 0CGADI analysis of the factorial function. State 1 consists of the top-level expression with an empty environment, store, and call cache. State 1 transitions to state 2 as the first call to `fact` is entered. The mode is concrete, so each argument is bound to a fresh address. Additionally, the transition introduces an initial entry for $body_{\texttt{fact}}$ into the call cache which includes its environment to recover the values of n and a within this call and a neutral SCG (i.e. the identity of SCG composition) $\{(\mathsf{n},\mathsf{n}) \mapsto \leq, (\mathsf{a},\mathsf{a}) \mapsto \leq\}$. 0CGADI can determine a precise value for the guard `(zero? n)`, and state 2 transitions to state 3. After evaluating the arguments in state 3, but before carrying out the recursive call, 0CGADI consults the call cache and discovers a preexisting entry for $body_{\texttt{fact}}$. 0CGADI produces the SCG $\{(\mathsf{n},\mathsf{n}) \mapsto <, (\mathsf{n},\mathsf{a}) \mapsto \leq\}$ by comparing the $\langle \mathsf{n}, \mathsf{a} \rangle$ values $\langle 4, 5 \rangle$ for this call to the values $\langle 5, 1 \rangle$ of the previous. (Notice that the comparison detects a to be no larger than n, but this relationship will prove spurious.) Composed with the initial SCG $\{(\mathsf{n},\mathsf{n}) \mapsto \leq, (\mathsf{a},\mathsf{a}) \mapsto \leq\}$ from the cache, 0CGADI obtains $\{(\mathsf{n},\mathsf{n}) \mapsto <, (\mathsf{n},\mathsf{a}) \mapsto \leq\}$ which denotes that n strictly descends from the first call to this one. With this knowledge, evaluation remains concrete and 0CGADI binds the argument values to fresh addresses during the transition to state 4. Because n strictly descends each iteration and each SCG witnesses its descent, each recursive call can be made concretely. As n decreases and a increases, the non-ascent from n to a detected earlier evaporates so that the SCGs in the cache deeper in the recursion report only the self-descent of n. After a few iterations, 0CGADI reaches state 12 in which n is bound to 0. State 12 then transitions to state 13, the exit of the tail recursive function, and the 0CGADI concludes with a precise value for a in hand.

This program also illustrates that 0CGADI—and CGADI generally—can remain concrete even if not all environment values are concrete. For instance, if the initial value of a was the abstract value $\top_{int}$, all subsequent values of a in the recursion would be as well. However, if n remained concrete,

$$
\begin{array}{rl}
e \in \mathbf{Exp} & ::= \mathbf{let}\, x \;=\; ce\, \mathbf{in}\, e \mid ce \\
ce \in \mathbf{CExp} & ::= ae(ae_1, \ldots, ae_n)^{\ell} \mid \mathbf{case}\, ae\, \mathbf{of}\, pat_1 \Rightarrow e_1 \,;\, \ldots \,;\, pat_n \Rightarrow e_n\, \mathbf{end} \mid ae \\
ae \in \mathbf{AExp} & ::= c \mid p \mid tag \mid x \mid \lambda(x_1, \ldots, x_n).e \\
pat \in \mathbf{Pat} & ::= tag(x_1, \ldots, x_n) \qquad\qquad tag \in \mathbf{Tag} = \text{an infinite set} \\
\ell \in \mathbf{Lab} & = \text{an infinite set} \qquad\qquad x \in \mathbf{Var} = \text{an infinite set}
\end{array}
$$

Fig. 3. Syntax for our $\mathcal{A}$-normal form language

0CGADI would still be able to detect its self-descent and thereby preserve concrete interpretation of control.

4 Language

We present CGADI for a call-by-value language in $\mathcal{A}$-normal form [9] with higher-order functions, tagged tuples, and primitives which provide operations over constant and other data. The use of $\mathcal{A}$-normal form is not essential to our technique—our implementation is over a standard direct-style language—but it reduces the size of the formalisms substantially. The syntax is presented in Figure 3.

A proper expression e either **let**-binds a call expression or is itself a call expression. A call expression ce is either a labelled application expression, a case expression, or an atomic expression. A case expression discriminates its argument against a finite sequence of patterns which comprise tags and a finite sequence of variables. We assume that all patterns in case expression are distinct up to the tag and number of variables. A atomic expression ae is either a constant c, a primitive p, a tag tag (denoting a data constructor), a variable reference x, or a λ term. Tags, labels, and variables are each drawn from an infinite set distinct from all other syntactic entities. We keep the set of constants and primitives abstract throughout the development, but explain how to integrate them at each point.

We assume constructs for mutually-recursive functions, which can be achieved via the Y combinator. We also assume booleans, which desugar to `true()` and `false()`, and conditional expressions, which desugar to **case** expressions.

A program pr is a closed expression. We assume that programs are alpha-renamed so that each binding instance of a variable is unique. We sometimes refer to the syntactic domain **Lam** which is the subdomain of **AExp** which includes only and all λ terms.

Store allocation will be a major component of the forthcoming semantics. To simplify the address space structure, we assume an infinite sequence of variables $X_0, X_1, \ldots$ distinct from all program variables which act as indices and support data allocation.

5 CGADI

CGADI combines a concrete and abstract semantics in superposition and employs a policy that dictates which semantics to apply at each evaluation point. Accordingly, our presentation of CGADI proceeds first with a concrete semantics (§5.1), then with an abstract semantics (§5.2), and finally with a hybridization of the two (§5.3).

Before we proceed, however, we overview a few commonalities shared by all of these semantics.

(1) Each semantics specifies call-by-value evaluation in a big-step style, as a relation between *configurations* and *results*. In the concrete and abstract semantics, configurations and results both consist of entirely-standard abstract machine components found in, e.g., the AAM

$$
\begin{aligned}
\mathit{cfg} \in \mathit{Config} &= \mathbf{Exp} \times \mathit{Env} \times \mathit{Store} \times \mathit{Time} \\
\rho \in \mathit{Env} &= \mathbf{Var} \rightharpoonup \mathit{Addr} \\
\alpha \in \mathit{Addr} &= \mathbf{Var} \times \mathit{Time} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup \mathit{Value} \\
\mathit{Time} &= \mathbf{Lab}^* \\
\mathit{Const} &\ \text{a potentially-infinite set} \\
\mathit{Variant} &= \mathbf{Tag} \times \mathit{Addr}^*
\end{aligned}
\qquad
\begin{aligned}
\mathit{rslt} \in \mathit{Result} &= \mathit{Value} \times \mathit{Store} \\
v \in \mathit{Value} &= \mathit{Const} + \mathit{Proc} + \mathit{Variant} \\
\mathit{proc} \in \mathit{Proc} &= \mathit{Prim} + \mathit{Cons} + \mathit{Clos} \\
\mathit{Prim} &\ \text{a potentially-infinite set} \\
\mathit{Cons} &= \mathbf{Tag} \\
\mathit{Clos} &= \mathbf{Lam} \times \mathit{Env}
\end{aligned}
$$

Fig. 4. The concrete semantics state space

framework. (In this big-step setting, no continuation component is necessary or present and the resulting analysis enjoys perfect stack precision.) The hybrid semantics has a few nonstandard elements, which we detail in its presentation.

(2) For each semantics, the configuration–result relation in each is decomposed to parallel the syntactic structure of the $\mathcal{A}$-normal form language. In particular, an evaluation relation is defined for each expression class—proper, call, and atomic—with subsumption rules to respect the syntactic hierarchy.

5.1 Concrete Semantics

The state space of the concrete semantics is presented in Figure 4. A configuration consists of an expression e to be evaluated, an environment ρ in which to evaluate it, a store σ modelling the heap, and a time t providing freshness for heap allocation.

A result consists of a value and a store where a value is either a constant $\mathsf{const}(c)$, a procedure $\mathsf{proc}(\mathit{proc})$, or a tagged tuple $\mathsf{vari}(\mathit{tag}, \langle \alpha_1, \ldots, \alpha_n \rangle)$.

The set of constants parameterizes the state space and includes, e.g., numbers and strings. A procedure *proc* is either a primitive, constructor, or closure. The set of primitives consists of constant operators, e.g. arithmetic, and also parameterize the state space. A constructor encapsulates the tag of the variant that will be constructed when it is applied. A closure is a λ term paired with its closing environment.

An environment is a finite map from variables x to addresses α. A store is a map from addresses to values v. A value address (x, t) binds a value bound to variable x at time t. A variant address (X_i, t) binds a value bound to index i at time t. A time is a finite sequence of call site labels. A result consists of a value and a store.

Figure 5 presents the evaluation relations.

Evaluation of a proper (let) expression is expressed as a judgment $e\,\rho\,\sigma\,t \Downarrow v\,\sigma'$. In particular, a **let** expression evaluates to the value of its body in an environment and store updated with the value of its bound expression (Body-Eval).

Evaluation of an atomic expression is expressed as a judgment $ae\,\rho\,\sigma\,t \Downarrow_a v\,\sigma'$. Its sole rule relies on the $\mathcal{A}$ metafunction (AE-Eval), defined as follows.

$$
\begin{aligned}
\mathcal{A}(c, \rho, \sigma) &= \mathsf{const}(c) & \mathcal{A}(p, \rho, \sigma) &= \mathsf{proc}(\mathsf{prim}(p)) \\
\mathcal{A}(\mathit{tag}, \rho, \sigma) &= \mathsf{proc}(\mathsf{cons}(\mathit{tag})) & \mathcal{A}(x, \rho, \sigma) &= \sigma(\rho(x))
\end{aligned}
$$

$$
\mathcal{A}(\lambda(x_1, \ldots, x_n).e, \rho, \sigma) = \mathsf{proc}(\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho))
$$

$$\textsc{Body-Eval}\ \frac{ce\,\rho\,\sigma\,t\ \Downarrow_c\ v\,\sigma' \qquad \alpha = (x,t) \qquad e\,\rho[x \mapsto \alpha]\,\sigma'[\alpha \mapsto v]\,t\ \Downarrow\ v\,\sigma''}{\mathbf{let}\,x\ =\ ce\,\mathbf{in}\,e\,\rho\,\sigma\,t\ \Downarrow\ v\,\sigma''}$$

$$\textsc{AE-Eval-Subsumption}\ \frac{ae\,\rho\,\sigma\,t\ \Downarrow_a\ v\,\sigma'}{ae\,\rho\,\sigma\,t\ \Downarrow_c\ v\,\sigma'}$$

$$\textsc{CE-Eval-Subsumption}\ \frac{ce\,\rho\,\sigma\,t\ \Downarrow_c\ v\,\sigma'}{ce\,\rho\,\sigma\,t\ \Downarrow\ v\,\sigma'}$$

$$\textsc{AE-Eval}\ \frac{\mathcal{A}(ae,\rho,\sigma) = v}{ae\,\rho\,\sigma\,t\ \Downarrow_a\ v\,\sigma}$$

$$\textsc{CE-Application-Eval}\ \frac{\begin{array}{c}\mathcal{A}(ae,\rho,\sigma) = \mathsf{proc}(proc) \\ \mathcal{A}(ae_i,\rho,\sigma) = v_i \qquad i = 1,\ldots,n \qquad proc\,\langle v_1,\ldots,v_n\rangle\,\sigma\,\ell :: t\ \Downarrow_{apply}\ v\,\sigma'\end{array}}{ae(ae_1,\ldots,ae_n)^\ell\,\rho\,\sigma\,t\ \Downarrow_c\ v\,\sigma'}$$

$$\textsc{CE-Case-Eval}\ \frac{\begin{array}{c}\mathcal{A}(ae,\rho,\sigma) = \mathsf{vari}(tag_i,\langle\alpha_1,\ldots,\alpha_{m_i}\rangle) \qquad \beta_j = (x_{i,j},t) \\ \sigma_1 = \sigma[\beta_j \mapsto \sigma(\alpha_j)] \qquad \rho_1 = \rho[x_{i,j} \mapsto \beta_j] \qquad j = 1,\ldots,m_i \qquad e_i\,\rho_1\,\sigma_1\,t\ \Downarrow\ v'\,\sigma'\end{array}}{\mathbf{case}\,ae\,\mathbf{of}\,tag_1(x_{1,1},\ldots,x_{1,m_1}) \Rightarrow e_1\,;\,\ldots\,;\,tag_n(x_{n,1},\ldots,x_{n,m_n}) \Rightarrow e_n\,\mathbf{end}\,\rho\,\sigma\,t\ \Downarrow_c\ v'\,\sigma'}$$

$$\textsc{Prim-Apply}\ \frac{p\,\langle v_1,\ldots,v_n\rangle\,\sigma\,t\ \downarrow\ v\,\sigma'}{\mathsf{prim}(p)\,\langle v_1,\ldots,v_n\rangle\,\sigma\,t\ \Downarrow_{apply}\ v\,\sigma'}$$

$$\textsc{Cons-Apply}\ \frac{\alpha_i = (X_i,t) \qquad \sigma_i = \sigma_{i-1}[\alpha_i \mapsto v_i] \qquad i = 1,\ldots,n}{\mathsf{cons}(tag)\,\langle v_1,\ldots,v_n\rangle\,\sigma_0\,t\ \Downarrow_{apply}\ \mathsf{vari}(tag,\langle\alpha_1,\ldots,\alpha_n\rangle)\,\sigma_n}$$

$$\textsc{Clos-Apply}\ \frac{\begin{array}{c}t' = t \\ \alpha_i = (x_i,t') \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \sigma_i = \sigma_{i-1}[\alpha_i \mapsto v_i] \qquad i = 1,\ldots,n \qquad e\,\rho_n\,\sigma_n\,t'\ \Downarrow\ v\,\sigma\end{array}}{\mathsf{clos}(\lambda(x_1,\ldots,x_n).e,\rho)\,\langle v_1,\ldots,v_n\rangle\,\sigma_0\,t\ \Downarrow_{apply}\ v\,\sigma}$$

Fig. 5. The evaluation relation

Constants and primitives evaluate to corresponding values. A tag evaluates to a constructor producing variants with that tag. A variable reference locates an address in the environment which locates a value in the store. A λ term evaluates to a closure.

Evaluation of a call expression is expressed as a judgment $ce\,\rho\,\sigma\,t\ \Downarrow_c\ v\,\sigma'$. A case call expression evaluates the scrutinee and selects the appropriate clause corresponding to its tag and arity, and takes on the value of the clause body in an extended environment and store (CE-Case-Eval). An application call expression atomically evaluations the operator and arguments and applies the opeartor value to a vector of arguments in the successor time (CE-Application-Eval). If the operator is a primitive, the result is determined by the primitive evaluation relation $\downarrow$, a parameter of the semantics. If the operator is a constructor, a new variant is allocated and returned (Cons-Apply); in this case, the address $\hat{\alpha}_i$ of argument i is derived from a unique variable X_i indicating its position (see §4), as well as the new time $\hat{t}'$. If instead the operator is a closure, the entry environment and store is produced and the value of this entry configuration becomes the value of the call (Clos-Apply).

Evaluation of an atomic expression in a call position follows the rule of atomic expression evaluation (AE-Eval-Subsumption) and of a call expression in general position follows the rule of call expression evaluation (CE-Eval-Subsumption).

$$\widehat{cfg} \in \widehat{Config} = \mathbf{Exp} \times \widehat{Env} \times \widehat{Store} \times \widehat{Time} \qquad \widehat{rslt} \in \widehat{Result} = \widehat{Value} \times \widehat{Store}$$
$$\hat{\rho} \in \widehat{Env} = \mathbf{Var} \rightharpoonup \widehat{Addr} \qquad \hat{v} \in \widehat{Value} = \widehat{Const} \times \widehat{Proc} \times \widehat{Variant}$$
$$\hat{\alpha} \in \widehat{Addr} = \mathbf{Var} \times \widehat{Time} \qquad \widehat{proc} \in \widehat{Proc} = \mathcal{P}(Prim) \times \mathcal{P}(Cons) \times \mathcal{P}(\widehat{Clos})$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \widehat{Value} \qquad \widehat{Clos} = \mathbf{Lam} \times \widehat{Env}$$
$$\widehat{Time} = \mathbf{Lab}^{\leq m} \qquad \widehat{Variant} = \mathbf{Tag} \times \widehat{Addr}^{*}$$
$$\widehat{Const} \text{ a potentially-infinite domain}$$

Fig. 6. The abstract semantics state space

The initial evaluation configuration is given by the injection function $\mathcal{I}$ defined $\mathcal{I}(pr) = pr\ \bot\ \bot\ \langle\rangle$ which yields an evaluation configuration with an empty environment and store and the initial empty time.

5.2 Abstract Semantics

The abstract semantics supports an orthodox control-flow analysis (CFA) and is systematically derived from the concrete, essentially using the AAM methodology. To compute the analysis in this big-step setting (in which continuations are implicit), we use the caching strategy of the *Abstracting Definitional Interpreters* (ADI) [4] work. In so doing, we obtain a computable interpretation which models the stack precisely.

The state space is presented in Figure 6. The overall structure of the state space is the largely the same as the concrete semantics's. The root difference is the finitization of the address space achieved by bounding the sequence of labels (constituting addresses) to length at most m. With a finite address space, an allocation may occur at an already-occupied address. To account for both the existing and new allocation, the store extension operation gives way to store join. To accommodate the join of values, particularly from different domains, the value domain shifts from a sum of concrete subdomains to a product of abstract subdomains. The new address representation percolates to environments and variants, then closures. A procedure itself is a set of primitives, tags (standing in for constructors), and closures. The structure of closures is lifted to abstract environments. A variant is a set of tagged sequences of variant addresses, rather than a single sequence of values. A store remains a map from addresses to values, albeit lifted to their abstract counterparts. The set of constants parameterizing the semantics is restricted to a domain with a lattice ordering with no infinite ascending chains. In the end, the state space structure remains largely intact, though nearly every part of it has been touched by the change to addresses.

Figure 7 presents the evaluation relations. There are only two essential differences between the concrete and abstract evaluation relation definitions.

(1) The Cons-Apply and Clos-Apply rules truncate the incoming time to at most m labels using the operator $\lfloor\cdot\rfloor_m$. Both of these rules use this truncated time to derive addresses for store allocation. Moreover, because only the CE-Application-Eval rule extends the time, the truncation applied by Clos-Apply upholds the invariant that times within evaluation configurations are limited to m labels.
(2) Store allocation is not performed using store update $\sigma[\alpha \mapsto v]$ in which the prior resident of α (if any) is *replaced* by v, but rather by *store join* $\hat{\sigma} \sqcup [\hat{\alpha} \mapsto \hat{v}]$ which is defined as $\hat{\sigma}_0 \sqcup \hat{\sigma}_1 = \lambda\hat{\alpha}.\hat{\sigma}_0(\hat{\alpha}) \sqcup \hat{\sigma}_1(\hat{\alpha})$ so that accessing $\hat{\alpha}$ produces $\hat{v}$ joined with $\hat{\sigma}$'s mapping for it.

$$\textsc{Body-Eval}\quad \frac{ce\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_c\,\hat{v}\,\hat{\sigma}' \qquad \hat{\alpha} = (x, \hat{t}) \qquad e\,\hat{\rho}[x \mapsto \hat{\alpha}]\,\hat{\sigma}' \sqcup [\hat{\alpha} \mapsto \hat{v}]\,\hat{t}\,\hat{\Downarrow}\,\hat{v}\,\hat{\sigma}''}{\textbf{let}\,x\;=\;ce\,\textbf{in}\,e\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}\,\hat{v}\,\hat{\sigma}''}$$

$$\textsc{AE-Eval-Subsumption}\quad \frac{ae\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_a\,\hat{v}\,\hat{\sigma}'}{ae\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_c\,\hat{v}\,\hat{\sigma}'}$$

$$\textsc{CE-Eval-Subsumption}\quad \frac{ce\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_c\,\hat{v}\,\hat{\sigma}'}{ce\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}\,\hat{v}\,\hat{\sigma}'}$$

$$\textsc{AE-Eval}\quad \frac{\hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) = \hat{v}}{ae\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_a\,\hat{v}\,\hat{\sigma}}$$

$$\textsc{CE-Application-Eval}\quad \frac{\widehat{proc} \in \downarrow_{\widehat{proc}}(\hat{v}) \qquad \hat{\mathcal{A}}(ae_i, \hat{\rho}, \hat{\sigma}) = \hat{v}_i \qquad i = 1, \ldots, n \qquad \widehat{proc}\,\langle \hat{v}_1, \ldots, \hat{v}_n \rangle\,\hat{\sigma}\,\ell :: \hat{t}\,\hat{\Downarrow}_{apply}\,\hat{v}\,\hat{\sigma}'}{ae(ae_1, \ldots, ae_n)^{\ell}\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_c\,\hat{v}\,\hat{\sigma}'}$$

$$\textsc{CE-Case-Eval}\quad \frac{\begin{array}{c}\hat{\mathcal{A}}(ae, \hat{\rho}, \hat{\sigma}) = \hat{v} \qquad (tag_i, \langle \hat{\alpha}_1, \ldots, \hat{\alpha}_{m_i} \rangle) \in \downarrow_{\widehat{vari}}(\hat{v}) \qquad \hat{\beta}_j = (x_{i,j}, \hat{t}) \\ \hat{\sigma}_1 = \hat{\sigma} \sqcup [\hat{\beta}_j \mapsto \hat{\sigma}(\hat{\alpha}_j)] \qquad \hat{\rho}_1 = \hat{\rho}[x_{i,j} \mapsto \hat{\beta}_j] \qquad j = 1, \ldots, m_i \qquad e_i\,\hat{\rho}_1\,\hat{\sigma}_1\,\hat{t}\,\hat{\Downarrow}\,\hat{v}'\,\hat{\sigma}'\end{array}}{\textbf{case}\,ae\,\textbf{of}\,tag_1(x_{1,1}, \ldots, x_{1,m_1}) \Rightarrow e_1\,;\,\ldots\,;\,tag_n(x_{n,1}, \ldots, x_{n,m_n}) \Rightarrow e_n\,\textbf{end}\,\hat{\rho}\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_c\,\hat{v}'\,\hat{\sigma}'}$$

$$\textsc{Prim-Apply}\quad \frac{p\,\langle \hat{v}_1, \ldots, \hat{v}_n \rangle\,\hat{\sigma}\,\hat{t}\,\hat{\downarrow}\,\hat{v}\,\hat{\sigma}'}{\mathsf{prim}(p)\,\langle \hat{v}_1, \ldots, \hat{v}_n \rangle\,\hat{\sigma}\,\hat{t}\,\hat{\Downarrow}_{apply}\,\hat{v}\,\hat{\sigma}'}$$

$$\textsc{Cons-Apply}\quad \frac{\hat{t}' = \lfloor \hat{t} \rfloor_m \qquad \hat{\alpha}_i = (X_i, \hat{t}') \qquad \hat{\sigma}_i = \hat{\sigma}_{i-1} \sqcup [\hat{\alpha}_i \mapsto \hat{v}_i] \qquad i = 1, \ldots, n}{\mathsf{cons}(tag)\,\langle \hat{v}_1, \ldots, \hat{v}_n \rangle\,\hat{\sigma}_0\,\hat{t}\,\hat{\Downarrow}_{apply}\,|\mathsf{vari}(tag, \langle \alpha_1, \ldots, \alpha_n \rangle)|\,\hat{\sigma}_n}$$

$$\textsc{Clos-Apply}\quad \frac{\begin{array}{c}\hat{t}' = \lfloor \hat{t} \rfloor_m \qquad \hat{\alpha}_i = (x_i, \hat{t}') \\ \hat{\rho}_i = \hat{\rho}_{i-1}[x_i \mapsto \hat{\alpha}_i] \qquad \hat{\sigma}_i = \hat{\sigma}_{i-1} \sqcup [\hat{\alpha}_i \mapsto \hat{v}_i] \qquad i = 1, \ldots, n \qquad e\,\hat{\rho}_n\,\hat{\sigma}_n\,\hat{t}'\,\hat{\Downarrow}\,\hat{v}\,\hat{\sigma}\end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \hat{\rho})\,\langle \hat{v}_1, \ldots, \hat{v}_n \rangle\,\hat{\sigma}_0\,\hat{t}\,\hat{\Downarrow}_{apply}\,\hat{v}\,\hat{\sigma}}$$

Fig. 7. Abstract evaluation rules

Soundness and Computability. Two essential characteristics of an abstract semantics are (1) soundness with respect to the concrete semantics and (2) computability. In abstract interpretation, soundness is mediated by a formal relationship between the concrete and abstract state spaces by way of a Galois connection. This relationship is standard, and we fold our presentation of it into the hybrid semantics. Anticipating a refinement ordering $\sqsubseteq$ over configurations and over results, soundness is expressed as follows.

Theorem 5.1 (Abstract Evaluation Soundness). *If* $\widehat{cfg}_0 \sqsubseteq \widehat{cfg}_1$ *and* $\widehat{cfg}_0\,\hat{\Downarrow}\,\widehat{rslt}_0$ *then* $\widehat{cfg}_1\,\hat{\Downarrow}\,\widehat{rslt}_1$ *where* $\widehat{rslt}_0 \sqsubseteq \widehat{rslt}_1$.

This theorem is proven by induction on the evaluation derivation.

$$\widetilde{cfg} \in \widetilde{Config} = \mathbf{Exp} \times Env \times \widetilde{Store} \times Time \times Mode \times Cache \qquad \widehat{rslt} \in \widehat{Result} = \widehat{Value} \times \widetilde{Store}$$
$$\tilde{v} \in \widetilde{Value} = Value + \widehat{Value} \qquad \tilde{\sigma} \in \widetilde{Store} = Addr \rightharpoonup \widetilde{Value} \qquad Mode = \{\mathsf{con}, \mathsf{abs}\}$$
$$Cache \text{ a lattice with no infinite ascending chains}$$

Fig. 8. The hybrid semantics state space

Computability follows from the finiteness of the execution space, making the number of reachable configurations finite. The analysis can be computed using a big-step caching algorithm (such as the one ADI provides); it is presented in the appendix.

5.3 Hybrid Semantics

The hybrid semantics combines the concrete and abstract semantics and employs a policy to safely toggle between them. This hybridization manifests first in the state space, presented in Figure 8. Like the concrete and abstract semantics, a hybrid configuration includes an expression, environment, store, and timestamp. In addition, it includes a *mode* which indicates the semantics under which the configuration should be evaluated and a path-specific *cache* which the parameterizing safety property instantiates with auxiliary data. The set of modes forms a lattice with the single relationship $\mathsf{con} \sqsubset \mathsf{abs}$. A hybrid value is either a concrete value or an abstract value. Similarly, a hybrid address is either a concrete address or an abstract address but, as $Addr$ contains $\widehat{Addr}$, the hybrid address set is simply $Addr$. The address and value definitions percolate to other state space components: stores and results are manifestly hybridized but environments and times are simply concrete.

Figure 9 presents the hybrid evaluation semantics. This semantics is defined in terms of the same evaluation relations—and even rules—as the previous semantics. It differs in that it (1) accounts for the evaluation mode, and (2) defers to a termination safety property.

5.3.1 Evaluation Mode. Just as a configuration evaluation is concrete or abstract, its resultant value is concrete or abstract. The *disposition* $\tilde{v}^\circ$ of a hybrid value $\tilde{v}$ is the precision of that value in terms of an evaluation mode. The semantics ensures that configuration evaluation does not yield a value whose disposition is stronger than its mode by *disposing* the value, i.e., downcasting it to a weaker disposition, if necessary. The *disposal* $\lceil \tilde{v} \rceil^M$ is the coarsening of $\tilde{v}$ to the disposition M.

$$\cdot^\circ : \widetilde{Value} \to Mode \qquad \lceil \cdot \rceil^\cdot : Mode \times \widetilde{Value} \to \widetilde{Value}$$

$$\mathsf{con}(v)^\circ = \mathsf{con} \quad \mathsf{abs}(\hat{v})^\circ = \mathsf{abs} \qquad \lceil \tilde{v} \rceil^{\mathsf{con}} = \tilde{v} \qquad \lceil \mathsf{con}(v) \rceil^{\mathsf{abs}} = \mathsf{abs}(|v|) \qquad \lceil \mathsf{abs}(\hat{v}) \rceil^{\mathsf{abs}} = \mathsf{abs}(\hat{v})$$

The disposition is used within the semantics to match the interpretation mode to the precision of a value on which control depends—e.g. the scrutinee of a **case** expression. The disposal is used to match the value precision to the mode requested by the context, such as with atomic evaluation. Atomic evaluation, via $\tilde{\mathcal{A}}$, produces a hybrid value.

$$\tilde{\mathcal{A}}(c, \rho, \tilde{\sigma}) = \mathsf{con}(\mathsf{const}(c)) \qquad \tilde{\mathcal{A}}(p, \rho, \tilde{\sigma}) = \mathsf{con}(\mathsf{proc}(\mathsf{prim}(p)))$$
$$\tilde{\mathcal{A}}(tag, \rho, \tilde{\sigma}) = \mathsf{con}(\mathsf{proc}(\mathsf{cons}(tag))) \qquad \tilde{\mathcal{A}}(x, \rho, \tilde{\sigma}) = \tilde{\sigma}(\rho(x))$$
$$\tilde{\mathcal{A}}(\lambda(x_1, \ldots, x_n).e, \rho, \tilde{\sigma}) = \mathsf{con}(\mathsf{proc}(\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho)))$$

Store update is (effectively) sensitive to the disposition of the resident value (if any) and the updated value. Given a refinement ordering on values by the reflexive, transitive closure of the rules

$$\bot \sqsubseteq \mathsf{con}(v) \text{ for all } v \qquad \mathsf{con}(v) \sqsubseteq \mathsf{abs}(\hat{v}) \text{ iff } |v| \sqsubseteq \hat{v} \qquad \mathsf{abs}(\hat{v}_0) \sqsubseteq \mathsf{abs}(\hat{v}_1) \text{ iff } \hat{v}_0 \sqsubseteq \hat{v}_1$$

$$\textsc{Body-Eval}\quad \frac{ce\,\rho\,\tilde\sigma\,t\,\mathsf{con}\,C \mathrel{\tilde\Downarrow_c} \tilde v\,\tilde\sigma' \qquad \alpha = (x,t) \qquad e\,\rho[x\mapsto\alpha]\,\tilde\sigma'[\alpha \mathrel{\tilde\mapsto} \tilde v]\,t\,M\,C \mathrel{\tilde\Downarrow} \tilde v\,\tilde\sigma''}{\mathbf{let}\,x\;=\;ce\,\mathbf{in}\,e\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow} \tilde v\,\tilde\sigma''}$$

$$\textsc{AE-Eval-Subsumption}\quad \frac{ae\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_a} \tilde v\,\tilde\sigma'}{ae\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_c} \tilde v\,\tilde\sigma'} \qquad \textsc{CE-Eval-Subsumption}\quad \frac{ce\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_c} \tilde v\,\tilde\sigma'}{ce\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow} \tilde v\,\tilde\sigma'}$$

$$\textsc{AE-Eval}\quad \frac{\tilde{\mathcal{A}}(ae,\rho,\tilde\sigma) = \tilde v}{ae\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_a} \lceil\tilde v\rceil^M\,\tilde\sigma}$$

$$\textsc{CE-Application-Eval}\quad \frac{\begin{array}{c}\tilde{\mathcal{A}}(ae,\rho,\tilde\sigma) = \tilde v \qquad proc \in\!\downarrow_{\widetilde{proc}}(\tilde v)\\ \tilde{\mathcal{A}}(ae_i,\rho,\tilde\sigma) = \tilde v_i \qquad i = 1,\ldots,n \qquad proc\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma\,\ell :: t\,M \sqcup \tilde v^\circ\,C \mathrel{\tilde\Downarrow_{apply}} \tilde v_f\,\tilde\sigma_f\end{array}}{ae(ae_1,\ldots,ae_n)^\ell\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_c} \tilde v_f\,\tilde\sigma_f}$$

$$\textsc{CE-Case-Eval}\quad \frac{\begin{array}{c}\tilde{\mathcal{A}}(ae,\rho,\tilde\sigma) = \tilde v \qquad (tag_i,\langle\alpha_1,\ldots,\alpha_{m_i}\rangle) \in\!\downarrow_{\widetilde{vari}}(\tilde v) \qquad M' = M \sqcup \tilde v^\circ \qquad \beta_j = (x_{i,j},t)\\ \tilde\sigma_1 = \tilde\sigma[\beta_j \mathrel{\tilde\mapsto} \lceil\tilde\sigma(\alpha_j)\rceil^{M'}] \qquad \rho_1 = \rho[x_{i,j}\mapsto\beta_j] \qquad j = 1,\ldots,m_i \qquad e_i\,\rho_1\,\tilde\sigma_1\,t\,M'\,C \mathrel{\tilde\Downarrow} \tilde v'\,\tilde\sigma'\end{array}}{\mathbf{case}\,ae\,\mathbf{of}\,tag_1(x_{1,1},\ldots,x_{1,m_1}) \Rightarrow e_1\,;\,\ldots\,;\,tag_n(x_{n,1},\ldots,x_{n,m_n}) \Rightarrow e_n\,\mathbf{end}\,\rho\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_c} \tilde v'\,\tilde\sigma'}$$

$$\textsc{Prim-Apply}\quad \frac{p\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma\,t \mathrel{\hat\downarrow} \tilde v\,\tilde\sigma'}{\mathsf{prim}(p)\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_{apply}} \lceil\tilde v\rceil^M\,\tilde\sigma'}$$

$$\textsc{Cons-Apply}\quad \frac{\alpha_i = (X_i,t) \qquad \tilde\sigma_i = \tilde\sigma_{i-1}[\alpha_i \mathrel{\tilde\mapsto} \lceil\tilde v_i\rceil^M] \qquad i = 1,\ldots,n}{\mathsf{cons}(tag)\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma_0\,t\,M\,C \mathrel{\tilde\Downarrow_{apply}} \lceil\mathsf{con}(\mathsf{vari}(tag,\langle\alpha_1,\ldots,\alpha_n\rangle))\rceil^M\,\tilde\sigma_n}$$

$$\textsc{Clos-Apply}\quad \frac{\mathsf{clos}(\lambda(x_1,\ldots,x_n).e,\rho)\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma\,t\,M\,C \;\rightarrow_{apply}\; e\,\rho'\,\tilde\sigma'\,t'\,M'\,C' \qquad e\,\rho'\,\tilde\sigma'\,t'\,M'\,C' \mathrel{\tilde\Downarrow} \tilde v'\,\tilde\sigma''}{\mathsf{clos}(\lambda(x_1,\ldots,x_n).e,\rho)\,\langle\tilde v_1,\ldots,\tilde v_n\rangle\,\tilde\sigma\,t\,M\,C \mathrel{\tilde\Downarrow_{apply}} \tilde v'\,\tilde\sigma''}$$

Fig. 9. The hybrid semantics

the update $\tilde\sigma[\alpha \mathrel{\tilde\mapsto} \tilde v]$ of a hybrid store $\tilde\sigma$ at address α with hybrid value $\tilde v$ is sensitive to how refined the entry of $\tilde\sigma$ is at α.

$$\tilde\sigma[\alpha \mathrel{\tilde\mapsto} \tilde v] = \begin{cases}\tilde\sigma[\alpha\mapsto\tilde v] \text{ if } \tilde\sigma(\alpha) \sqsubseteq \mathsf{con}(v) \text{ for some } v\\ \tilde\sigma \sqcup [\alpha\mapsto\tilde v] \text{ if } \mathsf{abs}(\hat v) \sqsubseteq \tilde\sigma(\alpha) \text{ for some } \hat v\end{cases}$$

Under this definition, a concrete value is replaced by the updated value whereas an abstract value is joined with the updated value. This policy allows the analyzer to perform strong update [3] when concrete precision is achieved.

The semantics degenerates to abstract mode at points of nondeterminism. In particular, if the operator of a call is an abstract value, the call itself is evaluated abstractly. Similarly, if the scrutinee of a **case** expression is abstract, the body is evaluated abstractly. While the presented concrete semantics are deterministic, hybridization of a nondeteriminstic concrete semantics can be achieved by explicitly abstracting at the points of nondeterminism within the semantics.

The CE-Application-Eval rule depends on the projection of a hybrid value into a set of concrete procedures.

$$\downarrow_{\widetilde{proc}} (\mathsf{con}(v)) = \begin{cases} \{proc\} \text{ if } v = \mathsf{proc}(proc) \\ \emptyset \text{ otherwise} \end{cases} \qquad \downarrow_{\widetilde{proc}} (\mathsf{abs}(\hat{v})) = \downarrow_{\widehat{proc}} (\hat{v})$$

$$\downarrow_{\widetilde{vari}} (\mathsf{con}(v)) = \begin{cases} \{(tag, \langle \alpha_1, \ldots, \alpha_n \rangle)\} \text{ if } v = \mathsf{vari}(tag, \langle \alpha_1, \ldots, \alpha_n \rangle) \\ \emptyset \text{ otherwise} \end{cases} \qquad \downarrow_{\widetilde{vari}} (\mathsf{abs}(\hat{v})) = \downarrow_{\widehat{vari}} (\hat{v})$$

5.3.2 Structural Mutation. Our framework supports structural mutation, which we illustrate through the use of boxes. This use entails

(1) including a box constructor with a single field, which the semantics is already defined to store-allocate;
(2) including the primitives unbox and setbox!, for which the semantics is structured to allow to specify the particular behavior; and
(3) specifying that behavior through two additional rules of primitive evaluation relation.

$$\textsc{Unbox}\ \frac{(\mathsf{box}, \langle \alpha \rangle) \in \downarrow_{\widetilde{vari}} (\tilde{v})}{\mathsf{unbox}\ \langle \tilde{v} \rangle\ \tilde{\sigma}\ t\ \hat{\downarrow}\ \tilde{\sigma}(\alpha)\ \tilde{\sigma}} \qquad \textsc{Set-Box!}\ \frac{(\mathsf{box}, \langle \alpha \rangle) \in \downarrow_{\widetilde{vari}} (\tilde{v}_0) \qquad \tilde{\sigma}' = \tilde{\sigma}[\alpha \mathrel{\tilde{\mapsto}} \tilde{v}_1]}{\mathsf{setbox!}\ \langle \tilde{v}_0, \tilde{v}_1 \rangle\ \tilde{\sigma}\ t\ \hat{\downarrow}\ \mathsf{con}(\mathsf{vari}(\mathsf{unit}, \langle\rangle))\ \tilde{\sigma}'}$$

The Unbox rule specifies that the unbox primitive, when applied to a single argument value, projects from that value box values constructed with a single address; the store value at that address becomes the resultant value. The Set-Box! rule specifies that the setbox! primitive, when applied to two argument values, projects box values from the first and updates the store at each argument address with the second argument value.

5.3.3 Safety Property. The Clos-Apply rules of the concrete and abstract semantics determine how a closure and its argument correspond to the entry evaluation configuration of the closure body. The hybrid semantics leaves this rule to be specified by a call guard which implements the $\tilde{\rightarrow}_{apply}$ relation. This choice provides the call guard with control over the call argument binding and the time and mode of the entry configuration. It is supported by a cache, which is left abstract in this semantics, which it can use to detect violations of a safety property that implies termination.

5.3.4 Soundness and Computability. Soundness and computability are two essential properties of a program analysis. We articulate each property here and discuss sufficient conditions for the call cache to establish them, but leave the proof in the appendix.

Theorem 5.2 (Hybrid Evaluation Soundness). *If* $|cfg| \sqsubseteq \widetilde{cfg}$ *and* $cfg \Downarrow rslt$ *then* $\widetilde{cfg} \tilde{\Downarrow} \widehat{rslt}$ *where* $|rslt| \sqsubseteq \widehat{rslt}$.

This theorem relies on an abstraction function for concrete configurations and a refinement relation for hybrid configurations.

Theorem 5.3 (Hybrid Evaluation Computability). *If the call cache has no infinite ascending chains and the call cache–store product increases on every novel call, then, for any program pr, only a finite number of distinct states are reachable from* $\tilde{I}(pr)$.

Because we are able to phrase computability in terms of reachable states, we can use a standard caching algorithm to compute CGADI, such as the one offered by the ADI framework. Although ultimately computability is a statement about the reachable configurations in the aggregate, CGADI ensures that this number is finite merely by ensuring that each path terminates.

5.3.5 Computational Complexity. Because CGADI lacks an *a priori* bound on execution, the computational complexity of the analysis depends on the behavior of the call guard. However, the ceiling on the computational complexity the call guard enables is high: even the Ackermann function obeys the SCT principle and is considered computable even though it is practically incomputable for even small inputs.

We conjecture that an effective remedy to the threat of high computational complexity is a proactive abstraction policy to attend the call guard, but we do not explore the design or effectiveness of such a mechanism in this work.

6 Reentrant Call Guard

One way to ensure that every path terminates is to bound the number of times each path is allowed to traverse any given block of code. This idea underlies the *reentrant call guard* which tracks which function bodies have been encountered (and at which abstract times) and degenerates to abstract evaluation when a function is re-entered. (It is straightforward to generalize this approach to allow multiple but bounded re-entries, but we illustrate with degenerating at the first reentry.) Although this approach manifestly precludes concrete evaluation of recursion, it is still able to maintain concrete precision on all non-recursive subevaluations. For instance, consider the program template to the right which, though it applies `apply0`, `apply1`, ..., `applyn` each more than once, applies each at most once on any evaluation path. Regardless of the value of n at which the template is instantiated, the reentrant call guard allows CGADI to trace the data flow completely precisely—which, as the discussion in §1 observed, off-the-shelf analyses cannot achieve—and this precision is achieved even if this code fragment is embedded within a larger recursive program.

```
(let* ([applyn (lambda (fn xn) (fn xn))]
       ...
      [apply2 (lambda (f2 x2) (apply3 f2 x2))]
      [apply1 (lambda (f1 x1) (apply2 f1 x1))])
  (+ (apply1 add1 12) (apply1 sub1 10)))
```

To formally present the guard, we define $\widetilde{Signature}$ as the set of expressions–time pairs and instantiate the $\widetilde{Cache}$ as the powerset lattice of $\widetilde{Signature}$.

$$\widetilde{Signature} = \mathbf{Exp} \times \widehat{Time} \qquad \widetilde{Cache}_R = \mathcal{P}(\widetilde{Signature})$$

For a finite program, there are a finite number of signatures, so this lattice has no infinite ascending chains.

Using this guard, there are two ways to make a $\tilde{\rightarrow}_{apply}$ judgment.

Clos-Apply-Not-Seen

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \qquad (e, t') \notin C \qquad C' = C \cup \{ (e, t') \} \\ \alpha_i = (x_i, t) \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \tilde{v}_i] \qquad i = 1, \ldots, n \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ M\ C\ \rightarrow_{apply}\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ M\ C'}$$

Clos-Apply-Seen

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \\ (e, t') \in C \qquad \alpha_i = (x_i, t') \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \lceil \tilde{v}_i \rceil^{\mathrm{abs}}] \qquad i = 1, \ldots, n \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ M\ C\ \rightarrow_{apply}\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ \mathrm{abs}\ C}$$

The Apply-Not-Seen rule ensures that, if the signature has not been encountered on this path, the evaluation mode of the call is as precise as possible, and adds the signature of the entry configuration to the cache. The Apply-Seen rule ensures that, if the signature has been encountered on this path, the evaluation mode of the call is abstract; the truncated abstract time is used and the abstract disposal of the arguments are bound.

7 Size-Change Call Guard

The reentrant call guard of the previous section is effective at allowing non-recursive evaluation to proceed concretely, but eagerly defects to abstract evaluation when recursion is detected. To accommodate recursion, we turn to the size-change termination (SCT) principle [20] which states that a program is size-change terminating if any infinite evaluation path would cause an infinite descent on well-founded data values. We present a call guard based on the SCT principle which monitors the size change between configurations to ensure that the SCT principle is upheld.

7.1 Size-Change Graphs

The size change between two states in a path is captured by a *size change graph* (SCG), a bipartite graph whose nodes represent state values and edges are annotated with the descent relationship between the state values. This descent relationship, an element of $\widetilde{Order} = \{<, \leq, >\}$, where $<$ denotes *strict* descent, $\leq$ *non-strict* (also called non-ascending), and $>$ *unknown*.

State values are those bound to free variables in the control expression of a configuration. We encode an SCG G as a map from pairs of variables to an element order, where each pair consists of a free variable in the source configuration control expression and a free variable in the destination configuration control expression.

Given two SCGs G_0 and G_1, where the target control expression in G_0 is the same as the source control expression in G_1, we can consider the composition $G_0; G_1$ which conveys the descent between arguments across the concatenation of the sequences.

We combine two descents with the binary operator $\oplus$ which is defined by the table on the right. The $\oplus$ operator gives the most precise descent while remaining conservative. If a value descends on two sequences and strictly descends on at least one of those sequences, it must descend strictly across their concatenation. If a value merely doesn't ascend on each, then it won't ascend across their concatenation, but it may not descend strictly. If a value ascends on either, then it may ascend on their concatenation.

$\oplus$	$<$	$\leq$	$>$
$<$	$<$	$<$	$>$
$\leq$	$<$	$\leq$	$>$
$>$	$>$	$>$	$>$

Using $\oplus$, we concatenate two SCGs with the binary operator ; defined as

$$G_0; G_1 = \lambda(x_0, x_1).\min_{x \in \mathbf{Var}}\{G_0(x_0, x) \oplus G_1(x, x_1)\}$$

where descents are ordered from most to least precise. The ; operator thus reflects the most precise descent relationship between two variables that can be had by way of an intermediate variable in the argument SCGs.

Lemma 7.1. *The SCG composition operator* ; *is associative.*

Lemma 7.2. *The SCG* ϵ *is the left and right unit of* ;.

Thus, we can view the domain $\widetilde{SCG}$ as a category wherein objects are variable sets and morphisms are SCGs. The identity morphism is ϵ and the composition operator is ;.

$$\widetilde{SCG} = \mathbf{Var} \times \mathbf{Var} \to \widetilde{Order}$$

We define the function $\mathrm{SCG} : \mathbf{Exp} \times Env \times Env \times \widetilde{Store} \to \widetilde{SCG}$ which constructs an SCG reporting the size change of environment values in the context of an expression and store.

$$\mathrm{SCG}(e, \rho_0, \rho_1, \sigma) = \lambda(x, y). \begin{cases} < & \text{if } x \in \mathrm{free}(e), y \in \mathrm{free}(e), \tilde{\sigma}(\rho_1(y)) \prec_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{\sigma}(\rho_0(x)) \\ \leq & \text{if } x \in \mathrm{free}(e), y \in \mathrm{free}(e), \tilde{\sigma}(\rho_1(y)) \preceq_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{\sigma}(\rho_0(x)) \\ > & \text{otherwise} \end{cases}$$

The definition relies on order, defined in the next subsection, which considers two denotable values in the context of a store, because the values may contain references into it.

A SCG G satisfies the SCT condition, denoted $SCC(G)$ when, if it is idempotent, it has a variable which descends on itself [20]. Formally,

$$SCC(G) \iff [G; G = G \implies \exists x \in \mathbf{Var}.G(x,x) =<]$$

This fact provides us with a strategy to detect potential nontermination in an evaluation path: maintain all SCGs G on a path and, at self loops, check whether any G is idempotent and does not have a arc of self-descent. If so, the program does not obey the SCT principle, and evaluation should degenerate to abstract evaluation.

7.2 Size-Change Call Guard

Formally, the size-change call guard is defined as follows.

$$\widetilde{Entry} = \mathcal{P}(\widetilde{SCG}) \times Env \qquad \widetilde{Cache}_{SCT} = \widetilde{Signature} \to \widetilde{Entry}_{\bot}^{\top}$$

A cache is a map from a $\widetilde{Signature}$ to an element of the flat $\widetilde{Entry}$ domain.

If a signature has not been encountered, the cache maps it to $\bot$. If the SCT condition has been violated at that signature, the cache maps it to $\top$. Otherwise, the cache maps it to a set of SCGs capturing the size change for each suffix of the evaluation path at this signature and the environment of the last configuration with this signature so that the size change across it and the next encounter can be calculated.

The transition judgments are defined as follows.

Apply-Not-Seen

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \qquad C(e,t') = \bot \qquad \alpha_i = (x_i, t) \\ \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \tilde{v}_i] \qquad i = 1, \ldots, n \qquad C' = C[(e,t') \mapsto \mathsf{entry}(\{\epsilon\}, \rho_f)] \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ \mathsf{con}\ C\ \rightarrow_b\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ \mathsf{con}\ C'}$$

Apply-Seen-Preserve

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \qquad C(e,t') = \mathsf{entry}(\mathbf{G}, \rho) \qquad \alpha_i = (x_i, t) \\ \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \tilde{v}_i] \qquad i = 1, \ldots, n \qquad G' = \mathsf{SCG}(e, \rho, \rho_f, \sigma_f) \\ \mathbf{G}' = \{G; G' : G \in \mathbf{G}\} \qquad \forall G \in \mathbf{G}'.SCC(G) \qquad C' = C[(e,t') \mapsto \mathsf{entry}(\mathbf{G}', \rho_f)] \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ \mathsf{con}\ C\ \rightarrow_b\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ \mathsf{con}\ C'}$$

Apply-Seen-Forfeit

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \qquad C(e,t') = \mathsf{entry}(\mathbf{G}, \rho) \qquad \alpha_i = (x_i, t) \\ \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \tilde{v}_i] \qquad i = 1, \ldots, n \qquad G' = \mathsf{SCG}(e, \rho, \rho_f, \sigma_f) \\ \mathbf{G}' = \{G; G' : G \in \mathbf{G}\} \qquad \exists G \in \mathbf{G}'.\neg SCC(G) \qquad C' = C[(e,t') \mapsto \top] \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ \mathsf{con}\ C\ \rightarrow_b\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ \mathsf{abs}\ C'}$$

Apply-Abstract-Explicit

$$\frac{\begin{array}{c} t' = \lfloor t \rfloor_m \\ C(e,t') = \top \qquad \alpha_i = (x_i, t') \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \lceil \tilde{v}_i \rceil^{\mathsf{abs}}] \qquad i = 1, \ldots, n \end{array}}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ \mathsf{con}\ C\ \rightarrow_b\ e\ \rho_n\ \tilde{\sigma}_n\ t'\ \mathsf{abs}\ C}$$

Apply-Abstract-Implicit

$$\frac{t' = \lfloor t \rfloor_m \qquad \alpha_i = (x_i, t) \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \tilde{\sigma}_i = \tilde{\sigma}_{i-1}[\alpha_i \tilde{\mapsto} \tilde{v}_i] \qquad i = 1, \ldots, n}{\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho_0)\ \langle \tilde{v}_1, \ldots, \tilde{v}_n \rangle\ \tilde{\sigma}_0\ t\ \mathsf{abs}\ C\ \rightarrow_b\ e\ \rho_f\ \tilde{\sigma}_f\ t'\ \mathsf{abs}\ C}$$

When a particular signature is first encountered at a call, the Apply-Not-Seen rule inserts an initial entry with a unit SCG and the configuration's environment. When a signature is re-encountered and has not violated the size change condition, the Apply-Seen-Preserve and Apply-Seen-Forfeit rules come into play. If, once the SCG of the self loop is calculated and composed with the previous SCGs in the path, each such SCG obeys the size change condition, the Apply-Seen-Preserve rule allows evaluation to continue in concrete mode. If, however, one of the SCGs violates the size change condition, the Apply-Seen-Forfeit rule causes evaluation to become abstract at this signature for the remainder of the path. The Apply-Abstract-Explicit rule governs the transition when the size change condition at this signature has been violated previously in the path. The Apply-Abstract-Implicit rule preserves incoming abstract evaluation across the call transition, but does not update the cache.

We first define an ordering on pairs of call caches and their closing stores.

$$\begin{aligned}(C_0, \tilde{\sigma}_0) < (C_1, \tilde{\sigma}_1) &\iff \forall (e, \hat{t}) \in \widetilde{\mathit{Signature}}.(C_0(e,t), \tilde{\sigma}_0) < (C_1(e,t), \tilde{\sigma}_1) \\ &\iff \tilde{\sigma}_0 \sqsubseteq \tilde{\sigma}_1 \wedge \{G; \mathsf{SCG}(e, \rho_0, \rho_1, \sigma_1) : G \in \mathbf{G}_0\} \sqsubseteq \mathbf{G}_1 \\ \mathbf{G}_0 \sqsubseteq \mathbf{G}_1 &\iff \forall G_0 \in \mathbf{G}_0.\exists G_1 \in \mathbf{G}_1.G_0 \sqsubseteq G_1\end{aligned}$$

This ordering is the same as the refinement relation except for $\mathsf{entry}(\mathbf{G}, \rho)$ entries. An entry $\mathsf{entry}(\mathbf{G}_0, \rho_0)$ is less than an entry $\mathsf{entry}(\mathbf{G}_1, \rho_1)$ if and only if composing each SCG in $\mathbf{G}_0$ with the size-change graph $\mathsf{SCG}(e, \rho_0, \rho_1, \sigma_1)$ yields a graph which refines one in $\mathbf{G}_1$. The graph $\mathsf{SCG}(e, \rho_0, \rho_1, \sigma_1)$ reflects the descent between the environment values. The idea is that the composition of G_0 with this graph produces a lower bound for the size-change summary of the path up to C_1. The condition that $\tilde{\sigma}_0$ refines $\tilde{\sigma}_1$ allows us to safely use $\tilde{\sigma}_1$ in the size change calculation and is automatically satisfied by evaluation.

Lemma 7.3. *The ordering $<$ on call caches has no infinite ascending chains.*

Since the domain of call caches is finite, it suffices to ensure that its codomain, entries, is free of infinite ascending chains. Any infinite ascending chain has a suffix $\mathsf{entry}(\mathbf{G}_1, \rho_1) < \mathsf{entry}(\mathbf{G}_2, \rho_2) < \ldots$. But this cannot exist since we assume a well-founded ordering on values, so, by the SCT principle, we cannot indefinitely choose ρ such that the size-change graph is not disturbed.

Lemma 7.4. *If proc $\langle \tilde{v}_0, \ldots, \tilde{v}_n \rangle$ $\tilde{\sigma}$ t M C $\rightarrow_{apply}$ e' ρ' $\tilde{\sigma}'$ t' M' C' then either $(C, \tilde{\sigma}) < (C', \tilde{\sigma}')$, $\tilde{\sigma} \sqsubset \tilde{\sigma}'$, or e' ρ' $\tilde{\sigma}'$ t' M' C' is a previously-seen configuration on this path.*

We have several cases:

(1) $C(e', \lfloor t' \rfloor_m) = \bot$, i.e., this signature has not been encountered on this path; or
(2) $C(e', \lfloor t' \rfloor_m) = \mathsf{entry}(\mathbf{G}, \rho_0)$ and
   (a) it does not detect a size-change violation; or
   (b) it does detect a size-change violation; or
   (c) $M = \mathsf{abs}$, i.e., evaluation is in abstract mode already; or
(3) $M' = \mathsf{abs}$, i.e., the entry evaluation is in abstract mode already; and
   (a) $C(e', \lfloor t' \rfloor_m) = \top$, i.e., the call cache already insisted on abstract mode; or
   (b) $C(e', \lfloor t' \rfloor_m) \neq \top$, i.e., it didn't.

In each case except for (2c) and (3a), the call cache increases. In cases (2c) and (3a), $M' = \mathsf{abs}$ so each binding was made in the store. If $\tilde{\sigma} \sqsubset \tilde{\sigma}'$, then we have our result. Otherwise, we have $C = C$ and $\tilde{\sigma} = \tilde{\sigma}'$. Because the store didn't increase, $C(e', \lfloor t' \rfloor_m) = \top$, implying that a call to a configuration with signature $(e', \lfloor t' \rfloor_m)$ must have been made previously in abstract mode on this path with the same environment. Hence, evaluation has seen this configuration previously on this path and it will be intercepted by the caching algorithm.

$$\textsc{Equality-Subsumption}\quad \frac{\tilde{v}_1 =_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{v}_0}{\tilde{v}_1 \preceq_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{v}_0}$$

$$\textsc{Inequal-Concrete-Concrete}\quad \frac{v_1 \prec_{\tilde{\sigma}}^{Value} v_0}{\mathsf{con}(v_1) \prec_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{con}(v_0)}$$

$$\textsc{Inequal-Abstract-Concrete}\quad \frac{\forall v_1 \in \gamma(\hat{v}_1).v_1 \prec_{\tilde{\sigma}}^{Value} v_0}{\mathsf{abs}(\hat{v}_1) \prec_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{con}(v_0)}$$

$$\textsc{Inequality-Subsumption}\quad \frac{\tilde{v}_1 \prec_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{v}_0}{\tilde{v}_1 \preceq_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{v}_0}$$

$$\textsc{Inequal-Concrete-Abstract}\quad \frac{\forall v_0 \in \gamma(\hat{v}_0).v_1 \prec_{\tilde{\sigma}}^{Value} v_0}{\mathsf{con}(v_1) \prec_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{abs}(\hat{v}_0)}$$

$$\textsc{Inequal-Abstract-Abstract}\quad \frac{\forall v_0 \in \gamma(\hat{v}_0).\forall v_1 \in \gamma(\hat{v}_1).v_1 \prec_{\tilde{\sigma}}^{Value} v_0}{\mathsf{abs}(\hat{v}_1) \prec_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{abs}(\hat{v}_0)}$$

$$\textsc{Equal-Concrete-Concrete}\quad \frac{v_1 =_{\tilde{\sigma}}^{Value} v_0}{\mathsf{con}(v_1) =_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{con}(v_0)}$$

$$\textsc{Equal-Concrete-Abstract}\quad \frac{\forall v_0 \in \gamma(\hat{v}_0).v_1 =_{\tilde{\sigma}}^{Value} v_0}{\mathsf{con}(v_1) =_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{abs}(\hat{v}_0)}$$

$$\textsc{Equal-Abstract-Concrete}\quad \frac{\forall v_1 \in \gamma(\hat{v}_1).v_1 =_{\tilde{\sigma}}^{Value} v_0}{\mathsf{abs}(\hat{v}_1) =_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{con}(v_0)}$$

$$\textsc{Equal-Abstract-Abstract}\quad \frac{\forall v_0 \in \gamma(\hat{v}_0).\forall v_1 \in \gamma(\hat{v}_1).v_1 =_{\tilde{\sigma}}^{Value} v_0}{\mathsf{abs}(\hat{v}_1) =_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{abs}(\hat{v}_0)}$$

$$\textsc{Equal-Base}\quad \frac{v_0 = v_1}{\mathsf{con}(v_0) =_{\tilde{\sigma}}^{\widetilde{Value}} \mathsf{con}(v_1)}$$

$$\textsc{Const}\quad \frac{c_1 < c_0}{\mathsf{const}(c_1) \prec_{\tilde{\sigma}}^{Value} \mathsf{const}(c_0)}$$

$$\textsc{Equal-Closure}\quad \frac{\forall x \in \mathsf{free}(\lambda(x_1,\ldots,x_n).e).\sigma(\rho_0(x)) =_{\tilde{\sigma}}^{Value} \sigma(\rho_1(x))}{\mathsf{proc}(\mathsf{clos}(\lambda(x_1,\ldots,x_n).e,\rho_0)) =_{\tilde{\sigma}}^{Value} \mathsf{proc}(\mathsf{clos}(\lambda(x_1,\ldots,x_n).e,\rho_1))}$$

$$\textsc{Equal-Variant}\quad \frac{\forall i \in \{1,\ldots,n\}.\sigma(\alpha_i) =_{\tilde{\sigma}}^{Value} \sigma(\beta_i)}{\mathsf{vari}(tag,\langle\alpha_1,\ldots,\alpha_n\rangle) =_{\tilde{\sigma}}^{Value} \mathsf{vari}(tag,\langle\beta_1,\ldots,\beta_n\rangle)}$$

$$\textsc{Closure-Subpart}\quad \frac{\mathsf{con}(v) \preceq_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{\sigma}(\rho(x))}{v \prec_{\tilde{\sigma}}^{Value} \mathsf{proc}(\mathsf{clos}(lam,\rho))}\ x \in \mathsf{free}(lam)$$

$$\textsc{Closure-Field}\quad \frac{\begin{array}{c}\forall x \in \mathsf{free}(lam).\exists y \in \mathsf{free}(lam).\sigma(\rho_0(x)) \preceq_{\tilde{\sigma}}^{Value} \sigma(\rho_1(y)) \\ \tilde{\sigma}(\rho_0(x)) \preceq_{\tilde{\sigma}}^{\widetilde{Value}} \tilde{\sigma}(\rho_1(x)) \qquad \exists y \in \mathsf{free}(lam).\sigma(\rho_0(z)) \prec_{\tilde{\sigma}}^{Value} \sigma(\rho_1(y))\end{array}}{\mathsf{proc}(\mathsf{clos}(lam,\rho_0)) \prec_{\tilde{\sigma}}^{Value} \mathsf{proc}(\mathsf{clos}(lam,\rho_1))}\ z \in \mathsf{free}(lam)$$

$$\textsc{Variant-Field}\quad \frac{\begin{array}{c}\forall i \in \{1,\ldots,n\}.\exists j \in \{1,\ldots,m\}.\sigma(\alpha_i) \preceq_{\tilde{\sigma}}^{Value} \sigma(\beta_j) \\ \exists j \in \{1,\ldots,m\}.\sigma(\alpha_k) \prec_{\tilde{\sigma}}^{Value} \sigma(\beta_j)\end{array}}{\mathsf{vari}(tag,\langle\alpha_1,\ldots,\alpha_n\rangle) \preceq_{\tilde{\sigma}}^{Value} \mathsf{vari}(tag,\langle\beta_1,\ldots,\beta_m\rangle)}\ k \in \{1,\ldots,n\}$$

$$\textsc{Closure-Subexpression}\quad \frac{lam_0 \in \mathsf{subexp}(lam_1) \qquad lam_0 \neq lam_1 \qquad \forall x \in \mathsf{free}(lam_0).\sigma(\rho_0(x)) \preceq_{\tilde{\sigma}}^{Value} \sigma(\rho_1(x))}{\mathsf{proc}(\mathsf{clos}(lam_0,\rho_0)) \prec_{\tilde{\sigma}}^{Value} \mathsf{proc}(\mathsf{clos}(lam_1,\rho_1))}$$

Fig. 10. Value ordering

7.3 Value Ordering

Figure 10 presents three orders on hybrid values: strict inequality, equality, and inequality. Each order

is parameterized by a store $\tilde{\sigma}$. The Inequality-Subsumption and Equality-Subsumption rules deem strict inequality and equality as inequality. Strict inequality and equality on hybrid values is lifted to concrete values by a set of eight rules, reflecting the eight possible combinations of concrete, abstract, and whether equality or inequality. The principle is merely that an abstract ordering holds with respect to an abstract value if the corresponding concrete ordering holds for all concretizations of that value. The Equal-Base rule includes structural equality in the equality judgment. The Equal-Variant and Equal-Closure rules ensure that variants and closures, respectively, with different addresses but the same structure are considered equal. The Const rule lifts a parameter order on constants into the inequality judgment; in our evaluation, we order numeric values by magnitude and string values by length.

A value is less than a variant if it is less than or equal to some field of the variant (Variant-Subpart). Similarly, a value is less than a closure if it is less than or equal to some value in the closure's environment (Closure-Subpart). For two variants that share a tag, one is less than another if each of its fields is less than equal to a field of the other and at least one field is strictly less than one of the other (Variant-Field). Similarly, for two closures over the same *lam*, one is less than another if each binding in one is less than or equal to a binding in the other and at least one field is strictly less than one of the other (Closure-Field). Finally, we include an ordering on closures where a closure over a subexpression is less than another closure if they agree on the bindings of the subexpression [17].

8 Empirical Evaluation

We deployed CGADI on two sets of benchmarks: a standard set drawn from the modern control-flow analysis literature [6, 12] and a subset of the Larceny R6RS benchmark suite which has been used as a static analysis benchmark. We do not report on programs with mutation nor larger programs designed to provoke the exponential behavior of k-CFA for $k > 0$. (For the latter, CGADI is able to handle them completely concretely and with fewer states.)

The evaluation code resides at the following repository.

https://github.com/byu-static-analysis-lab/cg-sct-cfa

8.1 Control-flow Analysis Benchmarks

We break these benchmarks up into three categories.

8.1.1 Pathological Benchmarks. These programs were crafted specifically to provoke the weaknesses of contemporary static analyses.

`facehugger` contains binding patterns which cause static analyses to conflate values and waste effort on spurious paths

`blur` contains binding patterns which lead static analyses to confuse types

`eta` contains binding patterns which cause static analyses to conflate base values

`kcfa2`, **`kcfa3`** synthesized to provoke worst-case behavior for k-CFA

8.1.2 Data and Control. These programs exhibit a mix of data and control encodings emblematic of modern idiomatic code.

`mj09` contains a mix of direct style and continuation-passing style code exercising the data–control interpretation

`sat-brute` a brute-force SAT solver parameterized by a functional formula

`map-pattern` contains applications of `map` on static inputs

`regex-derivative` an implementation of a derivative-based regular expression matcher [2]

8.1.3 Number Theory. These programs implement simple cryptosystems and the functionality that supports them.
`fermat` an implementation of the Fermat primality test
`solovay-strassen` an implementation of the Solovay-Strassen primality test
`rsa` an implementation of the RSA algorithm with static inputs
`rsa-abstract-plaintext` an implementation of the RSA algorithm with static exponents and a dynamic (unknown) plaintext

8.1.4 Comparison. We compare the performance of a standard stack-based[2] k-CFA and a hybridized stack-based k-CFA using a size-change call guard. We instantiate each analysis for $k = 0$ and $k = 1$ obtaining standard (albeit stack-based) 0CFA and 1CFA and hybridized 0CGADI and 1CGADI, respectively, each of which exhibits perfect stack precision as offered by the ADI framework. The abstract semantics for each analysis is tuned for maximum precision by using domains that can express concrete values and the "crush" store extension technique [4] which preserves precision through primitive operations. Each analysis was deployed with a path- and flow-insensitive global store.

In conservative static analysis, there is an inherent tension between precision and scalability. On one hand, results with higher precision are more actionable; on the other, maintaining precision increases the size of the explored execution space, which takes more time to complete. With this in mind, we compare the number of precise results and the size of the execution space explored by each analysis. For the purposes of this evaluation, a *precise* result is either a concrete result or an abstract result expressing a concrete value (a "singleton abstraction"). The pure abstract semantics may produce only the latter of course, but the hybrid semantics may produce both, and we break down results as concrete or singleton abstract.

Figure 11 presents a comparison of the number of precise results across analyses and programs. A *Singleton Abstract/Abstract* value is a precise abstract value produced by abstract evaluation of a configuration (the default in the pure abstract semantics and a possibility in the hybrid semantics); a *Singleton Abstract/Concrete* value is a precise abstract value produced by an (attempted) concrete evaluation of a configuration, possible only in the hybrid semantics; a *Concrete* value is of course a concrete value produced by the concrete evaluation of a configuration in the hybrid semantics. The histograms chart the number of precise values obtained by each analysis on each program with CFA on the left and CGADI on the right. 0CFA timed out on `regex-derivative` and 1CFA timed out on `sat-brute`, `regex-derivative`, and `solovay-strassen`; 1CGADI timed out on `solovay-strassen`. In most cases, CGADI produces a higher number of precise values—in some cases, much higher (note the log scale of the count axis)—and typically is able to produce concrete values. A crucial fact is that, because CGADI degenerates to CFA, CGADI is always at least as precise as CFA. Consequently, in the cases where the CFA produces a higher number of precise values than the corresponding CGADI, we can verify (in the next comparison) that it has encountered more configurations, demonstrating that it has needlessly multiplied the execution space.

Figure 12 presents a comparison of the number of configurations encountered across analyses and programs. The histograms chart the number of configurations in a 0CFA/0CGADI analysis (above) and a 1CFA/1CGADI (below). Golden regions indicate configurations evaluated in abstract mode and blue those evaluated in concrete mode. Hence, the solid gold bars on the left correspond to the entirely-abstract 0CFA/1CFA. On the programs crafted to exercise static analyzers, CGADI was able to maintain concrete evaluation for the entire evaluation of the program. The `facehugger` program is crafted to cause analyses to reuse addresses unless they employ abstract garbage collection [24].

[2]Stack-based k-CFA does not record the k-most-recent call sites but rather the top-k call sites on the stack, just as m-CFA [25] does. However, it retains the nested environment structure and binding behavior of k-CFA, contra m-CFA.

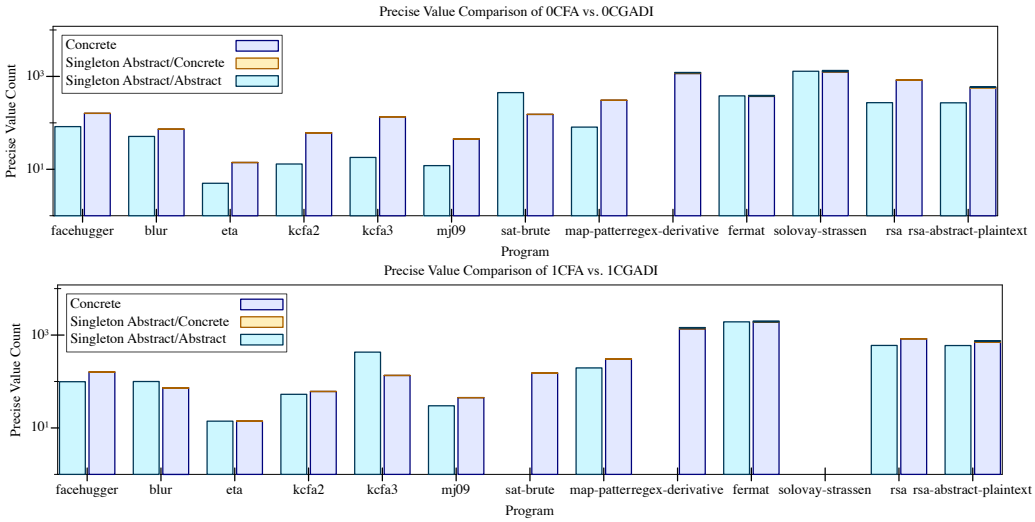


Fig. 11. Each graph presents a comparison between the number of precise results produced by CFA and a corresponding CGADI, broken down by program. The top graph presents the results for a context-insensitive analysis ($k = 0$) and the bottom for a context-sensitive analysis ($k = 1$). Precise results are concrete or singleton abstract from a concrete evaluation (both which only CGADI produces) or singleton abstract from an abstract evaluation (which CFA only produces).

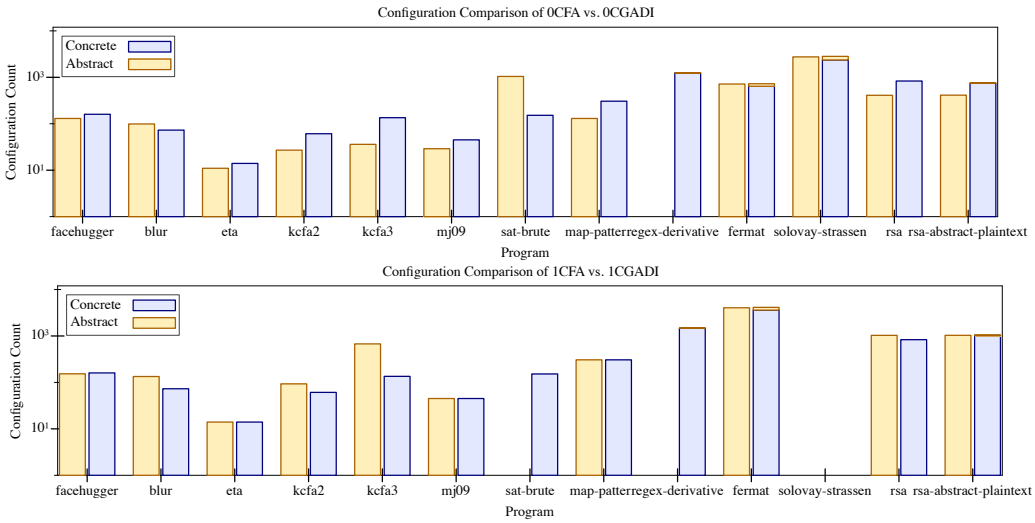


Fig. 12. Each graph presents a comparison between the number of configurations encountered by CFA and a corresponding CGADI, broken down by program. The top graph presents the results for a context-insensitive analysis ($k = 0$) and the bottom for a context-sensitive analysis ($k = 1$). A configuration is encountered in (and therefore evaluated in) concrete mode (in CGADI only) or abstract mode.

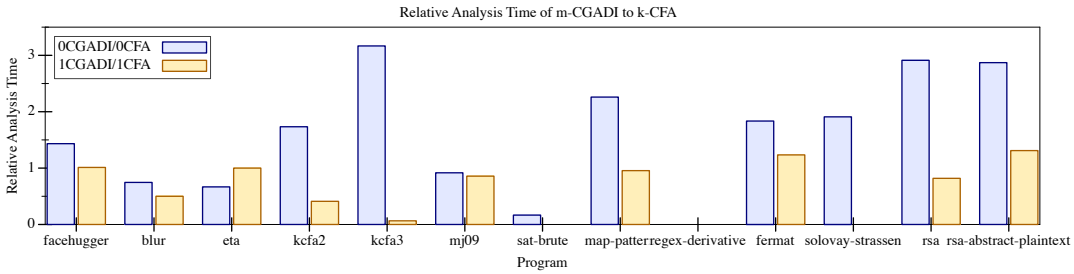


Fig. 13. This graph presents a relative comparison of running times for CFA and a corresponding CGADI. Blue bars graph the ratio of 0CGADI's running time to 0CFA's are gold bars graph that of 1CGADI's to 1CFA's. The time axis has a linear scale.

Because concrete evaluation in CGADI exhibits perfect binding precision, it is able to avoid address reuse. Morever, it is able to avoid reuse *despite the fact that its store is global*, a setting in which abstract garbage collection is completely ineffective. The `map-pattern` program includes maps over static lists. CGADI is able to build up the resulting lists with perfect fidelity as the input list decreases in size each recursive call, demonstrating the effectiveness of the size-change call guard. The `regex-derivative` program applies static patterns first to several static inputs and then to a dynamic input. Despite the dynamic input, CGADI is able to maintain concrete evaluation for the vast majority of the program. With respect to the raw number of states, 0CGADI exhibits about the same as or more states than 0CFA in general and 1CGADI exhibits about the same or less. As the number of states does not change for the programs on which CGADI remained entirely concrete, this flip is due to a mild state space explosion common with the addition of context sensitivity.

We ran the analysis with a timeout for each program run of two minutes, which is an order of magnitude longer than the longest successful analysis within the benchmark programs. Figure 13 presents a comparison of the running time across analyses and programs. The histograms chart the relative running time of 0CGADI to 0CFA (dark blue) and 1CGADI to 1CFA (gold). Compared times are the average of 30 runs of the analysis on each program. In most cases, 0CGADI's analysis times are higher than 0CFA's, being just over 3× higher at most. However, most of 1CGADI's analysis times are about at or lower than 1CFA's. Although not reflected in this figure, the memory and time overhead of call guards is negligible.

8.2 Larceny R6RS Benchmarks

We dispatched CGADI on a subset of the Larceny R6RS benchmark suite including `tak`, `cpstak`, `ack`, and others. 0CGADI was able to maintain concrete evaluation for the entire execution of every program except `primtest` which performs a call to `random` and therefore exhibits nondeterminism. As Scheme implementation benchmark programs, these programs are by and large deterministic and those that aren't, such as `primtest`, have extremely similar execution profiles across executions. Nevertheless, many static analyses use these programs as analysis benchmarks and on which recover much more (i.e. less-precise) behavior than single concrete execution. We see the fact that 0CGADI performs as a concrete interpreter as evidence of the sheer effectiveness of the size-change termination principle and the sensibility of its integration into the static analysis.

9 Related Work

This work is formulated in the framework of *Abstract Definitional Interpreters* [4] which itself derives from the seminal *Abstracting Abstract Machines* methodology of Van Horn and Might [32].

This work borrows the SCT machinery from Nguyen et al. [27] which uses dynamic size-change monitoring to conservatively detect when a particular function has entered an infinite sequence. Nguyen et al. [27] then abstract the semantics to obtain a static analysis which can reason about termination statically which expressly does not have to have SCT knowledge built-in. We repurpose their dynamic monitoring in a static setting and integrate SCT deeply within the semantics; the static setting also allows us to know more about called functions, which keeps SCT monitoring sound.

Toman and Grossman [31] present a framework for combining concrete and abstract interpretation aimed at analyzing programs themselves written in frameworks. Their work is based on the *state separation* hypothesis that framework-based programs partition both data/execution into framework-specific and program-specific segments and that data/execution within the framework segment is mostly concrete. Our work is motivated by similar goals and observations as Toman and Grossman [31]'s—that real programs execute in different phases and that some admit essentially concrete execution. However, the phase separation in their work is driven by the state separation hypothesis in which one region of the code is designated as concrete and the other as abstract. Our approach is more fine-grained, both statically and dynamically, and allows the same code to be evaluated in different modes at different analysis times. We intend to explore the efficacy of our approach on framework-based applications more directly in future work.

Another related work is the recent trace-based CFA [26] which derives a semantic account of constraint-based k-CFA. Like us, it recognizes the reliance constraint- and abstract machine-based formulations of CFA have on finite state spaces to ensure termination. In response, it proposes a new widening technique to permit infinite domains. It realizes this widening technique in ∇CFA which widens the output of evaluation when it detects a cycle and widens the input at the request of the analysis client. ∇CFA is thus similar to a CGADI using a reentrant call guard. With the size change call guard, CGADI is able to continue concrete evaluation even into recursive calls so long as the size-change condition is met. While we have not described it this way, the degeneration from concrete to abstract mode effects a kind of widening in the semantics, in which precision residing in an infinite space is abandoned to ensure (or accelerate) convergence.

Might and Manolios [23] show that an abstract interpretation's resource allocation policy may adapt to the conditions of the analysis itself (e.g. to preserve precision) and be proven sound *a posteriori*. Our work embodies a specific instance of their general result in that the concrete/abstract nature of a resource's allocation adapts to the advice of the call guard. Jenkins et al. [13] present a Galois unions, a framework in which one can realize semantics with varieties of precision in a lattice that the analysis ascends as it operates. At the bottom of this lattice resides the concrete semantics where the analysis begins. Galois unions are presented in an AAM-style small-step setting, and offer only a fixed number of steps as a transition threshold from concrete to abstract. Moreover, this work offers no means for the analysis to descend the lattice of semantics. It is possible to cast the concrete–abstract mode distinction in CGADI as a nearly trivial lattice of semantics. However, by utilizing size change information, CGADI offers a principled and dynamic transition criterion and, by working in a big-step setting, can descend the semantic lattice as well to recover precision. We consider it future work to generalize CGADI to richer lattices of semantics and to develop coherent critera to traverse them.

In object-oriented settings, there has recently been an explosion of work devoted to making more intelligent use of context, utilizing it only when needed to preserve precision. Wei and Ryder [37] present an adaptive analysis for JavaScript which applies an inexpensive points-to analysis to each function to determine the specific context-sensitive analysis it should apply in a subsequent stage. Jeong et al. [15] apply a data-driven learning algorithm to determine the sensitivity level to permit at each program site, up to a fixed size k, for call-site, object, and type sensitivity. In follow-on

work, Jeon et al. [14] use a similar data-driven approach to make more efficient use of the context in a k-limited analysi by intelligently selecting which k parts of the context should be recorded (instead of the k-most-recent), which they refer to as *tunneling*. Li et al. [21] present the Zipper algorithm to achieve selective context sensitivity which uses a principled approach to identify flow patterns that give rise to analysis imprecision. Eagle [22] applies context sensitivity selectively at the granularity of variables and allocation sites, using the results of a program pre-analysis to determine sensitivity levels. All of these works acknowledge and address the difficulty of selecting the appropriate pieces of context, which we see as analogous to the difficulty of knowing when to proactively abstract in our framework. Each of these works yields an analysis rooted in an abstract semantics and so it is apparently possible to apply call guards as presented in this work to obtain a hybridized analysis of concrete and abstract semantics.

10 Conclusion and Future Work

We have presented CGADI, an ADI-based framework which integrates concrete and abstract execution, which governs the transition from the former to the latter with a parameterizing call guard. We have presented two examples of call guards: a reentrant call guard which accounts for existing work within the framework, and a novel size-change call guard which uses the size-change principle of program termination to permit concrete execution to persist on paths that exhibit strict monotonic descent among values according to a specified metric.

To provide a reasonable and reliable complexity bound, future work may elaborate call guards to make explicit a *proactive* policy to transition to abstract execution, as well as explore the policy space to identify features of effective policies.

Modern languages feature rich control constructs which go beyond the simple call–return behavior captured by stack-precise models. Future work may extend the ADI framework to support such constructs, perhaps by supporting `call/cc` to provide support once and for all, as Vardoulakis and Shivers [34] do for CFA2, and by the same means. Once extended, subsequent future work may integrate the approach into the hybrid setting, in particular working out the interplay between a concrete and abstract treatment of non-local control.

A Computing the Abstract Semantics

An analysis of a program pr is a pair $(\$, R) : [\widehat{Config} \to \mathcal{P}(\widehat{Result})] \times \mathcal{P}(\widehat{Config})$ such that R contains the configurations reachable from the initial configuration $\hat{\mathcal{I}}(pr) = pr \perp \perp \langle\rangle$. and $\$$ is a *cache* mapping reachable configurations to their results. We say that $(\$, R) \models pr$ if and only if $\hat{\mathcal{I}}(pr) \hat{\Rightarrow} \hat{\varsigma} \implies \hat{\varsigma} \in R$ and $\hat{\mathcal{I}}(pr) \hat{\Rightarrow} \hat{\varsigma} \wedge \hat{\varsigma} \hat{\Downarrow} \widehat{rslt} \implies \widehat{rslt} \in \$$.

The *best analysis* $(\$^+, R^+)$ is defined as the least fixed point of the functional

$$\mathcal{F} = \lambda(\$, R). \bigsqcup_{\hat{\varsigma} \in R} (\{\hat{\varsigma} \mapsto \widehat{rslt} : \hat{\varsigma} \hat{\Downarrow}^{\$} \widehat{rslt}\}, \{\hat{\varsigma}' : \hat{\varsigma} \hat{\rightarrow}^{\$} \hat{\varsigma}'\})$$

where the initial configuration $\hat{\mathcal{I}}(pr)$ is assumed reachable. In other words, $(\$^+, R^+)$ is defined

$$(\$^+, R^+) = lfp(\lambda(\$, R).\mathcal{F}(\$, R) \sqcup (\perp, \{\hat{\mathcal{I}}(pr)\}))$$

In the definition of $\mathcal{F}$, the relations $\hat{\Downarrow}^{\$}$ and $\hat{\Rightarrow}^{\$}$ are the relations $\hat{\Downarrow}$ and $\hat{\Rightarrow}$ except that recursive references in the definition appeal directly to $(\$, R)$. Using these relations pulls all recursion outside of $\mathcal{F}$, bringing it into the view of the analysis's fixed point search.

To connect the functional $\mathcal{F}$ with the evaluation and reachability semantics, Darais [5] proves the following theorem:

Theorem A.1 (Algorithm Correctness [5]). *The analysis* $(\$^+, R^+)$ *is valid for program pr, i.e.,* $(\$^+, R^+) \models pr$.

The space of analyses is finite and $\mathcal{F}$ is monotonic, so the least fixed point can be computed by Kleene iteration. A convenient means to express and compute this analysis is an abstract definitional interpreter [4, 36] and our implementation, used to perform the evaluation in §8, takes this form.

B Soundness and Computability

We are now obliged to show that instantiating CGADI produces a sound and computable analysis. Our soundness argument will be standard. Our computability argument will not be, however: whereas typical computability arguments merely to the finiteness of the execution space, we appeal to the finiteness of paths.

B.1 Soundness

A sound analysis is a trustworthy analysis; a statement of soundness says that any behavior of the concrete semantics is simulated by the abstract semantics. Before we can correspond behaviors, however, we must correspond state spaces.

B.1.1 Abstraction. We define a family of abstraction functions $|\cdot|_X$ which map from the concrete to the abstract state space.

$$|e\ \rho\ \sigma\ t| = e\ \rho\ |\sigma|\ t\ \mathsf{con}\ \bot \qquad |v\ \sigma| = \mathsf{con}(v)\ |\sigma| \qquad |\sigma| = \lambda\alpha.\mathsf{con}(\sigma(\alpha))$$

A configuration abstracts componentwise; expressions are untouched; the store, mode, and call cache—not present in concrete configurations—take on their most-refined values (cf. the next section). A stores is "abstracted" to abstract each value within. A value is "abstracted" by tagging it as concrete. Abstracting a concrete configuration introduces a concrete mode and the most-refined call cache.

B.1.2 Refinement. We now define a family of refinement relations $\sqsubseteq$ which establish the relative precision between abstract components. Abstract components which may contain store references (addresses) are refined in the context of a store.

Figure 14 presents the family of refinement relations. Refinement of configurations and results lifts componentwise. Environments refine if their entries refine. Values within environments follow the concrete value refinement. Addresses within environments are looked up in the store and then follow the abstract value refinement. A value in an environment refines an address if the value abstracted refines the abstract value in the store. Refinement of denotable values follows the same pattern, but with no store lookup. Concrete closure follows environment refinement. Concrete constants and primitives refine with equality. Abstract values (a product of a base abstraction and a powerset of procedures) refine componentwise. The base refinement is inherited from the base abstraction and is a property of the host CFA. The abstract procedure refinement requires that every procedure in the refining set has a counterpart in the refined set which it refines. Concrete evaluation mode refines abstract evaluation mode. Call caches are refined as maps. SCGs are refined as maps according to $\widetilde{Order}$ refinement given in §7.1.

B.2 Evaluation Soundness

Having established an abstraction between the state spaces, and a refinement relation between abstract components, we move to showing that the abstract semantics simulates the concrete. Our strategy is to show that abstract evaluation and call preserve refinement; simulation quickly follows.

Configurations

$$e\ \rho_0\ \tilde{\sigma}_0\ t_0\ M_0\ C_0 \sqsubseteq e\ \rho_1\ \tilde{\sigma}_1\ t_1\ M_1\ C_1$$
$$\iff$$
$$\rho_0 \sqsubseteq \rho_1 \wedge \tilde{\sigma}_0 \sqsubseteq \tilde{\sigma}_1 \wedge M_0 \sqsubseteq M_1 \wedge (C_0, \tilde{\sigma}_0) \sqsubseteq (C_1, \tilde{\sigma}_1)$$

$$proc_0\ \langle \tilde{v}_{0,1}, \ldots, \tilde{v}_{0,n} \rangle\ \tilde{\sigma}\ t\ M_0\ C_0 \sqsubseteq proc_1\ \langle \tilde{v}_{1,1}, \ldots, \tilde{v}_{1,n} \rangle\ \tilde{\sigma}_1\ t_1\ M_1\ C_1$$
$$\iff$$
$$proc_0 \sqsubseteq proc_1 \wedge \langle \tilde{v}_{0,1}, \ldots, \tilde{v}_{0,n} \rangle \sqsubseteq \langle \tilde{v}_{1,1}, \ldots, \tilde{v}_{1,n} \rangle \wedge \tilde{\sigma}_0 \sqsubseteq \tilde{\sigma}_1 \wedge M_0 \sqsubseteq M_1 \wedge (C_0, \tilde{\sigma}_0) \sqsubseteq (C_1, \tilde{\sigma}_1)$$

Results

$$\tilde{v}_0\ \tilde{\sigma}_0 \sqsubseteq \tilde{v}_1\ \tilde{\sigma}_1 \iff \tilde{v}_0 \sqsubseteq \tilde{v}_1 \wedge \tilde{\sigma}_0 \sqsubseteq \tilde{\sigma}_1$$

Environments

$$\rho_0 \sqsubseteq \rho_1 \iff \forall x \in \mathsf{dom}(\rho_0).\rho_0(x) \sqsubseteq \rho_1(x)$$
$$(x, t_0) \sqsubseteq (x, t_1) \iff t_0 \sqsubseteq t_1$$

Times

$$t_0 \sqsubseteq t_1 \iff t_1 \text{ is a prefix of } t_0$$

Fig. 14. Abstract domain refinement

Lemma B.1 (Evaluation preserves refinement). *If $\widetilde{cfg} \sqsubseteq \widetilde{cfg}'$ and $\widetilde{cfg} \tilde{\Downarrow} \widehat{rslt}$ then $\widetilde{cfg}' \tilde{\Downarrow} \widehat{rslt}'$ where $\widehat{rslt} \sqsubseteq \widehat{rslt}'$.*

The preceding two lemmas are proven by mutual induction on the refining derivations. The proof also relies on metafunctions preserving refinement, such as bind:

With these lemmas in hand, we can prove the simulation theorem:

Theorem B.2 (Simulation). *If $|e\ \rho\ \sigma\ t| \sqsubseteq e\ \rho\ \tilde{\sigma}\ t\ M\ C$ and $e\ \rho\ \sigma\ t \Downarrow v\ \sigma'$ then $e\ \rho\ \tilde{\sigma}\ t\ M\ C \tilde{\Downarrow} \tilde{v}\ \tilde{\sigma}'$ where $|v\ \sigma'| \sqsubseteq \tilde{v}\ \tilde{\sigma}'$.*

Proven by appeal to the refinement of evaluation and the definition of abstraction (which produces the most-refined configurations and results).

The hybrid semantics is deterministic in the disposition of its result. That is, although a configuration may have multiple derivations, no configuration yields both a concrete and an abstract value within its result. Moreover, when a concrete value is produced, the hybrid semantics reflect the determinism of the concrete semantics, so that the two values are identical. This property is captured by the following theorem.

Theorem B.3. *If $\widetilde{cfg} \tilde{\Downarrow} \tilde{v}_0\ \tilde{\sigma}_0$ and $\widetilde{cfg} \tilde{\Downarrow} \tilde{v}_1\ \tilde{\sigma}_1$, then $\tilde{v}_0 = \mathsf{con}(v)$ if and only if $\tilde{v}_1 = \mathsf{con}(v)$, for some concrete value v.*

B.3 Reasoning about Nontermination Within Programs

We have defined our semantics in big-step style which does not provide facilities to reason about termination. Instead, a terminating evaluation is evidenced by a finite derivation that the configuration in question yields the purported result, and a nonterminating evaluation simply has no derivation at all.

$$\textsc{Bound-Reach}\quad \frac{}{\textbf{let}\, x \;=\; ce\, \textbf{in}\, e\, \rho\, \sigma\, t \;\Rightarrow\; ce\, \rho\, \sigma\, t} \qquad \textsc{Body-Reach}\quad \frac{ce\, \rho\, \sigma\, t \;\Downarrow_c\; v\, \sigma' \qquad \alpha = (x, t)}{\textbf{let}\, x \;=\; ce\, \textbf{in}\, e\, \rho\, \sigma\, t \;\Rightarrow\; e\, \rho[x \mapsto \alpha]\, \sigma'[\alpha \mapsto v]\, t}$$

$$\textsc{CE-Application-Reach}\quad \frac{\begin{array}{c}\mathcal{A}(ae, \rho, \sigma) = \mathsf{proc}(\mathsf{clos}(\lambda(x_1, \ldots, x_n).e, \rho)) \qquad \mathcal{A}(ae_i, \rho, \sigma) = v_i \\ \alpha_i = (x_i, t') \qquad \rho_i = \rho_{i-1}[x_i \mapsto \alpha_i] \qquad \sigma_i = \sigma_{i-1}[\alpha_i \mapsto v_i] \qquad i = 1, \ldots, n \qquad t' = \ell :: t\end{array}}{ae(ae_1, \ldots, ae_n)^\ell\, \rho\, \sigma\, t \;\Rightarrow\; e\, \rho_n\, \sigma_n\, t'}$$

$$\textsc{CE-Case-Reach}\quad \frac{\begin{array}{c}\mathcal{A}(ae, \rho, \sigma) = \mathsf{vari}(tag_i, \langle \alpha_1, \ldots, \alpha_{m_i} \rangle) \\ \beta_j = (x_{i,j}, t) \qquad \sigma_1 = \sigma[\beta_j \mapsto \sigma(\alpha_j)] \qquad \rho_1 = \rho[x_{i,j} \mapsto \beta_j] \qquad j = 1, \ldots, m_i\end{array}}{\textbf{case}\, ae\, \textbf{of}\, tag_1(x_{1,1}, \ldots, x_{1,m_1}) \Rightarrow e_1\, ;\, \ldots\, ;\, tag_n(x_{n,1}, \ldots, x_{n,m_n}) \Rightarrow e_n\, \textbf{end}\, \rho\, \sigma\, t \;\Rightarrow\; e_i\, \rho_1\, \sigma_1\, t}$$

Fig. 15. The reachability relation

To gain the ability to reason about termination, we follow the approach of Jones and Bohr [17] to define a notion of reachability between configurations. The notion we define expresses that, if cfg_0 reaches cfg_1, then the termination of cfg_0's evaluation depends on the termination of cfg_1. It captures the idea that, if one were writing the derivation of cfg_0 by hand, one would encounter the derivation of cfg_1. This encounter is meaningful whether or not the derivation of cfg_0 converges, or the derivation of cfg_1, for that matter.

We define the reachability relation $\Rightarrow$ as a judgment $e\, \rho\, \sigma\, t \;\Rightarrow\; e'\, \rho'\, \sigma'\, t'$ in Figure 15. A **let** expression configuration reaches a configuration focused on its bound expression (Bound-Reach). If that configuration evaluates to a result, the overall **let** configuration reaches a configuration focused on its body with the environment and store appropriately extended (Body-Reach). An application call expression configuration reaches the entry configuration of the call if the operator evaluates to a closure (CE-Application-Reach). A case call expression configuration reaches the configuration of the body of the selected clause (CE-Case-Reach).

Using this relation, we define the *configuration tree* $\mathcal{R}_{pr}$ of a program pr is the smallest set of configurations containing $\mathcal{I}(pr)$ that is closed under $\Rightarrow$, which is not necessarily finite. Whether or not the configuration tree is finite, an evaluation converges if and only if all of its evaluation paths converge. In short, nontermination is a path-based phenomenon, as captured by the following lemma.

Lemma B.4 (Nontermination is Sequential [17]). *For a program pr, $\mathcal{I}(pr) \Downarrow rslt$ for result $rslt$ if and only if $\mathcal{R}_{pr}$ has no infinite call chain $\mathcal{I}(pr) \Rightarrow cfg_1 \Rightarrow cfg_2 \Rightarrow \ldots$.*

This lemma provides us with a strategy to ensure the convergence of the analysis: if the analysis can guarantee that the evaluation along the path of a configuration chain is finite, it can simply allow it to converge; if, however, it cannot make that guarantee, it can apply abstraction to ensure that only a finite number of states occur on the path. The applied abstraction ensures that the path behavior is overapproximated by a finite number of states while the caching algorithm ensures that the analysis does not revisit any given state without bound.

B.4 Computability

CGADI is computable if it encounters a finite number of configurations. This we establish using Koñig's Lemma, which states that a rooted, finitely-branching tree which is infinite must have an

infinite path. The judgments for the evaluation and call relation define just such a tree, rooted at an initial configuration, and whose branches consist of evaluation paths—sequences of configurations guarded by a call cache. CGADI evaluation itself occurs in the context of an ADI-style caching algorithm which intercepts the evaluation of configurations already encountered on a path. Thus, we can show that no evaluation path is infinite by showing that evaluation can encounter only a finite number of configurations on it.

The proof relies on an ordering on call cache–store pairs which has no infinite ascending chains, which allows us to show that evaluation produces a monotonically-increasing sequence of call caches on an evaluation. We can then show that, where abstract transitions don't strictly increase the call cache, they increase the store or must reach a previously-seen configuration.

Lemma B.5. *Each evaluation path has only a finite number of distinct configurations.*

By Lemma 7.4, the call cache or store increases or the evaluation path starts repeating. By Lemma 7.3, there cannot be an infinite ascending sequence of call caches for a finite program. Thus, there is a finite number of increases before the evaluation path starts repeating.

Lemma B.6 (Computability). *From any configuration $\widetilde{cfg}$, only a finite number of distinct states are reachable.*

By Lemma B.5 and an application of König's Lemma.

Theorem B.7 (Hybrid Evaluation Computability). *If the call cache has no infinite ascending chains and the call cache–store product increases on every novel call, then, for any program pr, only a finite number of distinct states are reachable from $\tilde{\mathcal{I}}(pr)$.*

References

[1] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Isabel Garcia-Contreras, and Dusko Pavlovic. 2020. Abstract extensionality: on the properties of incomplete abstract interpretations. *Proc. ACM Program. Lang.* 4, POPL (2020), 28:1–28:28. https://doi.org/10.1145/3371096

[2] Janusz A. Brzozowski. 1964. Derivatives of Regular Expressions. *Journal of the ACM (JACM)* 11, 4 (Oct. 1964), 481–494. https://doi.org/10.1145/321239.321249

[3] David R. Chase, Mark N. Wegman, and F. Kenneth Zadeck. 1990. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation (PLDI), White Plains, New York, USA, June 20-22, 1990*, Bernard N. Fischer (Ed.). ACM, 296–310. https://doi.org/10.1145/93542.93585

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

[5] David Charles Darais. 2017. *Mechanizing Abstract Interpretation.* Ph. D. Dissertation. University of Maryland, College Park, MD, USA. https://doi.org/10.13016/M2J96097D

[6] Christopher Earl, Ilya Sergey, Matthew Might, and David Van Horn. 2012. Introspective pushdown analysis of higher-order programs. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming* (Copenhagen, Denmark) *(ICFP '12)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2364527.2364576

[7] Matthias Felleisen and Daniel P. Friedman. 1987. A Calculus for Assignments in Higher-Order Languages. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 314–325. https://doi.org/10.1145/41625.41654

[8] Matthias Felleisen and Daniel P. Friedman. 1987. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III, Ebberup, Denmark, 25-28 August 1986*, Martin Wirsing (Ed.). North-Holland, 193–222.

[9] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, Robert Cartwright (Ed.). ACM, 237–247. https://doi.org/10.1145/155090.155113

[10] Thomas Gilray, Michael D. Adams, and Matthew 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, ICFP 2016, Nara, Japan, September 18-22, 2016, Jacques Garrigue, Gabriele Keller, and Eijiro Sumii (Eds.). ACM, 407–420. https://doi.org/10.1145/2951913.2951936

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

[12] Dionna Amalie Glaze, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. 2014. Pushdown flow analysis with abstract garbage collection. *J. Funct. Program.* 24, 2-3 (2014), 218–283. https://doi.org/10.1017/S0956796814000100

[13] Maria Jenkins, Leif Andersen, Thomas Gilray, and Matthew Might. 2014. Concrete and Abstract Interpretation: Better Together. In *Proceedings of the 2014 Workshop on Scheme and Functional Programming*.

[14] Minseok Jeon, Sehun Jeong, and Hakjoo Oh. 2018. Precise and scalable points-to analysis via data-driven context tunneling. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 140:1–140:29. https://doi.org/10.1145/3276510

[15] Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. 2017. Data-driven context-sensitivity for points-to analysis. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 100:1–100:28. https://doi.org/10.1145/3133924

[16] Neil D. Jones and Nina Bohr. 2004. Termination Analysis of the Untyped lamba-Calculus. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings (Lecture Notes in Computer Science, Vol. 3091)*, Vincent van Oostrom (Ed.). Springer, 1–23. https://doi.org/10.1007/978-3-540-25979-4_1

[17] Neil D. Jones and Nina Bohr. 2008. Call-by-Value Termination in the Untyped lambda-Calculus. *Log. Methods Comput. Sci.* 4, 1 (2008). https://doi.org/10.2168/LMCS-4(1:3)2008

[18] Sven Keidel, Sebastian Erdweg, and Tobias Hombücher. 2023. Combinator-Based Fixpoint Algorithms for Big-Step Abstract Interpreters. *Proc. ACM Program. Lang.* 7, ICFP (2023), 955–981. https://doi.org/10.1145/3607863

[19] Sven Keidel, Casper Bach Poulsen, and Sebastian Erdweg. 2018. Compositional soundness proofs of abstract interpreters. *Proc. ACM Program. Lang.* 2, ICFP (2018), 72:1–72:26. https://doi.org/10.1145/3236767

[20] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. 2001. The size-change principle for program termination. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 81–92. https://doi.org/10.1145/360204.360210

[21] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2 (2020), 10:1–10:40. https://doi.org/10.1145/3381915

[22] Jingbo Lu, Dongjie He, and Jingling Xue. 2021. Eagle: CFL-Reachability-Based Precision-Preserving Acceleration of Object-Sensitive Pointer Analysis with Partial Context Sensitivity. *ACM Trans. Softw. Eng. Methodol.* 30, 4 (2021), 46:1–46:46. https://doi.org/10.1145/3450492

[23] Matthew Might and Panagiotis Manolios. 2009. A Posteriori Soundness for Non-deterministic Abstract Interpretations. In *Verification, Model Checking, and Abstract Interpretation, 10th International Conference, VMCAI 2009, Savannah, GA, USA, January 18-20, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5403)*, Neil D. Jones and Markus Müller-Olm (Eds.). Springer, 260–274. https://doi.org/10.1007/978-3-540-93900-9_22

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

[25] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '10)*. ACM, New York, NY, USA, 305–315. https://doi.org/10.1145/1806596.1806631

[26] Benoît Montagu and Thomas P. Jensen. 2021. Trace-based control-flow analysis. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 482–496. https://doi.org/10.1145/3453483.3454057

[27] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. 2019. Size-change termination as a contract: dynamically and statically enforcing termination for higher-order programs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, Kathryn S. McKinley and Kathleen Fisher (Eds.). ACM, 845–859. https://doi.org/10.1145/3314221.3314643

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

[29] John C. Reynolds. 1972. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM Annual Conference* (Boston, Massachusetts, United States). ACM.

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

[31] John Toman and Dan Grossman. 2019. Concerto: a framework for combined concrete and abstract interpretation. *Proc. ACM Program. Lang.* 3, POPL (2019), 43:1–43:29. https://doi.org/10.1145/3290356

[32] David Van Horn and Matthew Might. 2010. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1863543.1863553

[33] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: A Context-Free Approach to Control-Flow Analysis. In *Programming Languages and Systems*, Andrew D. Gordon (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 570–589. https://doi.org/10.2168/LMCS-7(2:3)2011

[34] Dimitrios Vardoulakis and Olin Shivers. 2011. Pushdown flow analysis of first-class control. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy (Eds.). ACM, 69–80. https://doi.org/10.1145/2034773.2034785

[35] Guannan Wei, Yuxuan Chen, and Tiark Rompf. 2019. Staged Abstract Interpreters: Fast and Modular Whole-Program Analysis via Meta-Programming. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 126 (Oct. 2019), 32 pages. https://doi.org/10.1145/3360552

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

[37] Shiyi Wei and Barbara G. Ryder. 2015. Adaptive Context-sensitive Analysis for JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic (LIPIcs, Vol. 37)*, John Tang Boyland (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 712–734. https://doi.org/10.4230/LIPICS.ECOOP.2015.712

Received 2025-02-27; accepted 2025-06-27