JFP **18** (5 & 6): 821–864, 2008. © 2008 Cambridge University Press
doi:10.1017/S0956796808006941 First published online 12 August 2008 Printed in the United Kingdom

Exploiting reachability and cardinality in higher-order flow analysis

MATTHEW MIGHT
Diagis, LLC, Atlanta, GA, USA
(*e-mail:* matt@diagis.com)

OLIN SHIVERS
Northeastern University, Boston, MA, USA
(*e-mail:* shivers@ccs.neu.edu)

Abstract

We present two complementary improvements for abstract-interpretation-based flow analysis of higher-order languages: (1) *abstract garbage collection* and (2) *abstract counting*.[1,2] Abstract garbage collection is an analog to its concrete counterpart: the analysis determines when an abstract resource has become unreachable, and then, re-allocates it as fresh. This prevents flow sets from joining during abstract interpretation, which has two immediate effects: (1) the precision of the interpretation increases and (2) its running time often falls. In abstract counting, the analysis tracks 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 knowledge, in turn, drives environment analysis, expanding the kind (rather than just the degree) of optimization available to the compiler.

1 Introduction

Two complementary ideas lie at the core of this work:

1. An abstract interpretation can make more efficient use of the finitized resources available by using the abstract analog to garbage collection.
2. By counting the number of concrete counterparts to an abstract resource, equality in the abstract state-space can imply equality in the concrete state-space.

In an abstract interpretation (Cousot & Cousot 1977), a smaller, often finite set of abstract elements represents an infinite set of concrete elements. In both the concrete space and the abstract space, some of these elements are addressed. When a concrete machine runs out of fresh addresses to allocate, it can either abort execution or attempt to garbage collect. However, when an abstract machine runs out of fresh

[1] A preliminary version of this work is appeared in the 2006 Proceedings of the International Conference on Functional Programming.

[2] This material is based upon work supported by the National Science Foundation under Grants No. 0638060 and 0438871.

addresses, the standard behaviour is to re-allocate an address already in use, thereby forcing multiple abstract values to reside at the same slot in an abstract store. (In fact, the analysis will frequently re-allocate an in-use abstract address even though free addresses are still available.) By garbage collecting these abstract addresses – setting the values associated with them back to empty sets – this merging is frequently avoidable.

Abstract counting exploits the fact that each abstract address represents a set of concrete addresses. By conservatively counting the number of elements in such a set, a simple yet effective principle often applies when dealing with sets of size one: if $\{x\} = \{y\}$, then $x = y$. Or, rephrased for our purposes, if two abstract addresses $\hat{a}_1$ and $\hat{a}_2$ are equal and each represents only a single concrete address, then the concrete addresses they represent must also be equal.

In an imperative setting, this principle yields a must-alias analysis. In a higher-order setting, which is the focus of this paper, this principle yields an environment analysis.

Let us review briefly the results of a flow analysis. A typical flow-analysis problem is to associate each expression in a programme with a conservative set of the values to which it may evaluate at run time. For example, on the following fragment,

```
(f i 0 a)
```

a flow analysis might produce:

- `f` is a closure over λ_{42} or λ_{314},
- `i` is an integer,
- `0` is the constant `0`,
- `a` is an array which was allocated either on line 13 or line 217 of the programme.

Inherent in a flow analysis is the possibility of false positives, e.g. it may actually be that the expression `f` never evaluates to a closure over λ_{314} at execution time.

Under variants of the standard 0CFA algorithm, these false positives happen because flow sets for return values and parameters merge. The monotonicity that irreversibly commits a traditional flow analysis to an expression-value association once that association is made compounds such merging. For instance, consider the following code:

```
(map f list-1)
(map g list-2)
```

Due to merging, 0CFA will conclude that the values flowing from `(map g list-1)` might also flow from `(map f list-2)`. As we will see, abstract garbage collection's strength comes, in part, from its ability to violate this monotonicity in a sound fashion.

Abstract counting approximates, for each abstract state, how many concrete resources each abstract address (which, in the coming model, includes both bindings and store locations) represents. When an abstract binding or location has an upper bound of one concrete counterpart, then the equality of this resource in the abstract space implies the equality of whatever it represents in the concrete space. In addition,

counting can also answer less abstract questions, such as: 'Given a variable, how many instances of this variable can be live at the same time?' Even in a first-order language, this information could justify the globalization of otherwise stack-allocated data, such as the variable `n`, and then, the parameter `x` in the following C code:

```
                          int n;                      int n, x;
int foo(int x) {    ⟹    int foo(int x) {    ⟹      int foo() {
  int n = x*x;              n = x*x;                    n = x*x;
  foo(n);                   foo(n);                     x = n;
}                         }                           foo();
                                                    }
```

In this example, the variable `n` is dead once the function `foo` is called, so the same slot in memory can be used for all instances of the variable `n`. The same argument applies to the parameter `x`, except that callers to `foo` must also place the first argument in the global variable `x`.

In a higher-order language, this information opens up optimizations such as super-β inlining. For example, suppose a flow analysis reports that at the call site `(f x)` only closures over the λ term `(λ (y) z)` are invoked. Is it safe to inline this λ term directly at this call site, turning it into `((λ (y) z) x)`? The answer depends on whether or not the value of the variable `z` captured in the closure will always be the same as when the call is made. If abstract counting reports that only one instance of the variable `z` exists at the time of the call, then clearly the binding of the variable `z` captured in the closure and the binding of the variable `z` at the call site are the same value, thereby making the inlining safe.

Before we begin, it is worth emphasizing the abstract-interpretive nature of the forthcoming analysis, ΓCFA. Readers familiar with the constraint-based formulation of k-CFA may be seeing this abstract-interpretive formulation for the first time, and are advised to pay careful attention to the differences.

In this work, we use both the terms *garbage collection* and *collecting semantics*. To avoid confusion, 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.

2 Conventions

For all of the domains used in this work, we assume the 'natural' meaning for the lattice operator $\sqcup$ 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 $A = \mathscr{P}(B)$, we define $\bot_A = \emptyset$ and $\top_A = B$; the order relation and join operator are then

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

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

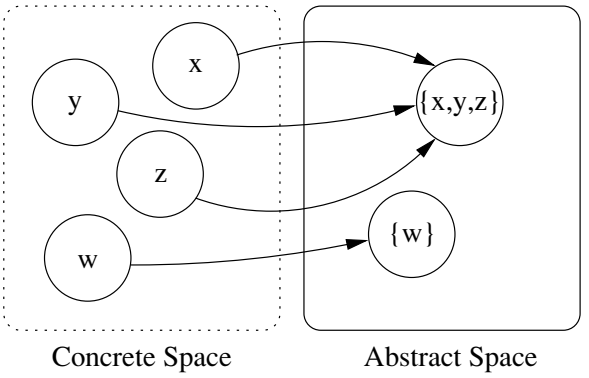


Fig. 1. Collisions in the concrete-to-abstract map.

an element outside of its domain, it yields the bottom element, $\perp$; thus, we get $dom(f) = \{x : f(x) \neq \perp\}$. 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 space represent multiple elements of the concrete space (Figure 1). It is 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 signs. The concrete set is the integers, $\mathbb{Z}$. The abstract set is the power set of signs, $\hat{\mathbb{Z}} = \mathscr{P}(\{-, 0, +\})$. The abstraction map $|\cdot| : \mathbb{Z} \to \hat{\mathbb{Z}}$ 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 the operator $\oplus : \hat{\mathbb{Z}} \times \hat{\mathbb{Z}} \to \hat{\mathbb{Z}}$. For example:

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

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

- Had we simply evaluated $4 + {}^{-}4$ and then abstracted, we would have $|4 + {}^{-}4| = \{0\}$. That is, abstract interpretation strictly over-approximated, 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 may act 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 the equality of their concrete counterparts. 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:

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

  This is the role of abstract counting: to determine when such an inference is valid, chiefly by determining when the abstract resource under consideration represents a singleton set – just as abstract 0 represents the set $\{0\}$.

Example: A traditional flow analysis This example looks at 0CFA (Shivers 1988; 1991) to highlight how overlap in the concrete-to-abstract mapping damages precision. The purpose of abstract garbage collection, presented later, is to opportunistically alleviate such merging. Consider a traditional, constraint-based control-flow analysis (Palsberg 1995) for the pure, call-by-value λ-calculus:

$$\begin{array}{rcll} e, f \in EXP & = & VAR + LAM + APP & \text{(expression)} \\ v \in VAR & = & \text{a set of identifiers} & \text{(variable)} \\ lam \in LAM & ::= & (\lambda\ (v)\ e) & (\lambda \text{ term}) \\ APP & ::= & (f\ e) & \text{(application)} \end{array}$$

Starting with the concrete, environment-based semantics given in the left-hand side of Figure 2, we can drop the environment component ρ and reformulate the semantics to arrive at the control-flow constraints given in the right-hand side of Figure 2. In this formulation, the expression $a \approx\!\!> b$ reads as 'b flows to a', or more precisely, 'a may evaluate to a closure over b'.

For the control-flow constraints, the [apply] rule states: 'If a λ term flows to the procedural position of an application, and a value flows to the body of that λ term, then that same value also flows out of the application'; the [eval-lambda] rule states: 'A λ term flows to itself"; and the [eval-var] rule states: 'If a λ term with formal v flows to a call site, then whatever flows to the argument of the call site also flows to the variable v'.

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

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

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

Fig. 2. A concrete big-step semantics for call-by-value λ-calculus (left), and its control-flow constraints for 0CFA (right). The variable ρ represents a variable-to-value environment.

The constraints for [apply] and [eval-var] help to illustrate why and where merging happens. The constraint for [apply] merges all values flowing out of the body of a λ term together, regardless of context, while the constraint for [eval-var] merges all values flowing to the formal v together, regardless of context.

By finding the least relation $\twoheadrightarrow$ such that the constraints in Figure 2 are satisfied, the relation $\twoheadrightarrow$ represents the flow-insensitive results of 0CFA (Shivers 1991). Consider the following fragment of Scheme code:

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

While analysing this fragment, 0CFA picks up the flow `x` $\twoheadrightarrow$ *lam* from the call site `(id lam)`. Next, it picks up the flow `x` $\twoheadrightarrow$ *lam*′ from the body of the `let*`. Because the variable `x` is the body of the identity function `id`, 0CFA thinks that the term *lam* and the term *lam*′ could be returned anywhere that the function `id` is called. As a result, 0CFA tells us that the above fragment could yield a closure over either the term *lam* or the term *lam*′, when, in fact, only a closure over the term *lam*′ is possible.

The root cause of this loss in precision is the way in which 0CFA handles environments under abstraction: 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 (frequently called *contours* (Shivers 1988; 1991; Wright & Jagannathan 1998; Jagannathan *et al.* 1998)) to be distinguishable from one another. Shivers' 1CFA (1991), for instance, uses a distinct abstract context for each call site. That is, when a λ term is invoked at a call site, the context for the binding made there is the call site itself. Agesen's CPA (1995), on the other hand, utilizes 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 programme. Given one of these finite sets, the purpose of abstract garbage collection is to make more efficient use of it. It turns out that abstract garbage collection then generalizes to other resources allocated during an abstract interpretation, including store locations, closures, list cells and time-stamps.

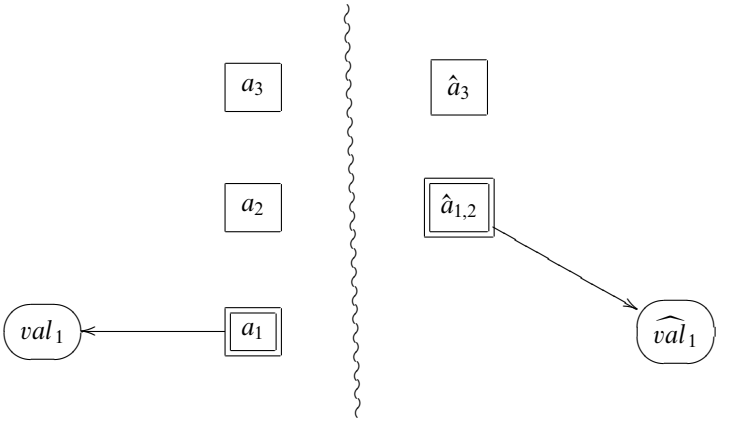
4 Abstract garbage collection and counting

In abstract interpretation, the state-space of the computation, and everything comprising it, is finite. Because addresses in the heap are members of that state-space, they too are finite. Consequently, allocations during abstract interpretation draw on that finite set of addresses, and, at some point (for any non-trivial programme), an abstract address that is already in use will be re-allocated.

Abstract garbage collection tackles the issue of scarcity by trying to make more efficient use of the abstract address space. As abstract addresses become unreachable, the analysis discards them *along with the resources or values which are reachable only via the these newly discarded addresses*. More precisely, when an address becomes unreachable, the set of values associated with it is reset to the empty set. Such behaviour is sound due to the principle that if an abstract binding has become unreachable within a state, then so must have all of the concrete bindings that it represents.

To help illustrate the concept, we will walk through several steps of execution for a three-address concrete machine and its two-address abstract counterpart. We use the more general term *address* to refer to entities such as bindings and store locations.

The diagram below encodes the initial state of the configuration for these two machines:



Square-edged boxes represent addresses: a_1, a_2 and a_3 in the concrete; $\hat{a}_{1,2}$ and $\hat{a}_3$ in the abstract heap. In this particular example, address a_1 and address a_2

abstract to address $\hat{a}_{1,2}$, while only address a_3 abstracts to address $\hat{a}_3$. Double boxes represent addresses in the root set from the perspective of garbage collection; that is, these are the addresses which are immediately touched by a machine state. For the purposes of this example, assume that there is a single register r (and its abstract equivalent $\hat{r}$) whose contents decide the root set – that is, for these machines, the root set is always a singleton. (One could view the register as pointing to the current environment record.) Rounded boxes represent values. In this case, the value val_1 is at address a_1, and its abstract counterpart $\widehat{val}_1$ is at the abstract counterpart of address a_1: $\hat{a}_{1,2}$. The machine could wind up in this state after the following pseudo-instructions:

$$*a_1 \leftarrow val_1$$
$$r \leftarrow a_1$$

At present, the abstract heap is a simulation of the concrete heap: for every concrete address to which a value is assigned, the abstract counterpart of that address has an abstraction of that value assigned to it. Because the abstract heap is a simulation of the concrete heap, this diagram is *sound*. It would be unsound if, for instance, the value $\widehat{val}_1$ were not present at address $\hat{a}_{1,2}$ in the abstract heap.

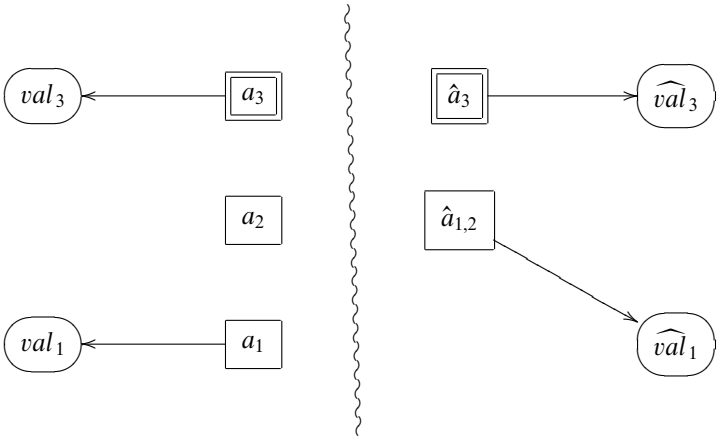
For the first step of execution, assign the value val_3 to address a_3; that is, execute the pseudo-instruction:

$$*a_3 \leftarrow val_3$$

In order to preserve soundness, we must assign its abstract counterpart $\widehat{val}_3$ to address $\hat{a}_3$. Directly thereafter, shift the root pointer to address a_3 in the concrete heap (and, hence, to address $\hat{a}_3$ in the abstract heap), as a result of the following pseudo-instruction:

$$r \leftarrow a_3$$

This results in the following diagram:

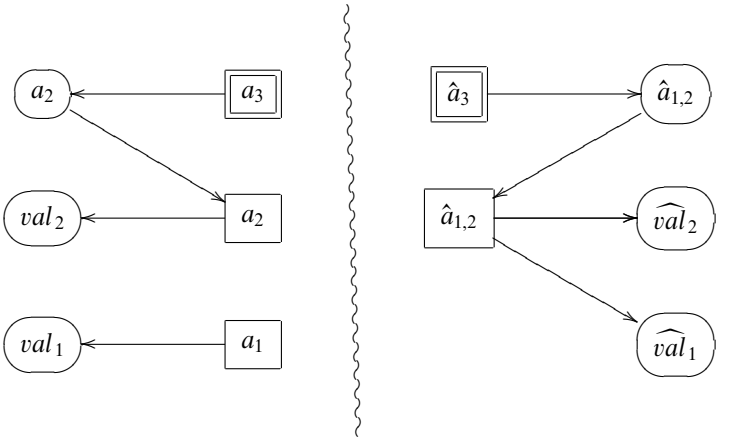


At this point, if we had a garbage collector available in the concrete, it would remove value val_1 from the configuration. Moving forward, we shall see how garbage collection (more precisely, abstract garbage collection) can actually improve the precision of an abstract interpretation.

For the next step of execution, assign the value val_2 to address a_2. Shortly thereafter, assign the pointer value a_2 to address a_3. That is, execute the following pseudo-instructions:

$$*a_2 \leftarrow val_2$$
$$*a_3 \leftarrow a_2$$

Once again, for soundness, we mirror the changes in the abstract. This results in the following diagram:



In this diagram, the damage to precision that results when a concrete address space is mapped to a smaller abstract address space is now apparent. In the concrete heap, the address a_2 points to value val_1. The abstract interpretation that we have been running simultaneously, however, now reasons that either value val_1 *or* val_2 could be at address a_2.

Part of the problem is abstract *zombies*. A zombie is an abstract value, which used to be unreachable (dead), that has once again become reachable (undead). In this diagram, the value $\widehat{val}_1$ has become a zombie. Zombies block optimisations such as run-time check removal (e.g. if val_1 were a cons cell and val_2 were nil). They can also increase the running time of the analysis. If, for example, the values $\widehat{val}_1$ and $\widehat{val}_2$ were procedures, and the abstract interpretation were to invoke the procedures sitting at address $\hat{a}_{1,2}$, the result would be a fork (Figure 3). Because forks further damage precision, this degrades into a vicious merge-fork-merge cycle.

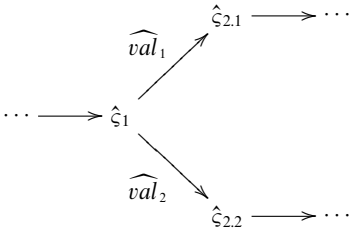
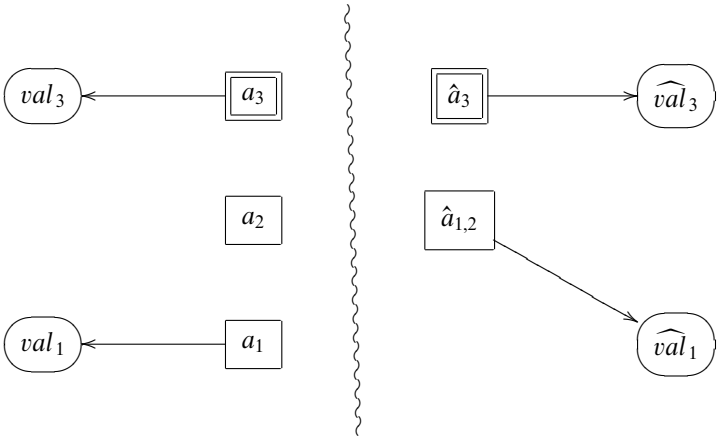
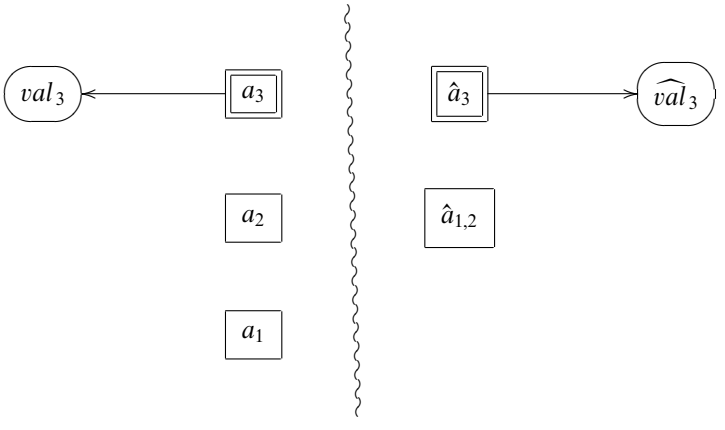


Fig. 3. A fork during analysis due to imprecision.

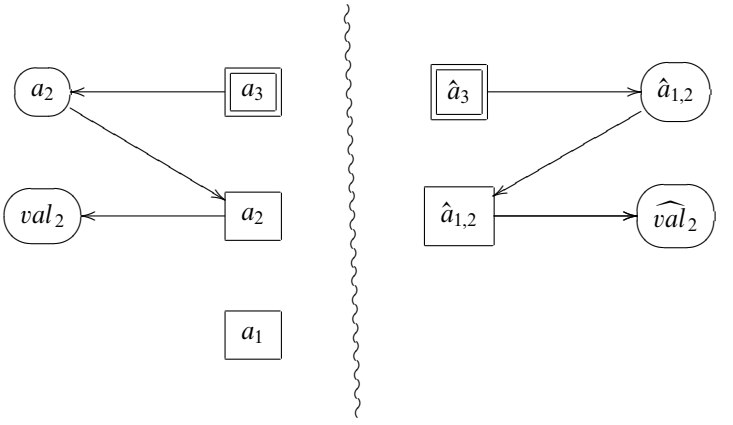
Fortunately, abstract garbage collection kills zombies before they become undead. Rewinding back to the pre-zombie configurations, we had:



Garbage collecting both the concrete heap and the abstract heap leads to:



Once again repeating the steps in the assignment of value val_2 to address a_2 and pointer value a_2 to address a_3 results in the following configurations:



In this final diagram, it is visually apparent that the abstract heap does not over-approximate the concrete heap. That is, due to the garbage collection of abstract zombies, no precision is lost. Before closing this example, note that in order for abstract garbage collection to be sound, we must also perform concrete garbage collection. If we garbage collect the abstract heap but not the concrete heap, there will exist values in the concrete with no abstract counterparts – a technical violation of soundness *even if* the concrete values are dead.

This exercise also suggests a way to improve the power[3] of the analysis: *abstract counting*. Re-run the example in this section, but with a count associated with each abstract address, starting at 0. (The address $\hat{a}_{1,2}$ starts off at a count of 1.) Each time an address is allocated (first used) in the concrete, increment the count of its corresponding abstract address. When an address gets garbage collected, reset its count to 0.

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 re-allocated, we can treat the address and whatever it points to as 'concrete'. Moreover, in the interim, equality for the address in the abstract heap implies equality for the corresponding address in the concrete heap.

By tracking the number of times an abstract address has been allocated and re-setting 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 9.

[3] We rank analyses on three axes: *speed*, *precision* and *power*. *Speed* refers to the running time of an analysis. *Precision*, for a fixed class of questions, refers to both (1) the frequency with which an analysis gives a definitive answer ('yes' or 'no' *v.* 'maybe') and (2) how tightly it constrains the set of possible programme behaviours. For example, the flow set $\{lam_{42}\}$ is more precise than the flow set $\{lam_{42}, lam_{314}\}$. *Power* refers to the class of questions which an analysis can answer. For example, many algorithms can answer may-alias questions, but it takes a more powerful analysis to answer must-alias questions.

5 Concrete semantics

In this section, we present a concrete, garbage-collecting semantics. Garbage collection trims the internal structure of intermediate states of execution; this allows its abstract interpretation (Cousot & Cousot 1977, 1979) to be similarly narrowed. We point out that if all we cared about was the return value of a programme (e.g. 314), we would not add garbage collection to a semantics – a semantics maps a programme to external observables; removing dead values via garbage collection does not change these. The reward for adding garbage collection to a semantics is entirely in the precision of the abstract interpretation.

Critical to our focus on higher-order languages, we extend the concept of garbage collection beyond its traditional realm of the store, as our garbage collector operates over environment structure.

5.1 Continuation-passing style

We define our analysis in terms of a continuation-passing style (CPS) representation. Using CPS simplifies the mathematics we develop, reducing the analysis to no more than its most essential ingredients. It is entirely possible to develop the analysis for a direct-style language, but this requires extra machinery that distracts from the presentation. As a side benefit, CPS will make handling control constructs such as `call/cc` and exceptions simpler.

CPS is λ-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*. The contract for p 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 multiplication. So, 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' λ-calculus in that:

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

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

Fig. 4. Semantic domains.

A side-by-side view of their grammars highlights these differences:

Direct-Style λ-calculus	CPS λ-calculus
$v \in VAR ::= identifier$	$v \in VAR ::= identifier$
$e, f \in EXP ::= v$	$e, f \in EXP ::= v \mid lam$
$\mid (\lambda\ (v_1 \cdots v_n)\ e)$	$lam \in LAM ::= (\lambda\ (v_1 \cdots v_n)\ call)$
$\mid (f\ e_1 \cdots e_n)$	$call \in CALL ::= (f\ e_1 \cdots e_n)$

where v is a variable, a member of the syntactic set VAR.

5.2 An environment-based CPS semantics

Even though we could use an ordinary λ-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.

The configurations of the small-step, environment-based semantics (Figure 5) range over the state-space described by the domains in Figure 4. Two kinds of states exist at the top level:

- *Eval* states. In *Eval* states, execution has reached a call site *call* in the context of a local environment β and some value environment *ve* at time t. Computations in this state await the evaluation of the function f into a procedure and the evaluation of the arguments $e_1, \ldots, e_n$ into values.
- *Apply* states. In *Apply* states, execution has reached the application of a closure (lam, β), or the *halt* continuation, to arguments $d_1, \ldots, d_n$. Execution proceeds by extending the closure's environment β with bindings for the formals in the λ term *lam*, and making an update into the variable environment.

The semantics given in Figure 5 have been simplified and factored in several ways to make reasoning about or abstracting the semantics simpler:

- *Time-stamps.* Each state contains a unique time-stamp. Making a transition increments this time-stamp via the function $succ : State \to Time$. The naturals

$$([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, t'), \text{ where}$$
$$proc = \mathscr{A}(f, \beta, ve)$$
$$d_i = \mathscr{A}(e_i, \beta, ve)$$
$$t' = succ(\varsigma)$$

$$((([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \beta), \boldsymbol{d}, ve, t) \Rightarrow (call, \beta', ve', t'), \text{ where}$$
$$\beta' = \beta[v_i \mapsto t] | free(call)$$
$$ve' = ve[(v_i, t) \mapsto d_i]$$
$$t' = succ(\varsigma)$$

Fig. 5. A small-step transition rule $\varsigma \Rightarrow \varsigma'$ for CPS, with a factored environment.

suffice for the set *Time* for the purpose of defining the meaning of a programme. For the purpose of proving the soundness of a particular analysis, it may be convenient to use ordered sets other than the naturals. This is why the function *succ* consumes the entire state and not merely the current time-stamp. The orderedness of time permits chronological reasoning in proofs, and time itself acts as a reliable source of 'freshness'. That is, for some strict partial order $<$, we require that:

$$t < succ(\ldots, t)$$

- *Factored environment*. The environment within each state is *binding factored*, which means that variables are mapped to values in two stages:
  1. A local binding-time environment, β, maps a variable to the time in which it was bound in this environment.
  2. A global binding-to-value environment, *ve*, maps a variable plus any time at which it was bound to its value at that time.[4]

  Factoring the environment removes recursion from the semantic domains, and enables reasoning about environment structure at the granularity of an individual *binding* – a variable/time pair. This also makes it clear that multiple bindings to the same variable can be simultaneously live: x can be bound at time 3 to 42, then bound again at time 94 to 217; both bindings can be captured in different closures. Mechanically, bindings behave exactly like the concrete addresses used in the previous section, with the value environment *ve* playing the role of heap. Consequently, bindings are the resource over which our garbage-collection algorithm will operate.[5]
- *Eval/apply factored transition*. The transition from state to state happens in two stages:
  1. *Argument evaluation*. The procedure and the arguments are evaluated.
  2. *Procedure application*. Evaluated arguments are bound to procedure formals.

[4] Note how this component *ve* increases monotonically over time. We call it 'global' because if we were coding an interpreter, the value environment *ve* would be pulled out of the state ς and implemented as a side-effected global table.

[5] The set *Time* is equivalent to the concrete contour set *CN* in Shivers' work (1991).

$$\mathscr{A} : EXP \times BEnv \times VEnv \rightharpoonup D$$
$$\mathscr{A}(v, \beta, ve) = ve(v, \beta(v))$$
$$\mathscr{A}(lam, \beta, ve) = (lam, \beta|free(lam))$$

Fig. 6. The function $\mathscr{A}$ evaluates arguments given a factored $BEnv/VEnv$ environment.

Factoring the transition in this fashion makes the addition of features such as `letrec`, primitive operations and conditionals simpler.

The argument-evaluation function $\mathscr{A}$ is shown in Figure 6. This function takes an argument and a factored-environment pair (β, ve) to a denotable value. Note how, when constructing a closure $(lam, \beta|free(lam))$, we trim the closure's environment by restricting its domain to those variables referenced by the closure's λ term, *lam*.

5.3 CPS as a state machine

Figures 4 and 5 show the semantic domains and the transition rules for our CPS semantics; these definitions, together with the ones for the function $\mathscr{A}$ in Figure 6, comprise a complete concrete semantics. For convenience, given some state ς, we will frequently refer to its components by subscripting the representative symbol with ς; 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 the small-step semantics for CPS 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 defined by a pair of axiom rules – there are no recursive inference rules. The time counter is clearly a 'machine clock' that assigns a unique, ordered time-stamp to each kind of state, and our semantic domains are not recursively defined.

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 a state. For our framework, *garbage collection* means finding the set of reachable bindings and restricting the domain of the global value environment *ve* to solely these bindings.

First, we define the set of bindings $\mathscr{T}(d)$ that value d immediately touches:

$$\mathscr{T}(lam, \beta) = \{(v, \beta(v)) : v \in dom(\beta)\}$$
$$\mathscr{T}(halt) = \{\}$$

A closure (lam, β) could potentially touch a binding (v, t) if the variable v is free in the term lam, and $\beta(v) = t$. In our semantics, we ensure that the domain of the local environment β is equal to the set of free variables in the term lam. We can extend the function $\mathscr{T}$ to objects such as states:

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

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

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

$$b_{\text{toucher}} \rightsquigarrow_{ve} b_{\text{touched}} \iff b_{\text{touched}} \in \mathscr{T}(ve(b_{\text{toucher}}))$$

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

$$\mathscr{R}(\varsigma) = \{b_{\text{reached}} : b_{\text{root}} \in \mathscr{T}(\varsigma) \text{ and } b_{\text{root}} \rightsquigarrow^*_{ve_\varsigma} b_{\text{reached}}\}$$

Now, we can define the GC function, $\Gamma : State \to State$:[6]

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

The function Γ removes unreachable bindings from the domain of the global value environment ve.

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

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

That is, the relation $\Rightarrow_\Gamma$ first performs a collection, and then, steps the execution forward. Our task in the next section will be to prove that this GC semantics is equivalent to the original semantics.

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

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

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.

[6] Recall that $f|X$ means 'the function f but only over the domain X'.

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 β. This can happen only if the top-level programme 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 value 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 the garbage-collecting semantics

We have two concrete operational semantics: an ordinary CPS semantics and a garbage-collecting CPS semantics. Our next task is a theory of correctness for relating these two machines. Ultimately, this means proving that the GC machine is a complete simulation of the original machine. Diagrammatically, this simulation looks like the following:[7]

$$\begin{array}{ccccccccccc}
\mathcal{I}(lam) & \Longrightarrow & \varsigma_1 & \Longrightarrow & \varsigma_2 & \Longrightarrow & \varsigma_3 & \Longrightarrow & \varsigma_4 & \Longrightarrow & \cdots \\
\equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \\
\mathcal{I}(lam) & \underset{\Gamma}{\Longrightarrow} & \varsigma'_1 & \underset{\Gamma}{\Longrightarrow} & \varsigma'_2 & \underset{\Gamma}{\Longrightarrow} & \varsigma'_3 & \underset{\Gamma}{\Longrightarrow} & \varsigma'_4 & \underset{\Gamma}{\Longrightarrow} & \cdots
\end{array}$$

On the path to this theorem, we will pull out lemmas that support the simulation and nurture intuition. On a first pass through this section, we recommend skipping the proofs while convincing yourself that the theorems and lemmas are intuitively correct.

Equivalence is the simulation relation between states that we need to preserve across transitions. We say that two states are *equivalent* if they have the same image under the function Γ:

Definition 6.1 (*Equivalent states*)
States ς_1 and ς_2 are **equivalent** iff $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$.

As required, this notion of equivalence preserves the value returned by the programme when the *halt* continuation is applied. Clearly, however, more than just the return value is preserved by this relation. Call sites, binding environments, procedures, arguments and time-stamps are also unchanged.

The first property that we define on a single state is *compactness*. A state is compact if environments found within it contain entries for exactly the variables required:

Definition 6.2 (*Compact state*)
A state ς is **compact** iff for each closure $(lam, \beta) \in range(ve_\varsigma)$, $free(lam) = dom(\beta)$, and:

- if $\varsigma = (call, \beta, ve, t)$, then $free(call) = dom(\beta)$, and

[7] Shortly, we will define *equivalence* ($\equiv$ in the diagram) as equality under garbage collection.

- if $\varsigma = (proc, \boldsymbol{d}, ve, t)$, then for each closure $(lam, \beta) \in \{proc, d_1, \ldots, d_n\}$, $free(lam) = dom(\beta)$.

The next property that we define on states is *well formedness.*

Definition 6.3 (*Well-formed state*)
A state ς is **well formed** iff

1. the state is compact,
2. every reachable binding has an entry, i.e. $\mathscr{R}(\varsigma) \subseteq dom(ve_\varsigma)$, and
3. no binding time found in the set $dom(ve_\varsigma)$ is higher than the current, t_ς.

The first requirement ensures that the programme has no free variables, and that all environments are minimal. The second requirement ensures that every reachable binding has a corresponding entry in the global value environment. The third requirement removes the possibility that a live slot in the global value environment will be smashed by a future binding step, and it lets us know that a binding created in the current time is fresh, i.e. distinct from all others.

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

Theorem 6.4
Well-formed states are not corrupt. That is, if a state ς is well formed, then either

- the transition $\varsigma \Rightarrow \varsigma'$ is legal,
- the state ς is a final state, or
- the state ς is stuck, but not corrupt.

Proof
By the definitions. □

Critically, if two states are well formed and equivalent, then either they transition together, or neither transitions.

With preliminaries taken care of, the first part of the simulation proof comes in two phases:

1. Prove that every well-formed state transitions to a well-formed state, or else, is final.
2. Prove that well formedness is preserved under garbage collection.

Combined, these demonstrate that garbage collection cannot introduce corruption.

Theorem 6.5
If a state ς is well formed and the transition $\varsigma \Rightarrow \varsigma'$ holds, then the new state ς' is well formed.

Proof
We consider only the reachable bindings property for well formedness. The other two properties are trivial. Assume that the state ς is well formed and the transition $\varsigma \Rightarrow \varsigma'$ holds. We divide into cases on ς.

- *Case* $\varsigma = ([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, t)$: Let the new state $\varsigma' = (proc, \boldsymbol{d}, ve, t')$. We must show that:

$$\mathscr{R}(\varsigma') \subseteq dom(ve)$$

  By the well formedness of the state ς, this reduces to showing that $\mathscr{R}(\varsigma') \subseteq \mathscr{R}(\varsigma)$. Choose a binding b^* in the set $\mathscr{R}(\varsigma')$; we will prove that this binding is also in the set $\mathscr{R}(\varsigma)$.

  Let $\langle b_0, b_1, \ldots, b_n \rangle$ be a path through the relation $\leadsto_{ve}^*$ such that $b_0 \in \mathscr{T}(\varsigma')$ and $b_n = b^*$. The new state ς' can touch the binding b_0 in one of two ways – as a member of the set $\mathscr{T}(proc)$, or as a member of the set $\mathscr{T}(d_i)$ for some i. Without loss of generality, assume that it was through the closure $proc$. We know that $proc = \mathscr{A}(f, \beta, ve) = (lam, \beta')$. We branch into the sub-cases induced by the definition of the evaluator $\mathscr{A}$.

  — *Subcase* f is a variable: In this case, the variable f is in $free(call)$, and hence, the binding $(f, \beta(f))$ is in the set of touched bindings $\mathscr{T}(\varsigma)$. From $b_0 \in \mathscr{T}(ve(f, \beta(f)))$, we have that $(f, \beta(f)) \leadsto_{ve} b_0$, and hence, that $\langle (f, \beta(f)), b_0, \ldots, b_n \rangle$ is a valid path which puts b^* in $\mathscr{R}(\varsigma)$.

  — *Subcase* f is a λ term: Let $b_0 = (v_0, t_0)$. We know that the variable v_0 is in the set $free(lam)$. Because $free(lam) \subseteq free(call)$, the binding (v_0, t_0) must also be in $\mathscr{T}(\varsigma)$. Hence, $\langle b_0, \ldots, b_n \rangle$ is a valid path which puts b^* in $\mathscr{R}(\varsigma)$.

- *Case* $\varsigma = ((([\![(\lambda\ (v_1 \cdots v_n)\ call)]\!], \beta), \boldsymbol{d}, ve, t)$: In this case, let the new state $\varsigma' = (call, \beta', ve', t')$. We must show that:

$$\mathscr{R}(\varsigma') \subseteq dom(ve')$$

  Choose a binding b^* in the set $\mathscr{R}(\varsigma')$; we will show that this binding is also in the set $dom(ve')$. We divide into cases on the freshness of b^*.

  — *Subcase* The binding b^* is fresh: Then, the binding time in b^* is the new time, t'. From the definition of the transition $\Rightarrow$, it is clearly in the domain of the new global environment ve'.

  — *Subcase* The binding b^* is not fresh: We will show that this binding was also in the domain of the old environment ve by finding that the binding was also reachable from the old state ς. The rest of this case then follows from the fact that $dom(ve) \subseteq dom(ve')$. Let $\langle b_0, \ldots, b_n \rangle$ be a path through the relation $\leadsto_{ve'}$, which puts b^* in the set $\mathscr{R}(\varsigma')$.

    1. First, we will show that either b_0 or b_1 is in $\mathscr{T}(\varsigma)$. Let $b_0 = (v_0, t_0)$.
       - Suppose $v_0 \in free(call)$ and this variable v_0 is not a bound formal. Then, v_0 was also free in the λ term, which means $(v_0, t_0) \in \mathscr{T}(proc) \subseteq \mathscr{T}(\varsigma)$.
       - Suppose alternately that $v_0 \in free(call)$ and that v_0 is a bound formal. Thus, for some argument i, $ve'(v_0, t_0) = d_i$, which means that $b_1 \in \mathscr{T}(d_i) \subseteq \mathscr{T}(\varsigma)$.
    2. Next, suppose $\langle b_1, \ldots, b_n \rangle$ did not form a valid path through $\leadsto_{ve}$ as well. Let i be the first index such that $ve(b_i) \neq ve'(b_i)$. By the definition of the transition relation $\Rightarrow$, it must be the case that the binding time

associated with b_i is the new time t'. This implies that either $b^* = b_0$, which is a contradiction, or that some time in the *middle* of the path is the fresh time t', which is also a contradiction. Hence, the path must have been valid through the relation $\rightsquigarrow_{ve}$ as well. Consequently, the binding b^* is in the set $\mathscr{R}(\varsigma)$, which, by well formedness, puts it in the domain of the old environment ve.

□

Next, we must show that performing a GC does not degrade well formedness.

Theorem 6.6
If the state ς is well formed, then the state $\Gamma(\varsigma)$ is well formed.

Proof
Assume that the state ς is well formed. By the definition of the reachability function $\mathscr{R}$, all paths starting from the set $\mathscr{T}(\varsigma)$ through the relation $\rightsquigarrow_{ve}$ are over the elements in the set $\mathscr{R}(\varsigma)$. Hence, any of these paths is also valid through the relation $\rightsquigarrow_{ve|\mathscr{R}(\varsigma)}$. As a result, $\mathscr{R}(\Gamma(\varsigma)) = \mathscr{R}(\varsigma) \subseteq dom(ve_\varsigma)$. Consequently, $\mathscr{R}(\Gamma(\varsigma)) = dom(ve_\varsigma|\mathscr{R}(\varsigma))$.
□

From the previous two theorems, we know that every state visited in both the GC and the non-GC semantics is well formed.

The following lemmas formalize intuition regarding garbage collection and reachability; first, if two states are equivalent, the bindings they reach are the same as well.

Lemma 6.7
If $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$, then $\mathscr{R}(\varsigma_1) = \mathscr{R}(\varsigma_2)$.

Note that the collection function Γ is idempotent:

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

We can also relate reachable bindings before and after transition:

Lemma 6.9 (*Containment*)
If ς is well formed and $\varsigma \Rightarrow \varsigma'$, then $(v,t) \notin ve_\varsigma$ and $(v,t) \in ve_{\varsigma'}$ implies that $t = t_{\varsigma'}$.

Or, in different words:

Corollary 6.10 (*Containment*)
If a state ς is well-formed and the transition $\varsigma \Rightarrow \varsigma'$ holds, then

- $\mathscr{R}(\varsigma') \subseteq \mathscr{R}(\varsigma)$ if ς is an eval state.
- $\mathscr{R}(\varsigma') - \{b : b \text{ is bound in this transition}\} \subseteq \mathscr{R}(\varsigma)$ if ς is an apply state.

The key inductive step in the simulation theorem is demonstrating that equivalence is preserved under transition:

Theorem 6.11 (*Complete simulation*)
If states ς_1 and ς_2 are well formed, and $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$, then either both states are terminal or $\varsigma_1 \Rightarrow_\Gamma \varsigma_1'$ and $\varsigma_2 \Rightarrow \varsigma_2'$ and $\Gamma(\varsigma_1') = \Gamma(\varsigma_2')$.

Proof
Assume that states ς_1 and ς_2 are well formed, and $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$. By the definition of the GC function Γ and well formedness, if one state is terminal, then so is the other. To avoid triple subscripts, let $\varsigma_i = (\ldots, ve_i, \ldots)$.

We consider only the case where the states are non-terminal. We must show that the subsequent states, ς_1' and ς_2', are equal under the GC function Γ; this reduces to showing the equality $ve_1'|\mathscr{R}(\varsigma_1') = ve_2'|\mathscr{R}(\varsigma_2')$, where the environment functions ve_1' and ve_2' are the global environments for the subsequent states; or, expanded:

$$ve_1|\mathscr{R}(\varsigma_1)[b_i \mapsto d_i]|\mathscr{R}(\varsigma_1') = ve_2[b_i \mapsto d_i]|\mathscr{R}(\varsigma_2')$$

By the Containment Lemma, this reduces to showing:

$$ve_1|\mathscr{R}(\varsigma_1') = ve_2|\mathscr{R}(\varsigma_2')$$

which, by $ve_1|\mathscr{R}(\varsigma_1) = ve_2|\mathscr{R}(\varsigma_2)$, reduces to showing:

$$\mathscr{R}(\varsigma_1') = \mathscr{R}(\varsigma_2')$$

We show this by contradiction. Suppose we could find a binding b^* that was in either the set $\mathscr{R}(\varsigma_1')$ or the set $\mathscr{R}(\varsigma_2')$ but not in both. Let the vector $\langle b_0, \ldots, b_n \rangle$ be the path justifying its membership. Let the index i be the lowest index such that the following does not hold:

$$ve_1|\mathscr{R}(\varsigma_1)[b_i \mapsto d_i](b_i) = ve_2[b_i \mapsto d_i](b_i)$$

Clearly, the binding b_i cannot be a fresh binding, so the condition must really be:

$$ve_1|\mathscr{R}(\varsigma_1)(b_i) = ve_2(b_i)$$

By the equivalence of states ς_1 and ς_2, this implies that the following does not hold:

$$ve_2|\mathscr{R}(\varsigma_2)(b_i) = ve_2(b_i)$$

And, this implies that $b_i \notin \mathscr{R}(\varsigma_2)$, which implies that $b_i \notin \mathscr{R}(\varsigma_1)$. But, if this were so, then the binding b_i could not be a member of the path justifying the membership of the binding b^* in either the set $\mathscr{R}(\varsigma_1')$ or $\mathscr{R}(\varsigma_2')$. □

7 Abstract semantics: ΓCFA

Thus far, we have developed a concrete, garbage-collecting semantics for CPS and proved its fidelity to the original semantics. Now, we shift gears and build a computable abstract semantics – our analysis – which approximates the concrete semantics. While it is possible to separate abstract GC and abstract counting, we add them both at the same time to avoid duplicating work. The machinery for abstract counting is encoded in a measure component: $\hat{\mu}$. The machinery for abstract garbage collection comes in the form of a state-to-state compaction function: $\hat{\Gamma}$. It is simple enough to tune parameters within this framework so that either feature is effectively 'turned off'. We term this combined framework ΓCFA a *garbage-collecting and counting flow analysis.*

The major components of this abstraction will be:

- An abstract domain for each concrete domain in Figure 4. The abstract counterpart for a given domain will be written with a hat on it, e.g. $\hat{D}$ is the abstraction of D. Figure 7 provides these domains.
- A family of abstraction functions – all written with the absolute-value-style notation $|\cdot|$ – which map elements from concrete domains (such as *State*, D, and *Clo*) into their corresponding abstract domains (such as $\widehat{State}$, $\hat{D}$ and $\widehat{Clo}$).
- An abstract garbage-collection function, $\hat{\Gamma} : \widehat{State} \to \widehat{State}$.
- Abstract transition rules $\approx\!\!>$ and $\approx\!\!>_\Gamma$ which approximate the concrete transition rules $\Rightarrow$ and $\Rightarrow_\Gamma$.[8]

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

$$|\varsigma| \sqsubseteq \hat{\varsigma} \implies |succ(\varsigma)| = \widehat{succ}(\hat{\varsigma})$$

By passing the state in as a parameter for abstract contour/time selection, changing the function $\widehat{succ}$ can alter the context sensitivity of the analysis. By leaving the exact structure and size of the set $\widehat{Time}$ unspecified, we allow the precision of the analysis, e.g. 0CFA, 1CFA, CPA, to be controlled externally.

The next significant change is the addition of an abstract binding counter, $\hat{\mu} \in \widehat{Measure}$, to each state. The value $\hat{\mu}(v, \hat{t})$ approximates how many concrete bindings are currently represented by the abstract binding $(v, \hat{t})$. For our work, we use three possible approximations – 0, 1 and ∞; that is, an abstract binding may represent no concrete bindings, at most a single concrete binding or an arbitrary number of concrete bindings.[9] An abstraction of the naturals, $\hat{\mathbb{N}} = \{0, 1, \infty\}$, represents these possibilities. We define the lattice operations for $\hat{\mathbb{N}}$ as: $\bot_{\hat{\mathbb{N}}} = 0$, $\top_{\hat{\mathbb{N}}} = \infty$, $\sqcup = \max$, $\sqcap = \min$ and $\sqsubseteq = \leqslant$.

Because $\sqsubseteq = \leqslant$, an abstract count is technically an *upper bound* on the number of concrete counterparts. That is, an abstract count of 1 means that there are either zero *or* one concrete counterparts. Over-approximations of a count result when a concrete binding is not reachable, but the abstract binding representing it still is. For most applications, an *upper bound* of 1 is just as good as knowing that there is exactly one counterpart, for when we are dealing with an abstract resource that has zero possible concrete counterparts, we have entered into strictly over-approximating state-space.

Percolating these changes through the rest of the domains leads to the abstract domains in Figure 7. 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 value environment $\widehat{ve}$ for a given abstract binding

[8] The symbol for the abstract transition relation $\approx\!\!>$ reads as 'makes an approximating transition to'.

[9] We are abusing our notation a bit here: the element ∞ does not mean an infinite number of bindings; it means an arbitrary (finite) number of bindings – 0 and 1 included.

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

Fig. 7. Abstract domains.

may need to represent multiple concrete values. This causes the domain of abstract denotable values $\hat{D}$ to become a power domain.

Combining the above leads to a natural definition for the abstract transition $\hat{\varsigma} \twoheadrightarrow \hat{\varsigma}'$:

$$
\begin{array}{c}
(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \twoheadrightarrow (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}'), \text{ where} \\
\widehat{proc} \in \hat{\mathscr{A}}(f, \hat{\beta}, \widehat{ve}) \\
\hat{d}_i = \hat{\mathscr{A}}(e_i, \hat{\beta}, \widehat{ve}) \\
\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})
\end{array}
$$

$$
\begin{array}{c}
((\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) \twoheadrightarrow (call, \hat{\beta}', \widehat{ve}', \hat{\mu}', \hat{t}'), \text{ where} \\
\hat{\beta}' = \hat{\beta}[v_i \mapsto \hat{t}] | free(call) \\
\widehat{ve}' = \widehat{ve} \sqcup [(v_i, \hat{t}) \mapsto \hat{d}_i] \\
\hat{\mu}' = \hat{\mu} \oplus [(v_i, \hat{t}) \mapsto 1] \\
\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})
\end{array}
$$

Note that the $\widehat{Eval}$-to-$\widehat{Apply}$ rule *branches* for each procedure. Here, the operator $\oplus$ is the natural abstraction of addition over $\hat{\mathbb{N}}$. The argument evaluator $\mathscr{A}$ abstracts directly to the abstract argument evaluator $\hat{\mathscr{A}}$:

$$
\begin{array}{c}
\hat{\mathscr{A}}(v, \hat{\beta}, \widehat{ve}) = \widehat{ve}(v, \hat{\beta}(v)) \\
\hat{\mathscr{A}}(lam, \hat{\beta}, \widehat{ve}) = \{(lam, \hat{\beta} | free(lam))\}
\end{array}
$$

It is worth taking a moment to point out where precision is lost. Putting the definitions of the transition relations $\Rightarrow$ and $\twoheadrightarrow$ side-by-side and looking at the

definitions of ve' and $\widehat{ve}'$, we notice a join ($\sqcup$) operation present in the abstract semantics that does not exist in the concrete semantics. (Recall that a value in the abstract space is a *set* of procedures, i.e. $\hat{D} = \mathscr{P}(Proc)$.) When the concrete semantics extend the environment ve, the new bindings are guaranteed to be fresh, because the current time has just increased. The abstract semantics, however, cannot extend the environment $\widehat{ve}$ to get $\widehat{ve}'$, because the bindings may *not* be fresh. If the analysis overwrote the value residing at $(v, \hat{t})$, then the analysis would no longer be sound, so instead, the analysis must merge the old and new values.

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

$$\begin{aligned}\widehat{\mathscr{T}}(lam, \hat{\beta}) &= \{(v, \hat{\beta}(v)) : v \in dom(\hat{\beta})\} \\ \widehat{\mathscr{T}}(halt) &= \{\} \\ \widehat{\mathscr{T}}\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} &= \widehat{\mathscr{T}}(\widehat{proc}_1) \cup \cdots \cup \widehat{\mathscr{T}}(\widehat{proc}_n)\end{aligned}$$

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

$$\begin{aligned}\widehat{\mathscr{T}}(call, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) &= \{(v, \hat{\beta}(v)) : v \in dom(\hat{\beta})\} \\ \widehat{\mathscr{T}}(\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) &= \widehat{\mathscr{T}}(\widehat{proc}) \cup \widehat{\mathscr{T}}(\hat{d}_1) \cup \cdots \cup \widehat{\mathscr{T}}(\hat{d}_n)\end{aligned}$$

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

$$\hat{b}_{\text{toucher}} \rightsquigarrow_{\widehat{ve}} \hat{b}_{\text{touched}} \iff \hat{b}_{\text{touched}} \in \widehat{\mathscr{T}}(\widehat{ve}(\hat{b}_{\text{toucher}}))$$

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

$$\widehat{\mathscr{R}}(\hat{\varsigma}) = \{\hat{b}_{\text{reached}} : \hat{b}_{\text{root}} \in \widehat{\mathscr{T}}(\hat{\varsigma}) \text{ and } \hat{b}_{\text{root}} \rightsquigarrow^*_{\widehat{ve}_{\hat{\varsigma}}} \hat{b}_{\text{reached}}\}$$

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

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

The chief difference between the abstract GC function $\hat{\Gamma}$ and the concrete collector Γ is that we also restrict the domain of the binding counter $\hat{\mu}$, effectively re-setting any unreachable bindings back to a count of 0.

With this, the abstract GC transition becomes

$$\frac{\hat{\Gamma}(\hat{\varsigma}) \twoheadrightarrow \hat{\varsigma}'}{\hat{\varsigma} \twoheadrightarrow_\Gamma \hat{\varsigma}'}$$

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

$$\hat{\mathscr{I}}(lam) = |\mathscr{I}(lam)|$$

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

Now that we have 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:

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

That is, we are not merging abstract bindings.

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 $\{lam'\}$.

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 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 programme pr as $\hat{\mathscr{V}}(pr)$. Eventual termination of the analysis is guaranteed because the space through which it roams, $\widehat{State}$, is finite.

Example: Abstract garbage collection This examples serves to illustrate abstract garbage collection of a single state in light of the heap diagrams from earlier. Consider the abstract state:

$$([\![\texttt{(f f halt)}]\!], \hat{\beta}, \widehat{ve}, \hat{t}_{\text{now}})$$

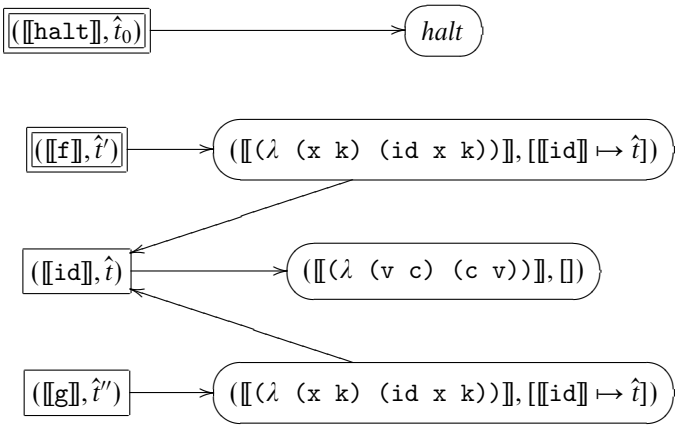
where the binding environment $\hat{\beta}$ is:

$$\hat{\beta} = [[\![\texttt{f}]\!] \mapsto \hat{t}', [\![\texttt{halt}]\!] \mapsto \hat{t}_0]$$

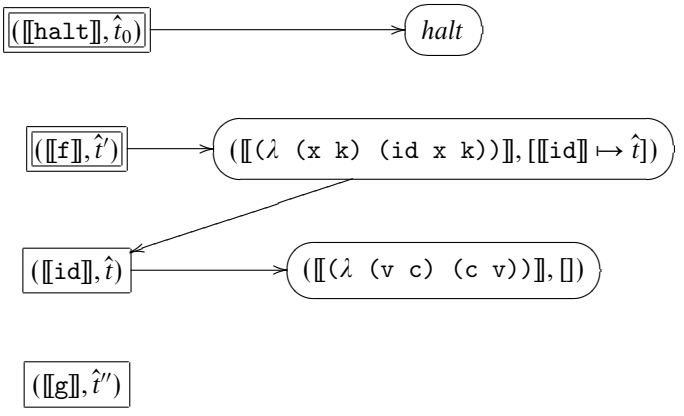
and the value environment $\widehat{ve}$ is:

$$\begin{aligned}
([\![\texttt{halt}]\!], \hat{t}_0) &\mapsto \{halt\} \\
([\![\texttt{id}]\!], \hat{t}) &\mapsto \{([\![\texttt{(}\lambda\texttt{ (v c) (c v))}]\!], [])\} \\
([\![\texttt{f}]\!], \hat{t}') &\mapsto \{([\![\texttt{(}\lambda\texttt{ (x k) (id x k))}]\!], [[\![\texttt{id}]\!] \mapsto \hat{t}])\} \\
([\![\texttt{g}]\!], \hat{t}'') &\mapsto \{([\![\texttt{(}\lambda\texttt{ (x k) (id x k))}]\!], [[\![\texttt{id}]\!] \mapsto \hat{t}])\}
\end{aligned}$$

Using the same schema as before, this value environment is visualized as:



After garbage collection, this value environment looks like:



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 is not hard to set up k-CFA for any k. It is also straightforward to set up Wright and Jagannathan's polymorphic splitting (1998) or Agesen's CPA (1995). By varying the set $\widehat{Time}$ and the next-time function $\widehat{succ}$, we can instantiate almost any conceivable variation on existing contour-selection strategies 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 speed and precision. The simplest policy, 'never GC', just gives us a counting flow analysis by abstract interpretation. The other extreme, which is to GC on every step, is simply not necessary: not every state is in danger of producing a *zombie*.

There are, however, still some benefits to GCing aggressively. For instance, there is a higher chance that the branch-termination check $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}_{\text{visited}}$ will succeed

than the check $\hat{\varsigma} \sqsubseteq \hat{\varsigma}_{\text{visited}}$. Moreover, the time cost of a GC does not appear to be significant, and implementation results suggest that GC costs are outweighed by the savings that we get from searching a smaller state space. (We will examine measurements to support this shortly.)

A sensible middle-ground policy 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 possible if we are about to add a binding for $(v, \hat{t})$, but $\hat{\mu}(v, \hat{t}) \geqslant 1$, or alternatively, $\widehat{ve}(v, \hat{t}) \neq \emptyset$.

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

Example: 0CFA v. *ΓCFA* Consider the direct-style code fragment:

```
(define (id x) x)
(id v1)
(id v2)
```

Clearly, the result of this programme is the value of `v2`. ΓCFA detects this fact. 0CFA, however, says it could be either the value of `v1` or `v2`. To see why these analyses diverge, let us desugar and CPS convert:

```
((λ (id)
   (id v1 (λ (_) (id v2 halt))))
 (λ (x k) (k x)))
```

Call this code fragment *call*.

To see the effect of GC, we will trace through abstract interpretation of this code for 0CFA context sensitivity, i.e. where the set $\widehat{Time}$ is a singleton. Simplifying matters, in 0CFA, binding environments ($\widehat{BEnv}$) disappear, value environments degenerate to $\widehat{VEnv} : VAR \to \mathscr{P}(LAM)$, and states no longer need time-stamps. (For this exercise, we also ignore the measure component $\hat{\mu}$.)

Suppose that *call* is evaluated (without abstract GC) in the abstract state $(call, \widehat{ve})$, where:

$$\begin{aligned} \widehat{ve}[\![\texttt{halt}]\!] &= \{halt\} \\ \widehat{ve}[\![\texttt{v1}]\!] &= \{\lambda_1\} \\ \widehat{ve}[\![\texttt{v2}]\!] &= \{\lambda_2\} \end{aligned}$$

In the subsequent *Apply* state, $([\![\texttt{(λ (id) ...)}]\!], \langle\{\lambda_{\text{id}}\}\rangle, \widehat{ve})$, we have:

$$\lambda_{\text{id}} = [\![\texttt{(λ (x k) (k x))}]\!]$$

This leads to the *Eval* state $([\![\texttt{(id v1 ...)}]\!], \widehat{ve}_1)$, where

$$\begin{aligned} \widehat{ve}_1[\![\texttt{halt}]\!] &= \{halt\} \\ \widehat{ve}_1[\![\texttt{v1}]\!] &= \{\lambda_1\} \\ \widehat{ve}_1[\![\texttt{v2}]\!] &= \{\lambda_2\} \\ \widehat{ve}_1[\![\texttt{id}]\!] &= \{\lambda_{\text{id}}\} \end{aligned}$$

Next, 0CFA enters an *Apply* state $(\lambda_{\text{id}}, \langle\{\lambda_1\}, \{\lambda_{\text{cont1}}\}\rangle, \widehat{ve}_1)$, where:

$$\lambda_{\text{cont1}} = [\![\texttt{(}\lambda\ \texttt{(_) (id v2 halt))}]\!]$$

The subsequent *Eval* state $([\![\texttt{(k x)}]\!], \widehat{ve}_2)$ is now in the body of the identity function, where:

$$\begin{aligned}
\widehat{ve}_2[\![\texttt{halt}]\!] &= \{halt\} \\
\widehat{ve}_2[\![\texttt{v1}]\!] &= \{\lambda_1\} \\
\widehat{ve}_2[\![\texttt{v2}]\!] &= \{\lambda_2\} \\
\widehat{ve}_2[\![\texttt{id}]\!] &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_2[\![\texttt{x}]\!] &= \{\lambda_1\} \\
\widehat{ve}_2[\![\texttt{k}]\!] &= \{\lambda_{\text{cont1}}\}
\end{aligned}$$

Next, control returns from the identity function to the continuation λ_{cont1}, leading to state $(\lambda_{\text{cont1}}, \langle\{\lambda_1\}\rangle, \widehat{ve}_2)$. This leads to the *Eval* state $([\![\texttt{(id v2 halt)}]\!], \widehat{ve}_3)$, where:

$$\begin{aligned}
\widehat{ve}_3[\![\texttt{halt}]\!] &= \{halt\} \\
\widehat{ve}_3[\![\texttt{v1}]\!] &= \{\lambda_1\} \\
\widehat{ve}_3[\![\texttt{v2}]\!] &= \{\lambda_2\} \\
\widehat{ve}_3[\![\texttt{id}]\!] &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_3[\![\texttt{x}]\!] &= \{\lambda_1\} \\
\widehat{ve}_3[\![\texttt{k}]\!] &= \{\lambda_{\text{cont1}}\} \\
\widehat{ve}_3[\![\texttt{_}]\!] &= \{\lambda_1\}
\end{aligned}$$

Next, 0CFA applies the identity function in state $(\lambda_{\text{id}}, \langle\{\lambda_2\}, \{halt\}\rangle, \widehat{ve}_3)$. Afterwards, 0CFA again evaluates the body of the identity function in the state $([\![\texttt{(k x)}]\!], \widehat{ve}_4)$, where:

$$\begin{aligned}
\widehat{ve}_4[\![\texttt{halt}]\!] &= \{halt\} \\
\widehat{ve}_4[\![\texttt{v1}]\!] &= \{\lambda_1\} \\
\widehat{ve}_4[\![\texttt{v2}]\!] &= \{\lambda_2\} \\
\widehat{ve}_4[\![\texttt{id}]\!] &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_4[\![\texttt{x}]\!] &= \{\lambda_1, \lambda_2\} \\
\widehat{ve}_4[\![\texttt{k}]\!] &= \{\lambda_{\text{cont1}}, halt\} \\
\widehat{ve}_4[\![\texttt{_}]\!] &= \{\lambda_1\}
\end{aligned}$$

At this point, the flow set for `x` has merged, and the flow set for `k` has merged. Consequently, this state has *two* successors: one applying the halt continuation to $\{\lambda_1, \lambda_2\}$, and one applying the continuation λ_{cont1} to $\{\lambda_1, \lambda_2\}$. Clearly, this second state is a spurious fork, and the merging of the flow sets for the variable `x` damaged the precision of the result: 0CFA says that either `v1` or `v2` could have returned from this programme.

Now, rewind back to the *Eval* state associated with environment $\widehat{ve}_3$; this is, the state $([\![\texttt{(id v2 halt)}]\!], \widehat{ve}_3)$. Diagrammatically, the environment $\widehat{ve}_3$ is represented

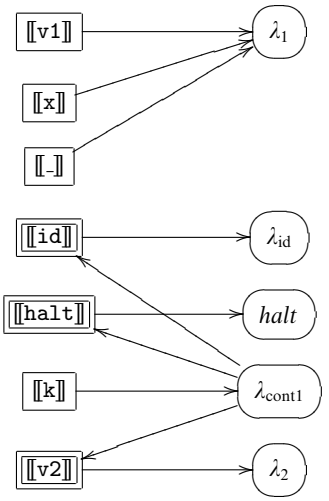


Fig. 8. Environment $\widehat{ve}_3$.

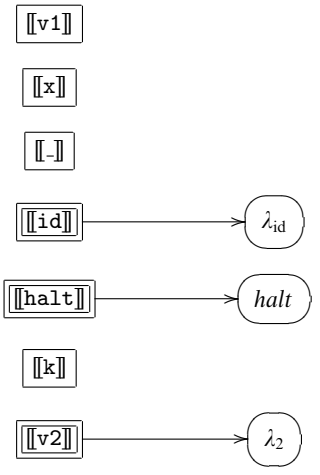


Fig. 9. Environment $\widehat{ve}'_3$.

in Figure 8. Clearly, only the bindings for the variables `id`, `halt` and `v2` are reachable from the root set.

Hence, after garbage collection, this environment is represented in Figure 9. Call this collected environment $\widehat{ve}'_3$. Running the abstract interpretation forward with this collected environment leads to the *Apply* state $(\lambda_{id}, \langle\{\lambda_2\}, \{halt\}\rangle, \widehat{ve}'_3)$. This, in turn, leads to the *Eval* state $([\![\texttt{(k x)}]\!], \widehat{ve}'_4)$, where:

$$\widehat{ve}'_4[\![\texttt{k}]\!] = \{halt\}$$
$$\widehat{ve}'_4[\![\texttt{x}]\!] = \{\lambda_2\}$$

Clearly, the next state is terminal, and no precision is lost in the result of $\{\lambda_2\}$.

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

Fig. 10. Concrete-to-abstract map: $|\cdot|_{\alpha} : \alpha \to \hat{\alpha}$.

8 Soundness of the abstract semantics

In this section, we demonstrate the soundness of the analysis. We have excised portions of the proofs that do not differ from an ordinary proof of correctness for a control-flow analysis. These portions are the same as the ones we have presented in earlier work (Shivers 1991; Might & Shivers 2006a). 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 semantics and the abstract semantics are formally connected by the abstraction operation $|\cdot|$ (Figure 10). Applied to an undefined value, $|\cdot|_D$ returns $\emptyset$. (An undefined value results if we apply an environment *ve* to a value outside its domain.)

The measure-abstraction function, $|\cdot|^{\mu} : VEnv \to \widehat{Measure}$ is

$$|ve|^{\mu} = \lambda \hat{b}.\widehat{size}\{b \in dom(ve) : |b| = \hat{b}\}$$

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

$$\widehat{size}\{x_1, \ldots, x_n\} = \begin{cases} n & n \in \{0, 1\} \\ \infty & \text{otherwise} \end{cases}$$

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

Now we are ready to define the simulation relation, $\mathscr{S} \subseteq \widehat{State} \times State$.

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

Since our abstract semantics can choose to GC or not to GC for any given transition, we have two obligations:

- Showing that the transition $\approx\!\!>_{\Gamma}$ simulates the transition $\Rightarrow_{\Gamma}$.
- Showing that the transition $\approx\!\!>$ simulates the transition $\Rightarrow_{\Gamma}$.

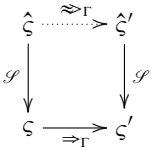
The first theorem on the road to these obligations demonstrates that the abstract collector $\hat{\Gamma}$ is a simulation of the concrete collector Γ:

Theorem 8.2 (*Simulation under collection*)
If $|\varsigma| \sqsubseteq \hat{\varsigma}$, then $|\Gamma(\varsigma)| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})$.

Proof
By Lemmas 8.13, 8.14 and 8.15. □

With this theorem, we can prove the first top-level obligation:

Theorem 8.3 (*Simulation under collecting transition*)
If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow_{\hat{\Gamma}} \varsigma'$, then a state $\hat{\varsigma}'$ exists such that $\hat{\varsigma} \twoheadrightarrow_{\Gamma} \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$. Diagrammatically:[10]



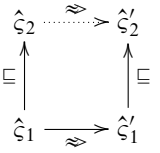
Proof
The proof of this theorem factors into two obligations:

1. If $|\varsigma| \sqsubseteq \hat{\varsigma}$, then $|\Gamma(\varsigma)| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})$.
2. If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then a state $\hat{\varsigma}'$ exists such that $\hat{\varsigma} \twoheadrightarrow \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$.

The first obligation is Theorem 8.2. The second obligation is the standard proof of correctness for a higher-order flow analysis augmented with Lemma 8.9. □

To prove the second top-level obligation, we will first prove two more general theorems. The first theorem shows that abstract transition $\twoheadrightarrow$ is monotonic:

Theorem 8.4 (*Monotonicity of abstract transition*)
If $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$ and $\hat{\varsigma}_1 \twoheadrightarrow \hat{\varsigma}_1'$, then a state $\hat{\varsigma}_2$ exists such that $\hat{\varsigma}_2 \twoheadrightarrow \hat{\varsigma}_2'$ and $\hat{\varsigma}_1' \sqsubseteq \hat{\varsigma}_2'$. Diagrammatically:



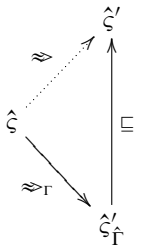
Proof
By cases on the type of the state $\hat{\varsigma}_1$. □

The monotonicity theorem is also what justifies the early termination test: once the analysis encounters a state which is weaker than an previously visited state, the analysis may terminate.

The next theorem states that the GC abstract transition is more precise than the normal abstract transition:

[10] The dotted line here means 'there exists a transition'.

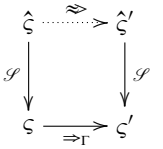
Theorem 8.5
If $\hat{\varsigma} \approx\!\!>_\Gamma \hat{\varsigma}'_{\hat{\Gamma}}$, then a state $\hat{\varsigma}'$ exists such that $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ and $\hat{\varsigma}'_{\hat{\Gamma}} \sqsubseteq \hat{\varsigma}'$. Diagrammatically:



Proof
By the fact that $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$ and Theorem 8.4. □

Putting this all together gives us the second top-level obligation:

Theorem 8.6 (*Simulation under transition*)
If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow_\Gamma \varsigma'$, then a state $\hat{\varsigma}'$ exists such that $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$. Diagrammatically:



Proof
By the previous three theorems. □

With these theorems, ΓCFA is sound to collect as few or as many unreachable bindings as deemed necessary on any given transition.

Given the lack of monotonicity in the configuration across transitions in the abstract semantics, the correctness arguments behind aggressive termination techniques in ΓCFA are more difficult to support. The aggressive cut-off condition in ΓCFA states that, during the state-space search, if the current state $\hat{\varsigma}$'s GC'd version, $\hat{\Gamma}(\hat{\varsigma})$, is more precise than a state already visited, ς^*, i.e. $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \varsigma^*$, then termination is sound. The soundness of this behaviour relies upon showing that the set of concrete states represented by the abstract state $\hat{\varsigma}$ and its collected version $\hat{\Gamma}(\hat{\varsigma})$ are, in fact, equal. The concretization function $Conc_\Gamma : \widehat{State} \to \mathscr{P}(State)$ maps an abstract state to the set of concrete states that it represents *for the garbage-collecting concrete semantics*:

Definition 8.7 (*GC concretization*)
The garbage-collected concretization of the state ς is:

$$Conc_\Gamma(\hat{\varsigma}) = \{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\varsigma}\}$$

That is, only the garbage-collected states are considered.

This means the soundness theorem for the aggressive cut-off reduces to:

Theorem 8.8
$Conc_\Gamma(\hat{\varsigma}) = Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$.

Proof
The theorem reduces to showing $\{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\varsigma}\} = \{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})\}$. We factor this into two obligations:

- First, we show $Conc_\Gamma(\hat{\varsigma}) \subseteq Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$. Choose a state ς from $Conc_\Gamma(\hat{\varsigma})$. We already know that $\Gamma(\varsigma) = \varsigma$. To prove the state ς's membership in $Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$, it suffices to show:

$$\begin{aligned} |\varsigma| &= |\Gamma(\varsigma)| \\ &\sqsubseteq \hat{\Gamma}|\varsigma| \\ &\sqsubseteq \hat{\Gamma}(\hat{\varsigma}) \end{aligned}$$

- Next, we must show $Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma})) \subseteq Conc_\Gamma(\hat{\varsigma})$. This holds by $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$.

□

The Zen of ΓCFA

$$\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$$

is true, while:

$$Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma})) \supseteq Conc_\Gamma(\hat{\varsigma})$$

is also true.

8.1 Supporting lemmas

The first lemma reasons about counters across transitions:

Lemma 8.9
If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$ and $\hat{\varsigma} \twoheadrightarrow \hat{\varsigma}'$ then $|ve_{\varsigma'}|^\mu \sqsubseteq \hat{\mu}_{\hat{\varsigma}'}$.

Proof
Suppose $|\varsigma| \sqsubseteq \hat{\varsigma}$, $\varsigma \Rightarrow \varsigma'$, and $\hat{\varsigma} \twoheadrightarrow \hat{\varsigma}'$. The case where ς is an *Eval* state is trivial, so suppose that ς is an *Apply* state. An abstract $\widehat{Apply}$ state has only one possible subsequent state, i.e. it will not fork. Let $\varsigma = (\ldots, ve, t)$, $\varsigma' = (\ldots, ve', t')$, and $\hat{\varsigma} = (\ldots, \hat{ve}, \hat{\mu}, \hat{t})$. By the apply-state schema, $ve' = ve[(v_i, t) \mapsto d_i]$. Thus:

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

□

The next series of lemmas relates touchable and reachable bindings in both the concrete state-space and the abstract state-space:

Lemma 8.10
If $|\varsigma| \sqsubseteq \hat{\varsigma}$, then $|\mathscr{T}(\varsigma)| \subseteq \widehat{\mathscr{T}}(\hat{\varsigma})$.

Proof
By cases on the structure of the state ς. □

Lemma 8.11
$|\mathscr{R}(\varsigma)| \subseteq \widehat{\mathscr{R}}(|\varsigma|)$.

Proof
Choose an abstract binding $\hat{b} \in |\mathscr{R}(\varsigma)|$. Let b be such that $|b| \sqsubseteq \hat{b}$ and $b \in \mathscr{R}(\varsigma)$. Let $\langle b_0, \ldots, b\rangle$ be a path that justifies $b \in \mathscr{R}(\varsigma)$. We can show by Lemma 8.10 and contradiction that the path $\langle |b_0|, \ldots, |b|\rangle$ must also justify $\hat{b} \in \widehat{\mathscr{R}}(|\varsigma|)$. □

Lemma 8.12
If $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$, then $\widehat{\mathscr{R}}(\hat{\varsigma}_1) \subseteq \widehat{\mathscr{R}}(\hat{\varsigma}_2)$.

Proof
By path-style reasoning similar to Lemma 8.11. □

The next few lemmas support simulation under GC:

Lemma 8.13
If $|\varsigma| \sqsubseteq \hat{\varsigma}$, then $|ve_\varsigma|\mathscr{R}(\varsigma)|^\mu \sqsubseteq \hat{\mu}_{\hat{\varsigma}}|\widehat{\mathscr{R}}(\hat{\varsigma})$.

Proof
Assume $|\varsigma| \sqsubseteq \hat{\varsigma}$. Then, $|ve_\varsigma|\mathscr{R}(\varsigma)|^\mu \sqsubseteq |ve_\varsigma|^\mu||\mathscr{R}(\varsigma)| \sqsubseteq \hat{\mu}_{\hat{\varsigma}}|\widehat{\mathscr{R}}(\hat{\varsigma})$. □

Lemma 8.14
$|ve|\mathscr{R}(\varsigma)| \sqsubseteq |ve|||\mathscr{R}(\varsigma)|$.

Proof
Choose any abstract binding $\hat{b}$.

$$
\begin{aligned}
|ve|\mathscr{R}(\varsigma)|(\hat{b}) &= \bigsqcup_{|b|=\hat{b}} |(ve|\mathscr{R}(\varsigma))(b)| \\
&= \bigsqcup_{\substack{|b|=\hat{b} \\ b\in\mathscr{R}(\varsigma)}} |ve(b)| \\
&\sqsubseteq \bigsqcup_{\substack{|b|=\hat{b} \\ \hat{b}\in|\mathscr{R}(\varsigma)|}} |ve(b)| \\
&= \textbf{if } \hat{b} \in |\mathscr{R}(\varsigma)| \textbf{ then } \bigsqcup_{|b|=\hat{b}} |ve(b)| \textbf{ else } \bot \\
&= (|ve|||\mathscr{R}(\varsigma)|)(\hat{b})
\end{aligned}
$$

□

Lemma 8.15
If $ve_1 \sqsubseteq ve_2$ and $\widehat{B}_1 \subseteq \widehat{B}_2$, then $\widehat{ve}_1|\widehat{B}_1 \sqsubseteq \widehat{ve}_2|\widehat{B}_2$.

Proof
By reasoning similar to Lemma 8.14. □

9 Applications

Now that we have a flow analysis instrumented with counting machinery, we turn to its application: environment analysis. (Control-flow analysis for which abstract garbage collection enhances precision offers a number of applications, including but certainly not limited to constant propagation, useless-variable elimination and induction-variable elimination (Shivers 1991).) Environment analysis drives globalization, lightweight closure conversion, super-β lambda propagation, super-β copy propagation and continuation promotion (Shivers 1991; Wand & Steckler 1944; Might & Shivers 2006a; Shivers & Might 2006).

Abstract counting can be brought to bear on environment analysis by the following theorem, which links the equality of bindings in the abstract space to the equality of their counterparts in the concrete space:

Theorem 9.1 (*Pinching*)
If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\hat{\mu}_{\hat{\varsigma}}(\hat{b}) = 1$, then for any two bindings $b_1, b_2 \in dom(ve_\varsigma)$ such that $|b_1| = \hat{b}$ and $|b_2| = \hat{b}$, $b_1 = b_2$.

Proof
Assume $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\hat{\mu}_{\hat{\varsigma}}(\hat{b}) = 1$. Choose any two bindings $b_1, b_2 \in dom(ve_\varsigma)$ such that $|b_1| = \hat{b}$ and $|b_2| = \hat{b}$. From $\hat{\mu}_{\hat{\varsigma}} \sqsupseteq |ve_\varsigma|^\mu$, we have:

$$\begin{aligned} 1 &= \hat{\mu}_{\hat{\varsigma}}(\hat{b}) \\ &\geqslant |ve_\varsigma|^\mu(\hat{b}) \\ &= \widehat{size}\{b \in dom(ve_\varsigma) : |b| = \hat{b}\} \\ &\geqslant \widehat{size}\left(\{b_1\} \cup \{b_2\}\right) \end{aligned}$$

which implies that the size of the set $\{b_1\} \cup \{b_2\}$ is 0 or 1. If the size is 0, then we cannot choose any such bindings, and the theorem holds vacuously. If the size is 1, then $b_1 = b_2$. □

Our next theorem relates reachable environments directly to one another:

Theorem 9.2 (*Environment*)
Given a sound state ς and a simulation $\hat{\varsigma}$ of it, if environments β_1 and β_2 are reachable in ς, and $|\beta_1|(v) = \hat{t} = |\beta_2|(v)$ and $\hat{\mu}_{\hat{\varsigma}}(v, \hat{t}) = 1$, then $\beta_1(v) = \beta_2(v)$.

Proof
Let β_1 and β_2 be reachable environments in ς. (By *reachable*, we mean that there exist bindings $b_1, b_2 \in \mathscr{R}(\varsigma)$ such that $(lam_1, \beta_1) = ve_\varsigma(b_1)$ and $(lam_2, \beta_2) = ve_\varsigma(b_2)$). Assume that $|\beta_1|(v) = \hat{t} = |\beta_2|(v)$ and $\hat{\mu}_{\hat{\varsigma}}(v, \hat{t}) = 1$. Because these environments are reachable, the bindings $(v, \beta_1(v))$ and $(v, \beta_2(v))$ must be reachable as well. Hence, these bindings are in the domain of the value environment ve. Thus, by the pinching theorem, $\beta_1(v) = \beta_2(v)$. □

We have recently shown (Shivers & Might 2006) how these analyses, applied to CPS representations, permit compilers to fuse together graphs of online transducers. We hope to apply this technology to programmes such as DSP systems, network

protocol stacks and graphics pipelines. The analyses we have presented in this paper were critical to the transducer-fusing transforms that we have demonstrated in that setting.

All of this leads to a super-β-inlining theorem:

Theorem 9.3
It is safe to inline the term lam' in place of the term f' in the programme pr if for each state $(\llbracket(f\ e_1 \cdots e_n)\rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t})$ in $\hat{\mathscr{V}}(pr)$ such that $f = f'$:

- $\hat{\mathscr{A}}(f, \hat{\beta}, \widehat{ve}) = \{(lam', \hat{\beta}')\}$,
- and for each $v \in free(lam')$:
  1. $\hat{\beta}(v) = \hat{\beta}'(v)$,
  2. $v \in free\llbracket(f\ e_1 \cdots e_n)\rrbracket$, and
  3. $\hat{\mu}(v, \hat{\beta}(v)) = 1 = \hat{\mu}(v, \hat{\beta}'(v))$.

9.1 Globalization

Globalization, as defined by Sestoft (1988), is the conversion of function parameters to global variables when the bindings to these parameters are known to have non-self-interfering lifetimes. Environmentally, globalization is effectively asking: would a globally scoped environment be equivalent to the lexically scoped environment for the variables in question?

Example: Globalization In the following function:

```
(define (f g i arr len)
 (if (< i len)
     (begin (g (array-ref arr i))
            (f g (+ i 1) arr len))))
```

if the procedure `g` never invokes the array-walking function `f` either directly or indirectly, then it is safe to transform this code into:

```
(define (f)
 (if (< i len)
     (begin (g (array-ref arr i))
            (set! i (+ i 1))
            (f))))
```

and invocations of the form (`f` *g i a l*) into:

```
(begin
 (set! g   g)
 (set! i   i)
 (set! arr a)
 (set! len l)
 (f))
```

Detecting when this is legal is straightforward for ΓCFA. A variable v is globalizable (in program pr) if there is never more than one simultaneously live binding to the variable in any state:

$$Globalizable(v, pr) \text{ iff } 1 \geqslant \bigsqcup_{\hat{\varsigma} \in \hat{\mathscr{V}}(pr)} \bigoplus_{\hat{t}} \hat{\mu}_{\hat{\varsigma}}(v, \hat{t})$$

Globalization via abstract counting offers the additional benefit over Sestoft's approach in that re-bindings of a variable to itself need not be counted as self-interfering bindings to the same variable.

10 Extensions

We can add primops and conditionals to the analysis in the standard way, which is described in Shivers' work (1991).

10.1 Explicit recursion

While the Y combinator is adequate for performing recursion, it is not difficult to modify the semantics and the analysis to handle an explicit construct for recursion similar to Scheme's `letrec`. Assuming appropriate modifications to the syntax, we can handle `letrec` with an $\widehat{Eval} \to \widehat{Eval}$ transition:

$$(\llbracket \texttt{(letrec ((}v_1\ lam_1\texttt{)) } \cdots \texttt{ (}v_n\ lam_n\texttt{)) } call\texttt{)} \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \approx\!\!> (call, \hat{\beta}', \widehat{ve}', \hat{\mu}', \hat{t}')$$

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

10.2 Stores, must-alias analysis and strong updates

Join is the enemy.
—Tom Reps, speaking at VMCAI 2007

We can add an abstract store to the analysis, and apply both abstract counting and abstract GC to it. With respect to abstract GC and abstract counting, the store behaves just like the value environment $\widehat{ve}$. That is, an abstract address $\hat{a}$ plays the role of an abstract binding.

The store itself is merely an extra component within the state, $\hat{\sigma} : \widehat{Addr} \to \hat{D}$, accessed by primops. The extended counter $\hat{\mu} : (\widehat{Addr} + \widehat{Bind}) \to \hat{\mathbb{N}}$ must then map abstract store addresses into the set $\hat{\mathbb{N}}$ as well. This implies that the reachability function $\widehat{\mathscr{R}}$'s range may contain both bindings and addresses. It also means that the adjacency relation $\rightsquigarrow$ must be parameterized by both the value environment *and* the store.

Consider what the direct analog to the pinching theorem would be for this store:

Theorem 10.1 (*Must alias*)
If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\hat{\mu}_{\hat{\varsigma}}(\hat{a}) = 1$, then for any two $a_1, a_2 \in dom(\sigma_\varsigma)$ such that $|a_1| = \hat{a}$ and $|a_2| = \hat{a}$, $a_1 = a_2$.

Ordinarily, if two abstract addresses $\hat{a}_1$ and $\hat{a}_2$ are equal, the most we can say is that their concrete counterparts *may* alias. (If instead $\hat{a}_1 \neq \hat{a}_2$, then we can infer that they *must not* alias.) If, however, $\hat{a}_1 = \hat{a}_2$ *and* $\hat{\mu}(\hat{a}_1) = 1$, then their concrete counterparts *must alias*.

With this theorem, it becomes possible to justify a sound *strong update* to the abstract store. A *weak update* merges values, costing us precision:

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \cdots]$$

In the absence of additional information about the address $\hat{a}$, the only sound behaviour is a weak update. In contrast, a strong update *overwrites* the value in the abstract store:

$$\hat{\sigma}' = \hat{\sigma}[\hat{a} \mapsto \cdots]$$

Clearly, additional information is required to make this sound.

In ΓCFA, if the abstract count of an address is one, and the address is being mutated, then a strong update is sound. For instance, suppose that at the call site:

```
(set-cell! loc val k)
```

the following conditions hold:

$$\hat{a} = \widehat{ve}(loc, \hat{\beta}\, loc)$$
$$\hat{\mu}(\hat{a}) = 1$$

Then, instead of performing:

$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{d}]$$

ΓCFA can perform:

$$\hat{\sigma}' = \hat{\sigma}[\hat{a} \mapsto \hat{d}]$$

By identical reasoning, if we add a Scheme-style `set!` to CPS, then it is also possible to perform a strong update on a value environment $\widehat{ve}$ when the abstract count of the binding in question is one. For example, suppose that at the call site:

```
(set! x val k)
```

the condition $\hat{\mu}(x, \hat{\beta}x) = 1$ holds. Then, we may soundly perform a strong update:

$$\widehat{ve}' = \widehat{ve}[(x, \hat{\beta}x) \mapsto \hat{d}]$$

Strong updates are particularly useful for preserving the precision of a flow analysis when variables are initialized to a null value, and then, assigned shortly thereafter.

11 Implementation and evaluation

Using Haskell, we have implemented ΓCFA for a subset of R5RS Scheme. This section briefly describes the implementation and presents results from running it in practice. The implementation uses the Larceny front end to perform macro

expansion, and it currently supports primitive operations, `set!`, `letrec`, a side-effecting store, vectors and `call/cc`. Both abstract GC and abstract counting operate on the store as well as the value environment.

We caution that the purely functional nature of the implementation means that timing numbers may be inflated by the $O(\log n)$ penalty imposed by such programming. Thus, while there is some room to improve the absolute timing numbers for both the control and experimental groups, the *ratio* of control to experimental timings provide a basis for comparing performance.

The implementation permits user-defined values of k for k-CFA context sensitivity: the past k call sites form the current contour/abstract time. For comparison purposes, only $k = 0$ was tested.

11.1 $\widehat{Conf}$*-widening*

In the pure-CPS ΓCFA, each abstract state's *machine configuration*, hereafter known as a member of the domain $\widehat{Conf}$, is a pair: the binding-to-value environment $\widehat{ve}$ and the counter $\hat{\mu}$. In the implementation, this machine configuration includes a store component $\hat{\sigma}$ as well. The two criteria for admitting a component into the machine configuration are:

1. Both *Eval* and *Apply* states have it.
2. It is susceptible to abstract GC and abstract counting.

(In ΔCFA (Might & Shivers 2006a), the log $\hat{\delta}$ qualifies as part of the configuration.)

Unlike an ordinary abstract interpretation's transfer function, such as the relation $\approx\!\!>$, the GC transition relation $\approx\!\!>_\Gamma$ does not increase the configuration monotonically. While the non-monotonicity aids precision, it can also increase the path sensitivity of ΓCFA to levels not required for flow or environment analysis: that is, the added sensitivity simply does not improve the result. Sometimes, this increase in path sensitivity comes at the cost of increased running time.

Widening the configuration mid-analysis discards path sensitivity while retaining increased precision and lowering analysis run time. *Widening*, in this case, refers to deliberately joining the current configuration with another configuration before making a transition. For other applications of ΓCFA, such as model checking (Might *et al.*2007), the increased path sensitivity is useful for verifying temporal correctness properties.

To $\widehat{Conf}$-widen during the state-space search, if the current state is $(call, \hat{\beta}, \widehat{c}, \hat{t})$ and this state is not covered by the visited set, then before making the transition, compute the widening configuration $\widehat{c}'$ (according to criteria below), and replace the current state with $(call, \hat{\beta}, \widehat{c} \sqcup \widehat{c}', \hat{t})$.

Our implementation supports three levels of $\widehat{Conf}$-widening:

- *Per-point*. With a per-point configuration, all states at the same *call* site share the same configuration. That is, the algorithm employs a global, side-effected table $\tau : CALL \to \widehat{Conf}$. If $\hat{\varsigma}_{\text{curr}}$ is the current state, the widening algorithm

is:

$$
\begin{aligned}
(call, \hat{\beta}, \widehat{c}_{\text{curr}}, \hat{t}) &\leftarrow \hat{\varsigma}_{\text{curr}} \\
\widehat{c}_{\text{point}} &\leftarrow \tau[call] \\
\widehat{c}_{\text{new}} &\leftarrow \widehat{c}_{\text{point}} \sqcup \widehat{c}_{\text{curr}} \\
\tau[call] &\leftarrow \widehat{c}_{\text{new}} \\
\hat{\varsigma}_{\text{curr}} &\leftarrow (call, \hat{\beta}, \widehat{c}_{\text{new}}, \hat{t})
\end{aligned}
$$

- *Per-context*. With a per-context configuration, all states with the same $(call, \hat{\beta}, \hat{t})$ triple share a heap. (In a 0CFA setting, per-context and per-point become the same.)
- *Per-programme*. With a per-programme configuration, all states share the same widening configuration.

11.2 Garbage collection frequency and granularity

The implementation supports two policies for garbage collection:

- *Eager*. Each state is garbage collected every time.
- *Never*. No state is ever garbage collected.

Other strategies are permissible as well such as garbage collecting only when zombie creation is imminent. In our experience, however, the time penalty of garbage collection as reported by profiling was negligible compared with the cost of termination checking; hence, we opted for a policy of garbage collection on every transition.

11.3 Measurements

Table 1 provides measurements on a suite of benchmarks. The machine used for evaluation is a 2 GHz Intel Core Duo with 2 GB RAM running Mac OS X.

Each benchmark was measured both with and without garbage collection enabled. The $\#\lambda$'s column indicates the number of functions in the *pre*-CPS-converted code. (CPS conversion inflates the number of functions.)

We took three metrics for each run:

- *Single*: The percentage of variables (that is, 0CFA-level bindings) marked as never exceeding a count of one. This metric approximates the improvement in precision and the power of ΓCFA as an environment analysis.[11]

[11] CPS conversion, A-Normalization and other argument-flattening transformations do not necessarily inflate or deflate the singleness measure since the fresh variables introduced may be live across a recursive call, causing deflation, or not live, causing inflation. In general, if a transformation can minimize the number of variables live across a recursive call, e.g. by moving local computations after it, ΓCFA's singleness precision will improve. The CPS transformation used in our measurements made no attempt to perform such optimizations.

Benchmark	#λ's	Without GC			With GC		
		Single	States	Time	Single	States	Time
`put-double`	39	16%	4746	3s	90%	1723	1s
`integrate-fringe`	49	22%	12549	12s	96%	4012	4s
`integrate-stream`	45	10%	18556	24s	82%	8067	11s
`perm`	35	6%	127656	145s	95%	18166	9s
`lattice*`	36	10%	41413	94s	91%	8533	10s
`earley*`	90	–	–	>30min	94%	43034	138s
`sboyer†`	44	–	–	>30min	99%	14846	120s
`nboyer†`	43	–	–	>30min	99%	23108	144s

Table 1. *Metrics for ΓCFA, k-level 0. A star * denotes per-context configuration-widening. A dagger † denotes per-programme configuration widening.*

- *States*: The number of states traversed before termination. The reduction in states approximates the improvement in false-positive-branch reduction.
- *Time*: The time until termination.

The high single percentages for the GC-enabled runs are explained by the fact that most variables have short, non-self-interfering lifetimes. The low single percentages for the non-GC-enabled runs are explained by the fact that, without GC, only variables bound once for the entire run of the program, e.g. globals, are marked single.

Analyses taking longer than 30 min were aborted. However, on a quad-Xeon machine with 16 GB RAM, the `nboyer` benchmark was re-run with per-context heap widening *and* GC turned on: after 6 h and 689,897 states visited, ΓCFA terminated. This indicates that per-programme configuration widening may be critical in scaling ΓCFA to larger programs.

12 Related work

The work we have 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 (1991). 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 (1995) and Wright and Jagannathan's polymorphic splitting (1998). That is, the ΓCFA framework should be able to take nearly any contour-selection strategy and make it more precise.

Shivers (1991) 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 that we have presented here could be considered as a sort of 'opportunistic reflow analysis'. Our work is further differentiated by a proof of correctness. (We suspect that 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 (1986), continued by Chase *et al.*'s strong update (1990) and generalized by Jagannathan (1998). Our abstract counter $\hat{\mu}$ and reachability function $\widehat{\mathscr{R}}$ 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.
- Our reachability function 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 (Might & Shivers 2006a; Might & Shivers 2007), 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 (1994). 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-β inlining, to motivate the need for their environment analysis. Hannan (1995) 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 is based on the body of work that develops the CPS-as-intermediate-representation thesis. The foundational work here is by Steele (1978). Shivers' earlier work (1991) in CPS-based analysis has provided the basic framework for the techniques we have presented in this article. 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 do not 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.

Acknowledgments

Suresh Jagannathan very kindly pointed out some important related work that helped the development of our ideas. We thank Ben Chambers, Daniel Harvey and our anonymous reviewers for the conference version of this paper (Might & Shivers 2006b), whose detailed feedback made for a much better paper. We also thank Julia Lawall and our anonymous reviewers for their extensive and insightful feedback on this version of the paper. This work was funded by the NSF's Science of Design programme; we are grateful for their support.

References

Agesen, O. (1995) The cartesian product algorithm: simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP 1995*, pp. 2–26.

Chase, D. R., Wegman, M., & Zadeck, F. K. 1990 (June) Analysis of pointers and structures. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 296–310.

Cousot, P. & Cousot, R. 1977 (January) Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, vol. 4, pp. 238–252.

Cousot, P. & Cousot, R. 1979 (January) Systematic design of program analysis frameworks. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, vol. 6, pp. 269–282.

Hannan, J. (1995) Type systems for closure conversion. In *Proceedings of Workshop on Types for Program Analysis*, pp. 48–62.

Hudak, P. 1986 (August) A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pp. 351–363.

Jagannathan, S., Thiemann, P., Weeks, S., & Wright, A. K. 1998 (January) Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pp. 329–341.

Might, M. & Shivers, O. 2006a (January) Environment analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006)*, pp. 127–140.

Might, M. & Shivers, O. 2006b (September) Improving flow analyses via ΓCFA: abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)*, pp. 13–25.

Might, M. & Shivers, O. (2007) Analyzing environment structure of higher-order languages using frame strings. *Theoretical Computer Science*, **375**(1–3), 137–168.

Might, M., Chambers, B., & Shivers, O. 2007 (January) Model checking via ΓCFA. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2007)*, pp. 59–73.

Palsberg, J. (1995) Closure analysis in constraint form. *ACM Trans. Programming Languages Systems*, **17**(1), 47–62.

Sestoft, P. 1988 (October) *Replacing Function Parameters by Global Variables.* M.Phil. thesis, Copenhagen, Denmark: DIKU, University of Copenhagen.

Shivers, O. 1988 (June) Control-flow analysis in scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (pldi)*, pp. 164–174.

Shivers, O. 1991 (May) *Control-Flow Analysis of Higher-Order Languages.* Ph.D. thesis, Pittsburgh, PA: School of Computer Science, Carnegie-Mellon University. Technical Report CMU-CS-91-145.

Shivers, O. & Might, M. 2006 (June) Continuations and transducer composition. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 295–307.

Steele, Jr., & Guy L. 1978 (May) *RABBIT: A Compiler for SCHEME.* M.Phil. thesis, Cambridge, MA: Artificial Intelligence Laboratory, Massachusetts Institute of Technology. Technical report AI-TR-474.

Wand, M. & Steckler, P. 1994 (January) Selective and lightweight closure conversion. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, vol. 21, pp. 435–445.

Wright, A. K. & Jagannathan, S. (1998) Polymorphic splitting: an effective polyvariant flow analysis. *ACM Trans. Programming Languages Systems*, **20**(1), 166–207.