# Improving Flow Analyses via ΓCFA

## Abstract Garbage Collection and Counting

Matthew Might
Georgia Institute of Technology
mattm@cc.gatech.edu

Olin Shivers
Northeastern University
shivers@ccs.neu.edu

## Abstract

We present two independent and complementary improvements for flow-based analysis of higher-order languages: (1) *abstract garbage collection* and (2) *abstract counting*, collectively titled ΓCFA.

Abstract garbage collection is an analog to its concrete counterpart: we determine when an abstract resource has become unreachable, and then reallocate it as fresh. This prevents flow sets from merging in the abstract, which has two immediate effects: (1) the precision of the analysis is increased, and (2) the running time of the analysis is frequently reduced. In some nontrivial cases, we achieve an order of magnitude improvement in precision and time *simultaneously*.

In abstract counting, we track how many times an abstract resource has been allocated. A count of one implies that the abstract resource momentarily represents only one concrete resource. This, in turn, allows us to perform environment analysis and to expand the kinds (rather than just the degree) of optimizations available to the compiler.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Optimization

***General Terms*** Languages

***Keywords*** Gamma-CFA, program analysis, flow analysis, environment analysis, functional languages, lambda calculus, superbeta, inlining, CPS, continuations, abstract garbage collection, abstract counting

## 1. Introduction

Experience in the field of flow analysis leads to a perception that speed and precision are fundamental tradeoffs. 0CFA is cubic in the worst case, and the more precise 1CFA is exponential. In general, the next step up the ladder of precision, $(k + 1)$-CFA, is always slower then the one beneath it, $k$-CFA. Given this information alone, the tradeoff seems natural.

Our message is just the opposite: speed and precision are not necessarily tradeoffs; in many cases, higher speed is a direct consequence of higher precision. What the conventional wisdom is missing is a key fact about the nature of (im)precision: the manner in which imprecision reinforces itself during a flow analysis through an ever-worsening feedback loop.

To get a feel for how this happens, take a look at the following Scheme fragment:

```
(let ((x y))
  ((if (equal? x y) f g) z))
```

Clearly the condition `(equal? x y)` is always true, which means that `f` always executes and `g` never will. A flow analysis such as 0CFA, however, will often miss this information and take the branch to `g` anyway. Let's investigate why this is so.

During the analysis, multiple values could flow to `y` at different times. For the moment, suppose just the constant `0` has flowed to `y`. From this, 0CFA then infers that `0` also flows to `x`. Temporarily, 0CFA knows that `x`'s value is restricted to the set {`0`} and that `y`'s is restricted to the set {`0`}, and this is enough information to infer that `(equal? x y)` holds.

Moving forward, let another constant, `5`, also flow to `y`. The `5` is then seen flowing to `x`. Now the set of possible values for `x` is the set {`0`, `5`}, with the same set for `y`. When 0CFA looks at the condition `(equals? x y)`, it doesn't know if it's comparing `0` to `0`, `0` to `5` or `5` to `5`. Being a conservative analysis, 0CFA then chases down *both* branches of the `if`. As a result, all of the time that 0CFA spends investigating the `g` branch is completely wasted, since it doesn't correspond to something which can happen in any execution. *All* of the information generated on the `g` branch is spurious and serves to further taint the analysis. Making matters worse, this tainted information can cause other spurious forks during the analysis, leading to a vicious cycle of ever-worsening information.

We present a way to end the cycle by intermittently and safely improving precision during analysis. This tightens up the set of values to which an expression might evaluate. These tighter bounds (better information) decrease the chance that a spurious branch will be investigated during analysis. The mechanism we use to accomplish this is a form of abstract garbage collection applied to the environment structure.

Once we install the abstract garbage collector, however, another opportunity to improve appears in the form of *abstract counting*. The idea here is to count (abstractly) the number of concrete objects to which a given abstract object corresponds. When this count is one, the abstract object is effectively concrete. With this information, we can make inferences that exceed the power of ordinary control-flow analysis. For instance, suppose we're told that two sets of values, $S_1$ and $S_2$, are equal, but we don't know what's in either set. What could we say about the contents of these sets if we were told that the size of both sets was one? We could say, *without knowing what's in the sets*, that any member of $S_1$ is equal to any member of $S_2$ and *vice versa*. By adapting this reasoning to a control-flow analysis, we gain the ability to perform environment analysis as well.



*ICFP '06* September 16–21, 2006, Portland, Oregon, USA.

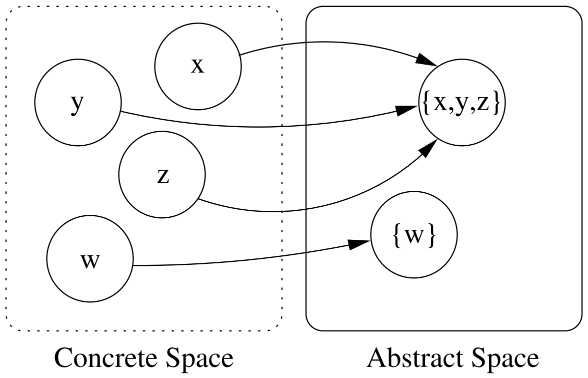



**Figure 1.** Collisions in the concrete-to-abstract map.

Our work here defines *abstract garbage collection* and *abstract counting* for environment structure, proves the correctness of both and provides results from a running implementation. Our primary contribution is a framework for performing:

1. An abstract-garbage-collecting control-flow analysis.
2. An abstract-counting-based environment analysis.

For nontrivial examples, we are able to achieve large speedups for both 0CFA and 1CFA. At the same time, we manage to improve precision by an order of magnitude. We'll discuss connections to other work in more detail in a later section.

## 2. Conventions

For all of the domains used in this work, we assume the "natural" meaning for the lattice operators $\sqcap$ and $\sqcup$ as well as the relation $\sqsubseteq$; that is, a point-wise lifting (for functions), or an index-wise lifting (for vectors and tuples). We also assume an implicit and appropriate top $\top$ and bottom $\bot$ element for domains that need them. For a power domain $A = \mathcal{P}(B)$, we define $\bot_A = \emptyset$ and $\top_A = B$; the order relation and the meet and join operators are then

$$X \sqsubseteq_A Y \text{ iff } \forall x \in X : \exists y \in Y : x \sqsubseteq_B y$$
$$X \sqcup_A Y = X \cup Y$$
$$X \sqcap_A Y = \{b \in X \cup Y : \{b\} \sqsubseteq_A X \text{ and } \{b\} \sqsubseteq_A Y\}.$$

The vertical bar '|' operator denotes function restriction, *i.e.*, $f|X$ is the function $f$ defined at most over elements in the set $X$. When a function is applied to an element outside of its domain, it yields $\bot$; thus, we get $dom(f) = \{x : f(x) \neq \bot\}$. The function *free* returns the set of free variables for a given piece of syntax. We use boldface to denote vectors, *i.e.*, $\boldsymbol{d} = \langle d_1, \ldots, d_n \rangle$. The "absolute value" notation $|x|$ should be read and interpreted as "the abstraction of $x$." The function $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ is the function $f$ except that when applied to $x_i$, it yields $y_i$. Operators are implicitly lifted point-wise over ranges for functions; that is: if $\oplus : Y \times Y \to Y$ and $f, g : X \to Y$, then $f \oplus g = \lambda x. f(x) \oplus g(x)$.

## 3. The problem: too many pigeons

During an analysis performed through abstract interpretation, it is typically the case that an infinite, concrete space in which computation occurs is compressed into some finite, abstract space. It is inevitable, then, that some elements of the abstract domain represent multiple elements of the concrete space (Figure 1). It's this overlapping in the abstract that leads to imprecision in reasoning.

***Example: Abstract integers*** To get a better feel for the problem, consider an abstraction of the integers to their sign. The concrete set is the integers, $\mathbb{Z}$. The abstract set is the power set of signs, $\widehat{\mathbb{Z}} = \mathcal{P}(\{-, 0, +\})$. The abstraction map $|\cdot| : \mathbb{Z} \to \widehat{\mathbb{Z}}$ in this case is:

$$|z| = \begin{cases} \{-\} & z < 0 \\ \{0\} & z = 0 \\ \{+\} & z > 0. \end{cases}$$

The addition operator, $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, abstracts naturally to $\oplus : \widehat{\mathbb{Z}} \times \widehat{\mathbb{Z}} \to \widehat{\mathbb{Z}}$. For example:

$$\begin{aligned} \{0\} \oplus \{0, +\} &= \{0, +\} \\ \{+\} \oplus \{+\} &= \{+\} \\ \{+\} \oplus \{-\} &= \{-, 0, +\} \\ \{+, -\} \oplus \{0\} &= \{-, +\}. \end{aligned}$$

Suppose we wish to analyze the expression $4 + -4$ with an abstract interpretation. To do so, we evaluate $|4| \oplus |-4|$, and get back $\{-, 0, +\}$. At this point, it's worth noting several things:

- Had we simply evaluated $4 + -4$ and then abstracted, we would have $|4 + -4| = \{0\}$. That is, abstract interpretation conservatively overapproximated, even though a tighter answer was possible.
- The set $\{0\}$ has only one concrete counterpart: 0. So, if we can find a tighter way to do abstract interpretation, then the abstract interpretation may in some cases yield the exact concrete result.
- Because $\{0\}$ has only one concrete counterpart, it acts as if it were concrete. That is,

  $$\{0\} \oplus \hat{z} = \hat{z} \oplus \{0\} = \hat{z},$$

  in which case, no precision is lost.
- When comparing abstract values, we cannot ordinarily infer concrete equality from the abstract. That is,

  $$|z_1| \subseteq \hat{z}_1 \text{ and } |z_2| \subseteq \hat{z}_2 \text{ and } \hat{z}_1 = \hat{z}_2 \not\Longrightarrow z_1 = z_2,$$

  *unless* the abstract values correspond to one concrete element:

  $$|z_1| \subseteq \hat{z}_1 \text{ and } |z_2| \subseteq \hat{z}_2 \text{ and } \hat{z}_1 = \hat{z}_2 = \{0\} \implies z_1 = z_2.$$

***Example: A simple flow analysis*** When using abstract interpretation for a flow analysis, we encounter a similar set of problems. Consider control-flow analysis for the pure, call-by-value $\lambda$-calculus given by the following grammar:

$$\begin{aligned} e, f \in EXP ::= &\; v \\ \mid &\; (\lambda\ (v)\ e) \\ \mid &\; (f\ e) \end{aligned}$$

Starting with the concrete, environment-based semantics given in the left-hand side of Figure 2, we can drop the environment component $\rho$ to arrive at the abstract, control-flow constraints given in the right-hand side of Figure 2.

By finding the least relation $\gtrsim$ such that the constraints in Figure 2 are satisfied, the relation $\gtrsim$ represents the results of Shivers' 0CFA [8]. For example, with the following program:

```
((λ (x) (x x)) (λ (y) (y y)))
```

We get x $\gtrsim$ $[\![$`(λ (y) (y y))`$]\!]$ and y $\gtrsim$ $[\![$`(λ (y) (y y))`$]\!]$. Now, take the following program fragment:

```
(let* ((id     (λ (x) x))
       (unused (id lam)))
  (id lam′))
```

While analyzing this fragment, 0CFA picks up x $\gtrsim$ *lam* from the call `(id lam)`. Next, it picks up x $\gtrsim$ *lam′* from the body of the `let*`. Because `x` is the body of `id`, 0CFA thinks that *lam* and *lam′* could be returned anywhere that `id` is called. As a result, 0CFA tells

$$\frac{\begin{array}{l}(f,\rho) \Rightarrow ([\![ (\lambda\ (v)\ e_b) ]\!], \rho') \\ (e,\rho) \Rightarrow d \\ (e_b, \rho'[v \mapsto d]) \Rightarrow (lam, \rho'')\end{array}}{([\![ (f\ e) ]\!], \rho) \Rightarrow (lam, \rho'')} \quad \text{[apply]} \quad \left\{ \frac{f \ggg [\![ (\lambda\ (v)\ e_b) ]\!] \qquad e_b \ggg lam}{[\![ (f\ e) ]\!] \ggg lam} \right.$$

$$(lam, \rho) \Rightarrow (lam, \rho) \quad \text{[eval-lambda]} \quad lam \ggg lam$$

$$(v, \rho) \Rightarrow \rho(v) \quad \text{[eval-var]} \quad \left\{ \begin{array}{l} \text{For every application term } (f\ e)\text{:} \\ \dfrac{f \ggg [\![ (\lambda\ (v)\ e_b) ]\!] \qquad e \ggg lam}{v \ggg lam} \end{array} \right.$$

**Figure 2.** A concrete big-step semantics for call-by-value $\lambda$-calculus (left), and its abstract control-flow constraints for 0CFA (right).

us that the above fragment could yield either *lam* or *lam*′, when in fact, only *lam*′ is possible.

The root cause of this loss of precision is the way in which environments are handled in the abstract: all bindings to a given variable are merged together. To alleviate this over-approximation, more sophisticated flow analyses arrange for bindings made in different contexts (known as *contours*) to be distinguishable from one another. Shivers' 1CFA, for instance, uses a distinct abstract context for each call site. That is, when a $\lambda$ term is invoked at a call site, the context for the binding made there is the call site itself. Agesen's [1] CPA, on the other hand, utilizes the cartesian product of the types of the arguments for a contour. The main idea behind these solutions is to create a finite set of abstract contexts in which bindings may occur. As a result, only bindings sharing the same abstract context merge.

In the end, all of these approaches still suffer the same problem: the set of abstract contours is finite, so some merging is inevitable for any nontrivial program. Our work is concerned, in part, with how to improve the precision of an analysis based on these abstract contour sets. Our methods, however, generalize to other resources allocated during an abstract interpretation, including store locations, list cells and timestamps.

## 4. Abstract garbage collection and counting

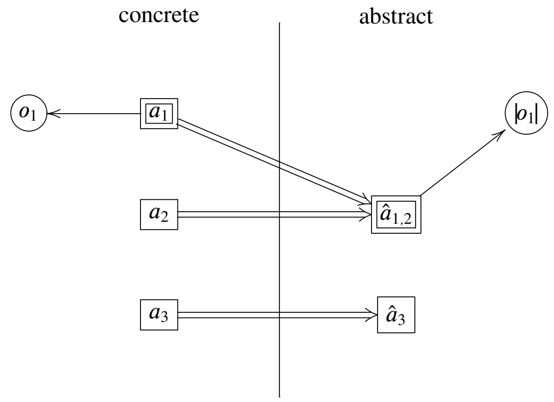
Note: To avoid confusion between the ideas of *garbage collection* and *collecting semantics*, both of which are used in this work, we will avoid using the term "collecting" or "collection" in an unqualified context. We will use the term "GC" when we mean something related to garbage collection.

In a flow analysis, abstract bindings (introduced later) are a finite resource. Ultimately, we run out of them, and when we do, we have to pick one to reuse. In doing so, we end up merging values associated with abstract bindings together. Just as with concrete store locations, however, we can apply GC to this finite resource in the abstract. Whenever we find that some abstract bindings have become unreachable, we are able to safely reallocate them as fresh.

In this section, we'll take a high-level view of abstract garbage collection, in order to build some intuition for later sections.

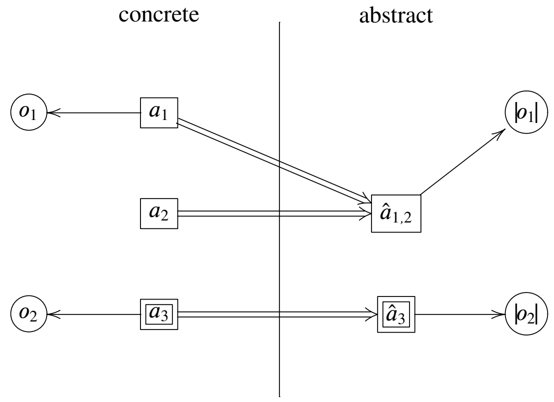
Take a concrete machine with three addresses: $a_1$, $a_2$ and $a_3$. We can imagine an abstract version of this machine with only two abstract addresses, $\hat{a}_{1,2}$ and $\hat{a}_3$. In this example, $\hat{a}_{1,2}$ is the abstract address for $a_1$ and $a_2$, and $\hat{a}_3$ is the abstract address for $a_3$.

Start by allocating a new object, $o_1$, to address $a_1$. The resulting heap (left) and its abstract counterpart (right) look like this:




The root pointer, presumably coming from a register or the stack, is represented by a double box. The double arrow $\Rightarrow$ means "abstracts to." Consequently, $\hat{a}_{1,2}$ points to $|o_1|$, the abstract counterpart of $o_1$.

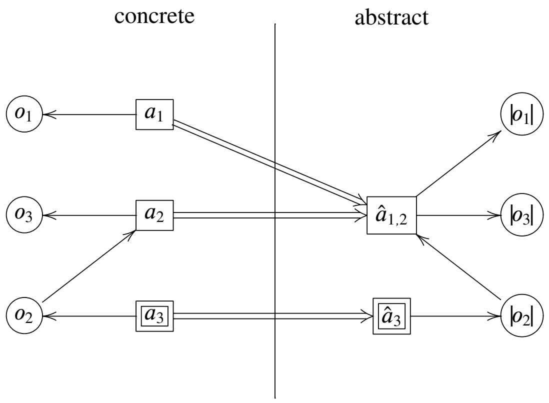
Next, allocate a new object, $o_2$, to address $a_3$. Simultaneously, shift the root pointer to $a_3$. Note that with the move of the root pointer, $o_1$ has become "unreachable." *Without* garbage collection (in the concrete or in the abstract), we would then have the following setup:[1]




Now we're ready to get into trouble. Allocate a third object, $o_3$, and give it the fresh address $a_2$ in the concrete. This means that $|o_3|$, its abstract counterpart, is allocated to $\hat{a}_{1,2}$. Also, form a pointer from

[1] *With* garbage collection, we would remove $o_1$ and $|o_1|$, because there is no path from the root to either.

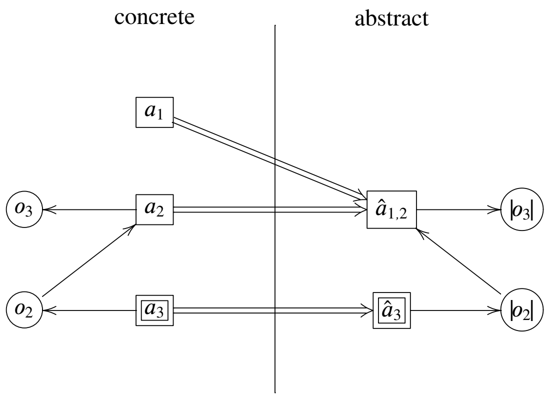
$o_2$ to $o_3$. This gives us the following picture:




Note how the concrete and the abstract have diverged—how the abstract has begun to overapproximate the concrete. The abstract address $\hat{a}_{1,2}$ now claims to have two objects to which it could point, $|o_1|$ and $|o_3|$, when in fact, only $|o_3|$ is possible.

In effect, $|o_1|$ is a *zombie*—a previously dead (unreachable) object which has become live again. Whereas previously there was no path from the root $\hat{a}_3$ to $|o_1|$, we now have the path $\hat{a}_3 \to |o_2| \to \hat{a}_{1,2} \to |o_1|$. Note also that the concrete version of this path does not exist. In general, zombies are created when the abstract address pointing to a dead object is reallocated.

Now, reset to the machine state described by the second diagram. Garbage collect $o_1$ and $|o_1|$, and then repeat the last round of changes. This leads to the following image:




By performing garbage collection in the concrete and the abstract, we have prevented overapproximation.

This exercise hints at another way to enhance the analysis, not in precision, but in power. When an abstract address is freshly allocated, it corresponds to only one concrete address. As a result, from the time in which an abstract address is freshly allocated to the time in which it is reallocated, we can treat the address and whatever it points to as "concrete." Moreover, in the interim, equality for the address in the abstract implies equality for the address in the concrete.

By tracking the number of times an abstract address has been allocated and resetting the count to zero if it gets garbage collected, we have a mechanism for performing *environment* analysis. We briefly review the utility of environment analysis in Section 10. We term the mechanism of counting abstract allocations *abstract counting*.

# 5. A concrete semantics

In this section, we develop a concrete, garbage-collecting semantics. A slight but important twist here is that our GC operates over environment structure rather than a store. The analysis we develop later is then just an abstract interpretation [3] of these semantics.[2]

Here, we start off with an environment-based semantics for the call-by-value, multi-argument $\lambda$-calculus, and through a series of transformations, we will evolve a semantics that suits our purposes. Readers, especially those already comfortable with continuation-passing style (CPS), may safely skip the intervening semantics and go directly to the final version, if they wish. The walk through the transformation is included purely to help build understanding of our semantics in gradual steps.

## 5.1 CPS

As a design constraint, we would like our analysis be able to handle languages that provide full continuations. Partly to achieve this, we will define our analysis in terms of a continuation-passing style representation. Using CPS will also, as we'll see, simplify the mathematics we develop. CPS is $\lambda$-calculus with a simple restriction: function calls do not return—they are one-way control transfers. Instead of returning, each procedure $p$ takes an additional argument, another procedure known as $p$'s *continuation*. $P$'s contract is that it will invoke the continuation supplied by the caller, passing it the "return" value that $p$ computed. Thus, instead of writing

```
(* (+ w x) (- y z))
```

which would require the `+` and `-` procedures to return values to their calling context, we write

```
(+ w x (λ (a) (- y z (λ (b) (* a b k)))))
```

where $k$ is the continuation for the top-level multiply. Again, the new contract for the `-` procedure is, "The procedure `-` takes three arguments: two numbers, $i$ and $j$, and a continuation $k$. It computes the difference $i - j$, and passes this value to procedure $k$." Thus the continuation $k$ passed to a procedure $p$ encodes, as a procedure, $p$'s calling context; the continuation represents "the rest of the computation" to be performed after $p$ is done.

The procedures-do-not-return stricture is reflected in the grammar for CPS, which differs from the traditional, or "direct-style" $\lambda$-calculus in that:

- call forms may only appear as the body of a $\lambda$ expression;
- $\lambda$ expressions can only have call forms as their body; and
- the arguments to a call form must be variable references or $\lambda$ expressions.

A side-by-side view of their grammars highlights the differences between direct-style and CPS:

$$
\overbrace{\begin{array}{l} e, f \in \mathit{EXP} ::= v \\ \qquad\qquad\;\; \mid \; (\lambda\ (v^*)\ e) \\ \qquad\qquad\;\; \mid \; (f\ e^*) \end{array}}^{\text{Direct-Style } \lambda\text{-calculus}}
\qquad
\overbrace{\begin{array}{l} e, f \in \mathit{EXP} \;\; ::= v \mid \mathit{lam} \\ \mathit{lam} \in \mathit{LAM} \;\; ::= (\lambda\ (v^*)\ \mathit{call}) \\ \mathit{call} \in \mathit{CALL} ::= (f\ e^*) \end{array}}^{\text{CPS } \lambda\text{-calculus}}
$$

where $v$ is a variable, a member of $\mathit{VAR}$.

To translate a call-by-value direct-style program into an equivalent CPS program, we can use the following translator $T$, which

[2] Ordinarily, one wouldn't add garbage collection to a semantics. A semantics is meant to map a program to a result, and adding garbage collection does not change this result. The reward for adding GC is invisible until we perform the abstract interpretation.

takes a direct-style term and a continuation $q$ awaiting its result:[3]

$$T\,[\![v]\!]\;q = [\![(q\ v)]\!]$$
$$T\,[\![(\lambda\ (v)\ e)]\!]\;q = [\![(q\ (\lambda\ (v\ v')\ call))]\!] \quad \text{where } call = T\ e\ v'$$
$$T\,[\![(f\ e)]\!]\;q = T\ f\ [\![(\lambda\ (v')\ call)]\!] \quad \text{where } call = T\ e\ [\![(\lambda\ (v'')\ (v'\ v''\ q))]\!].$$

In the above, primed variables ($v'$, $v''$) are fresh variables.

### 5.2 Deriving an environment-based CPS semantics

***Specialising to CPS*** Even though we could use an ordinary $\lambda$-calculus semantics to interpret CPS, its syntactic restrictions permit a much simpler interpretation, one in which "function call" is explicitly modelled as a one-way control transfer. Begin with an environment-based semantics, $\Rightarrow \subseteq (EXP \times Env) \times (EXP \times Env)$, for direct-style $\lambda$-calculus:

$$\frac{(f,\rho) \Rightarrow^* ([\![(\lambda\ (v_1 \cdots v_n)\ e)]\!], \rho') \qquad (e_i,\rho) \Rightarrow^* d_i}{([\![(f\ e_1 \cdots e_n)]\!], \rho) \Rightarrow (e, \rho'[v_i \mapsto d_i])} \quad \text{(app-prog)}$$

$$(lam,\rho) \Rightarrow (lam,\rho) \quad \text{(lambda-eval)}$$

$$(v,\rho) \Rightarrow \rho(v). \quad \text{(var-eval)}$$

An environment $\rho$ is a function $VAR \to D$; the set of denotable values $D$ just contains closures—$\lambda$ terms paired with an environment. Evaluating an expression $e$ in this semantics consists of finding $(lam,\rho)$ such that $(e,[]) \Rightarrow^* (lam,\rho)$.

Now, assume that this semantics is being used to interpret a program written in CPS, and consider what becomes degenerate. In CPS, execution begins with the (app-prog) rule on some top-level *call* form. The (app-prog) rule will never be used to satisfy an antecedent of the (app-prog) rule in CPS, as neither the procedure $f$ nor any argument $e_i$ can be a call form. Consequently, the antecedents in the (app-prog) rule are immediately satisfied with the rules (lambda-eval) or (var-eval).

Noting that the (lambda-eval) and (var-eval) rules are trivial, their degenerate, CPS-specific semantics becomes:

$$\mathcal{A}(v,\rho) = \rho(v)$$
$$\mathcal{A}(lam,\rho) = (lam,\rho).$$

That is, the -eval rules have been captured in the argument evaluator function $\mathcal{A}$. The remaining transition rule (app-prog) becomes:

$$([\![(f\ e_1 \cdots e_n)]\!], \rho) \Rightarrow (call, \rho'[v_i \mapsto d_i])$$

$$\text{where} \begin{cases} ([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \rho') = \mathcal{A}(f,\rho) \\ d_i = \mathcal{A}(e_i,\rho). \end{cases}$$

At this point, we have a specialized concrete semantics for CPS, but we need to make additional changes before we can add garbage collection.

***Adding a time counter*** It will be useful later on to add a time counter $t$ to our transition system, giving us:

$$([\![(f\ e_1 \cdots e_n)]\!], \rho, t) \Rightarrow (call, \rho'[v_i \mapsto d_i], t+1)$$

$$\text{where} \begin{cases} ([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \rho') = \mathcal{A}(f,\rho) \\ d_i = \mathcal{A}(e_i,\rho). \end{cases}$$

This time counter is a resource that is guaranteed to always be fresh, a property we utilize in the next transformation. Times are also ordered, which permits chronological reasoning in proofs.

***Factoring the environment structure*** Certain headaches are avoidable by removing the recursion from the domains here. Recursive domains enter our semantics because environments, $\rho$, have the structure $Env = VAR \to LAM \times Env$. By factoring the environment into a binding environment $\beta : VAR \to Time$ and a variable environment $ve : VAR \times Time \to D$, the recursion goes away. The binding half of the environment, $\beta$, yields the time $\beta(v)$ that some variable $v$ was bound for some particular lexical context. The global environment half, $ve$, keeps track of all the variable/value bindings made during execution of the program: it maps a variable/binding-time pair to its associated value $ve(v,t)$. By "tagging" variable bindings in $ve$ with their binding times, multiple, distinct bindings of the same variable can coexist in a $ve$ environment. We call $ve$ "global" because it is a single, shared component of machine state, while the local $\beta$ environments are what we find in closures, where $\rho$ used to be. Thus, instead of accessing a variable's value with $\rho(v)$, we now fetch it with $ve(v,\beta(v))$.

Putting this all together results in the following semantics:

$$([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, t) \Rightarrow (call, \beta'[v_i \mapsto t'], ve[(v_i,t') \mapsto d_i], t')$$

$$\text{where} \begin{cases} ([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \beta') = \mathcal{A}(f,\beta,ve) \\ d_i = \mathcal{A}(e_i,\beta,ve) \\ t' = t+1. \end{cases}$$

Because the environment has been split, we have to modify the evaluation function $\mathcal{A}$. The new definition of $\mathcal{A}$ is shown in Figure 4. With this factoring, we can also now refer to a particular *binding*, a variable/time coupling. Bindings are somewhat like the concrete addresses used the previous section, and $ve$ is somewhat like a store. In fact, bindings are the resource over which our garbage collection algorithm will operate. [4]

***Eval/apply transitions*** A final factoring (Figure 3) of these semantics simplifies the integration of extra features beyond our simple core language, such as `letrec`, basic values, primops, conditionals and a store. This factoring splits the transition rule into two stages: (1) an eval stage when the arguments to a call are evaluated, and (2) an apply stage when a procedure is applied to a vector of values.

If we were to compose the eval transition with the apply transition, we would end up with the original transition relation.

### 5.3 CPS as a state machine

Figures 5 and 3 show the semantic domains and the transition rules for our CPS semantics; these definitions, together with ones for the $\mathcal{A}$ function in Figure 4 comprise our complete, final concrete semantics. For convenience, given some state $\varsigma$, we will frequently refer to its components by subscripting them with $\varsigma$; that is, $\varsigma = (\ldots, ve_\varsigma, t_\varsigma)$. A primitive continuation *halt* has been added to the set of values; execution terminates when the *halt* continuation is applied.

Note that something pleasant happened when we specialised this semantics from the original direct-style rules: the final small-step semantics defines a simple state machine, one which alternates, tick-tock, between $(call, \beta, ve, t)$ eval states, and $(proc, \boldsymbol{d}_{\text{args}}, ve, t)$ apply states. The machine-like nature of the system is captured by the fact that the transition system is now defined by a pair of axiom rules—there are no recursive inference rules. The time counter is now clearly a "machine clock" that assigns a unique, ordered timestamp to each kind of state, and our semantic domains are no longer recursively defined.

[3] The transform $T$ given here handles the single-argument direct-style $\lambda$-calculus. Generalizing $T$ to handle the multi-argument $\lambda$-calculus we use is not difficult.

[4] At this point, our set *Time* has become equivalent to the concrete contour set *CN* in Shivers' work [8].

$$([\![ (f\ e_1 \cdots e_n) ]\!], \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, t{+}1)$$

$$\text{where} \begin{cases} proc = \mathcal{A}(f, \beta, ve) \\ d_i = \mathcal{A}(e_i, \beta, ve) \end{cases}$$

$$(([\![ (\lambda\ (v_1 \cdots v_n) ]\!]\ call), \beta), \boldsymbol{d}, ve, t) \Rightarrow (call, \beta', ve', t)$$

$$\text{where} \begin{cases} \beta' = \beta[v_i \mapsto t] \\ ve' = ve[(v_i, t) \mapsto d_i] \end{cases}$$

**Figure 3.** A small-step semantics for CPS, with a factored environment representation: (left) the eval-state transition, and (right) the apply-state transition.

$$\mathcal{A} : \mathit{EXP} \times \mathit{BEnv} \times \mathit{VEnv} \to D$$
$$\mathcal{A}(v, \beta, ve) = ve(v, \beta(v))$$
$$\mathcal{A}(lam, \beta, ve) = (lam, \beta)$$

**Figure 4.** The $\mathcal{A}$ function evaluates argument expressions given a factored *BEnv*/*VEnv* environment.

$$\begin{array}{rcll}
\varsigma \in \mathit{State} &=& \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} &=& \mathit{CALL} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Time} \\
\mathit{Apply} &=& \mathit{Proc} \times D^* \times \mathit{VEnv} \times \mathit{Time} \\
\beta \in \mathit{BEnv} &=& \mathit{VAR} \rightharpoonup \mathit{Time} \\
b \in \mathit{Bind} &=& \mathit{VAR} \times \mathit{Time} \\
ve \in \mathit{VEnv} &=& \mathit{Bind} \rightharpoonup D \\
proc \in \mathit{Proc} &=& \mathit{Clo} + \{halt\} \\
clo \in \mathit{Clo} &=& \mathit{LAM} \times \mathit{BEnv} \\
d \in D &=& \mathit{Proc} \\
t \in \mathit{Time} &=& \text{an infinite set of times (contours)}
\end{array}$$

**Figure 5.** Semantic domains

Defining the meaning of our language as a small-step operational semantics exposes the intermediate states of the computation, including the environment structure we made explicit with our factored *VEnv*/*BEnv* representation. This sets us up to use abstract interpretation to reason statically about these states. All we need to do now is add garbage collection.

### 5.4 Adding GC transitions to the semantics

Before we can define garbage collection, we need to define more basic notions, such as the touchability of a value by a binding, the adjacency of bindings and the bindings reachable from some entity. For our framework, *garbage collection* means finding the set of reachable bindings and restricting the domain of the global variable environment *ve* to solely these bindings.

First, we define the bindings $\mathcal{T}(d)$ touched by some value $d$:

$$\mathcal{T}(lam, \beta) = \{(v, \beta(v)) : v \in free(lam)\}$$
$$\mathcal{T}(halt) = \{\}.$$

A closure $(lam, \beta)$ could potentially touch a binding $(v, t)$ if $v$ is free in *lam*, and if $\beta(v) = t$. We can extend the function $\mathcal{T}$ to objects such as states:

$$\mathcal{T}(call, \beta, ve, t) = \{(v, \beta(v)) : v \in free(call)\}$$
$$\mathcal{T}(proc, \boldsymbol{d}, ve, t) = \mathcal{T}(proc) \cup \mathcal{T}(d_1) \cup \cdots \cup \mathcal{T}(d_n).$$

In essence, a binding is touched by an entity if the binding is *immediately* reachable by that entity.

With this notion of touch, we can define the *adjacency* relation over bindings:

$$b \rightsquigarrow_{ve} b' \iff b' \in \mathcal{T}(ve(b)).$$

The set $\mathcal{R}(\varsigma)$ of bindings *reachable* from state $\varsigma$ is simply all the bindings we can reach from $\varsigma$ with chains of $\rightsquigarrow_{ve}$ links:

$$\mathcal{R}(\varsigma) = \{b' : b \in \mathcal{T}(\varsigma) \text{ and } b \rightsquigarrow^*_{ve_\varsigma} b'\}.$$

Now we can define the GC function, $\Gamma : \mathit{State} \to \mathit{State}$:

$$\Gamma(\varsigma) = \begin{cases} (proc, \boldsymbol{d}, ve|\mathcal{R}(\varsigma), t) & \varsigma = (proc, \boldsymbol{d}, ve, t) \\ (call, \beta, ve|\mathcal{R}(\varsigma), t) & \varsigma = (call, \beta, ve, t). \end{cases}$$

The function $\Gamma$ removes unreachable bindings from the domain of the global variable environment *ve*.

Using this, we can define the alternate, GC transition rule, $\Rightarrow_\Gamma$:

$$\frac{\Gamma(\varsigma) \Rightarrow \varsigma'}{\varsigma \Rightarrow_\Gamma \varsigma'}.$$

That is, $\Rightarrow_\Gamma$ first performs a collection, and then steps the execution forward. When a GC is "deemed appropriate," the transition can be made with this rule instead of with the regular transition, $\Rightarrow$.

Because we are agnostic as to when a GC is done, we define a new, nondeterministic transition relation $\Rrightarrow$ for this semantics as the union of $\Rightarrow$ and $\Rightarrow_\Gamma$.

Before proceeding, we need to tidy up loose ends such as the injection of a program into an initial state, and the concept of a final state. The injection function $\mathcal{I} : \mathit{LAM} \to \mathit{State}$ injects a $\lambda$ term accepting the halt continuation into an initial state:

$$\mathcal{I}(lam) = ((lam, \bot_{BEnv}), \langle halt \rangle, \bot_{VEnv}, t_0).$$

A *final state* is one applying the *halt* continuation to a singleton argument vector containing the final result: $(halt, \langle d_{\text{result}} \rangle, ve, t)$.

Execution may also end by arriving at a stuck state, of which we distinguish three kinds:

**Mismatch** A mismatch stuck state is an apply state in which the number of arguments supplied does not match the number of arguments required. This is a result of programmer error.

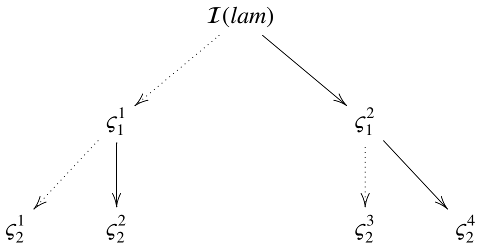
**Undefined variable** An undefined-variable stuck state is an eval state in which a variable argument is not in the domain of the lexical contour environment $\beta$. This can happen only if the top-level program has a free variable, also a programmer error.

**Corrupted environment** A corrupted-environment stuck state is an eval state in which a required binding is not in the domain of the global variable environment *ve*. As part of showing correctness, we demonstrate that this can never happen.

We call a state *terminal* if it is final or stuck.

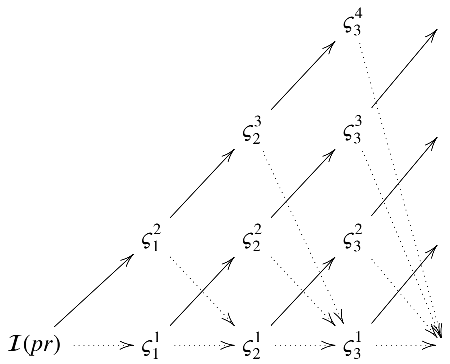
## 6. Correctness of garbage-collecting semantics

The transformation from a non-GC semantics to a GC semantics is nontrivial, and we are obliged to show correctness. As we mentioned, we leave it open whether $\Rightarrow$ or $\Rightarrow_\Gamma$ is used for any given transition, so the semantics are nondeterministic. Given that we have two choices from any given state, we can visualize the execution space as a binary tree:



Here, dotted arrows represent GC-and-step-forward transitions ($\Rightarrow_\Gamma$) and solid arrows represent ordinary transitions ($\Rightarrow$). What we mean by the term *correctness* in this case is that the result obtained is independent of the path taken through this tree.

Realizing that this "tree" is in reality the following DAG hints at how we might go about doing this:



In words, the GC transitions from any given depth of the tree always merge into the same state. This happens because the GC function $\Gamma$ takes all states at a given depth to the same state. That is, the GC function partitions states into equivalence classes. Moreover, it's trivial to show that all members of a given equivalence class would yield the same result if they were final. Our focus for the remainder of this section becomes formalizing and demonstrating what we've just described.

Thus our roadmap for correctness is as follows:

1. Show that soundness (defined shortly) is preserved under $\Rightarrow$ transitions, and then under GC.
2. Show that all states at any given depth of the execution tree are equivalent.
3. Show that if one state at some depth transitions, then all states at that depth transition.
4. Using the above, show that the result obtained on any path through the execution tree is equivalent to the result obtained on any other path.

From this, it also follows that if one branch doesn't terminate, no branch terminates.

**A word on proof technique**. Many of the proofs in this section require us to show that if some path is valid over some adjacency relation $\rightsquigarrow_{ve}$ then the path is also valid over some other adjacency relation $\rightsquigarrow_{ve'}$. Often, we show this using contradiction: we choose the first binding which invalidates the path for $\rightsquigarrow_{ve'}$ and then show that it cannot actually invalidate the path. Later on, in showing the correctness of the analysis, we'll make use of a similar path-based technique.

The *soundness* of a state captures the essence of well-formedness for a given state:

**Definition 6.1** (Soundness)**.** *A state $\varsigma$ satisfies $Sound(\varsigma)$ iff*

$$\mathcal{R}(\varsigma) \subseteq dom(ve_\varsigma).$$

In words, a state is sound if it has a valid entry in its global environment for every binding it could conceivably reach.

We say that two states are *equivalent* if they are equal under GC:

**Definition 6.2.** *States $\varsigma_1$ and $\varsigma_2$ are equivalent iff $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$.*

Our first theorem rules out a corrupted-environment error for sound states.

**Theorem 6.3.** *Sound states are not corrupt.*
*That is, if a state $\varsigma$ is sound, then either*

- $\varsigma \Rightarrow \varsigma'$,
- *$\varsigma$ is a final state, or*
- *$\varsigma$ is stuck, but not corrupt.*

*Proof.* By the definitions. ☐

Soundness is important because if two states are sound and equivalent, then either both transition, or neither transitions. This ultimately lets us show that if any state at some depth in the execution tree transitions, then all states at that depth transition.

The following lemma relates touching and reaching:

**Lemma 6.4** (Containment)**.** *If $\varsigma \Rightarrow \varsigma'$, and $\mathcal{T}(\varsigma') \subseteq \mathcal{R}(\varsigma)$, then $\mathcal{R}(\varsigma') \subseteq \mathcal{R}(\varsigma)$.*

*Proof.* By contradiction and the definitions of $\mathcal{T}$ and $\mathcal{R}$. ☐

The next lemma states that once a binding is dead, it cannot come back to life.

**Lemma 6.5** (No Zombies)**.** *If $(v, t) \notin \mathcal{R}(\varsigma)$, and $\varsigma \Rightarrow \varsigma'$, then*

- *$\varsigma$ is an eval state and $(v, t) \notin \mathcal{R}(\varsigma')$, or*
- *$\varsigma$ is an apply state and if $t < t_\varsigma$, then $(v, t) \notin \mathcal{R}(\varsigma')$.*

*Proof.* Assume $\varsigma \Rightarrow \varsigma'$. We proceed by cases on $\varsigma$:

Case $\varsigma = ((lam, \beta), \boldsymbol{d}, ve, t)$: Thus, $\varsigma' = (call, \beta', ve', t)$. Choose any $(v_n, t_n) \notin \mathcal{R}(\varsigma)$ such that $t_n < t$. We proceed by contradiction, so suppose that $(v_n, t_n) \in \mathcal{R}(\varsigma')$. Now, let $\langle (v_0, t_0), \ldots, (v_n, t_n)\rangle$ be a path through $\rightsquigarrow_{ve_{\varsigma'}}$ from $\mathcal{T}(\varsigma')$.

Suppose the root of the path, $(v_0, t_0)$, has $t_0 = t$. Then we know that this root binding was just created, and hence that $ve(v_0, t_0) = d_i$ for some $i$. From the definition of $\mathcal{T}(\varsigma)$, $\mathcal{T}(d_i) \subseteq \mathcal{T}(\varsigma)$, and so $\langle (v_1, t_1), \ldots, (v_n, t_n)\rangle$ is a legitimate path through $\rightsquigarrow_{ve}$ from $\mathcal{T}(\varsigma)$— the path is legitimate because no element past $(v_0, t_0)$ in the path can have a time equal to $t$, and because $ve$ and $ve'$ differ only by bindings for time $t$.

Now, suppose that $t_0 < t$. From this, we can infer that $v_0$ is a free variable in both $lam$ and in $call$. Hence, $(v_0, t_0) \in \mathcal{T}(\varsigma)$. Consequently, the path $\langle (v_0, t_0), \ldots, (v_n, t_n)\rangle$ is also valid through $\rightsquigarrow_{ve}$.

Hence, under either supposition, $(v_n, t_n) \in \mathcal{R}(\varsigma)$, which is a contradiction.

*Case* $\varsigma = ([\![ (f\ e_1 \cdots e_n) ]\!], \beta, ve, t)$:

Thus, $\varsigma' = (proc, \boldsymbol{d}, ve', t + 1)$. In this case, $ve = ve'$. Consequently, we can reduce this case to showing $\mathcal{T}(\varsigma') \subseteq \mathcal{R}(\varsigma)$. By definition,

$$\mathcal{T}(\mathcal{A}(e, \beta, ve)) = \begin{cases} \{(v, \beta(v)) : v \in free(e)\} & e \in LAM \\ \mathcal{T}(ve(e, \beta(e)) & e \in VAR. \end{cases}$$

Now, we show that $\mathcal{T}(proc) \subseteq \mathcal{R}(\varsigma)$. Either $f \in LAM$ or $f \in VAR$. If $f \in LAM$, by $free(f) \subseteq free(call)$, $\mathcal{T}(proc) \subseteq$

$\mathcal{R}(\varsigma)$. If instead $f \in VAR$, then $(f, \beta(f)) \in \mathcal{T}(\varsigma)$, and hence, $\mathcal{T}(ve(f, \beta(f))) \subseteq \mathcal{R}(\varsigma')$. Either way, $\mathcal{T}(\mathcal{A}(f, \beta, ve)) \subseteq \mathcal{R}(\varsigma)$. An identical argument shows for each $d_i$ that $\mathcal{T}(d_i) \subseteq \mathcal{R}(\varsigma)$.

This case now follows from Lemma 6.4. □

The No Zombies Lemma captures a fundamental constraint on program behavior, and therefore acts as the workhorse for many of our proofs.

Now we can show that soundness is preserved across non-GC transitions:

**Theorem 6.6.** *If a state $\varsigma$ is sound and $\varsigma \Rightarrow \varsigma'$, then $\varsigma'$ is sound.*

*Proof.* Assume $\varsigma$ is sound and $\varsigma \Rightarrow \varsigma'$. We proceed by cases on the structure of $\varsigma$:

*Case* $\varsigma = ([\![ (f\ e_1 \cdots e_n) ]\!], \beta, ve, t)$: Soundness is established by

$$\mathcal{R}(\varsigma') \subseteq \mathcal{R}(\varsigma) \subseteq dom(ve) = dom(ve').$$

Taking this chain of relations from left to right: $\mathcal{R}(\varsigma') \subseteq \mathcal{R}(\varsigma)$ follows from the No Zombies Lemma; $\mathcal{R}(\varsigma) \subseteq dom(ve)$ because $\varsigma$ is sound; and $dom(ve) = dom(ve')$ simply because eval-state transitions do not alter the $ve$ component of the state, so $ve = ve'$.

*Case* $\varsigma = (([\![ (\lambda\ (v_1 \cdots v_n)\ call) ]\!], \beta), \boldsymbol{d}, ve, t)$:

In this case, $\varsigma' = (call, \beta', ve', t)$. Again, we must show that $\mathcal{R}(\varsigma') \subseteq dom(ve')$. Choose $(v', t') \in \mathcal{R}(\varsigma')$.

*Subcase* $t' < t$: By the No Zombies Lemma, we have that $(v', t') \in \mathcal{R}(\varsigma)$. By this and the soundness of $\varsigma$, we know that $(v', t') \in dom(ve)$. By this and the apply state schema, we know that $(v', t') \in dom(ve')$.

*Subcase* $t' = t$: In other words, this binding is fresh for $\varsigma'$. In this case, $(v', t')$ is clearly in $dom(ve')$. □

Most importantly, we also show that performing a GC does not degrade soundness:

**Theorem 6.7.** *If $\varsigma$ is sound, then $\Gamma(\varsigma)$ is sound.*

*Proof.* Assume $Sound(\varsigma)$. By the definition of $\mathcal{R}$, all paths starting from $\mathcal{T}(\varsigma)$ through the relation $\leadsto_{ve}$ are over the elements in $\mathcal{R}(\varsigma)$. Hence, any of these paths is also valid through the relation $\leadsto_{ve|\mathcal{R}(\varsigma)}$. As a result, $\mathcal{R}(\Gamma(\varsigma)) = \mathcal{R}(\varsigma) \subseteq dom(ve_\varsigma)$. Consequently, $\mathcal{R}(\Gamma(\varsigma)) = dom(ve_\varsigma | \mathcal{R}(\varsigma))$. □

Because the initial state is sound, we can now claim that every state on every possible branch through the computation is also sound.

We can also show that performing a GC on a GC'd state doesn't change anything:

**Lemma 6.8.** $\Gamma(\Gamma(\varsigma)) = \Gamma(\varsigma)$.

Now, we can show that all states at a given depth in the tree have the same image under the GC function, $\Gamma$. The key inductive step is the following theorem:

**Theorem 6.9.** *If $\varsigma_1$ and $\varsigma_2$ are sound, and $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$, then either both states are terminal, or $\varsigma_1 \Rightarrow \varsigma_1'$ and $\varsigma_2 \Rightarrow \varsigma_2'$ and $\Gamma(\varsigma_1') = \Gamma(\varsigma_2')$ .*

*Proof.* Assume $Sound(\varsigma_1)$, $Sound(\varsigma_2)$ and $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$. By the definition of $\Gamma$ and soundness, if one state is terminal, then so is the other. [To avoid triple subscripts, let $\varsigma_i = (\ldots, ve_i, \ldots)$.]

We now handle the case where they are non-terminal. We must show that their subsequent states, $\varsigma_1'$ and $\varsigma_2'$ are equal under the GC function $\Gamma$, which reduces to showing $ve_1' | \mathcal{R}(\varsigma_1') = ve_2' | \mathcal{R}(\varsigma_2')$, where $ve_1'$ and $ve_2'$ are the variable environments for the subsequent states. Before proceeding, we note that by $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$, we can infer $\mathcal{R}(\varsigma_1) = \mathcal{R}(\varsigma_2)$.

We proceed by contradiction, so suppose we can find a binding $(v', t')$ such that $(ve_1' | \mathcal{R}(\varsigma_1'))(v', t') = d_1$ but that we had $(ve_2' | \mathcal{R}(\varsigma_2'))(v', t') = d_2 \neq d_1$. From this supposition, we know that $(v', t') \in dom(ve_1')$ and $(v', t') \in \mathcal{R}(\varsigma_1')$. We proceed by cases on what could cause this inequality.

*Case* $ve_2'(v', t') \neq d_1$. By the apply state schema, we can rule out the case where $t'$ is fresh for $\varsigma_2'$. Thus, $ve_1(v', t') \neq ve_2(v', t')$. From this, we can infer that $(v', t')$ is in neither $\mathcal{R}(\varsigma_1)$ nor $\mathcal{R}(\varsigma_2)$. By the No Zombies Lemma, we would then have that $(v', t') \notin \mathcal{R}(\varsigma_1')$, except that this is a contradiction.

*Case* $(v', t') \notin \mathcal{R}(\varsigma_2')$. Let $\langle (v_0, t_0), \ldots, (v_n, t_n) \rangle$ be a path to $(v', t')$ from $\mathcal{T}(\varsigma_1')$ through $\leadsto_{ve_1'}$; that $(v', t') \in \mathcal{R}(\varsigma_1')$ guarantees this path exists. Let $k$ be the lowest index in the path such that $\langle (v_0, t_0), \ldots, (v_k, t_k) \rangle$ is not a valid path for $\mathcal{R}(\varsigma_2')$. The equality of $\varsigma_1'$ and $\varsigma_2'$ over $\mathcal{T}$ means that $k \neq 0$. By the apply state schema, $t_k$ cannot be fresh for $\varsigma_2'$. Thus, $ve_1(v_{k-1}, t_{k-1}) \neq ve_2(v_{k-1}, t_{k-1})$, which means that $(v_{k-1}, t_{k-1}) \notin \mathcal{R}(\varsigma_1)$. However, this lets us conclude, by the No Zombies Lemma, that $(v_{k-1}, t_{k-1}) \notin \mathcal{R}(\varsigma_1')$ either, which is clearly not the case. Thus, the path must be valid for $\mathcal{R}(\varsigma_2')$ as well, but this would imply that $(v', t') \in \mathcal{R}(\varsigma_2')$, which is a contradiction. □

From all of this, we can conclude that the $n$th children from any two branches of execution are equivalent:

**Theorem 6.10.** *If $\mathcal{I}(lam) \Rightarrow^n \varsigma_1$ and $\mathcal{I}(lam) \Rightarrow^n \varsigma_2$, then $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$.*

## 7. Abstract semantics: ΓCFA

Thus far, we have developed a concrete, garbage-collecting semantics for CPS and proved its correctness. In this section, we shift gears and build an abstract semantics—our analysis—which simulates the concrete. While it is possible to separate abstract GC and abstract counting, we add them both at the same time to avoid duplicating work. It is simple enough to tune parameters within this framework so that either feature is effectively "turned off" through degeneracy. We term this combined framework ΓCFA.

The major components of this abstraction will be:

- An abstract domain for each concrete domain from Figure 5. The abstract counterpart for a given domain will be written with a hat on it, *e.g.*, $\widehat{D}$ is the abstraction of $D$.
- A family of abstraction functions—all written with the absolute-value-style notation $|\cdot|$—which maps elements from concrete domains (such as *State*, $D$, and *Clo*) into their corresponding abstract domains (such as $\widehat{State}$, $\widehat{D}$ and $\widehat{Clo}$).
- An abstract collection function, $\widehat{\Gamma} : \widehat{State} \to \widehat{State}$.
- Abstract transition relations, $\approx\!>$ and $\approx\!>_{\widehat{\Gamma}}$ which simulate the concrete transitions.

To abstract the semantics, we begin by making the set of times finite, giving us the set $\widehat{Time}$. We also need an abstract "successor" function, $\widehat{succ} : \widehat{Time} \to \widehat{Time}$ which is constrained so that:

$$|t| \sqsubseteq \widehat{t} \implies |t+1| \sqsubseteq \widehat{succ}(\widehat{t}).$$

By leaving the exact structure and size of $\widehat{Time}$ unspecified, we allow the precision of the analysis, *e.g.*, 0CFA, 1CFA, CPA, to be controlled externally.[5]

[5] We should actually define the concrete successor function so that it takes in the current state when choosing a next time. This then allows the abstract successor function to take in the abstract state when making its decision. Since this adds a few tedious distractions to the proof of correctness, we opt to keep it simple here. We have shown detailed abstract-contour selection machinery in previous work [8].

$$
\begin{aligned}
\hat{\varsigma} \in \widehat{State} &= \widehat{Eval} + \widehat{Apply} \\
\widehat{Eval} &= CALL \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Count} \times \widehat{Time} \\
\widehat{Apply} &= \widehat{Proc} \times \widehat{D}^* \times \widehat{VEnv} \times \widehat{Count} \times \widehat{Time} \\
\hat{\beta} \in \widehat{BEnv} &= VAR \to \widehat{Time} \\
\hat{b} \in \widehat{Bind} &= VAR \times \widehat{Time} \\
\widehat{ve} \in \widehat{VEnv} &= \widehat{Bind} \to \widehat{D} \\
\widehat{proc} \in \widehat{Proc} &= \widehat{Clo} + \{halt\} \\
\widehat{clo} \in \widehat{Clo} &= LAM \times \widehat{BEnv} \\
\hat{d} \in \widehat{D} &= \mathcal{P}(\widehat{Proc}) \\
\hat{\mu} \in \widehat{Count} &= \widehat{Bind} \to \widehat{\mathbb{N}} \\
\hat{n} \in \widehat{\mathbb{N}} &= \{0, 1, \infty\} \\
\hat{t} \in \widehat{Time} &= \text{a finite set of abstract times}
\end{aligned}
$$

**Figure 6.** Abstract Domains

The next significant change is the addition of an abstract binding counter, $\hat{\mu} \in \widehat{Count}$, to each state. Given an abstract binding $(v, \hat{t})$, the value $\hat{\mu}(v, \hat{t})$ approximates how many concrete bindings $(v, \hat{t})$ currently represents. For our work, we use three possible approximations—0, 1 and $\infty$; that is, an abstract binding may represent no concrete bindings, a single concrete binding or an arbitrary number of concrete bindings.[6] An abstraction of the naturals, $\widehat{\mathbb{N}} = \{0, 1, \infty\}$, represents these possibilities. We define the lattice operations for $\widehat{\mathbb{N}}$ as: $\bot_{\widehat{\mathbb{N}}} = 0$, $\top_{\widehat{\mathbb{N}}} = \infty$, $\sqcup = \max$, $\sqcap = \min$ and $\sqsubseteq = \leq$.

Percolating these changes through the rest of the domains leads to the abstract domains in Figure 6. The compression of the infinite set *Time* into the finite set $\widehat{Time}$ causes each abstract binding to represent multiple concrete bindings. As a result, the entry in an abstract global variable environment $\widehat{ve}$ for a given abstract binding may need to represent multiple concrete values. This causes the domain of abstract denotable values $\widehat{D}$ to become a power domain.

Combining the above leads to a natural definition for the abstract transition relation, $\approx\!>$:

$$([\![ (f\ e_1 \cdots e_n) ]\!], \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \approx\!> (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \widehat{succ}(\hat{t}))$$

$$\text{where } \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve}) \\ \hat{d}_i = \widehat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve}) \end{cases}$$

$$(([\![ (\lambda\ (v_1 \cdots v_n)\ call) ]\!], \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) \approx\!> (call, \hat{\beta}', \widehat{ve}', \hat{\mu}', \hat{t})$$

$$\text{where } \begin{cases} \hat{\beta}' = \hat{\beta}[v_i \mapsto \hat{t}] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \hat{t}) \mapsto \hat{d}_i] \\ \hat{\mu}' = \hat{\mu} \oplus [(v_i, \hat{t}) \mapsto 1]. \end{cases}$$

Here, the operator $\oplus$ is the natural abstraction of addition over $\widehat{\mathbb{N}}$. The argument evaluator $\mathcal{A}$ abstracts naturally to $\widehat{\mathcal{A}}$:

$$\widehat{\mathcal{A}}(v, \hat{\beta}, \widehat{ve}) = \widehat{ve}(v, \hat{\beta}(v))$$
$$\widehat{\mathcal{A}}(lam, \hat{\beta}, \widehat{ve}) = \{(lam, \hat{\beta})\}.$$

We add garbage collection to the abstract semantics with the same steps we used in the concrete case. First, we define what it means for an abstract value to touch an abstract binding, with the function $\widehat{\mathcal{T}}$:

$$\widehat{\mathcal{T}}(lam, \hat{\beta}) = \{(v, \hat{\beta}(v)) : v \in free(lam)\}$$
$$\widehat{\mathcal{T}}(halt) = \{\}$$
$$\widehat{\mathcal{T}}\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} = \widehat{\mathcal{T}}(\widehat{proc}_1) \cup \cdots \cup \widehat{\mathcal{T}}(\widehat{proc}_n).$$

As before, we can extend the notion of touching to abstract states:

$$\widehat{\mathcal{T}}(call, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) = \{(v, \hat{\beta}(v)) : v \in free(call)\}$$
$$\widehat{\mathcal{T}}(\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) = \widehat{\mathcal{T}}(\widehat{proc}) \cup \widehat{\mathcal{T}}(\hat{d}_1) \cup \cdots \cup \widehat{\mathcal{T}}(\hat{d}_n).$$

The abstraction of the binding-to-binding adjacency relation looks nearly the same:

$$\hat{b} \rightsquigarrow_{\widehat{ve}} \hat{b}' \iff \hat{b}' \in \widehat{\mathcal{T}}(\widehat{ve}(\hat{b})).$$

The abstract reachable-bindings function, $\widehat{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Bind})$, looks nearly identical to its concrete counterpart $\mathcal{R}$, as well:

$$\widehat{\mathcal{R}}(\hat{\varsigma}) = \{\hat{b}' : \hat{b} \in \widehat{\mathcal{T}}(\hat{\varsigma}) \text{ and } \hat{b} \rightsquigarrow^*_{\widehat{ve}_{\hat{\varsigma}}} \hat{b}'\}.$$

Now we can define the abstract GC function, $\widehat{\Gamma} : \widehat{State} \to \widehat{State}$:

$$\widehat{\Gamma}(\hat{\varsigma}) = \begin{cases} (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}|\widehat{\mathcal{R}}(\hat{\varsigma}), \hat{\mu}|\widehat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) \\ (call, \hat{\beta}, \widehat{ve}|\widehat{\mathcal{R}}(\hat{\varsigma}), \hat{\mu}|\widehat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (call, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}). \end{cases}$$

The chief difference between $\widehat{\Gamma}$ and the concrete version $\Gamma$ is that we also restrict the domain of the binding counter $\hat{\mu}$, effectively resetting any unreachable bindings back to a count of 0.

With this, the abstract GC transition becomes

$$\frac{\widehat{\Gamma}(\hat{\varsigma}) \approx\!> \hat{\varsigma}'}{\hat{\varsigma} \approx\!>_{\widehat{\Gamma}} \hat{\varsigma}'}.$$

To run the analysis, we first inject a program *lam* into an abstract state using $\widehat{\mathcal{I}} : LAM \to \widehat{State}$:

$$\widehat{\mathcal{I}}(lam) = |\mathcal{I}(lam)|.$$

We'll define the abstraction operator $|\cdot|$ in the next section.

We need no notion of a final state for the abstract semantics, as we are not particularly interested in the actual value produced by the computation. To run the analysis then consists of collecting (in the sense of a *collecting semantics* rather than GC) all of the states

**Where precision is lost**. It's worth taking a moment to point out where precision is lost. If we put the definitions of $\Rightarrow$ and $\approx\!>$ side-by-side and looked at the definitions of $ve'$ and $\widehat{ve}'$, we'd notice a join ($\sqcup$) operation present in the abstract that does not exist in the concrete. In the concrete, every time we extend $ve$, the bindings added are guaranteed to be fresh, because we just bumped up the current time. In the abstract, we can't "extend" $\widehat{ve}$ to get $\widehat{ve}'$, because the bindings may *not* be fresh. As we try to insert an entry for $(v, \hat{t})$ into $\widehat{ve}$, something may already be there. If we overwrote the value lying at $(v, \hat{t})$, our analysis would no longer be sound, so instead, we must merge the old and new values.

Returning to the example in the introduction, after analyzing the fragment the first time, $\widehat{ve}([\![\texttt{x}]\!], \hat{t})$ would be $\{\texttt{0}\}$, and after the second time, $\widehat{ve}([\![\texttt{x}]\!], \hat{t})$ would be $\{\texttt{0}, \texttt{5}\}$.

Returning to the model developed in Section 4, the concrete addresses are members of *Bind*, the abstract addresses are members of $\widehat{Bind}$, the concrete objects are $D$ and the abstract objects are members of $\widehat{D}$. The functions $ve$ and $\widehat{ve}$ themselves model the connections in the "heaps."

[6] We are abusing our notation a bit here: the element $\infty$ doesn't mean an infinite number of bindings; it means an *unknown* or *arbitrary* number of bindings.

**How precision is saved**. Now that we've integrated abstract garbage collection, we can discuss its role in improving precision. Suppose the abstract interpretation is on the verge of adding a new binding for $(v, \hat{t})$ in $\widehat{ve}$. Either $\widehat{ve}(v, \hat{t}) = \bot$, in which case this binding has been collected since its last allocation (or never allocated at all), or some value is already sitting at $(v, \hat{t})$ in $\widehat{ve}$. Note that if nothing is at $(v, \hat{t})$, then:

$$\widehat{ve} \sqcup [(v, \hat{t}) \mapsto \hat{d}] \equiv \widehat{ve}[(v, \hat{t}) \mapsto \hat{d}].$$

That is, we are not merging abstract bindings.

Returning to the example in the introduction, after analyzing the fragment the first time, $\widehat{ve}([\![\mathtt{x}]\!], \hat{t})$ would be $\{\mathtt{0}\}$. It's likely, however, that before returning to this point, garbage collection would reset $\widehat{ve}([\![\mathtt{x}]\!], \hat{t})$ to $\bot$. Thus, during the second pass, $\widehat{ve}([\![\mathtt{x}]\!], \hat{t})$ would be $\{\mathtt{5}\}$.

Peeking back at the `id` example in Section 3, we can motivate how ΓCFA (with a 0CFA-level contour set) yields the more precise answer: that only $lam'$ is in the flow set for the return value. After the first call to `id`, `x` is $\{lam\}$. Directly after this call, however, that binding to `x` is unreachable, and `x` can be reset to $\bot$. Thus, when interpretation reaches the second call to `id`, there is no merging of $\{lam\}$ and $\{lam'\}$.

reachable from the initial state on any path. In practice, we can stop collecting on any given path if (1) the current state is stuck, or (2) we have already visited a state that approximates (via $\sqsubseteq$) the current state. We refer to the set of abstract states reached by a program $pr$ as $\hat{\mathcal{V}}(pr)$. Eventual termination of the analysis is guaranteed because the space through which it roams, $\widehat{State}$, is finite.

### 7.1 Choices impacting precision

We left the set of abstract times constrained but unspecified, so that we can vary precision externally. If we use a singleton set for $\widehat{Time}$, we end up with 0CFA. We can instead let $\widehat{Time}$ be the set of call sites, and then have the successor function $\widehat{succ}$ choose the current call site as the next "time." This gives us 1CFA. Generalizing further, it's not hard to set up $k$-CFA for any $k$. It's also straightforward to set up Wright and Jagannathan's polymorphic splitting [12] or Agesen's CPA [1]. By varying $\widehat{Time}$ and $\widehat{succ}$, we can instantiate almost any conceivable variation on existing analyses and have it "GCified."

There are a number of policy choices available for deciding when to perform a GC transition, each with a different impact on precision. The simplest policy, "never GC," just gives us an ordinary control-flow analysis. The other extreme, which is to GC on every step, could be considered too aggressive because the act of performing a GC transition throws away some information that could be useful to the optimizer when trying to perform Super-$\beta$ copy propagation [8].[7] The time cost of a GC does not appear to be significant, and implementation results reveal that any GC costs are handily outweighed by the savings we get from searching a smaller state space. The policy we use when deciding whether or not to make a GC is: "perform a GC transition if and only if zombie creation would be imminent otherwise." Zombie creation is imminent if we are about to add a binding for $(v, \hat{t})$, but $\hat{\mu}(v, \hat{t}) \geq 1$, or alternatively, $\widehat{ve}(v, \hat{t}) \neq \bot$.

If desired, we can effectively turn abstract counting off by setting $\hat{\mathbb{N}} = \{\infty\}$.

## 8. Correctness of the abstract semantics

In this section, we demonstrate the correctness of the analysis. We have excised portions of the proofs which do not differ from an ordinary proof of correctness for a control-flow analysis. These portions are the same as the ones we've presented in earlier work [8, 7]. To show the correctness of the abstract semantics, we must show that they simulate the concrete semantics. The first step in this process is defining the simulation relation, and for that, we need to define our abstraction map.

The concrete and the abstract are formally connected by the abstraction operation, $|\cdot|$:

$$\begin{aligned}
|(call, \beta, ve, t)|_{Eval} &= (call, |\beta|, |ve|, \mathcal{M}(ve), |t|)\\
|(proc, \boldsymbol{d}, ve, t)|_{Apply} &= (|proc|, |\boldsymbol{d}|, |ve|, \mathcal{M}(ve), |t|)\\
|\langle d_1, \ldots, d_n\rangle|_{D^*} &= \langle |d_1|_D, \ldots, |d_n|_D\rangle\\
|d|_D &= \{|d|_{Proc}\}\\
|halt|_{Proc} &= halt\\
|clo|_{Proc} &= |clo|_{Clo}\\
|(lam, \beta)|_{Clo} &= (lam, |\beta|)\\
|(v, t)|_{Bind} &= (v, |t|)\\
|\beta|_{BEnv} &= \lambda v.|\beta(v)|\\
|ve|_{VEnv} &= \lambda(v, \hat{t}).\bigsqcup_{|t|=\hat{t}} |ve(v, t)|_D
\end{aligned}$$

where the abstract counter creator, $\mathcal{M} : VEnv \to \widehat{Count}$ is

$$\mathcal{M}(ve) = \lambda\hat{b}.\widehat{size}\{b' \in dom(ve) : |b'| = \hat{b}\},$$

and the abstract set-size function $\widehat{size}$ is

$$\widehat{size}(S) = \begin{cases} size(S) & size(S) \in \{0, 1\}\\ \infty & \text{otherwise.}\end{cases}$$

For a set $S$ whose elements are abstractable, $|S| = \{|s| : s \in S\}$.

Now we're ready to define the simulation relation, $\mathcal{S} \subseteq \widehat{State} \times State$.

**Definition 8.1** (Simulates)**.** *An abstract state $\hat{\varsigma}$ simulates a concrete state $\varsigma$, written $\mathcal{S}(\hat{\varsigma}, \varsigma)$, iff $|\varsigma| \sqsubseteq \hat{\varsigma}$.*

The key steps for correctness now become a matter of showing that the simulation is preserved under transition and under GC:

**Theorem 8.2.** *If $\mathcal{S}(\hat{\varsigma}, \varsigma)$ and $\varsigma \Rightarrow \varsigma'$, then there exists an abstract state $\hat{\varsigma}'$ such that $\hat{\varsigma} \approx\!> \hat{\varsigma}'$ and $\mathcal{S}(\hat{\varsigma}', \varsigma')$. Diagrammatically:*

$$\begin{tikzcd}
\hat{\varsigma} \arrow[r, "\mathcal{S}"] \arrow[d, dashed, "\approx>"'] & \varsigma \arrow[d, "\Rightarrow"] \\
\hat{\varsigma}' \arrow[r, "\mathcal{S}"'] & \varsigma'
\end{tikzcd}$$

*Proof.* With the exception of the binding-counter component, this is a straightforward proof for the correctness of a control-flow

[7] This seems counterintuitive; it arises because the GC semantics reaps away environment structure that is irrelevant to the program *as it is*—but this structure might become relevant to proving the safety of the code-transform we are contemplating. Consider the Super-$\beta$ optimization which seeks to change a reference $r$ in the syntax tree from some variable `x` to some other variable `z`, perhaps to make `x` into a useless variable.
In order to do this, the analysis must show that `x` is always equal to `z` whenever control reaches point $r$ in the program. Performing this transformation adds `z` to the free-variable set of position $r$ in the syntax tree, as well as the free-variable sets of $r$'s parents. However, while analysing the *original* program, at some point on the control path to the `x` reference at $r$, the `z` binding may become dead, and so, by the time control actually reaches $r$, its binding will have been garbage collected—which kills our ability to reason about `z`'s equality with `x`.

analysis. Correctness of the binding-counter component follows from the lemmas below. □

**Theorem 8.3.** *If $\mathcal{S}(\widehat{\varsigma}, \varsigma)$, then $\mathcal{S}(\widehat{\Gamma}(\widehat{\varsigma}), \Gamma(\varsigma))$.*

*Proof.* By Lemma 8.8. □

Proving these theorems reduces to the following lemmas.

**Lemma 8.4.** *If $\mathcal{S}(\widehat{\varsigma}, \varsigma)$, $\varsigma \Rightarrow \varsigma'$, $\widehat{\varsigma} \approx> \widehat{\varsigma}'$, and $\mathcal{M}(ve_{\varsigma}) \sqsubseteq \widehat{\mu}_{\widehat{\varsigma}}$ then $\mathcal{M}(ve_{\varsigma'}) \sqsubseteq \widehat{\mu}_{\widehat{\varsigma}'}$.*

*Proof.* Suppose $\mathcal{S}(\widehat{\varsigma}, \varsigma)$, $\varsigma \Rightarrow \varsigma'$, $\widehat{\varsigma} \approx> \widehat{\varsigma}'$, and $\mathcal{M}(ve_{\varsigma}) \sqsubseteq \widehat{\mu}_{\widehat{\varsigma}}$. The case where $\varsigma$ is an eval state is trivial, so suppose $\varsigma$ is an apply state. Let $\varsigma = (\dots, ve, t)$, $\varsigma' = (\dots, ve', t)$, and $\widehat{\varsigma} = (\dots, \widehat{ve}, \widehat{\mu}, \widehat{t})$. By the apply-state schema, $ve' = ve[(v_i, t) \mapsto d_i]$. Thus:

$$\begin{aligned}
\mathcal{M}(ve') &= \lambda\widehat{b}.\widehat{size}\{b \in dom(ve') : |b| = \widehat{b}\} \\
&= \lambda\widehat{b}.\widehat{size}\{b \in dom(ve) : |b| = \widehat{b}\} \\
&\quad \cup \ \{b \in dom([(v_i, t) \mapsto d_i]) : |b| = \widehat{b}\} \\
&= \lambda\widehat{b}.\widehat{size}\{b \in dom(ve) : |b| = \widehat{b}\} \\
&\quad \oplus \ \widehat{size}\{b \in dom([(v_i, t) \mapsto d_i]) : |b| = \widehat{b}\} \\
&\sqsubseteq \widehat{\mu} \oplus \mathcal{M}([(v_i, t) \mapsto d_i]) \\
&= \widehat{\mu} \oplus [(v_i, \widehat{t}) \mapsto 1].
\end{aligned}$$

□

**Lemma 8.5.** *If $\mathcal{S}(\widehat{\varsigma}, \varsigma)$, then $|\mathcal{T}(\varsigma)| \sqsubseteq \widehat{\mathcal{T}}(\widehat{\varsigma})$.*

*Proof.* By cases on the structure of $\varsigma$. □

**Lemma 8.6.** $|\mathcal{R}(\varsigma)| \sqsubseteq \widehat{\mathcal{R}}(|\varsigma|)$.

*Proof.* For this case, the operator $\sqsubseteq$ effectively becomes $\subseteq$. Choose an abstract binding $\widehat{b} \in |\mathcal{R}(\varsigma)|$. Let $b$ be such that $|b| \sqsubseteq \widehat{b}$ and $b \in \mathcal{R}(\varsigma)$. Let $\langle b_0, \dots, b\rangle$ be a path that justifies $b \in \mathcal{R}(\varsigma)$. We can show by Lemma 8.5 and contradiction that the path $\langle |b_0|, \dots, |b|\rangle$ must also justify $\widehat{b} \in \widehat{\mathcal{R}}(|\varsigma|)$. □

**Lemma 8.7.** *If $\widehat{\varsigma}_1 \sqsubseteq \widehat{\varsigma}_2$, then $\widehat{\mathcal{R}}(\widehat{\varsigma}_1) \sqsubseteq \widehat{\mathcal{R}}(\widehat{\varsigma}_2)$.*

*Proof.* By reasoning similar to Lemma 8.6. □

**Lemma 8.8.** *If $\mathcal{S}(\widehat{\varsigma}, \varsigma)$, then $\mathcal{M}(ve_{\varsigma}|\mathcal{R}(\varsigma)) \sqsubseteq \widehat{\mu}_{\widehat{\varsigma}}|\mathcal{R}(\widehat{\varsigma})$.*

*Proof.* Assume $\mathcal{S}(\widehat{\varsigma}, \varsigma)$. Then,

$$\mathcal{M}(ve_{\varsigma}|\mathcal{R}(\varsigma)) = \mathcal{M}(ve) \big| |\mathcal{R}(\varsigma)| \sqsubseteq \widehat{\mu}_{\widehat{\varsigma}}|\widehat{\mathcal{R}}(\widehat{\varsigma}).$$

□

**Lemma 8.9.** $\big| ve|\mathcal{R}(\varsigma) \big| \sqsubseteq |ve| \big| |\mathcal{R}(\varsigma)|$.

*Proof.* Choose any abstract binding $\widehat{b}$.

$$\begin{aligned}
|ve \big| \mathcal{R}(\varsigma)|(\widehat{b}) &= \bigsqcup_{|b|=\widehat{b}} |(ve|\mathcal{R}(\varsigma))(b)| \\
&= \bigsqcup_{|b|=\widehat{b}} |\textbf{if } b \in \mathcal{R}(\varsigma) \textbf{ then } ve(b) \textbf{ else } \bot| \\
&\sqsubseteq \bigsqcup_{|b|=\widehat{b}} |\textbf{if } \widehat{b} \in |\mathcal{R}(\varsigma)| \textbf{ then } ve(b) \textbf{ else } \bot| \\
&= \textbf{if } \widehat{b} \in |\mathcal{R}(\varsigma)| \textbf{ then } \bigsqcup_{|b|=\widehat{b}} |ve(b)| \textbf{ else } \bot \\
&= (|ve| \big| |\mathcal{R}(\varsigma)|)(\widehat{b}).
\end{aligned}$$

□

**Lemma 8.10.** *If $ve_1 \sqsubseteq ve_2$ and $\widehat{B}_1 \sqsubseteq \widehat{B}_2$, then $\widehat{ve}_1|\widehat{B}_1 \sqsubseteq \widehat{ve}_2|\widehat{B}_2$.*

*Proof.* By reasoning similar to Lemma 8.9. □

## 9. Extensions

We can add primops and conditionals to the analysis in the standard way, which is described in Shivers' work [8]. Assuming appropriate modifications to the syntax, we can handle `letrec` with an $\widehat{Eval} \to \widehat{Eval}$ transition:

$$\begin{aligned}
&([\![\texttt{(letrec ((}v_i\ lam_i\texttt{))}\ call\texttt{)}]\!], \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \\
&\qquad\qquad\qquad\qquad \approx> (call, \widehat{\beta}', \widehat{ve}', \widehat{\mu}', \widehat{t}')
\end{aligned}$$

$$\text{where} \begin{cases}
\widehat{t}' = \widehat{succ}(\widehat{t}) \\
\widehat{\beta}' = \widehat{\beta}[v_i \mapsto \widehat{t}'] \\
\widehat{d}_i = \widehat{\mathcal{A}}(lam_i, \widehat{\beta}', \widehat{ve}) \\
\widehat{ve}' = \widehat{ve} \sqcup [(v_i, \widehat{t}') \mapsto \widehat{d}_i] \\
\widehat{\mu}' = \widehat{\mu} \oplus [(v_i, \widehat{t}') \mapsto 1].
\end{cases}$$

We can also add a store to the semantics and apply abstract counting and abstract GC to it. The store itself is merely an extra component within the state, accessed by primops. The counter $\widehat{\mu}$ must then also map abstract store locations into $\widehat{\mathbb{N}}$. This implies that the reaching function $\widehat{\mathcal{R}}$'s range may contain both bindings and locations. It also means that the adjacency relation $\widehat{\leadsto}$ must be parameterized by both the variable environment and the store.

## 10. Applications

Control-flow analysis, for which we enhance precision, offers a number of applications, including but certainly not limited to constant propagation, useless-variable elimination and induction-variable elimination.

Environment analysis, which we perform through abstract counting, enables a more exotic array of optimizations, including lightweight closure conversion, Super-$\beta$ lambda propagation, Super-$\beta$ copy propagation and continuation promotion [8, 11, 7, 9].

We have recently shown [9] how these analyses, applied to CPS representations, permit compilers to fuse together graphs of online transducers. We hope to apply this technology to programs such as DSP systems, network protocol stacks and graphics pipelines. The analyses we've presented in this paper were critical to the transducer-fusing transforms we have demonstrated in that setting.

Abstract counting can be brought to bear on environment analysis by the following theorem, which provides the environment condition allowing us to infer concrete environment equality.

**Theorem 10.1** (Environment condition)**.** *Take two abstract environments $\widehat{\beta}_1$ and $\widehat{\beta}_2$ that are reachable from the same abstract state. Let $\widehat{\mu}$ be the abstract binding counter from this state. Let $\beta_i$ be any concrete environment corresponding to $\widehat{\beta}_i$. Then, $\beta_1(v) = \beta_2(v)$ if $\widehat{\beta}_1(v) = \widehat{\beta}_2(v)$ and $\widehat{\mu}(v, \widehat{\beta}_1(v)) = \widehat{\mu}(v, \widehat{\beta}_2(v)) = 1$.*

*Proof.* By the definition of the simulation relation. □

## 11. Results and implementation

We have a prototype implementation written in Haskell which accepts a small subset of direct-style Scheme. It allows a choice of 0CFA or 1CFA on the command-line and the option to turn off abstract GC or abstract counting. The source code itself is a close mapping of the mathematics, plus strictness annotations for efficiency.

Table 1 measures improvement in analytic precision resulting from just abstract GC. To get a general feel for the improvement in precision, we include the number of states visited for ordinary $k$-CFA and our ΓCFA. Generally speaking, fewer states represents

| Program | 0CFA | | 0CFA+GC | | 1CFA | | 1CFA+GC | |
|---|---|---|---|---|---|---|---|---|
| | States | Time | States | Time | States | Time | States | Time |
| `fact-tail` | 28 | $\epsilon$ | 28 | $\epsilon$ | 28 | $\epsilon$ | 28 | $\epsilon$ |
| `fact-y-combinator` | 130 | $\epsilon$ | 80 | $\epsilon$ | 202 | $\epsilon$ | 110 | $\epsilon$ |
| `nested-loops` | 80 | $\epsilon$ | 71 | $\epsilon$ | 214 | $\epsilon$ | 82 | $\epsilon$ |
| `put-double-coroutines` | 3339 | 7m 51s | 808 | 17s | 9460 | 56m 30s | 1813 | 57s |
| `integrate-fringe-coroutines` | 7428 | 41m 31s | 1619 | 1m 06s | 31741 | 12h 31m 08s | 6340 | 6m 55s |
| `integrate-stream-coroutines` | 11540 | 3h 11m 54s | 2066 | 2m 46s | 27032 | >12h | 7055 | 9m 09s |

**Table 1.** Improvements for $k$-CFA and GC without abstract counting. We report the number of states reached during analysis as well as running time of the analysis. In both cases, lower values are better. A time of $\epsilon$ means that the analysis finished in less than a second.

| Program | 0CFA+GC | | 1CFA+GC | |
|---|---|---|---|---|
| | Inlines w/o Counting | Inlines w/Counting | Inlines w/o Counting | Inlines w/Counting |
| `fact-tail` | 2 | 4 | 2 | 4 |
| `fact-y-combinator` | 4 | 8 | 4 | 8 |
| `nested-loops` | 4 | 10 | 4 | 10 |
| `put-double-coroutines` | 28 | 55 | 28 | 55 |
| `integrate-fringe-coroutines` | 45 | 77 | 45 | 77 |
| `integrate-stream-coroutines` | 46 | 72 | 47 | 76 |

**Table 2.** Improvement in lambda/constant propagation due to abstract counting. All runs used abstract garbage collection. We report the number of expressions which can be replaced by a $\lambda$ term/constant in the CPS-converted code. When abstract counting is enabled, $\lambda$ terms containing free variables may become eligible for (Super-$\beta$) inlining.

higher precision. Timing measurements were conducted on a 2.0 GHz Pentium 4 machine with 2 GB RAM running Fedora Core 4 Linux.[8]

For the more complicated examples, we were able to achieve order of magnitude improvements in every case. The improvements were less dramatic for the toy examples (such as `fact-*`) because they offered less opportunity to improve: their small size left less room for imprecision to compound itself. Most examples make heavy use of Church encodings for constructs such as lists and option types. This is appropriate for testing as it makes the control-flow analyzer's job *more* difficult (but we have no compelling reason for using these encodings; it is simply an artifact of our crude, prototype compiler). The `*-coroutines` examples make liberal use of `call/cc` to build cooperatively multithreaded communication channels for stream processing [9].

Table 2 provides measurements for abstract counting's ability to inline $\lambda$ terms or constants in place of variables. The number of inlines with and without counting is given for comparison. All examples in this table were run with abstract garbage collection enabled.

## 12. Related work

The work we've developed in this paper lies at the confluence of three lines of research: (1) prior work in control-flow analyses; (2) prior work in environment analyses; and (3) prior work in continuation-passing style representations.

From a control-flow analysis perspective, these techniques descend from the broader body of work in higher-order control-flow analysis, such as Shivers' development of the $k$-CFA hierarchy [8]. By remaining agnostic to the structure of the abstract contour set, our GC framework is orthogonal to, and synergistic with, most of the subsequent innovations in CFA, such as Agesen's CPA [1] and Wright and Jagannathan's polymorphic splitting [12]. That is, the ΓCFA framework should be able to take nearly any control-flow analysis and make it more precise.

Shivers [8] introduced the term "environment analysis," the higher-order analog to must-alias analysis for variables and environments. His initial solution, reflow analysis, operates on the same principle underlying our work: inferring when an abstract object has only one corresponding concrete object. He achieves this by selectively allocating a single unique abstract contour *once* at a point of interest during the analysis. For the remainder of the analysis, this abstract contour is then effectively equivalent to a concrete contour. This approach, however, suffers from the drawback that the analysis must be re-run for each point of interest, and it does not have the benefit of GC to improve precision. The techniques we've presented here could be considered as a sort of "opportunistic reflow analysis." Our work is further differentiated by a proof of correctness. (We suspect the proof techniques we employed to show the correctness of abstract counting could be employed to show the correctness of reflow analysis.)

With regard to must-alias analysis, our GC and counting analyses are related to the line of work initiated by Hudak's abstract reference counting [5], continued by Chase's [2] strong update and generalized by Jagannathan [6]. Our abstract counter $\widehat{\mu}$ and reachability function $\widehat{\mathcal{R}}$ are quite similar to Jagannathan's cardinality maps and reachmaps; in fact, Jagannathan described his technique as "an abstract form of garbage collection." Of the work that we know, Jagannathan is the first to use abstract garbage collection in a higher-order analysis, and also the first to perform environment/must-alias analysis through the notion of "singleness." In these ways, his result is the closest to our own; it differs from our work in that:

- Our analysis supports polyvariance.
- Our analysis is a fundamental shift in granularity from the variable level to the binding level.
- We operate over CPS rather than direct style, which makes it simple to use an operational semantics for performing our analysis, instead of constraint-solving.

[8] In the last test case, the benchmark machine lacked sufficient uptime and availability (> 12 hours) to finish the 1CFA control test. We were still able to compute the number of states visited on a dual 4 GHz Athlon in 6 hours.

- We need no explicit support for "strong update," as GC provides exactly the same effect. That is: $\rho \sqcup [x \mapsto y] = \rho[x \mapsto y]$ when $\rho(x) = \bot$.
- Our reachability analysis is computed on-the-fly rather than once, and we do not need to run multiple iterations of the analysis to achieve the best results possible.

In other work [7], we have developed a technique, ΔCFA, for performing environment analysis using abstract frame strings. Like other environment analyses, ΔCFA relies upon the ability to infer concrete equality from certain abstract conditions. Both abstract GC and counting are orthogonal to and synergistic with ΔCFA. In practice, we have observed very significant improvements in speed and precision when we added these techniques to our ΔCFA trials.

A second line of work regarding environment analysis was initiated by Wand and Steckler's use of invariance sets [11]. Their analysis is not (outwardly) rooted in the notion of determining concrete equality from the abstract, but rather in determining which variables must remain unchanged—*invariant*—across machine transitions. Wand and Steckler also introduced lightweight closure conversion, a cousin of Shivers' Super-$\beta$ inlining, to motivate the need for their environment analysis. Hannan [4] later translated this technique to a type system. The invariance-set approach to environment analysis, however, suffers from an inability to handle certain common cases, such as when a closure escapes its context of creation.

Our analysis also draws on the body of work that supports the CPS-as-intermediate-representation thesis. The foundational work here is by Steele [10]. Shivers' earlier work [8] in CPS-based analysis has provided the basic framework for the techniques we've developed here. CPS lends itself to analysis based on a state-collecting abstract interpretation because it corresponds so naturally to a state machine. In the context of our GC operations, having a simple state machine means that we can freeze execution at intermediate states, perform a GC, and then resume. We could achieve this in a non-CPS setting, with a semantics based on context grammars or progress-establishing inference rules, but it would complicate the analysis and its correctness proofs. With CPS, we don't have to add machinery to our semantics to handle evaluation context, or worry about or reference subcomputations appearing in a justification tree for a given machine step.

## 13. Future work

We are currently in the process of adding these analyses to an industrial-strength compiler for full SML. With this, we hope to obtain measurements on precision and time improvements for very large code bases. We are also curious to see if there are the performance tradeoffs involved in using Hudak's abstract reference counting [5] rather than the tracing-style garbage collector presented here. Reference counting has the benefit of providing timely deallocation, which could then spread apart the points where need-driven tracing GC would have to be invoked by the analysis.

## Acknowledgments

Suresh Jagannathan very kindly pointed out some important related work that helped the development of our ideas. We'd also like to thank Ben Chambers, Daniel Harvey and our anonymous reviewers, whose detailed feedback made for a much better paper.

## References

[1] AGESEN, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP 1995* (1995), pp. 2–26.

[2] CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. Analysis of Pointers and Structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, New York, June 1990), pp. 296–310.

[3] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California, Jan. 1977), vol. 4, pp. 238–252.

[4] HANNAN, J. Type Systems for Closure Conversion. In *Workshop on Types for Program Analysis* (1995), pp. 48–62.

[5] HUDAK, P. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, Aug. 1986), pp. 351–363.

[6] JAGANNATHAN, S., THIEMANN, P., WEEKS, S., AND WRIGHT, A. K. Single and loving it: Must-alias analysis for higher-order languages. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (San Diego, California, January 1998), pp. 329–341.

[7] MIGHT, M., AND SHIVERS, O. Environment Analysis via ΔCFA. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Charleston, South Carolina, January 2006), pp. 127–140.

[8] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[9] SHIVERS, O., AND MIGHT, M. Continuations and transducer composition. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Canada, June 2006).

[10] STEELE JR., G. L. RABBIT: a compiler for SCHEME. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[11] WAND, M., AND STECKLER, P. Selective and lightweight closure conversion. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Portland, Oregon, January 1994), vol. 21, pp. 435–445.

[12] WRIGHT, A. K., AND JAGANNATHAN, S. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*, 1 (January 1998), 166–207.