Newly-Single and Loving It: Improving Higher-Order Must-Alias Analysis with Heap Fragments

KIMBALL GERMANE, Brigham Young University, USA
JAY MCCARTHY, University of Massachusetts at Lowell, USA

Theories of higher-order must-alias analysis, often under the guise of environment analysis, provide deep behavioral insight. But these theories—in particular those that are most insightful otherwise—can reason about recursion only in limited cases. This weakness is not inherent to the theories but to the frameworks in which they're defined: machine models which thread the heap through evaluation. Since these frameworks allocate each abstract resource in the heap, the constituent theories of environment analysis conflate co-live resources identified in the abstract, such as recursively-created bindings. We present heap fragments as a general technique to allow these theories to reason about recursion in a general and robust way. We instantiate abstract counting in a heap-fragment framework and compare its performance to a precursor entire-heap framework. We also sketch an approach to realizing binding invariants, a more powerful environment analysis, in the heap-fragment framework.

CCS Concepts: • **Software and its engineering** → *Compilers.*

Additional Key Words and Phrases: control-flow analysis, must-alias analysis, abstract counting, heap fragments

ACM Reference Format:
Kimball Germane and Jay McCarthy. 2021. Newly-Single and Loving It: Improving Higher-Order Must-Alias Analysis with Heap Fragments. *Proc. ACM Program. Lang.* 5, ICFP, Article 96 (August 2021), 28 pages.
https://doi.org/10.1145/3473601

1 HIGHER-ORDER MUST-ALIAS ANALYSIS

The ability to tell when two references[1] in higher-order programs *must* alias unlocks a host of optimizations—both mundane [Might 2007a; Shivers 1991] and intricate [Might 2010; Steckler and Wand 1997]—and automated verification techniques [Might 2007b; Might et al. 2007].

But control-flow analysis alone does not provide this ability. Control-flow analysis (CFA) computes an abstract *over*approximation of the values to which each expression evaluates, and so implicitly computes only may-alias facts: two expressions *may* alias if their abstract values overlap.

Neither do techniques designed for first-order programs provide this ability. Several aspects of higher-order program behavior are qualitatively different than their first-order counterparts. Binding behavior is a prime example as, in higher-order programs, bindings may be captured in closures and flow as data.

Thus, higher-order must-alias analysis requires facilities designed specifically to cope with higher-order phenomena like binding behavior. (We examine this requirement more in § 2.)

[1]In addition to variable references, we consider expressions in the programming language and its metalanguage to be references.

Authors' addresses: Kimball Germane, Brigham Young University, Provo, Utah, USA, kimball@cs.byu.edu; Jay McCarthy, University of Massachusetts at Lowell, Lowell, Massachusetts, USA.

2475-1421/2021/8-ART96
https://doi.org/10.1145/3473601

1.1 The State of the Art

Presently, the most powerful and general theories of higher-order must-alias analysis are formulated in a framework based on abstract machines. This framework is a precursor to the *Abstracting Abstract Machines* (AAM) [Van Horn and Might 2010] framework and essentially an instantiation of it. (Every property of AAM significant to us here is also a property of this framework.)

AAM is nearly ideal for expressing higher-order must-alias analyses for several reasons:

- It offers tunable polyvariance which analyses can both access and influence [Gilray et al. 2016a; Might and Manolios 2009].
- It permits analyses to introspect and act on the current machine state to increase analysis precision (e.g. garbage collection [Earl et al. 2012; Might and Shivers 2006b] and strong update [Chase et al. 1990]).
- It allows analyses to incorporate semantic ghost state or synchronize with sister analyses to track ephemeral properties [Might 2007b, 2010; Might and Shivers 2006a].

But, along with these compelling capabilities, each must-alias analysis formulated in AAM inherits an acute limitation: recursion. To illustrate, consider the program to the right. The variable n is bound by the recursive loop and, because n is referenced after the recursive call, the number of co-live bindings to n is not static. AAM, however, can distinguish between only a static number of co-live bindings, determined by its level of polyvariance. Thus, while it is obvious from our vantage that each instance of n is local to its binding invocation, AAM-based must-alias analyses conflate instances of n across invocations.

```
(lambda (m)
  (letrec ([loop
             (lambda (n)
               (if0 n
                 n
                 (+ (loop (- n 1))
                    n)))])
    (loop m)))
```

In some cases, a seemingly-benign program transformation circumvents this limitation. In this program, for instance, a transposition of the arguments to + renders n dead by the recursive call, so that only a static number (one, in fact) of bindings are co-live at any given time. But cases like this are little reason to rejoice: a seemingly-benign program transformation can just as easily activate this limitation. And it applies only in some cases, as the continuation of the recursive call may genuinely depend on such bindings.

The limited ability AAM-based analyses have to reason about recursion is intrinsic to the AAM framework itself. To see why, we need to examine what this framework both provides and prescribes. The AAM framework provides a recipe to move from a concrete interpreter to a sound, computable abstract interpreter, where each is defined in terms of a small-step abstract machine. To provide this, the AAM framework prescribes redirecting all abstract machinery through the heap. AAM then induces computability by finitizing the heap's address space, thereby finitizing the execution space. (It ensures soundness by permitting multiple values to reside at a single address and nondeterministically choosing which to return.)

Because all abstract machinery allocated in the heap, the heap serves as a path-global repository of analysis information or, more pessimistically, a path-global chokepoint of precision.

1.2 Heap Fragments

To address this limitation, we use the recently-developed technique of decomposing the heap into *heap fragments* [Germane and Adams 2020]. With this technique, no abstract state by itself maintains the entire heap. Instead, each abstract state maintains only the fragment of the heap relevant to it, determined conservatively by reachability from its environment. The careful construction, separation, and combination of heap fragments simulates the presence of the entire heap. We discuss heap fragments formally, including how to incorporate mutation, in § 6.

An analysis of the preceding program which utilized heap fragments would analyze each invocation of `loop` in a fragment including bindings for (only) `loop` and `n`—those reachable from its environment. Because neither of these bindings escape, the analysis is able to fully isolate each binding. We walk through several examples in more depth in § 3.6.

However, Germane and Adams's formulation of heap fragments prevents it from being applied to the finite-state models AAM produces. These models don't provide enough structure to manipulate fragments in a sound and precise way. It builds instead on the pushdown models produced by stack-precise CFAs, now a standard approach to CFA [Darais et al. 2017; Gilray et al. 2016b; Johnson and Van Horn 2014; Vardoulakis and Shivers 2010; Wei et al. 2018]. Pushdown models guarantee that procedures return precisely to their point of call, which provides the structure necessary to effectively use heap fragments.

The shift from the finite-state models produced by AAM to the pushdown models produced by stack-precise CFAs implies a shift from execution snapshots embodied by an abstract state ς to execution summaries embodied by a pair $\varsigma \Downarrow r$ of a configuration ς and result r.

This shift has non-trivial ramifications on the encoding of must-alias relationships. As a snapshot of execution, an abstract state encodes must-alias relationships within its structure. By contrast, an execution summary encodes the *change in* must-alias relationships across the execution it summarizes. To successfully use the heap fragment technique for must-alias analysis, our primary task will be to reformulate snapshot-based theories of it into summary-based theories.

1.3 Contributions

To show the effectiveness of heap fragments, we do the following:

- We devise HFAC (§ 7), a heap fragment-based CFA which incorporates abstract counting [Might and Shivers 2006b]. We show how to compute it (§ 8), prove it sound (§ 9), and empirically evaluate its performance against the original AAM-based formulation of abstract counting (§ 11).
- We outline a minor modification to HFAC to implement binding invariants [Might 2010] which unlocks more powerful optimizations (§ 10).
- We discuss how Facchinetti et al. [2017]'s use of *relative store fragments* compares to a heap fragment-based CFA built on the ΔCFA [Might and Shivers 2006a] theory of environments (§ 12).

2 THE CHARACTER OF HIGHER-ORDER MUST-ALIAS ANALYSIS

Palsberg [1995] characterized higher-order analysis generally by the equation

$$\textit{higher-order analysis} = \textit{first-order analysis} + \textit{closure analysis}$$

which remarks on the qualitative effect of λ with respect to a program's control flow. λ is indeed a control construct but is also an environment construct: the evaluation of a λ captures ambient environment bindings and transports them wherever the enclosed λ flows. Thus, λ has a qualitative effect on environment behavior as well.

Environment behavior is arguably as fundamental to overall higher-order program behavior as control flow—so fundamental that Shivers [Shivers 1991, Ch. 8] designated *environment analysis* for environments the analogue of control-flow analysis for control flow. Environment analysis is intrinsic to higher-order must-alias analysis both because it answers must-alias questions about environment bindings and because many properly must-alias questions hinge on environment questions since bindings themselves refer to must-alias scrutinees. Let's take a closer look at each of these. (The following examples are meant to illustrate only the complications environment behavior can introduce and not instances that can be reasoned about in no other way.)

Environment behavior is subtle, and Shivers' now-canonical example of its subtlety [Shivers 1988] remains illustrative: In the program to the right, almost any CFA can tell that, when (h) has control, only closures over (λ () x) are bound to h. Given that x, the free variable captured in the closure, is in scope at (h), is appears that (λ () x) can be inlined at (h). But doing so changes the program's result from 42 to 0 and is thus unsafe. Where does this reasoning fall short? The result of the inner call to f captures a *local* binding of x, the capturing closure providing a vehicle for its escape. This closure flows into the outer call to f, a distinct invocation with its own binding of x. Thus two distinct bindings of x are reachable from the outer call's environment, a possibility the prior analysis failed to consider.

```
(let ([f (λ (x h) (if (zero? x) (h) (λ () x)))])
  (f 0 (f 42 #f)))
```

Now appropriately wary of surface-level equivalence in environments, we may fail to determine in the program to the right that the targets of set-box! and unbox are in fact precisely the same box whose exact value is known (and likely in hand) at (unbox b). Sufficiently powerful environment analysis reactivates our ability to perform must-alias reasoning.

```
(let ([b (box 42)])
  (let ([f (λ (mutate! get) (begin (mutate! 35) (unbox (get))))])
    (f (λ (x) (set-box! b x)) (λ () b))))
```

Not only can must-alias reasoning be used to justify optimizations, as in the preceding example, but it can also be used *during* the analysis to improve precision of the underlying flow analysis. For example, when an analysis reaches the set-box! in the program to the left, it can conservatively add the value of x to the values of b or, if it can tell that b must alias b0, it can *replace* the current value of b with x. In the former and typical case without must-alias reasoning available, the flow analysis concludes that the unbox can produce 42 *or* 35, whereas it concludes only 35 in the latter. Such imprecision is not contained to only its point of introduction; Might and Shivers [2006b] discuss how it can lead to other breaches of precision which in turn lead to others, creating a negative feedback loop that damages the overall precision of the analysis.

```
(let ([f (λ (b x) (set-box! b x))])
  (let ([b0 (box 42)])
    (begin (f b0 35) (unbox b0))))
```

3 ABSTRACT COUNTING WITHOUT AND WITH HEAP FRAGMENTS

Now that we've seen a few examples of environment behavior, we can take a closer look at how to reason about it. Since Shivers first designated it, a wide variety of theories of environment analysis have been developed [Bergstrom et al. 2014; Facchinetti et al. 2017; Germane and Might 2017; Might 2010; Might and Shivers 2006a,b; Shivers 1991; Steckler and Wand 1997]. In this section, we focus on abstract counting, one of the most powerful and general of these theories, both without and with heap fragments.

3.1 Abstract Counting

In a CFA, a finite number of abstract resources represent a possibly unbounded number of concrete resources. For instance, a central resource in a CFA is the finite pool of abstract addresses. At any given time, an abstract address $\hat{\alpha}$ may represent zero concrete addresses (i.e. it represents unallocated memory), one concrete address, or multiple concrete addresses.

To be conservative in general, we must assume that an abstract resource represents multiple concrete resources. When accessing an abstract address in the heap, for example, we must assume that it represents multiple addresses.

However, if we knew that $\hat{\alpha}$ represented only a single concrete address, then we would know that any two references to $\hat{\alpha}$ must be referring to precisely the same concrete object. If $\hat{\alpha}$ houses an environment binding, we can apply this fact to safely equivocate bindings, enabling inlining and copy propagation [Might 2007a, Ch. 10]. If $\hat{\alpha}$ instead houses the contents of a box, we can apply this fact to safely overwrite the previous box value with a new one rather than joining the two values, a technique known as *strong update* [Chase et al. 1990].

The idea behind abstract counting is simply to include with each heap entry an abstract count of the number of times that address has been allocated: 0, 1, or ∞ (representing multiple).

3.2 Garbage Collection

Abstract counting allows an analysis to introspect to know when its abstraction is precise but offers no way to recover precision. In practice, abstract garbage collection [Might and Shivers 2006b] nearly always accompanies abstract counting. Abstract garbage collection works just like concrete garbage collection: the environment and continuation registers of the abstract state are crawled to obtain a root set and all heap entries not transitively reachable from this root set are collected. Its purpose however is entirely different. In the concrete, garbage collection in an abstract machine has no semantic effect by definition. In the abstract, garbage collection increases analysis precision by expunging stale heap entries before their addresses are reused, thereby avoiding multiple simultaneous allocations at these addresses.

3.3 Limitations of AAM-Based Abstract Counting

With a better understanding of abstract counting, let's see how it fares on the example program from the introduction. We'll assume a 0CFA so that the address space is simply the set of program variables, but our reasoning applies to polyvariant CFAs too.

We apply this program to $\top_{num}$, the abstract value representing a fixed but unknown number. As the program is entered, `m` is bound and represents exactly one concrete binding. Evaluation of the **`letrec`** does the same for `loop`. At this point, no bindings have been made to `n`. After the function and arguments have been evaluated, but before the call has been made, the abstract state is garbage collected. Due to the **`letrec`**, the binding of `loop` is reachable from the closure bound to it. The binding of `m` is unreachable from that closure, its own value, and the (empty) continuation and is reaped from the heap. When the `loop`'s closure is entered, `n` acquires its first abstract binding and has an abstract count of 1. Because $\top_{num}$, the value of `n`, may or may not be 0, the analysis explores both branches. In the first, the call returns $\top_{num}$. The second calls `loop` recursively. Again, once the function and argument are evaluated but before the call is made, the heap is garbage collected. The function is the closure bound to `loop` which reaches `loop`. The value is $\top_{num}$ which doesn't reach anything. The continuation reaches `n`, because `n` is live after the call. Each binding in the heap is reachable, so garbage collection doesn't reap anything. Now the analysis makes the call, binding `n`. Now `n`'s binding in the heap, which previously represented a single concrete binding, represents multiple concrete bindings. Its abstract count is incremented to ∞ accordingly. Analysis of this invocation proceeds just as the first invocation's. The abstract state reached when the third call is made is identical to the second's, so the analysis concludes with two abstract invocations: one representing the first when `n`'s abstract binding represents a single concrete binding and one representing the unbounded others when `n`'s abstract binding represents multiple concrete bindings.

It's also more clear now why transposing the arguments to `+` improves the precision: when `n` is not referenced after the recursive call, it's not reachable by the continuation and can be reaped before the recursive call is made. When that call is made and binds `n`, it does so in a heap with no resident binding of `n`, so its binding has count 1.

3.4 Stack Precision and Garbage Collection are Not Enough

By producing a pushdown model of control flow rather than a finite-state one, stack-precise CFAs [Vardoulakis and Shivers 2010] boast a (colloquial) quantum leap in precision. In exchange for this precision, stack-precise CFAs turn control of the continuation—once a component of the abstract state—over to the framework, where access to it is limited. Despite this limited access, several formulations of abstract garbage collection have been achieved in this setting, using a variety of means to provide the stack root set of addresses to the analysis [Darais et al. 2017; Earl et al. 2012; Johnson and Van Horn 2014].

But even abstract counting adapted to this setting would be unable to reliably reason about recursion. While some spuriously-large abstract counts would likely disappear due to its higher precision, others would necessarily remain. After all, some bindings are genuinely necessary to the continuation and a more precise modelling of the continuation will only underscore such cases.

3.5 Heap Fragments

Heap fragments remove the dependence of evaluation on the continuation entirely. Let's see how with some examples. To simplify our presentation, we will assume 0CFA for all of our examples, so that variables serve as addresses. Accordingly, we omit environments and timestamps.

A heap fragment analysis of the program (1) begins in an empty heap fragment, denoted $\emptyset$ above the program. Once the analysis determines that `(λ (x) x)` is the next expression to evaluate, it decomposes the **`let`** into the `(λ (x) x)` itself and its continuation (2) and associates $\emptyset$ to each (seen above the continuation). The analysis then evaluates `(λ (x) x)` to produce $(\{$`(λ (x) x)`$\}, \emptyset)$, a pair consisting of an abstract closure and its heap fragment restricted to bindings reachable from it. This result is combined with the heap fragment of the continuation to produce heap fragment (3) under which the analysis proceeds to evaluate expression (3). Once again, the analysis focuses on the next expression, `(f 10)` and separates it from its continuation, seen at (4). The binding for `f` is reachable from both the expression and its continuation, so the heap fragment (4) doesn't shrink. To evaluate `(f 10)`, the analysis evaluates `f` and `10` into results $(\{$`(λ (x) x)`$\}, \emptyset)$ and $(\{10\}, \emptyset)$ respectively. The heap fragment associated with the function ($\emptyset$) is extended with a binding for `x` associated with the argument, and all bindings reachable from the argument in its heap fragment are copied in. The analysis then proceeds to evaluate the function body `x` under the heap fragment $\{(x, \{10\})\}$ which immediately produces the result $(\{10\}, \emptyset)$. This result is bound in heap fragment (5) under which expression (5) is evaluated. This expression decomposes to `(f a)` and continuation (6). Both `f` and `a` are reachable from the call expression so it is evaluated in the heap fragment with bindings for both. Only `a` is reachable from continuation (6), so `f`'s binding is absent from heap fragment (6).

(1) $\emptyset$
```
(let* ([f (λ (x) x)]
       [a (f 10)]
       [b (f a)])
  (+ a b))
```
(2) $\emptyset$
```
(let* ([f •]
       [a (f 10)]
       [b (f a)])
  (+ a b))
```
(3) $\{(f, \{$`(λ (x) x)`$\})\}$
```
(let* ([a (f 10)]
       [b (f a)])
  (+ a b))
```
(4) $\{(f, \{$`(λ (x) x)`$\})\}$
```
(let* ([a •]
       [b (f a)])
  (+ a b))
```
(5) $\{(f, \{$`(λ (x) x)`$\}), (a, \{10\})\}$
```
(let* ([b (f a)])
  (+ a b))
```
(6) $\{(a, \{10\})\}$
```
(let* ([b •])
  (+ a b))
```
(7) $\{(a, \{10\}), (b, \{10\})\}$
```
(+ a b)
```

`f` evaluates to $(\{$`(λ (x) x)`$\}, \emptyset)$, as before, and `a` happens to evaluate to $(\{10\}, \emptyset)$. The analysis has made this call before, and once it binds `x` to 10 in the empty environment, it will realize it, having encountered that precise configuration previously. Rather than reevaluating the function body, it simply returns the same result as before which is bound to `b` in the continuation's heap fragment. The analysis finally comes to expression (7) in the heap fragment (7) and the analysis produces, under a typical abstraction of numbers, $(\top_{num}, \emptyset)$.

Although we called this closed expression a program, it could just as well have been a subexpression of a larger program. Because the use of heap fragments makes analysis and garbage collection independent of the continuation, analysis of any closed expression, no matter where it appears, takes place in an empty heap. More generally, this property of heap fragments makes analysis dependent only on the bindings reachable from the free variables of the expression itself, and not its continuation. To wit, the analysis was able to reuse the summary from the first call to `f` at the second call even though they had different continuations with different reachable bindings. In this way, the use of heap fragments makes it more likely that summaries can be reused.

3.6 Heap Fragment Abstract Counting

To formulate abstract counting with heap fragments, we need to transition from the snapshot-centric account of AAM-based CFAs to the summary-centric account of stack-precise CFAs. In particular, we need to go from a snapshot state ς whose heap contains abstract counts to a summary pair $\varsigma \Downarrow r$ whose configuration ς and result r encode a change in abstract count.

We achieve this by distinguishing the roles of the configuration heap fragment (part of ς) and the result heap fragment (part of r). In the configuration heap fragment, each heap entry has an associated abstract count with precisely the same meaning as in an AAM-based CFA. Each entry in the result heap fragment has an abstract count too, but it is used to encode the relationship to the corresponding binding in the configuration heap fragment. Let's look at some examples to illustrate.

Let's pick up evaluation of the program on the left at `(λ (y) (+ y x))`. The heap fragments of both this expression and its continuation are $\{(\mathsf{x}, (\{42\}, 1))\}$ where the 1 indicates the abstract count of `x`'s binding. Clearly the `x`'s bindings in both heap fragments are the same, a fact we want the analysis to preserve. The result of evaluating this λ is $(\{$`(λ (y) (+ y x))`$\}, \{(\mathsf{x}, (\{42\}, 1))\})$. Its heap will be joined with the continuation's and extended with a binding for `f`. When the heaps are joined, the analysis will be tasked with joining $(\{42\}, 1)$ from the continuation heap and $(\{42\}, 1)$ from the result heap. The count 1 in the result heap indicates that it is the same binding as the continuation heap, and the join is simply $(\{42\}, 1)$.

```
(let ([x 42])
  (let ([f (λ (y) (+ y x))])
    (f x)))
```

Suppose however that the result heap fragment has a binding with count 1 but that was made during evaluation and therefore guaranteed to be distinct from those in the configuration heap fragment. For instance, consider the call `(const 10)` in the program to the right. Once the call is entered, the analysis evaluates `(λ (y) x)` in the heap fragment $\{(\mathsf{x}, (\{10\}, 1))\}$ immediately producing the λ itself (as a closure) with that precise heap. But the binding to `x` in the result was made during the call, a fact that needs to be communicated to the configuration heap. To communicate this, as the result is passed back, `x`'s count acquires a "freshness" flag, transforming from 1 to 1^F. This freshness flag is considered when joining the result back into the configuration. There is no binding for `x` currently in the configuration, so the join is effectively of $(\emptyset, 0)$ on the left and $(\{10\}, 1^F))$ on the right, which becomes $(\{10\}, 1))$ in the combined heap. That is, the freshness flag is removed once the continuation heap is introduced to its associated binding.

```
(let ([const (λ (x) (λ (y) x))])
  (let ([f (const 10)]
        [g (const 20)])
    (+ (f 30) (g 40))))
```

Now consider the call `(const 20)` which, by similar reasoning, results in the closure `(λ (y) x)` and the heap fragment $(\{20\}, 1^F))$. In this case, the continuation's heap contains a binding for `x` already, reachable from `f`, with a count of 1. Thus, the join of `x`'s binding is $(\{10\}, 1)$ on the left and $(\{20\}, 1^F))$ on the right Because `x`'s binding in the result is known to be distinct from `x`'s binding in the continuation, `x` must now be bound to multiple distinct concrete addresses, and the join is $(\{10, 20\}, \infty))$.

These examples cover the interesting cases. More mundanely, when the count on the left or the right is ∞, the result is ∞.

3.7 Effect Logs

Heap fragments communicate allocation through explicit value flow. But, classically, the heap not only accumulates allocation but also implicitly records mutative effects. By threading the heap through an evaluation path, the heap communicates these effects from one part of evaluation to another. For instance, evaluation of the expression bound to `y` in the program on the right not only yields 10 but also has an effect on the box `b` which the continuation relies on. The box `b` is not part of the result, however, and therefore not part of the heap fragment.

```
(let ([b (box 42)])
  (let ([y (begin
             (set-box! b 35)
             10)])
    (+ (unbox b) y)))
```

To ensure that effects actually have an effect, each evaluation records an effect log of the mutations that occur. Each result contains not only a value and associated heap fragment but also an effect log. When integrating a result with a continuation heap fragment, the first step is to replay the log to bring that heap fragment up to date.

We'll now look at an example which illustrates both effect logs and their interaction with abstract counts. Consider the analysis of the program to the left at the `(set-box! b 35)`. At this point, the heap fragment of both this expression and the continuation is $\{(\mathsf{b}, (\{\mathsf{box}(\hat{\alpha}_{42})\}, 1)), (\hat{\alpha}_{42}, (\{42\}, 1))\}$. That is, `b` is bound to a box whose contents are accessible at address $\hat{\alpha}_{42}$ (derived from `42`, the initialization expression of the box). The value of the box—i.e. the value at address $\hat{\alpha}_{42}$—is currently 42. The evaluation of a `set-box!` doesn't return an interesting value and, in any case, its value is ignored in this example. Its evaluation does produce an effect log with a single entry. In this case, that log is $\{(\hat{\alpha}_{42}, (\{35\}, 1))\}$. The analysis represents an effect log just like a result heap fragment, and replaying it on the continuation's heap fragment is very similar to joining it. In the continuation's fragment, $\hat{\alpha}_{42}$'s binding is $(\{42\}, 1)$ and, in the effect log, $\hat{\alpha}_{42}$'s binding is $(\{35\}, 1)$. When we join, with $(\{42\}, 1)$ on the left and $(\{35\}, 1)$ on the right, we can tell from the abstract counts that these are the values of precisely the same concrete binding. We are therefore justified in replacing the value on the left with the value on the right so that their join is $(\{35\}, 1)$, an instance of strong update [Chase et al. 1990]).

```
(let ([b (box 42)])
  (begin
    (set-box! b 35)
    (unbox b)))
```

4 LANGUAGE

With intuition in hand, we new proceed to define HFAC formally. We define our semantics over an ANF λ calculus [Flanagan et al. 1993] extended with boxes.

$$\begin{aligned} e &::= \mathsf{let}\, x = ce \,\mathsf{in}\, e \mid ae \\ ce &::= (ae\ ae) \mid \mathsf{box}\, ae \mid \mathsf{setbox!}\, ae\ ae \mid \mathsf{unbox}\, ae \mid ae \\ ae &::= \lambda x.e \mid x \end{aligned}$$

We assume that programs are *alphatized*—i.e., that all binding instances are distinct—which guarantees that bindings made at distinct program points remain distinct. We also assume that each expression bears a unique label to distinguish it from otherwise-identical expressions, which provides the same sort of guarantee for box allocations, which are indexed (in part) by expression. Both of these assumptions are satisfied by ANF normalization: the former by ensuring that the converter α-converts identical binding instances to fresh names, the latter by binding each subexpression to a distinct name, which serves as a de-facto label.

4.1 ANF Disturbs Reachability

Sabry and Felleisen [1994] remark that this normalization process has no effect on the results of a dataflow analysis but merely gives each intermediate value a name and orders expressions so that the evaluation order is reflected in the program structure. However, this process does affect what is naively reachable from the continuation. To illustrate, let's consider the non-ANF program which uses the make-box procedure to make two boxes, bound to b1 and b2. Consider the behavior of an AAM-based 0CFA with abstract counting on this program: The contents of b2 will use the same heap address as the contents of b1, derived from (box v). Since b1 is reachable from the continuation, its contents persist in the heap when b2 is created, so their co-live contents share that address. Consequently, from the analysis's perspective, the update of the contents of b1 via set-box! could apply to b2 as well, so the value 20 is added to the resident values 42 and 35, and does not replace them. In a heap fragment-based 0CFA with abstract counting, the nested **let** binding b2 creates an implicit local continuation frame which isolates b2 from b1. Since b2 doesn't outlive this frame, the analysis is able to keep it distinct from b1. Thus, the assignment to b1 is strong and the value 20 *replaces* the previous value 42.

```
(let ([make-box (λ (v) (box v))])
  (let ([b1 (make-box 42)])
    (let ([x (let ([b2 (make-box 35)])
               (unbox b2))])
      (let ([ignore (set-box! b1 20)])
        (unbox b1 20)))))
```

However, ANF normalization of this program lifts b2's binding into the sequence of bindings as seen on the right. This lift removes the creation of the local continuation frame and integrates its evaluation with its parent's. Thus, when b2 is bound, b1 is reachable from the continuation frame. As a consequence, their contents are merged and the abstract count of their contents saturates to ∞.

```
(let ([make-box (λ (v) (box v))])
  (let ([b1 (make-box 42)])
    (let ([b2 (make-box 35)])
      (let ([x (unbox b2))])
        (let ([ignore (set-box! b1 20)])
          (unbox b1 20)))))
```

This example illustrates that heap fragments defined in terms of naive reachability remain sensitive to certain structural transformations, even as they are oblivious to others. However, this sensitivity occurs only locally within each invocation; heap fragments robustly isolate inter-invocation bindings of an ANF program, including the kind induced by recursion. A straightforward intraprocedural analysis can recover the liveness information necessary to circumvent this sensitivity.

We expect the bindings of ANF-introduced variables to always be singleton since they only ever occur once in the program. Jagannathan et al. [1998] observe that, even with this expectation, its realization is not entirely trivial, since bindings can be captured by the continuation. By using heap fragments, we avoid any dependence on the continuation and confirm in our evaluation that each such binding was determined singleton (§ 11).

$$
\begin{array}{ll}
\varsigma \in \mathit{Config} = \mathit{Heap} \times \mathit{Env} \times \mathit{Exp} \times \mathit{Time} & r \in \mathit{Result} = D \times \mathit{Store} \times \mathit{Time} \\
\sigma \in \mathit{Heap} = \mathit{Addr} \to D & d \in D = \mathit{Closure} + \mathit{Box} \\
\rho \in \mathit{Env} = \mathit{Var} \rightharpoonup \mathit{Addr} & \mathit{Closure} = \mathit{Lam} \times \mathit{Env} \\
t \in \mathit{Time} & \mathit{Box} = \mathit{Addr} \\
\alpha \in \mathit{Addr} &
\end{array}
$$

Fig. 1. State space for the reference semantics

$$
\textsc{Let}\quad \frac{\sigma_0\, \rho\, ce\, t_0 \Downarrow_{ref} d_0\, \sigma_1\, t_1 \qquad \alpha = \mathrm{alloc}(\mathrm{var}(x), t_1) \qquad \sigma_1[\alpha \mapsto d_0]\, \rho[x \mapsto \alpha]\, e\, t_1 \Downarrow_{ref} d\, \sigma_2\, t_2}{\sigma_0\, \rho\, \mathrm{let}\, x = ce\, \mathrm{in}\, e\, t_0 \Downarrow_{ref} d\, \sigma_2\, t_2}
$$

$$
\textsc{App}\quad \frac{\begin{array}{c}\mathrm{closure}(\lambda x.e, \rho_0) = \mathcal{A}_{ref}(\sigma_0, \rho, ae_0) \qquad d_1 = \mathcal{A}_{ref}(\sigma_0, \rho, ae_1) \\ t_1 = \mathrm{tick}((ae_0\, ae_1), t_0) \qquad \alpha = \mathrm{alloc}(\mathrm{var}(x), t_1) \qquad \sigma_0[\alpha \mapsto d_1]\, \rho_0[x \mapsto \alpha]\, e\, t_1 \Downarrow_{ref} d\, \sigma_1\, t_2\end{array}}{\sigma_0\, \rho\, (ae_0\, ae_1)\, t_0 \Downarrow_{ref} d\, \sigma_1\, t_2}
$$

$$
\textsc{Atomic-Exp}\quad \frac{}{\sigma\, \rho\, ae\, t \Downarrow_{ref} \mathcal{A}_{ref}(\sigma, \rho, ae)\, \sigma\, t}
\qquad
\textsc{Box}\quad \frac{d = \mathcal{A}_{ref}(\sigma, \rho, ae) \qquad \alpha = \mathrm{alloc}(\mathrm{exp}(ae), t)}{\sigma\, \rho\, \mathrm{box}\, ae\, t \Downarrow_{ref} \mathrm{box}(\alpha)\, \sigma[\alpha \mapsto d]\, t}
$$

$$
\textsc{Set-Box!}\quad \frac{\mathrm{box}(\alpha) = \mathcal{A}_{ref}(\sigma, \rho, ae_0) \qquad d = \mathcal{A}_{ref}(\sigma, \rho, ae_1)}{\sigma\, \rho\, \mathrm{setbox!}\, ae_0\, ae_1\, t \Downarrow_{ref} \mathrm{closure}(\lambda \mathrm{x.x}, \bot)\, \sigma[\alpha \mapsto d]\, t}
\qquad
\textsc{Unbox}\quad \frac{\mathrm{box}(\alpha) = \mathcal{A}_{ref}(\sigma, \rho, ae)}{\sigma\, \rho\, \mathrm{unbox}\, ae\, t \Downarrow_{ref} \sigma(\alpha)\, \sigma\, t}
$$

Fig. 2. Big-step rules for the reference semantics

5 REFERENCE SEMANTICS

To ensure that the intricate separation and recombination of heap fragments faithfully implements the expected semantics, we define a standard reference semantics which does not decompose the heap.

Figure 1 defines the semantic constituents. A configuration ς is a 4-tuple consisting of a heap σ, an environment ρ, an expression e, and a timestamp t. From a configuration, evaluation produces a result r consisting of a denotable d, a heap σ, and a timestamp t. A heap σ maps addresses α to denotables d, and we often treat them extensionally as a set of address–denotable pairs. The alloc function produces addresses from a variable or expression and timestamp.

$$\mathrm{alloc} : (\mathit{Var} + \mathit{Exp}) \times \mathit{Time} \to \mathit{Addr}$$

We leave addresses and the definition of alloc abstract and require only that alloc be injective. A timestamp t is a sequence of call sites of the interpreted program. A denotable d is either a closure—a λ paired with an environment—or a box which consists solely of an address. An environment ρ is a finite map from variables to addresses.

Evaluation produces big-step judgements of the form $\varsigma \Downarrow_{ref} r$ which summarize the derivation. Figure 2 presents the rules which establish such judgements. We evaluate a let by evaluating the bound call expression, making a fresh binding of its value, and evaluating the body, the result of which becomes the result of the entire expression. We evaluate an application by evaluating the function expression to a closure and argument expression to a value. We obtain the next evaluation timestamp using tick, which derives it from the call site itself and current timestamp.

$$\mathsf{tick} : \mathit{App} \times \mathit{Time} \to \mathit{Time}$$

We leave its definition abstract and, like alloc, require only that it's injective. For a k-CFA-style context abstraction, we can instantiate $\mathit{Time} = \mathit{App}^*$ so that the timestamp is a finite sequence of application sites and tick as the identity function so that the timestamp records the application sites as they are encountered during evaluation. We evaluate an atomic expression using $\mathcal{A}_{ref}$, defined

$$\mathcal{A}_{ref}(\sigma, \rho, x) = \sigma(\rho(x)) \qquad\qquad \mathcal{A}_{ref}(\sigma, \rho, \lambda x.e) = \mathsf{closure}(\lambda x.e, \rho)$$

which simply looks up variables and closes λs with an environment.

The next three rules implement state. The Box rule allocates a fresh location in the heap using the timestamp and the box expression and yields a box with that now-populated location.[2] The Set-Box! rule identifies a box and a denotable d. It updates the heap so that the previously-allocated address α now locates d. It returns the identity function as a dummy value. Finally, the Unbox rule identifies a box and yields the denotable to which its address points.

This semantics treats the heap in a completely standard way. In particular, each configuration and result contains an entire heap and the rules thread the heap through evaluation, communicating allocations and state changes to later evaluation.

6 HEAP FRAGMENT SEMANTICS

Armed by the reference semantics with a ground truth of evaluation, we now define a semantics which decomposes the heap into fragments and propagates side effects via an effect log. The primary aim of this semantics, codified later by theorem, is to faithfully reconstitute global heap behavior by propagating local heap activity from one fragment to another.

The state space is largely the same as the reference semantics. Heap fragments σ replace full heaps in configurations and results, and results accrue an effect log ξ. A heap fragment is structurally the same as a heap. An effect log is a sequence of mutation entries (α, d, σ_d) consisting of an address α, denotable d, and heap fragment σ_d.

$$\varsigma \in \widetilde{\mathit{Config}} = \mathit{Fragment} \times \mathit{Env} \times \mathit{Exp} \times \mathit{Time} \qquad r \in \widetilde{\mathit{Result}} = D \times \mathit{Fragment} \times \mathit{Log} \times \mathit{Time}$$

$$\sigma \in \mathit{Fragment} = \mathit{Addr} \to D \qquad \xi \in \mathit{Log} = \mathit{Entry}^* \qquad \mathit{Entry} = \mathit{Addr} \times D \times \mathit{Fragment}$$

As with the reference semantics, evaluation in the heap fragment semantics is defined in terms of a big-step judgement $\varsigma \Downarrow r$ which summarizes its derivation. The rules which establish these judgements ensure that configurations and results are minimized by garbage collection. Figure 3 defines garbage collection. The gc_{in} function garbage collects a configuration by restricting the environment ρ to only the free variables of e and restricting the heap fragment σ to only the addresses reachable from $\rho|_e$. The set $\mathcal{R}_\rho(\sigma, \rho)$ of addresses reachable from ρ in σ is defined in terms of the reflexive, transitive closure of $\rightsquigarrow_\sigma$ which relates when one address can reach another through one dereference in σ. The relation $\rightsquigarrow_\sigma$ itself is defined in terms of the set of addresses that a value d touches, given by $\mathcal{T}$. The gc_{out} function garbage collects a result heap fragment σ_d by restricting it to the addresses reachable from its corresponding value d.

In addition to producing a result value d and corresponding heap fragment σ_d, each evaluation also produces an effect log ξ. The effect log records the mutations performed during evaluation;

[2]We assume some scheme to distinguish otherwise-identical expressions, such as expression labels.

$$\mathrm{gc}_{in}(\sigma,\rho,e) = \mathrm{gc}^*_{in}(\sigma,\rho|_e,e) \qquad \mathrm{gc}^*_{in}(\sigma,\rho,e) = (\sigma|_{\mathcal{R}_\rho(\sigma,\rho)},\rho,e) \qquad \mathrm{gc}_{out}(d,\sigma_d) = (d,\sigma_d|_{\mathcal{R}_d(\sigma_d,d)})$$

$$\mathcal{R}_\rho(\sigma,\rho) = \{\alpha' : \alpha \in \mathrm{rng}(\rho), \alpha \rightsquigarrow^*_\sigma \alpha'\} \qquad \mathcal{T}(\mathrm{box}(\alpha)) = \{\alpha\}$$

$$\mathcal{R}_d(\sigma_d,d) = \{\alpha' : \alpha \in \mathcal{T}(d), \alpha \rightsquigarrow^*_{\sigma_d} \alpha'\} \qquad \mathcal{T}(\mathrm{closure}(\lambda x.e,\rho)) = \mathrm{rng}(\rho)$$

$$\alpha \rightsquigarrow_\sigma \alpha' \iff \alpha' \in \mathcal{T}(\sigma(\alpha))$$

$$\mathcal{R}_{box}(\sigma) = \{\alpha : \mathrm{box}(\alpha) \in \mathrm{rng}(\sigma)\}$$

$$\mathrm{gc}_\xi(\xi,\sigma) = \xi|_{\mathcal{R}_{box}(\sigma)}$$

Fig. 3. The gc_{in}, gc_{out}, and gc_ξ functions

each triple (α, d, σ_d) denotes an update to heap location α to value (d, σ_d). We denote the replay of an effect log ξ on a heap fragment σ by $\xi(\sigma)$ defined

$$\langle\rangle(\sigma) = \sigma \qquad\qquad ((\alpha,d,\sigma_d) :: \xi)(\sigma) = (\xi(\sigma) \cup \sigma_d)[\alpha \mapsto d]$$

The effect logs ξ_0 and ξ_1 of a sequence of evaluations can be composed, denoted $\xi_0 \circ \xi_1$, by simply appending the logs.

If a particular evaluation allocates a box, mutates its contents, and then discards it, the effect log will have a record of the mutation of a box unreachable by the configuration heap fragment σ. To remove such records, the gc_ξ function in Figure 3 garbage collects a result effect log ξ by restricting the entries to those whose target is reachable by the configuration heap fragment. The semantic rules use gc_ξ to ensure that each result effect log contains only entries relevant to the configuration.

Since the union operation treats the store extensionally, we are obliged to show effect logs maintain the functionality of the heap, even amid composition and garbage collection.

Theorem 6.1. *If $\sigma\ \rho\ e\ t_0 \Downarrow (d,\sigma_d)\ \xi\ t_1$, then, for all $(\alpha,\_,\_) \in \xi$, $\alpha \in \mathrm{rng}(\sigma)$ and $\xi(\sigma)$ is a function.*

This result follows from the fact that, for $\alpha \in \mathrm{dom}(\xi(\sigma)) \cap \mathrm{dom}(\sigma_d)$, $\xi(\sigma)(\alpha) = \sigma_d(\alpha)$, which we establish by induction on the derivation.

The heap fragment semantic rules can be seen in Figure 4. The atomic evaluation function $\mathcal{A}$ produces a pair (d, σ_d) of a denotable d and its minimal heap fragment σ_d. The Atomic-Exp rule is a thin wrapper over this function and yields an empty effect log.

The Box rule evaluates the value to box, allocates an address for it, and yields a box with its minimal heap fragment. The Unbox rule evaluates a box itself and yields its value and minimal heap fragment produced by gc_{out}. Both of these rules yield an empty effect log. The Set-Box! rule yields the identity function (a closed value) and a correspondingly empty heap fragment. Rather than recording its mutation in the heap, the Set-Box! rule records it in a single-entry effect log, where $[\alpha \mapsto (d,\sigma_d)]$ denotes the sequence $\langle(\alpha,d,\sigma_d)\rangle$. This is the only rule which creates a new effect log, as opposed to the Let and App rules which adjust and combine them.

The Let rule first evaluates its let-bound expression in a minimal configuration. This evaluation ensures the resultant log mutates only boxes reachable from σ (minimized to *ce* and ρ). It replays this log to bring σ in accord with σ_{d0} before extending σ by (d, σ_{d0}) at a fresh address. It then evaluates the let body in that extended heap fragment, minimized by gc_{in}. The effect log ξ_1 produced by its evaluation is garbage-collected with respect to σ and composed with ξ_0.

The App rule atomically evaluates the function and argument, allocates a fresh address to bind the argument, and evaluates the function body in a minimal configuration binding the argument.

Let

$$\frac{\alpha = \mathrm{alloc}(\mathrm{var}(x), t_1) \qquad \begin{array}{l} \mathrm{gc}_{in}(\sigma, \rho, ce)\ t_0 \Downarrow (d_0, \sigma_{d0})\ \xi_0\ t_1 \\ \mathrm{gc}_{in}(\xi_0(\sigma)[\alpha \mapsto (d_0, \sigma_{d0})], \rho[x \mapsto \alpha], e)\ t_1 \Downarrow (d, \sigma_d)\ \xi_1\ t_2 \end{array}}{\sigma\ \rho\ \mathrm{let}\ x = ce\ \mathrm{in}\ e\ t_0 \Downarrow (d, \sigma_d)\ \mathrm{gc}_{\xi}(\xi_1, \sigma) \circ \xi_0\ t_2}$$

App

$$\frac{\begin{array}{c} (\mathrm{closure}(\lambda x.e, \rho_0), \sigma_{d0}) = \mathcal{A}(\sigma, \rho, ae_0) \qquad (d_1, \sigma_{d1}) = \mathcal{A}(\sigma, \rho, ae_1) \qquad t_1 = \mathrm{tick}((ae_0\ ae_1), t_0) \\ \alpha = \mathrm{alloc}(\mathrm{var}(x), t_1) \qquad \mathrm{gc}_{in}(\sigma_{d0}[\alpha \mapsto (d_1, \sigma_{d1})], \rho_0[x \mapsto \alpha], e)\ t_1 \Downarrow (d, \sigma_d)\ \xi\ t_2 \end{array}}{\sigma\ \rho\ (ae_0\ ae_1)\ t_0 \Downarrow (d, \sigma_d)\ \xi\ t_2}$$

Box

$$\frac{(d, \sigma_d) = \mathcal{A}(\sigma, \rho, ae) \qquad \alpha = \mathrm{alloc}(\mathrm{exp}(ae), t)}{\sigma\ \rho\ \mathrm{box}\ ae\ t \Downarrow (\mathrm{box}(\alpha), \bot[\alpha \mapsto (d, \sigma_d)])\ \langle\rangle\ t}$$

Unbox

$$\frac{(\mathrm{box}(\alpha), \sigma_\alpha) = \mathcal{A}(\sigma, \rho, ae)}{\sigma\ \rho\ \mathrm{unbox}\ ae\ t \Downarrow \mathrm{gc}_{out}(\sigma_\alpha(\alpha), \sigma_\alpha)\ \langle\rangle\ t}$$

Set-Box!

$$\frac{(\mathrm{box}(\alpha), \sigma_\alpha) = \mathcal{A}(\sigma, \rho, ae_0) \qquad (d, \sigma_d) = \mathcal{A}(\sigma, \rho, ae_1)}{\sigma\ \rho\ \mathrm{setbox!}\ ae_0\ ae_1\ t \Downarrow (\mathrm{closure}(\lambda \mathrm{x.x}, \bot), \bot)\ [\alpha \mapsto (d, \sigma_d)]\ t}$$

Atomic-Exp

$$\frac{}{\sigma\ \rho\ ae\ t \Downarrow \mathcal{A}(\sigma, \rho, ae)\ \langle\rangle\ t}$$

Fig. 4. The heap fragment semantics

The result of the call is the result of the function body evaluation. The effect log of the call is the effect log of the function body evaluation, restricted to entries reachable from the call heap fragment σ.

6.1 Faithful Heap Reconstruction

Although the heap management of this semantics is significantly different to that of the reference semantics, the results that each produces are the same modulo the heap. Now we establish that the heap fragment semantics and reference semantics agree on produced results, modulo the pruning due to garbage collection. In particular, a concrete configuration yields a result if and only if the heap fragment configuration produced by gc_{in} produces that result mapped through gc_{out}.

Theorem 6.2. *If σ_0 is closed under reachability with respect to ρ, then $\sigma_0\ \rho\ e\ t_0 \Downarrow_{ref}\ d\ \sigma_1\ t_1$ if and only if $\mathrm{gc}_{in}(\sigma_0, \rho, e)\ t_0 \Downarrow \mathrm{gc}_{out}(d, \sigma_1)\ \xi\ t_1$ where $\xi(\sigma_0) \subseteq \sigma_1$*

7 HEAP FRAGMENT ABSTRACT COUNTING

Armed with a concrete heap fragment semantics that faithfully captures heap behavior, we now turn to obtaining an abstract heap fragment semantics which incorporates abstract counting. We present the semantics in this section and, in the sections following, show how to compute the analysis and prove it sound.

7.1 State Space

We arrive at computability by finitizing the heap fragment address space, a standard technique in both finite-state [Van Horn and Might 2010] and pushdown [Darais et al. 2017] CFAs, and by finitizing the log space. Figure 5 presents the abstract state space.

To finitize the address space, we finitize the space of timestamps from which addresses are derived. We abstract *Time* as $\widehat{Time}^k = App^{\leq k}$ which denotes sequences of at-most k applications, where k is a parameter of the analysis. With $\widehat{Time}^k$ bounded, the address space $\widehat{Addr}$ becomes

$$\hat{\varsigma} \in \widehat{Config} = \widehat{Fragment} \times \widehat{Env} \times Exp \times \widehat{Time} \qquad \hat{r} \in \widehat{Result} = \hat{D} \times \widehat{FragmentR} \times \widehat{Log} \times \widehat{Time}$$

$$\begin{aligned}
\hat{\sigma} \in \widehat{Fragment} &= \widehat{Addr} \rightarrow \hat{D} \times \hat{\mathbb{N}} & \hat{\mathbb{N}} &= \{0, 1, \infty\} \\
\hat{\sigma}_d \in \widehat{Fragment}_{result} &= \widehat{Addr} \rightarrow \hat{D} \times \hat{\mathbb{N}}^F & \hat{\mathbb{N}}^F &= \{0, 1, 1^F, \infty\} \\
\hat{d} \in \hat{D} &= \mathcal{P}(\hat{V}) & & \\
\hat{v} \in \hat{V} &= \widehat{Closure} + \widehat{Box} & \widehat{Closure} &= Lam \times \widehat{Env} \\
\hat{\rho} \in \widehat{Env} &= Var \rightharpoonup \widehat{Addr} & \widehat{Box} &= \widehat{Addr} \\
\hat{\xi} \in \widehat{Log} &= \widehat{Fragment}_{result} & \hat{t} \in \widehat{Time}^k &= App^{\leq k}
\end{aligned}$$

Fig. 5. Finite state space of the abstract heap fragment semantics

bounded and, in turn, so do the spaces of boxes, environments, closures, and values. With $\widehat{Addr}$ bounded, $\widehat{\text{alloc}}$ produces a finite set of addresses. With a finite number of addresses, we cannot obtain a fresh address at will and therefore some abstract values will share an address. We abstract $\hat{D}$ to a set of values $\hat{V}$, each of which is a closure or box.

Configuration heap fragments include an abstract count for each heap entry. Result heap fragments include an abstract count extended to include the count 1^F denoting an abstract address which represents a single concrete address which was allocated during evaluation.

To finitize the space of effect logs, we represent them as heap fragments which represent a compound update to a store. Like result heap fragments, abstract effect logs carry an extended abstract count with each entry. Depending on the abstract counts of the constituents, some of the updates are strong, replacing the value at their address, and others aren't, merely joining with it.

7.2 Semantics

The abstract semantics is defined in terms of a judgement $\hat{\sigma}\ \hat{\rho}\ e\ \hat{t} \ \hat{\Downarrow}\ (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}\ \hat{t}$. Figure 6 presents the set of rules which establish this judgement, which are in one-to-one correspondence with the set of concrete rules. Like the concrete semantics, the abstract semantics uses garbage collection to ensure that configurations and results are minimized and the $\widehat{\text{gc}}_{in}$ $\widehat{\text{gc}}_{out}$ functions are straightforward abstract adaptations of their concrete counterparts. (The $\widehat{\text{gc}}_{\xi}$ function is not as straightforward due to the drastically different representation of abstract effect logs.)

We work our way through the rules from least- to most-affected by the abstraction. The Atomic-Exp rule for evaluating atomic expressions is unchanged, modulo the use of $\hat{\mathcal{A}}$ instead of $\mathcal{A}$.

The Unbox and Set-Box! rules nondeterministically produce results for each box denoted by the atomic expression. The Box rule allocates an address $\hat{\alpha}$ and uses mark to possibly mark it as fresh in the result heap.

The App rule is nondeterministic in the function. Any entry for $\hat{\alpha}$ in result heap and effect log are marked as fresh since $\hat{\alpha}$ is allocated during evaluation of the rule. Otherwise, the App rule perfectly corresponds to its concrete counterpart.

The Let rule marks $\hat{\alpha}$ in the result heap and effect log just as it does for App. In addition, it uses mark* to propagate freshness from the intermediate result heap to the final result heap and effect log.

We define the abstract operations in the following sections.

$$\textsc{Let}\quad \frac{\hat{\alpha} = \widehat{\mathsf{alloc}}(\mathsf{var}(x), \hat{t}_1) \qquad \begin{array}{c}\widehat{\mathsf{gc}}_{in}(\hat{\sigma}_0, \hat{\rho}, ce)\ \hat{t}_0 \mathrel{\hat{\Downarrow}} (\hat{d}_0, \hat{\sigma}_{d0})\ \hat{\xi}_0\ \hat{t}_1 \\ \widehat{\mathsf{gc}}_{in}(\hat{\xi}_0(\hat{\sigma}_0)[\hat{\alpha} \mapsto (\hat{d}_0, \hat{\sigma}_{d0})], \hat{\rho}[x \mapsto \hat{\alpha}], e)\ \hat{t}_1 \mathrel{\hat{\Downarrow}} (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}_1\ \hat{t}_2\end{array}}{\hat{\sigma}_0\ \hat{\rho}\ \mathsf{let}\, x = ce\, \mathsf{in}\, e\ \hat{t}_0 \mathrel{\hat{\Downarrow}} (\hat{d}, \mathsf{mark}^*(\mathsf{mark}(\hat{\sigma}_d, \hat{\alpha}), \hat{\sigma}_{d0}))\ \widehat{\mathsf{gc}}_{\xi}(\mathsf{mark}^*(\mathsf{mark}(\hat{\xi}_1, \hat{\alpha}), \hat{\sigma}_{d0}), \hat{\sigma}_0) \hat{\circ} \hat{\xi}_0\ \hat{t}_2}$$

$$\textsc{App}\quad \frac{\begin{array}{c} (\hat{d}_0, \hat{\sigma}_{d0}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_0) \\ \mathsf{closure}(\lambda x.e, \hat{\rho}_0) \in \hat{d}_0 \qquad (\hat{d}_1, \hat{\sigma}_{d1}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_1) \qquad \hat{t}_1 = \widehat{\mathsf{tick}}((ae_0\ ae_1), \hat{t}_0) \\ \hat{\alpha} = \widehat{\mathsf{alloc}}(\mathsf{var}(x), \hat{t}_1) \qquad \widehat{\mathsf{gc}}_{in}(\hat{\sigma}_{d0}[\hat{\alpha} \mapsto (\hat{d}_1, \hat{\sigma}_{d1})], \hat{\rho}_0[x \mapsto \hat{\alpha}], e)\ \hat{t}_1 \mathrel{\hat{\Downarrow}} (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}\ \hat{t}_2 \end{array}}{\hat{\sigma}\ \hat{\rho}\ (ae_0\ ae_1)\ \hat{t}_0 \mathrel{\hat{\Downarrow}} (\hat{d}, \mathsf{mark}(\hat{\sigma}_d, \hat{\alpha}))\ \mathsf{mark}(\hat{\xi}, \hat{\alpha})\ \hat{t}_2}$$

$$\textsc{Box}\quad \frac{(\hat{d}, \hat{\sigma}_d) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae) \qquad \hat{\alpha} = \widehat{\mathsf{alloc}}(\mathsf{exp}(ae), \hat{t})}{\hat{\sigma}\ \hat{\rho}\ \mathsf{box}\, ae\ \hat{t} \mathrel{\hat{\Downarrow}} (\{\mathsf{box}(\hat{\alpha})\}, \mathsf{mark}(\bot[\hat{\alpha} \mapsto (\hat{d}, \hat{\sigma}_d)], \hat{\alpha}))\ \langle\rangle\ \hat{t}} \qquad \textsc{Unbox}\quad \frac{(\hat{d}, \hat{\sigma}_d) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae) \qquad \mathsf{box}(\hat{\alpha}) \in \hat{d}}{\hat{\sigma}\ \hat{\rho}\ \mathsf{unbox}\, ae\ \hat{t} \mathrel{\hat{\Downarrow}} \widehat{\mathsf{gc}}_{out}(\hat{\sigma}_d(\hat{\alpha}), \hat{\sigma}_d)\ \langle\rangle\ \hat{t}}$$

$$\textsc{Set-Box!}\quad \frac{(\hat{d}_0, \hat{\sigma}_{d0}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_0) \qquad \mathsf{box}(\hat{\alpha}) \in \hat{d}_0 \qquad (\hat{d}, \hat{\sigma}_d) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_1)}{\hat{\sigma}\ \hat{\rho}\ \mathsf{setbox!}\, ae_0\ ae_1\ \hat{t} \mathrel{\hat{\Downarrow}} (\{\mathsf{closure}(\lambda \mathrm{x.x}, \bot)\}, \bot)\ [\hat{\alpha} \mapsto (\hat{d}, \hat{\sigma}_d)]\ \hat{t}} \qquad \textsc{Atomic-Exp}\quad \frac{}{\hat{\sigma}\ \hat{\rho}\ ae\ \hat{t} \mathrel{\hat{\Downarrow}} \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae)\ \langle\rangle\ \hat{t}}$$

Fig. 6. The abstract heap fragment semantics

7.3 Heap Fragment Join and Effect Log Replay

The *right-biased join* of a heap fragment $\hat{\sigma}$ and result heap fragment $\hat{\sigma}_d$, denoted $\hat{\sigma} \overrightarrow{\sqcup} \hat{\sigma}_d$ combines $\hat{\sigma}$ and $\hat{\sigma}_d$ with a bias toward the entries in $\hat{\sigma}_d$. We define both heap fragment extension and effect log replay in terms of a right-biased join as

$$\hat{\sigma}[\hat{\alpha} \mapsto (\hat{d}, \hat{\sigma}_d)] = \hat{\sigma} \overrightarrow{\sqcup} \hat{\sigma}_d \overrightarrow{\sqcup} \{(\hat{\alpha}, (\hat{d}, 1^F))\} \qquad \hat{\xi}(\hat{\sigma}) = \hat{\sigma} \overrightarrow{\sqcup} \hat{\xi} \qquad \hat{\xi}_0 \hat{\circ} \hat{\xi}_1 = \mathsf{mark}^*(\emptyset \overrightarrow{\sqcup} \hat{\xi}_1 \overrightarrow{\sqcup} \hat{\xi}_0, \hat{\xi}_1)$$

where $\overrightarrow{\sqcup}$ associates to the left. Notice in particular that, when the heap is extended, the newly-allocated variable $\hat{\alpha}$ inhabits a singleton heap fragment with a fresh abstract count. As we'll see, this ensures that the extension operation properly accounts for any previous bindings of $\hat{\alpha}$.

The $\overrightarrow{\sqcup}$ operator is defined fundamentally in terms of heap fragment entries and then lifted to heap fragments themselves.

$$\begin{array}{ccc}
(\bot, 0) \overrightarrow{\sqcup} (\bot, 0) = (\bot, 0) & (\hat{d}_0, 1) \overrightarrow{\sqcup} (\bot, 0) = (\hat{d}_0, 1) & (\hat{d}_0, \infty) \overrightarrow{\sqcup} (\hat{d}_1, \hat{n}) = (\hat{d}_0 \sqcup \hat{d}_1, \infty) \\
(\bot, 0) \overrightarrow{\sqcup} (\hat{d}, 1^F) = (\hat{d}, 1) & (\hat{d}_0, 1) \overrightarrow{\sqcup} (\hat{d}_1, 1) = (\hat{d}_1, 1) & \\
(\bot, 0) \overrightarrow{\sqcup} (\hat{d}, \infty) = (\hat{d}, \infty) & (\hat{d}_0, 1) \overrightarrow{\sqcup} (\hat{d}_1, 1^F) = (\hat{d}_0 \sqcup \hat{d}_1, \infty) & \\
 & (\hat{d}_0, 1) \overrightarrow{\sqcup} (\hat{d}_1, \infty) = (\hat{d}_0 \sqcup \hat{d}_1, \infty) &
\end{array}$$

When the abstract count on the left is 0 (left column), the result is the right except that a fresh count of 1 becomes simply a count of 1. This case applies when an evaluation allocates and returns a new binding that is not present in the initial heap. There is no case for an abstract count of 1 on the right because the semantics are designed such that a count of 1 on the right implies a count of 1 on the left.

When the abstract count on the left is 1 (middle column), we have a case for each possible abstract count on the right. A count of 0 means that a value isn't present and the value on the left is preserved. A count of 1 means that the entries refer to the same single address. In the case of a heap join, $\hat{d}_0$ and $\hat{d}_1$ are the same value, and $\hat{d}_0 \sqcup \hat{d}_1 = \hat{d}_1$. In the case of an effect log replay, the right entry may represent either a join or a strong update. In either case, we take the right value. This behavior when the count on the left and right are both 1 is the key reason we can use the same operation for heap join and log replay. A count of 1^F means that a fresh concrete address was allocated during evaluation when the initial heap already had a entry for it. Once joined, the heap has multiple entries and count ∞.

When the abstract count on the left is ∞ (right column), the value on the right is joined and the count remains ∞.

A right-biased join on heap fragments is simply a join on entries:

$$\hat{\sigma} \overrightarrow{\sqcup} \hat{\sigma}_d = \lambda\hat{\alpha}.\hat{\sigma}(\hat{\alpha}) \overrightarrow{\sqcup} \hat{\sigma}_d(\hat{\alpha})$$

7.4 Freshness Marking

The mark function ensures that a particular entry in a result heap is marked as fresh. It is defined as

$$\mathsf{mark}(\hat{\sigma}_d, \hat{\alpha}) = \lambda\hat{\alpha}'. \begin{cases} (\hat{d}, 1^F) \text{ if } \hat{\sigma}_d(\hat{\alpha}) = (\hat{d}, 1) \text{ and } \hat{\alpha} = \hat{\alpha}' \\ \hat{\sigma}_d(\hat{\alpha}') \text{ otherwise} \end{cases}$$

such that the target address is marked fresh if it has count 1 in the given heap fragment, and is untouched otherwise.

The mark* function ensures that any entries with count 1 in the result heap fragment but were fresh in the intermediate heap fragment are marked fresh.

$$\mathsf{mark}^*(\hat{\sigma}_d, \hat{\sigma}_{d_0}) = \lambda\hat{\alpha}. \begin{cases} (\hat{d}, 1^F) \text{ if } \hat{\sigma}_d(\hat{\alpha}) = (\hat{d}, 1) \text{ and } \hat{\sigma}_{d_0}(\hat{\alpha}) = (\hat{d}, 1^F) \\ \hat{\sigma}_d(\hat{\alpha}) \text{ otherwise} \end{cases}$$

This freshening is necessary to propagate the freshness across an intermediate heap. Consider the sequence of events: An initial heap has no entry for $\hat{\alpha}$. The evaluation of the let-bound expression results in a heap fragment in which $\hat{\alpha}$ has count 1^F. (The semantics ensures that such an address cannot return with count 1 since the initial heap has no entry for it.) When the initial heap is extended with an entry for $\hat{\alpha}$, its count is 1. Now suppose that the evaluation of the body results in a heap fragment in which $\hat{\alpha}$ has count 1. Such a count indicates that it is the same binding as the heap initial to the body evaluation. However, that binding is known to be fresh with respect to the evaluation of let expression as a whole. Thus, we freshen it to maintain the invariant that fresh addresses are so flagged.

7.5 Effect Log Garbage Collection

Because abstract effect logs are collapsed into a heap, effect log garbage collection, performed by $\widehat{\mathrm{gc}}_{\xi}$, is defined

$$\widehat{\mathrm{gc}}_{\xi}(\hat{\xi}, \hat{\sigma}) = \hat{\xi}|_{\hat{\mathcal{R}}_{box}(\hat{\sigma})} \text{ where } \hat{\mathcal{R}}_{box}(\hat{\sigma}) = \bigcup_{\hat{d} \in \mathrm{rng}(\hat{\sigma})} \{\hat{\alpha}' : \mathsf{box}(\hat{\alpha}) \in \hat{d}, \hat{\alpha} \rightsquigarrow_{\hat{\sigma}} \hat{\alpha}'\}$$

so that a garbage-collected effect log maintains all entries reachable from box addresses of a reference heap fragment. (Here, reachability between addresses is transparently lifted to cope with sets of values.)

$$\textsc{Refl}\quad \frac{}{\varsigma \Uparrow \varsigma}$$

$$\textsc{Let-Body}\quad \frac{\begin{array}{c} \mathsf{gc}_{in}(\sigma, \rho, ce)\; t_0 \Downarrow (d, \sigma_d)\; \xi\; t_1 \\ \alpha = \mathsf{alloc}(\mathsf{var}(x), t_1) \qquad \mathsf{gc}_{in}(\xi(\sigma)[\alpha \mapsto (d, \sigma_d)], \rho[x \mapsto \alpha], e)\; t_1 \Uparrow \varsigma \end{array}}{\sigma\; \rho\; \mathsf{let}\, x = ce\, \mathsf{in}\, e\; t_0 \Uparrow \varsigma}$$

$$\textsc{Let-App}\quad \frac{\begin{array}{c} (\mathsf{closure}(\lambda x.e, \rho_0), \sigma_{d0}) = \mathcal{A}(\sigma, \rho, ae_0) \qquad (d_1, \sigma_{d1}) = \mathcal{A}(\sigma, \rho, ae_1) \qquad t_1 = \mathsf{tick}((ae_0\; ae_1), t_0) \\ \alpha = \mathsf{alloc}(\mathsf{var}(x), t_1) \qquad \mathsf{gc}_{in}(\sigma_{d0}[\alpha \mapsto (d_1, \sigma_{d1})], \rho_0[x \mapsto \alpha], e)\; t_1 \Uparrow \varsigma \end{array}}{\sigma\; \rho\; \mathsf{let}\, x = (ae_0\; ae_1)\, \mathsf{in}\, e\; t_0 \Uparrow \varsigma}$$

Fig. 7. The concrete heap fragment reachability semantics

$$\textsc{Refl}\quad \frac{}{\hat{\varsigma} \hat{\Uparrow} \hat{\varsigma}}$$

$$\textsc{Let-Body}\quad \frac{\begin{array}{c} \widehat{\mathsf{gc}}_{in}(\hat{\sigma}, \hat{\rho}, ce)\; \hat{t}_0 \hat{\Downarrow} (\hat{d}, \hat{\sigma}_d)\; \hat{\xi}\; \hat{t}_1 \\ \hat{\alpha} = \mathsf{alloc}(\mathsf{var}(x), \hat{t}_1) \qquad \widehat{\mathsf{gc}}_{in}(\hat{\xi}(\hat{\sigma})[\hat{\alpha} \mapsto (\hat{d}, \hat{\sigma}_d)], \hat{\rho}[x \mapsto \hat{\alpha}], e)\; \hat{t}_1 \hat{\Uparrow} \hat{\varsigma} \end{array}}{\hat{\sigma}\; \hat{\rho}\; \mathsf{let}\, x = ce\, \mathsf{in}\, e\; \hat{t}_0 \hat{\Uparrow} \hat{\varsigma}}$$

$$\textsc{Let-App}\quad \frac{\begin{array}{c} (\hat{d}_0, \hat{\sigma}_{d0}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_0) \\ \mathsf{closure}(\lambda x.e, \hat{\rho}_0) \in \hat{d}_0 \qquad (\hat{d}_1, \hat{\sigma}_{d1}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_1) \qquad \hat{t}_1 = \mathsf{tick}((ae_0\; ae_1), \hat{t}_0) \\ \hat{\alpha} = \mathsf{alloc}(\mathsf{var}(x), \hat{t}_1) \qquad \widehat{\mathsf{gc}}_{in}(\hat{\sigma}_{d0}[\hat{\alpha} \mapsto (\hat{d}_1, \hat{\sigma}_{d1})], \hat{\rho}_0[x \mapsto \hat{\alpha}], e)\; \hat{t}_1 \hat{\Uparrow} \hat{\varsigma} \end{array}}{\hat{\sigma}\; \hat{\rho}\; \mathsf{let}\, x = (ae_0\; ae_1)\, \mathsf{in}\, e\; \hat{t}_0 \hat{\Uparrow} \hat{\varsigma}}$$

Fig. 8. The abstract heap fragment reachability semantics

8 COMPUTING THE ANALYSIS

Each semantics we have presented, including the abstract semantics, has been in terms of a big-step relation. This representation is somewhat at odds with an abstract interpretation because it accounts for intermediate computations only for overall convergent computations. For example, a big-step evaluation relation says nothing about the evaluation of Ω because it diverges, but we would expect an abstract interpretation of this program to report on reachable states at least.

Darais [2017] provides a generic framework to overcome these limitations and achieve an abstract interpretation of a big-step semantics.

The first step is to augment the big-step evaluation relation with a big-step reachability relation which specifies reachable configurations. Figure 7 presents the concrete heap fragment reachability relation. For our ANF language, there are only a few rules. The Refl rule says that a configuration reaches itself. The Let-App rule says that a configuration that reaches a let binding an application reaches the call's body. The Let-Body rule says that a configuration that reaches a let for which evaluation of the bound expression converges reaches its body. We define the same kind of relation for our abstract semantics, seen in Figure 8.

The reachability semantics can be brought into correspondence with a reference small-step semantics to ensure that it does indeed recover intermediate configurations. (See Darais [2017] for a detailed walkthrough of such a development.) We elide this development but do desire the property that the abstract reachability semantics reaches a corresponding intermediate configuration to each one reached by the concrete reachability semantics.

Theorem 8.1 (Abstract Reachability Semantics Soundness). *If $\varsigma \Uparrow \varsigma'$ and $|\varsigma|_{config} \sqsubseteq \hat{\varsigma}$ then $\hat{\varsigma} \hat{\Uparrow} \hat{\varsigma}'$ where $|\varsigma'|_{config} \sqsubseteq \hat{\varsigma}'$.*

This result follows easily from the soundness of our abstract evaluation semantics with respect to the concrete evaluation semantics, established in § 9.

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 $\hat{\varsigma}_{pr}$ and \$ is a *cache* mapping reachable configurations to their results. We say that $(\$, R) \models pr$ if and only if $\hat{\varsigma}_{pr} \hat{\Uparrow} \hat{\varsigma} \implies \hat{\varsigma} \in R$ and $\hat{\varsigma}_{pr} \hat{\Uparrow} \hat{\varsigma} \land \hat{\varsigma} \hat{\Downarrow} \hat{r} \implies \hat{r} \in \$(\hat{\varsigma})$.

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

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

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

$$(\$^+, R^+) = lfp(\lambda(\$, R).\mathcal{F}(\$, R) \sqcup (\bot, \{\hat{\varsigma}_{pr}\}))$$

In the definition of $\mathcal{F}$, the relations $\hat{\Downarrow}^{(\$,R)}$ and $\hat{\Uparrow}^{(\$,R)}$ are the relations $\hat{\Downarrow}$ and $\hat{\Uparrow}$ 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 [2017] proves the following theorem:

Theorem 8.2 (Algorithm Correctness [Darais 2017]). *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 [Darais et al. 2017; Wei et al. 2018] and our implementation, discussed further in § 11, takes this form.

9 SOUNDNESS OF THE ANALYSIS

The abstract semantics is sound if its judgements are in accord with the concrete semantics. For instance, if an abstract summary indicates that a closure in the result is precisely the same as one in the configuration (by an abstract count of 1 for all of its captured bindings) and the abstract semantics is sound, then it must be the case that a closure in a corresponding concrete result is precisely the same as one in the corresponding configuruation.

The entire development of this section is to arrive at the following general soundness theorem.

Theorem 9.1. *If $|\varsigma| \sqsubseteq \hat{\varsigma}$ then $\varsigma \Downarrow r$ only if there exists $\hat{r}$ such that $|r|^{\varsigma} \sqsubseteq \hat{r}$ and $\hat{\varsigma} \hat{\Downarrow} \hat{r}$.*

We proceed by defining the abstractions and relationships and then turn to proving it.

9.1 Abstraction

Each element of the concrete state space has a corresponding element in the abstract state space, accessed by a member of the family of abstraction operators $|\cdot|_X$, defined in Figure 9. A configuration abstracts component-wise. A heap fragment abstracts to a pair-valued map keyed by an abstract address: the first component is the abstracted and joined collection of values located by each concrete address that abstracts to it; the second component is the abstract cardinality of the number of addresses, given by card:

$$\text{card}(\emptyset) = 0 \qquad \text{card}(\{\alpha\}) = 1 \qquad \text{card}(\{\alpha_1, \alpha_2, \ldots, \alpha_n\}) = \infty \text{ for } n > 1$$

$$|(\sigma,\rho,e,t)|_{config} = (|\sigma|_{fragment}, |\rho|_{env}, e, |t|_{time})$$

$$|(d,\sigma_d,\xi,t)|^{(\sigma,\rho,e,t)}_{result} = (|d|_D, |\sigma_d|^{\sigma}_{fragmentr}, |\xi|^{\sigma}_{log}, |t|_{time})$$

$$\begin{aligned} |\mathsf{box}(\alpha)|_D &= \{\mathsf{box}(|\alpha|_{addr})\} & |(\mathsf{var}(x),t)|_{addr} &= (\mathsf{var}(x), |t|_{time}) \\ |\mathsf{closure}(\lambda x.e,\rho)|_D &= \{\mathsf{closure}(\lambda x.e, |\rho|_{env})\} & |(\mathsf{exp}(e),t)|_{addr} &= (\mathsf{exp}(e), |t|_{time}) \\ |\rho|_{env} &= \lambda x.|\rho(x)|_{addr} & |t|^k_{time} &= \lfloor t \rfloor_k \end{aligned}$$

$$\begin{aligned} |\sigma|_{fragment} &= \lambda\hat{\alpha}.(\bigsqcup_{\hat{\alpha}=|\alpha|_{addr}} |\sigma(\alpha)|_D, \mathsf{card}(\{\alpha : \alpha \in \mathrm{dom}(\sigma), \hat{\alpha} = |\alpha|_{addr}\})) \\ |\sigma_d|^{\sigma}_{fragmentr} &= \lambda\hat{\alpha}.(\bigsqcup_{\hat{\alpha}=|\alpha|_{addr}} |\sigma_d(\alpha)|_D, \mathsf{card}^F(\{\alpha : \alpha \in \mathrm{dom}(\sigma_d), \hat{\alpha} = |\alpha|_{addr}\}, \sigma)) \\ |\xi|^{\sigma}_{log} &= |\mathsf{fold}(\xi)|^{\sigma}_{fragmentr} \end{aligned}$$

Fig. 9. The family of abstraction operators

That is, the abstract cardinality of empty and singleton sets is their concrete cardinality; the abstract cardinality of any other set is ∞. A box abstracts to a singleton box with an abstracted address. A closure abstracts to a singleton closure with an abstracted environment. An environment abstracts point-wise. An address abstracts to a pair of its first component and its abstracted time. A time abstracts to its length-k prefix, where k is a parameter of the analysis governing polyvariance. A result abstracts component-wise though the abstractions of its heap fragment and log each depend on a given reference heap fragment. The result heap fragment abstracts just as a configuration heap fragment, but uses card^F to report freshness with respect to a reference heap fragment:

$$\begin{aligned} \mathsf{card}^F(\emptyset,\sigma) &= 0 \\ \mathsf{card}^F(\{\alpha\},\sigma) &= 1 \text{ if } \alpha \in \mathrm{dom}(\sigma) \\ \mathsf{card}^F(\{\alpha\},\sigma) &= 1^F \text{ if } \alpha \notin \mathrm{dom}(\sigma) \\ \mathsf{card}^F(\{\alpha_1,\alpha_2,\ldots,\alpha_n\},\sigma) &= \infty \text{ for } n > 1 \end{aligned}$$

The log abstracts first by collapsing it using fold into a heap fragment and abstracting it with respect to a reference heap fragment.

$$\mathsf{fold}(\langle\rangle) = \emptyset \qquad \mathsf{fold}((\alpha,d,\sigma_d) :: \xi) = (\mathsf{fold}(\xi) \cup \sigma_d)\ \overrightarrow{\cup}\ \{(\alpha,d)\}$$

where $\overrightarrow{\cup}$ is a concrete counterpart to $\overrightarrow{\sqcup}$ defined

$$\sigma_0\ \overrightarrow{\cup}\ \sigma_1 = \sigma_0|_{\mathrm{dom}(\sigma_0)\setminus\mathrm{dom}(\sigma_1)} \cup \sigma_1$$

which combines heap fragments σ_0 and σ_1 ensuring that the entries of σ_1 take precedence in the result.

9.2 Refinement

Just as abstraction is defined by a family of operators, refinement is defined by a family of relations, seen in Figure 10. We explicitly name each member in ambiguous circumstances, but otherwise leave it implicit.

$$
\begin{aligned}
(\hat{\sigma}_0, \hat{\rho}, e, \hat{t}) \sqsubseteq_{config} (\hat{\sigma}_1, \hat{\rho}, e, \hat{t}) &\iff \hat{\sigma}_0 \sqsubseteq_{fragment} \hat{\sigma}_1 \\
\hat{\sigma}_0 \sqsubseteq_{fragment} \hat{\sigma}_1 &\iff \forall \hat{\alpha}.\hat{\sigma}_0(\hat{\alpha}) \sqsubseteq \hat{\sigma}_1(\hat{\alpha}) \\
\hat{d}_0 \sqsubseteq_D \hat{d}_1 &\iff \hat{d}_0 \subseteq \hat{d}_1 \\
(\hat{d}_0, \hat{\sigma}_{d_0}, \hat{\xi}_0, \hat{t}) \sqsubseteq_{result} (\hat{d}_1, \hat{\sigma}_{d_1}, \hat{\xi}_1, \hat{t}) &\iff \hat{d}_0 \sqsubseteq_D \hat{d}_1 \wedge \hat{\sigma}_{d_0} \sqsubseteq_{fragmentr} \hat{\sigma}_{d_1} \wedge \hat{\xi}_0 \sqsubseteq_{log} \hat{\xi}_1 \\
\hat{\sigma}_{d_0} \sqsubseteq_{fragmentr} \hat{\sigma}_{d_1} &\iff \forall \hat{\alpha}.\hat{\sigma}_{d_0}(\hat{\alpha}) \sqsubseteq \hat{\sigma}_{d_1}(\hat{\alpha}) \\
\hat{\xi}_0 \sqsubseteq_{log} \hat{\xi}_1 &\iff \forall \hat{\alpha}.\hat{\xi}_0(\hat{\alpha}) \sqsubseteq \hat{\xi}_1(\hat{\alpha}) \\
0 \sqsubseteq_{\mathbb{N}} 1 \sqsubseteq_{\mathbb{N}} 1^F &\sqsubseteq_{\mathbb{N}} \infty
\end{aligned}
$$

Fig. 10. The family of refinement relations

9.3 Simulation

The key relationship between the concrete and abstract heap fragment semantics is *simulation*: if the concrete heap fragment semantics takes a (big) step, the abstract semantics should take a corresponding step. We formally capture this relationship as the following theorem.

THEOREM 9.2. *If* $|\varsigma| \sqsubseteq \hat{\varsigma}$ *then* $\varsigma \Downarrow r$ *only if there exists* $\hat{r}$ *such that* $|r|^{\varsigma} \sqsubseteq \hat{r}$ *and* $\hat{\varsigma} \,\hat{\Downarrow}\, \hat{r}$.

As written, this theorem is standard except for the dependence of the abstraction of the result on the configuration itself. The proof proceeds by induction over the derivation of $\varsigma \Downarrow r$. We sketch each case to reveal the required lemmas, most of which show that each operator on the concrete state space has a relationship with its corresponding operator on the abstract state space across the abstraction.

We begin with the base cases:

For the ATOMIC-EXP rule, we need that atomic evaluation commutes with abstraction.

LEMMA 9.3. *For all* σ, ρ, *and* ae, $|\mathcal{A}(\sigma, \rho, ae)|_D \sqsubseteq \hat{\mathcal{A}}(|\sigma|_{frag}, |\rho|_{env}, ae)$.

For the BOX rule, we need that bath heap extension and allocation commute with abstraction.

LEMMA 9.4. *For all* σ, α, d, *and* σ_d, $|\sigma[\alpha \mapsto (d, \sigma_d)]|_{frag} \sqsubseteq |\sigma|_{frag}[|\alpha|_{addr} \mapsto (|d|_D, |\sigma_d|^{\sigma}_{fragr})]$.

LEMMA 9.5. *For all* ae *and* t, $|\text{alloc}(\text{exp}(ae), t)|_{addr} \sqsubseteq \widehat{\text{alloc}}(\text{exp}(ae), |t|_{time})$.

For the SET-BOX! rule, we need that an effect log with a single entry commutes with abstraction.

LEMMA 9.6. *For all* σ, α, d, *and* σ_d, $|[(\alpha, d, \sigma_d)]|^{\sigma}_{log} \sqsubseteq [(|\alpha|_{addr}, |d|_D, |\sigma_d|^{\sigma}_{fragr})]$.

For the UNBOX rule, we need that garbage collection of a result commutes with abstraction. Of course, we will also need that garbage collection of a configuration commutes with abstraction too.

LEMMA 9.7. *For all* σ, ρ, *and* e, $|\text{gc}_{in}(\sigma, \rho, e)| \sqsubseteq \widehat{\text{gc}}_{in}(|\sigma|_{frag}, |\rho|_{env}, e)$.

LEMMA 9.8. *For all* d *and* σ, $|\text{gc}_{out}(d, \sigma)| \sqsubseteq \widehat{\text{gc}}_{out}(|d|_D, |\sigma|_{frag})$.

Now onto the inductive cases:

For the APP rule, we require that the advancement of time commutes with abstraction.

LEMMA 9.9. *For all* $(ae_0\ ae_1)$ *and* t, $|\text{tick}((ae_0\ ae_1), t)|_{time} \sqsubseteq \widehat{\text{tick}}((ae_0\ ae_1), |t|_{time})$.

An effect log ξ is *consistent* with a heap fragment σ if replaying ξ on σ yields a function. The semantics ensure that produced logs are consistent with their corresponding heap fragments.

For the LET rule, we need commutation results about effect log composition and that mark and mark* introduce refinement.

LEMMA 9.10. *For all σ and ξ_0 that are consistent and ξ_1 consistent with $\xi_0(\sigma)$, $|\xi_1 \circ \xi_0|^{\sigma}_{log} \sqsubseteq |\xi_1|^{\xi_0(\sigma)}_{log} \hat{\circ} |\xi_0|^{\sigma}_{log}$.*

10 BINDING INVARIANTS

A CFA applied to the program to the right will conclude that only closures over `(λ () y)` flow to `f` yet `(λ () y)` remains ineligible for inlining because its free variable `y` is not in scope at `(f)`. However, its binding is equivalent to `z`'s and `z` *is* in scope at `(f)`. Thus, one could rematerialize `(λ () z)` to replace `f` and preserve the program's behavior.

```
(let ([z 42])
  (let ([f (let ([y z])
              (λ () y))])
    (f)))
```

This optimization is an instance of *higher-order rematerialization* introduced by Might [2010]. To support this optimization, Might generalized the task of environment analysis from determining whether the same variable in different environments has the same binding to determining whether two bindings are equivalent, whether or not they are associated with the same variable. An analysis which can complete this more general task can justify this and similarly-powerful optimizations.

We can bring HFAC up to the task of the generalized environment analysis with a minor modification. The following modified rules for let expressions and application take precedence when the argument is a reference to some variable y.

LET-INVARIANT

$$\frac{(\hat{d}_0, \hat{\sigma}_{d0}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, y) \qquad |\hat{\sigma}(\hat{\rho}(y))| = 1 \qquad \widehat{\mathsf{gc}}_{in}(\hat{\xi}_0(\hat{\sigma}_0)[\hat{\rho}(y) \mapsto (\hat{d}_0, \hat{\sigma}_{d0})], \hat{\rho}[x \mapsto \hat{\rho}(y)], e)\ \hat{t}_1 \Downarrow (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}_1\ \hat{t}_2}{\hat{\sigma}_0\ \hat{\rho}\ \mathsf{let}\ x = y\ \mathsf{in}\ e\ \hat{t}_0 \Downarrow (\hat{d}, \mathsf{mark}^*(\hat{\sigma}_d, \hat{\sigma}_{d0}))\ \widehat{\mathsf{gc}}_{\xi}(\mathsf{mark}^*(\hat{\xi}_1, \hat{\sigma}_{d0}), \hat{\sigma}_0) \hat{\circ} \hat{\xi}_0\ \hat{t}_2}$$

APP-INVARIANT

$$\frac{\begin{array}{cccc}(\hat{d}_0, \hat{\sigma}_{d0}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, ae_0) & \mathsf{closure}(\lambda x.e, \hat{\rho}_0) \in \hat{d}_0 & |\hat{\sigma}(\hat{\rho}(y))| = 1 & (\hat{d}_1, \hat{\sigma}_{d1}) = \hat{\mathcal{A}}(\hat{\sigma}, \hat{\rho}, y) \\ \hat{t}_1 = \widehat{\mathsf{tick}}((ae_0\ y), \hat{t}_0) & \multicolumn{3}{c}{\widehat{\mathsf{gc}}_{in}(\hat{\sigma}_{d0}[\hat{\rho}(y) \mapsto (\hat{d}_1, \hat{\sigma}_{d1})], \hat{\rho}_0[x \mapsto \hat{\rho}(y)], e)\ \hat{t}_1 \Downarrow (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}\ \hat{t}_2}\end{array}}{\hat{\sigma}\ \hat{\rho}\ (ae_0\ y)\ \hat{t}_0 \Downarrow (\hat{d}, \hat{\sigma}_d)\ \hat{\xi}\ \hat{t}_2}$$

In either of these cases, if the binding of the variable y has abstract count 1, then a fresh address for x is not allocated. Instead, the current address of y is used. Notice that we must still evaluate the argument and extend the heap with it, but we are guaranteed to maintain precision because $\hat{\mathcal{A}}$ does not perform any allocation. When we want to know whether two different variables have the same binding, we can simply ensure that their abstract addresses are the same and have count 1.

We evaluate some aspects of binding invariants in § 11.4.

11 HFAC IMPLEMENTATION AND EVALUATION

We implemented HFAC as an abstract definitional interpreter [Darais et al. 2017] to evaluate its performance. We evaluate our implementation of HFAC against an AAM-based k-CFA with abstract counting (AAM-AC) on two different benchmark suites:

(1) the original abstract counting benchmark suite [Might and Shivers 2006b], which includes programs eligible for transducer fusion [Shivers and Might 2006]; and
(2) a set of R6RS Scheme programs designed to elicit a variety of analysis behaviors.

For each program, we ran each analysis in both context-insensitive ($k = 0$) and context-sensitive ($k = 1$) modes and, for each analysis, we applied garbage collection at each transition to ensure maximum precision. In addition to tabulating the size of the explored state space and the time taken to do so, we also tabulate the number of inlinings an constant propagations justified by the analysis.

A λ may be inlined at a site if only closures over it flow to that site, each of its free variables are in scope at that site, and the bindings in the closure environment are identical to the bindings at the site environment. Per Might and Shivers [2006b], we appealed to the environment bindings' abstract counts to establish the third condition. Bindings for ANF-introduced variables had count 1, as expected.

11.1 Extending the Language

Running HFAC on our benchmark suites requires that we extend it to support letrec, multiple parameter functions, primitive operations, conditional branching, and call/cc.

11.1.1 letrec. To add support for letrec to HFAC, we need to ensure only that the reachability computation respects letrec's scope rules. For our benchmarks, we rewrite definition contexts, which typically consist of function definitions and mutable variable declarations, to instances of letrec. In some cases, however, such a translation leads to a use of letrec with a "serious" expression providing the bound value. We support these more general forms by implementing letrec operationally in terms of let and set!, which is particularly well-behaved with the ability to perform strong update.

11.1.2 Multiple-Parameter Functions. Supporting functions which accept multiple parameters introduces no technical issues but does require that we, e.g., propagate freshness with respect to each parameter when the the result of a function is obtained.

11.1.3 Primitive Operations. To analyze the R6RS Scheme programs, we implemented several dozen primitive operations. Impure operations, such as `vector-set!` and `set-cdr!`, can potentially leverage abstract counts to perform strong update. In practice, the array or recursive structure is folded into a single abstract address, preventing the conditions under which strong update applies.

11.1.4 Conditional Branching. If the analysis cannot narrow a guard value to `\#true` or `\#false`, then it must analyze both branches. If the abstract count of the guard value is 1, however, it can sharpen the guard value in each branch for the value necessary to have taken it. This capability is of limited utility in this setting, since guard values are rarely scrutinized multiple times, but this policy can assist verification machinery integrated within the analysis [Might 2007c; Might et al. 2007].

11.1.5 call/cc. The continuation-based transducer programs require support for call/cc, which we add using the technique of Vardoulakis and Shivers [2011]. Uses of call/cc interfere with the abstract count summaries produced by HFAC. When calling a non-local continuation, there is no necessary correspondence between the bindings in the result heap fragment and those in the continuation's. We reflect this disconnect in the abstract count by flagging each address as fresh in the result heap fragment, which has the effect of distinguishing addresses in the result heap fragment from those in the continuation's heap fragment.

11.2 ΓCFA Benchmark

The original abstract counting evaluation [Might and Shivers 2006b] considers a small suite of programs which exhibit recursive patterns as well as heavy use of *call/cc*. Figure 11 tabulates the number of states/configurations, analysis time, and inlinings justified by each analysis. For each program and precision level, HFAC is able to justify more inlinings. For smaller programs, AAM-AC and HFAC exhibit similar performance in terms of analysis time and, to a lesser degree, the size of the explored state space. For the larger programs, which include uses of call/cc, performance diverges: HFAC takes significantly less time and encounters fewer distinct states/configurations. Because of the somewhat stark difference, we manually verified the coverage of each analysis.

	0CFA						1CFA					
	AAM-AC			HFAC			AAM-AC			HFAC		
Program	States	Time	Inlines	Configs	Time	Inlines	States	Time	Inlines	Configs	Time	Inlines
fact-tail	14	ϵ	2	15	ϵ	3	20	ϵ	2	15	ϵ	3
fact-y-combinator	30	ϵ	4	32	ϵ	5	49	1ms	4	43	2ms	5
nested-loops	23	ϵ	4	55	1ms	8	57	1ms	4	61	3ms	8
put-double-coroutines	417	34ms	44	148	3ms	64	862	172ms	44	149	6ms	65
integrate-fringe-coroutines	925	146ms	60	180	5ms	77	2281	1500ms	63	191	9ms	81
integrate-stream-coroutines	1070	184ms	55	295	12ms	61	2039	921ms	56	209	11ms	76

Fig. 11. Comparison between an AAM-AC and HFAC for both $k = 0$ and $k = 1$ call-site sensitivities on the original abstract counting benchmark suite. The table presents the number of states/configurations of each analysis, the median analysis time of three runs, and the number of inlines and constant propagations the flow analysis and abstract counting together justify.

We observe that the analysis size of HFAC stays essentially the same when moving from $k = 0$ to $k = 1$. Much of this is due to the use of flat domains for atomic values, a choice made to replicate the original AAM-AC benchmark. When $k = 0$, earlier heaps (and heap fragments) contain constants and later heaps contain $\top_X$ for a particular domain X. Thus, this abstraction offers a small degree of polyvariance by itself. When $k = 1$, the call-site sensitivity becomes saturated roughly as the constants become $\top_X$, offering little more discriminating power. This behavior completely explains the results of `fact-tail` which has the same number of abstract states for each k.

We don't see the same increase for HFAC as for AAM-AC on the transducer programs because of their intricate use of continuations. Because AAM-AC heap-allocates continuations, distinct uses of call/cc are susceptible to a combinatorial explosion as evaluation paths are explored for each combination of allocations and abstract counts. (The AAM-AC implementation used subsumption testing rather than naive equality testing for states, which reduced the number of states significantly.) In contrast, HFAC specifically does not allocate first-class continuations in its heap which diminishes its state space significantly and increases its precision.

11.3 R6RS Benchmark

Our second benchmark suite includes a variety of R6RS Scheme programs crafted to exercise CFAs in particular ways, such as to test its polyvariance capabilities (e.g. `eta`, `blur`, `kcfa-2`, `kcfa-3`), its ability to cull stale abstract resources via garbage collection (e.g. `facehugger`), its performance under pathological constructions (e.g. `sat-1`, `sat-2`, `sat-3`), and its performance on larger and more-representative programs (e.g. `regex`, `earley`). Figure 12 tabulates the number of states/configurations, analysis time, and inlinings justified by each analysis.

We observe that HFAC is more precise than AAM-AC on these programs, when both analyses succeed. On `boyer`, a logic programming benchmark, only HFAC at $k = 0$ succeeds, with each other analysis timing out. On `regex`, a derivative-based regular expression matcher, AAM-AC times out at $k = 1$ simply due to the large state space; HFAC itself produces a relatively large model for `regex` at $k = 1$. On `cpstak`, a CPS version of the Tak function, HFAC times out at $k = 1$. This timeout occurs in part because HFAC treats the continuation encoded as a first-class function as data. This encoding prevents both the control-flow aspect of the analysis from managing it and the heap fragment aspect of the analysis from separating any bindings. AAM-AC succeeds at $k = 1$ because the AAM-AC implementation uses subsumption testing for states whereas the HFAC implementation uses naive equality. (This difference explains why the explored state space of AAM-AC is smaller than for HFAC on `tak` as well.)

With the exception of `tak` and `cpstak`, HFAC takes the same or less time than AAM-AC to produce its model for the same k. In some cases, the difference is substantial. `sat-2`, a brute-force SAT solver, is an extreme example on which AAM-AC takes 8 seconds to explore over 13,000 states

	0CFA						1CFA					
	AAM-AC			HFAC			AAM-AC			HFAC		
Program	States	Time	Inlines	Configs	Time	Inlines	States	Time	Inlines	Configs	Time	Inlines
eta	13	ϵ	6	12	ϵ	8	13	ϵ	6	12	ϵ	8
blur	36	ϵ	12	28	ϵ	17	55	1ms	12	32	1ms	17
facehugger	32	ϵ	9	43	ϵ	11	75	1ms	9	43	ϵ	11
kcfa-2	21	ϵ	5	19	ϵ	8	21	ϵ	5	19	ϵ	8
kcfa-3	45	ϵ	7	39	ϵ	11	45	1ms	7	39	1ms	11
church	7823	8s	24	237	11ms	24	1478	295ms	25	72	3ms	26
mj09	23	ϵ	6	21	ϵ	13	27	ϵ	6	21	ϵ	13
ack	39	ϵ	4	43	ϵ	6	571	40ms	4	58	1ms	6
tak	43	1ms	5	1281	149ms	7	1487	481ms	5	2222	345ms	7
cpstak	53	1ms	6	201	12ms	11	1688	561ms	6	–	–	–
map	101	1ms	14	40	ϵ	16	110	3ms	14	67	2ms	16
flatten	34	ϵ	3	15	ϵ	5	919	135ms	3	53	1ms	5
loop2-1	23	ϵ	4	55	1ms	8	57	1ms	4	61	3ms	8
loop2-2	31	ϵ	4	71	3ms	8	75	3ms	4	77	7ms	8
state	14	ϵ	3	13	ϵ	5	20	ϵ	3	13	ϵ	5
sat-1	142	5ms	7	55	ϵ	29	857	61ms	7	78	2ms	29
sat-2	452	27ms	10	43	1ms	23	13,530	8s	10	43	2ms	23
sat-3	249	14ms	11	42	ϵ	26	12,384	6s	11	42	2ms	26
regex	7375	3s	53	553	26ms	111	–	–	–	2491	420ms	111
boyer	–	–	–	1166	2s	103	–	–	–	–	–	–
deriv	106	3ms	5	57	1ms	12	536	60ms	5	158	11ms	12
earley	354	38ms	41	253	20ms	58	1963	1s	41	644	141ms	58
mbrotZ	236	13ms	16	120	3ms	30	1020	289ms	16	175	17ms	30

Fig. 12. Comparison between an AAM-AC and HFAC for both $k = 0$ and $k = 1$ call-site sensitivities on a variety of R6RS programs. The table presents the number of states/configurations of each analysis, the median analysis time of three runs, and the number of inlines and constant propagations the flow analysis and abstract counting together justify. A dash indicates a failure to complete the analysis after 10 minutes.

at $k = 1$ whereas HFAC takes 2 milliseconds to encounter 43 configurations. For this particular example, the difference arises from the exponential number of candidate solutions the program tests, each of which is reachable from the stack.

11.4 Binding Invariants

In this section, we report on an initial evaluation of binding invariants. As with inlining, we separate the question of whether a binding invariant is usable from the question of whether it is useful. The criteria given in § 10 answer the former question. The answer to the latter depends on the policy which determines whether to use the detected invariant or create a fresh binding.

For first question, we report on the number of usable binding invariants encountered in each analysis. For the second, we report on the performance of a naive policy which always uses the detected invariant; however, this report is intended only to give an impression of the effectiveness of the naive policy and not of binding invariants in general. To assess the general effectiveness of binding invariants requires a comparison between different principled policies or a formulation of binding invariants which allows the choice to be made by the user at the point the binding information is needed.

Figure 13 tabulates the number of configurations, analysis time, and inlinings justified by HFAC without binding invariants against HFAC with binding invariants (abbreviated HFBI), using the aforementioned naive policy. In addition, it tabulates the number of times each analysis detected a binding invariant. In a few cases, the analysis detected no binding invariants while, in several of the more general programs, it detected hundreds and, in case, thousands. The high number of detected binding invariants underscores the need for an effective policy to determine whether the

	0CFA							1CFA						
	HFAC			HFBI				HFAC			HFBI			
Program	States	Time	Inlines	Configs	Time	Detected	Inlines	States	Time	Inlines	Configs	Time	Detected	Inlines
eta	12	ϵ	8	12	ϵ	0	8	12	ϵ	8	12	ϵ	0	8
blur	28	ϵ	17	32	ϵ	8	14	32	1ms	17	34	1ms	8	14
facehugger	43	ϵ	11	43	ϵ	3	7	43	ϵ	11	43	ϵ	3	7
kcfa-2	19	ϵ	8	19	ϵ	8	8	19	ϵ	8	19	ϵ	8	8
kcfa-3	39	ϵ	11	39	ϵ	24	11	39	1ms	11	39	1ms	24	11
church	237	11ms	24	172	4ms	109	19	72	3ms	26	75	2ms	24	20
mj09	21	ϵ	13	21	ϵ	2	13	21	ϵ	13	21	ϵ	2	13
ack	43	ϵ	6	43	ϵ	2	6	58	1ms	6	62	1ms	3	6
tak	1281	149ms	7	–	–	–	–	2222	345ms	7	5145	848ms	2944	7
cpstak	201	12ms	11	–	–	–	–	–	–	–	–	–	–	–
map	40	ϵ	16	62	1ms	6	16	67	2ms	16	67	2ms	6	16
flatten	15	ϵ	5	15	ϵ	0	5	53	1ms	5	53	1ms	0	5
loop2-1	55	1ms	8	55	1ms	11	8	61	3ms	8	55	3ms	11	8
loop2-2	71	3ms	8	106	4ms	21	8	77	7ms	8	106	10ms	21	8
state	13	ϵ	5	13	ϵ	2	5	13	ϵ	5	13	ϵ	2	5
sat-1	55	ϵ	29	55	ϵ	9	29	78	2ms	29	78	2ms	9	29
sat-2	43	1ms	23	43	1ms	8	21	43	2ms	23	43	2ms	8	21
sat-3	42	ϵ	26	42	ϵ	8	26	42	2ms	26	42	2ms	8	26
regex	553	26ms	111	1098	48ms	301	111	2491	420ms	111	2383	217ms	547	111
boyer	1166	2s	103	1586	2s	157	103	–	–	–	–	–	–	–
deriv	57	1ms	12	57	1ms	0	12	158	11ms	12	158	11ms	0	12
earley	253	20ms	58	621	35ms	217	56	644	141ms	58	765	155ms	231	56
mbrotZ	120	3ms	30	120	3ms	8	24	175	17ms	30	175	20ms	14	24

Fig. 13. Comparison between HFAC with and without binding invariants (HFBI) for both $k = 0$ and $k = 1$ call-site sensitivities on a variety of R6RS programs. The table presents the number of configurations of each analysis, the median analysis time of three runs, and the number of inlines and constant propagations the flow analysis and abstract counting together justify. For HFBI, it also presents the number of binding invariants detected during analysis. A dash indicates a failure to complete the analysis after 10 minutes.

invariant should be used. The naive policy in each case decreases (or leaves unchanged) the number of propagations (e.g. inlines) the analysis is able to justify, because the use of a binding invariant changes the variables the environment questions consider—variables in general less likely to be in scope.

Interestingly, the naive policy's use of binding invariants had a non-negligible effect on the size of the models produced by the analysis. `map`, `loop2-2`, and `boyer` see 50% increases in size, `tak`, `regex` at $k = 0$, and `early` see 100% or more, but `church` at $k = 0$ and `regex` at $k = 1$ see decreases in size.

12 RELATED WORK

This work builds directly on the heap fragment technique of Germane and Adams [2020]. In terms of CFA, we slightly generalize their technique by permitting structural mutation and not just variable mutation. We repurpose this technique for must-alias analysis and find it to be nearly ideal: its independence from the stack allows it to maintain precision even in recursive settings and yields a qualitatively more-precise analysis.

The heap fragment technique is analogous to the combination garbage collection/abstract counting (GC/AC) technique [Might and Shivers 2006b] in that it increases speed and precision of the control-flow analysis/must-alias analysis (CFA/MAA) to which it is applied. Our technique is not an alternative to GC/AC to achieve the same kind of analysis improvement but rather one we can apply to GC/AC, for example, to qualitatively improve their effectiveness. While it is straightforward to enhance a CFA/MAA formulated in an operational framework with GC/AC, enhancing CFA/MAA with heap fragments requires the underlying CFA to be stack-precise and requires a summarization-based account of the MAA.

Like Might and Shivers [2006b], HFAC is similar to the higher-order must-alias analysis of Jagannathan et al. [1998] in that it maintains a count of variable bindings and incorporates abstract reachability and garbage collection. We share the differences too: HFAC is polyvariant, obtains full precision after only one run, and is formulated operationally. HFAC is further distinguished by providing a summary-based account of abstract counting. Finally, like Jagannathan et al. [1998], our analysis separates the bindings reachable from a call from those reachable only from its continuation. For this reason, both analyses can reason more capably about recursion than standard abstract counting.

12.1 Environment Analysis

Several theories of environment analysis, which can be used to answer must-alias questions in limited settings, have been developed since Might and Shivers's abstract counting. We review several representative ones.

Facchinetti et al. [2017] use the notion of relative store fragments to identify singleton abstractions (i.e. abstractions with count 1) in the setting of a demand-driven analyzer. A relative store fragment annotates each variable with a kind of delta frame string (à la ΔCFA [Might and Shivers 2006a]) relative to the current program point. HFAC is spiritually similar to this analysis where the former extends a kind of localized heap with one instrument of must-alias analysis—abstract counts—and the latter with another—delta frame strings. In fact, their use of delta frame strings likely looks similar to the heap fragment technique instantiated for delta frame strings rather than abstract counts. However, the demand-driven framework of relative store fragments is different enough from our setting—an exhaustive, stack-precise CFA—that porting it over would likely not be trivial.

Unchanged variable analysis (UVA) [Bergstrom et al. 2014] transforms environment binding questions into graph reachability questions. In particular, UVA determines that a variable's binding is unchanged at a destination site (e.g. a call site) from a given source site when no control flow path from the source to the destination passes through a rebinding of that variable. This criterion allows UVA to answer environment questions using only the CFA-produced control flow model. However, Bergstrom et al. recognize that, because their technique cannot distinguish between environments created on different control-flow paths, it is more limited than dedicated analyses such as abstract counting or ΔCFA [Might and Shivers 2006a].

ΔCFA [Might and Shivers 2006a] is a theory of environment analysis based on stack change. A ΔCFA analysis tracks the net stack change between each pair of points in a control flow graph and determines which variable rebindings the stack change implies. Originally formulated for finite-state control-flow models, ΔCFA, like ΓCFA, has only limited ability to reason about recursion. Germane and Might [2017] apply ΔCFA's environment theory to pushdown control-flow models to increase this ability. Like UVA, their application does not required dedicated environment analysis machinery within the implementation and instead analyzes the pushdown control-flow model to answer environment questions. It is unknown to us how effective this theory would be applied to a pushdown model obtain via heap fragments.

13 DISCUSSION AND CONCLUSION

By isolating the evaluation of an expression from its continuation, the use of heap fragments provides a qualitative increase to reasoning power about bindings which mingle with recursion. Given that recursion is pervasive in functional programs, we believe this increase makes the heap fragment-based analysis worth its formal weight. However, isolating bindings with heap fragments makes it difficult to determine global properties about bindings. For instance, it is not apparent how to determine whether only one concrete binding ever exists at once for a particular abstract binding, in which case the binding could be globalized [Sestoft 1989].

REFERENCES

Lars Bergstrom, Matthew Fluet, Matthew Le, John Reppy, and Nora Sandler. 2014. Practical and Effective Higher-Order Optimizations. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/2628136.2628153

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

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

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

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

Leandro Facchinetti, Zachary Palmer, and Scott F. Smith. 2017. Relative Store Fragments for Singleton Abstraction. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10422)*, Francesco Ranzato (Ed.). Springer, 106–127. https://doi.org/10.1007/978-3-319-66706-5_6

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

Kimball Germane and Michael D. Adams. 2020. Liberate Abstract Garbage Collection from the Stack by Decomposing the Heap. In *Programming Languages and Systems - 29th European Symposium on Programming, ESOP 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25-30, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12075)*, Peter Müller (Ed.). Springer, 197–223. https://doi.org/10.1007/978-3-030-44914-8_8

Kimball Germane and Matthew Might. 2017. A Posteriori Environment Analysis with Pushdown Delta CFA. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017)*. Association for Computing Machinery, New York, NY, USA, 19–31. https://doi.org/10.1145/3009837.3009899

Thomas Gilray, Michael D. Adams, and Matthew Might. 2016a. 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

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016b. 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

Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew K. Wright. 1998. Single and Loving It: Must-Alias Analysis for Higher-Order Languages. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, David B. MacQueen and Luca Cardelli (Eds.). ACM, 329–341. https://doi.org/10.1145/268946.268973

James Ian Johnson and David Van Horn. 2014. Abstracting abstract control. In *Proceedings of the 10th ACM Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '14)*. ACM, New York, NY, USA, 11–22. https://doi.org/10.1145/2661088.2661098

Matthew Might. 2007a. *Environment analysis of higher-order languages.* Ph.D. Dissertation. Georgia Institute of Technology. http://hdl.handle.net/1853/16289

Matthew Might. 2007b. Logic-Flow Analysis of Higher-Order Programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. Association for Computing Machinery, New York, NY, USA, 185–198. https://doi.org/10.1145/1190216.1190247

Matthew Might. 2007c. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Nice, France) *(POPL '07)*. ACM, New York, NY, USA, 185–198. https://doi.org/10.1145/1190216.1190247

Matthew Might. 2010. Shape Analysis in the Absence of Pointers and Structure. In *Verification, Model Checking, and Abstract Interpretation*, Gilles Barthe and Manuel Hermenegildo (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 263–278. https://doi.org/10.1007/978-3-642-11319-2_20

Matthew Might, Benjamin Chambers, and Olin Shivers. 2007. Model Checking Via GammaCFA. In *Verification, Model Checking, and Abstract Interpretation, 8th International Conference, VMCAI 2007, Nice, France, January 14-16, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4349)*, Byron Cook and Andreas Podelski (Eds.). Springer, 59–73. https://doi.org/10.1007/978-3-540-69738-1_4

Matthew Might and Panagiotis Manolios. 2009. *A posteriori* soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2009)*. Savannah, Georgia, USA. https://doi.org/10.1007/978-3-540-93900-9_22

Matthew Might and Olin Shivers. 2006a. Environment Analysis via ΔCFA. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Charleston, South Carolina, USA) *(POPL '06)*. Association for Computing Machinery, New York, NY, USA, 127–140. https://doi.org/10.1145/1111037.1111049

Matthew Might and Olin Shivers. 2006b. 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

Jens Palsberg. 1995. Closure Analysis in Constraint Form. *ACM Trans. Program. Lang. Syst.* 17, 1 (Jan. 1995), 47–62. https://doi.org/10.1145/200994.201001

Amr Sabry and Matthias Felleisen. 1994. Is Continuation-Passing Useful for Data Flow Analysis?. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/178243.178244

Peter Sestoft. 1989. Replacing Function Parameters by Global Variables. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, Joseph E. Stoy (Ed.). ACM, 39–53. https://doi.org/10.1145/99370.99374

Olin Shivers. 1988. Control-Flow Analysis in Scheme. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI), Atlanta, Georgia, USA, June 22-24, 1988*, Richard L. Wexelblat (Ed.). ACM, 164–174. https://doi.org/10.1145/53990.54007

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

Olin Shivers and Matthew Might. 2006. Continuations and transducer composition. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 295–307. https://doi.org/10.1145/1133981.1134016

Paul A. Steckler and Mitchell Wand. 1997. Lightweight Closure Conversion. *ACM Trans. Program. Lang. Syst.* 19, 1 (Jan. 1997), 48–86. https://doi.org/10.1145/239912.239915

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

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

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

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