Model Checking via ΓCFA

Matthew Might, Benjamin Chambers, and Olin Shivers

{mattm,bjchamb}@cc.gatech.edu shivers@ccs.neu.edu

Abstract. We present and discuss techniques for performing and improving the model-checking of higher-order, functional programs based upon abstract interpretation [4]. We use continuation-passing-style conversion to produce an abstractable state machine, and then utilize abstract garbage collection and abstract counting [9] to indirectly prune false branches in the abstract state-to-state transition graph. In the process, we generalize abstract garbage collection to *conditional garbage collection*; that is, we collect values which an ordinary reaching-based collector would have deemed live when it is provable that such values will never be referenced. In addition, we enhance *abstract counting*, and then exploit it to more precisely evaluate conditions in the abstract.

Key words: Abstract interpretation, static analysis, abstract counting, abstract garbage collection, ΓCFA, higher-order languages

1 Introduction

We are interested in analysing and verifying the behavior of programs written in call-by-value, higher-order programming languages based on the λ-calculus, such as Scheme or Standard ML. (However, techniques developed for this class of languages can be profitably adapted for other higher-order languages, such as Haskell or Java.) Our goal is to describe the construction of a model checker for higher-order programs in such a way that it is eligible to achieve precision enhancements by garbage collecting "dead" environment structure in the abstract state space traversed by the program.

We decompose building a garbage-collecting model checker for a higher-order language into four steps:

1. Convert the language's semantics into state-to-state rules of the form $\varsigma \Rightarrow \varsigma'$.
2. Axiomatize the rules by modelling control explicitly, *i.e.*, with continuations.
3. Instrument the resulting state machine with garbage collection.
4. Construct an abstract interpretation of this machine's transition relation.

The abstract state-to-state transition that results induces a finite, directed graph between abstract states, which sets the stage for model checking. However, the abstraction that makes the state-space finite and hence checkable, can obscure the property we seek, and so render the entire analysis useless. Folding states in the concrete state space together introduces spurious paths; if these spurious paths admit the possibility of "bad" behavior, then our computable abstract analysis will erroneously conclude that a correct program might give rise to incorrect behavior. The program succeeds, but the analysis has failed.

We address this problem with Step (3) above: garbage-collecting elements of a machine state (such as its environment structure and bound values) permits the abstract interpretation to prune false branches from the state space's transition graph. To get a feel for the reduction in the state space, consider the following doubly nested loop, written in a direct-style Scheme:

```
(letrec ((lp1 (λ (i x)
                (if (= 0 i)  x
                    (letrec ((lp2 (λ (j f y) (if (= 0 j)
                                                 (lp1 (- i 1) y)
                                                 (lp2 (- j 1) f
                                                      (f y))))))
                      (lp2 10 (λ (n) (+ n i)) x))))))
  (lp1 10 0))
```

Figure 1 shows the flow-sensitive, context-sensitive abstract transition graphs generated by this loop first without, and then with, abstract garbage collection. Garbage-collecting environment structure during the exploration of the abstract state space yields an order of magnitude improvement in the size of the state space—enough so that the doubly-nested structure of the loop is *visually* apparent from the second graph. (Besides the improvement in analytic precision, we also get a secondary benefit in that the processor time and memory space needed to explore the abstract state space are also greatly reduced.)

Abstract garbage collection sets the stage for another technique known as *abstract counting* [9]. With abstract counting, we track the "cardinality" of an abstract object; that is, we track whether an abstract object currently represents zero, one or more than one concrete values. Suppose we were to use sets of concrete values for our abstract values. Ordinarily, if abstract value A were equal to abstract value B, we could not infer that any concrete value $a \in A$ is equal to any concrete value $b \in B$, *except* for the case where A and B have size one. The ability to transfer abstract equality to concrete equality allows us to more precisely evaluate conditions, *e.g.* `(= x y)`, in the abstract.

In previous work [9], we developed a higher-order flow-analysis framework, ΓCFA, which synergistically combines abstract counting and abstract garbage collection as we've just outlined above. The benefit of combining the two is that we can use abstract counts to reason more precisely about reachable values during abstract garbage collection. This, in turn, increases the chance that we can cut off even more branches from the abstract transition graph.

Our purpose in this paper is to show how ΓCFA technology can be applied to the problem of model-checking software written in higher-order languages. Our technical contributions are:

1. Enhancing abstract garbage collection by switching from *reachability* to *usability* as the criterion for liveness. That is, our garbage collector discards abstract values and environment structure which are "reachable," but whose use is dominated by conditions which have become unsatisfiable. We term this *conditional garbage collection.*

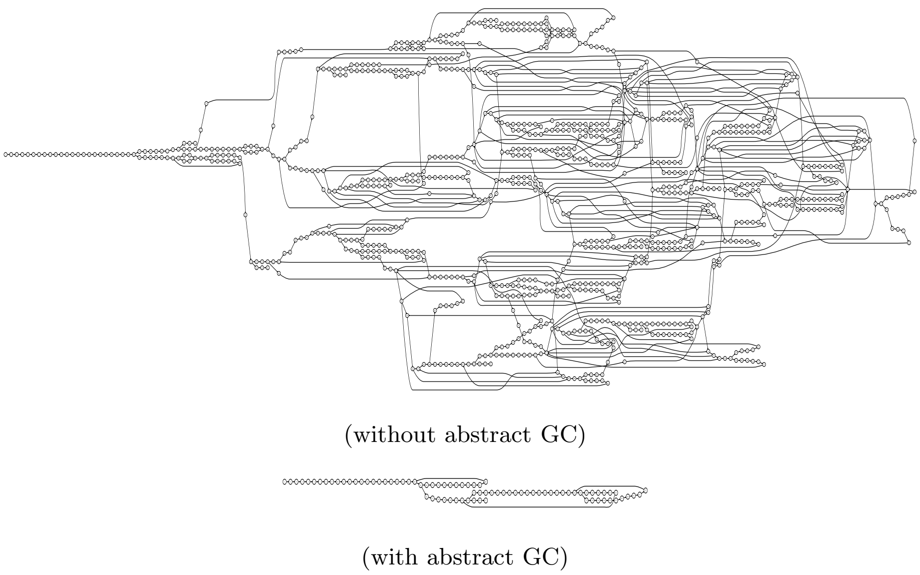


Fig. 1. These images are abstract state-to-state transition graphs generated from the same doubly nested loop. Construction of the top graph did not utilize abstract garbage collection. The bottom graph is the result of garbage collecting at each step.

2. Using abstract counting to more precisely evaluate conditionals during abstract garbage collection. We also improve the precision of abstract counting by accounting for objects that remain invariant across transitions.

2 CPS

Our first task in preparing a program for model checking is to put it into a continuation-passing style (CPS) representation [1, 6, 12]. In CPS, function calls do not return; they are one-way control transfers. Further, all control structures (call, return, loops, exceptions, and so forth) are encoded using this restricted mechanism. Among other benefits, CPS reifies implicit control context, thus rendering it into a form that can be handled by the abstract garbage-collection machinery we'll be using.

The grammar for our particular CPS representation is given in Figure 2. Note that our language has some syntactic structure more reminiscent of A-Normal Form (ANF) [10] than minimal CPS: it includes an explicit `if` conditional form, instead of encoding conditionals as primitive procedures that take multiple continuation arguments, and we also have a `let` form for binding variables to the results of "trivial" expressions, which can be trees of primop applications whose leaves are variables, constants and λ terms. We also provide a `letrec` form for defining mutually-recursive functions, and a `halt` form that terminates the com-

$$
\begin{aligned}
const \in CONST &= \mathbb{Z} + \{\texttt{\#f}\} \\
prim \in PRIM &= \{\texttt{+}, \texttt{*}, \texttt{equal?}, \texttt{<}, \ldots\} \\[1ex]
v \in VAR &::= \text{a set of identifiers} \\[1ex]
e, f \in EXP &::= v \mid const \\
&\mid \ (\lambda\ (v_1 \cdots v_n)\ call) \\
&\mid \ (prim\ v_1 \cdots v_n) \\[1ex]
call \in CALL &::= (f\ e_1 \cdots e_n) \\
&\mid \ (\texttt{if}\ e_c\ e_t\ e_f) \\
&\mid \ (\texttt{let}\ ((v\ e))\ call) \\
&\mid \ (\texttt{letrec}\ ((v\ lam)^*)\ call) \\
&\mid \ (\texttt{halt}\ e)
\end{aligned}
$$

Fig. 2. A grammar for restricted CPS. Programs are alphatised terms with no free variables, *i.e.*, any two binding variables are distinct.

putation, providing its final result. Note the signature syntactic distinction of a CPS representation: the arguments e_i to a function call $(f\ e_1 \cdots e_n)$ cannot themselves be function calls—that would require function calls to return a value, which CPS does not permit.

3 Generating the abstract state graph with ΓCFA

Our objective in this section is to create a computable, *finite* abstract transition relation—that is, a small-step operational semantics for our CPS language whose set of possible machine states is finite. (We skip over the development of the corresponding concrete semantics. It is completely standard, and can, in any event, be inferred from the abstract semantics.) Figure 3 gives the state-space for ΓCFA.

The set $\widehat{State}$ is the set of possible abstract states—the nodes in the forthcoming abstract transition graph. We distinguish two kinds of states: $\widehat{Eval}$ states and $\widehat{Apply}$ states. In an $\widehat{Eval}$ state, execution has reached a call site, *e.g.* $(f\ e_1 \cdots e_n)$, where the function f and its arguments e_i need *evaluation*. In an $\widehat{Apply}$ state, execution has reached the *application* of a procedure to a vector of argument values.

In $\widehat{Eval}$ states, arguments are evaluated under the current environment, which is decomposed into a "local" variable-to-binding portion ($\widehat{BEnv}$) and a "global" binding-to-value portion ($\widehat{VEnv}$) [11]. Given a factored environment $(\widehat{\beta}, \widehat{ve})$, a variable maps to a value in two stages: (1) the time of its binding in the current environment $\widehat{\beta}$ is found: $\widehat{\beta}(v)$; and (2) the value attached to the variable at this time is looked up: $\widehat{ve}(v, \widehat{\beta}(v))$. Consequently, the binding $(v, \widehat{t})$ acts as a reference to this value. (When using a binding in this referential sense, we refer to it as

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

Fig. 3. The abstract state-space.

a member of $\widehat{Ref}$ to emphasize the distinction.) We also sometimes refer to the variable environment $\widehat{ve}$ as the *abstract heap*.

The set of abstract denotable values ($\widehat{D}$) is the power set of basic values ($\widehat{Bas}$) and procedures ($\widehat{Proc}$). The finite set $\widehat{Time}$ takes the place of Shivers' contour set [11]; consequently, the context-sensitivity of the analysis depends on the choice of the set $\widehat{Time}$ and the next-time function, $\widehat{succ} : \widehat{State} \times \widehat{Time} \rightarrow \widehat{Time}$.

Up to now, our semantic domains have been completely standard for a higher-order control-flow analysis; we now introduce the extra machinery that gives our ΓCFA abstract semantics the ability to engage in abstract garbage collection and counting. Every state features a counter map $\widehat{\mu}$. For a binding $\widehat{b}$ and counter $\widehat{\mu}$, the count $\widehat{\mu}(\widehat{b})$ approximates how many *concrete* bindings the abstract binding $\widehat{b}$ represents. The set of approximate counts is $\{0, 1, \infty\}$, where the symbol ∞ denotes any number greater than one, and the operator $\oplus$ is the natural abstraction of addition.

The argument-evaluation function $\widehat{\mathcal{A}} : EXP \times \widehat{BEnv} \times \widehat{VEnv} \rightarrow \widehat{D}$ is:

$$
\begin{aligned}
\widehat{\mathcal{A}}(const, \widehat{\beta}, \widehat{ve}) &= \{const\} \\
\widehat{\mathcal{A}}(v, \widehat{\beta}, \widehat{ve}) &= \widehat{ve}(v, \widehat{\beta}(v)) \\
\widehat{\mathcal{A}}(lam, \widehat{\beta}, \widehat{ve}) &= \{(lam, \widehat{\beta})\} \\
\widehat{\mathcal{A}}([\![\texttt{(}prim\ v_1 \cdots v_n\texttt{)}]\!], \widehat{\beta}, \widehat{ve}) &= \widehat{\mathcal{O}}(prim)\langle \widehat{\mathcal{A}}(v_1, \widehat{\beta}, \widehat{ve}), \ldots, \widehat{\mathcal{A}}(v_n, \widehat{\beta}, \widehat{ve})\rangle
\end{aligned}
$$

where the function $\widehat{\mathcal{O}} : PRIM \rightarrow (\widehat{D}^* \rightarrow \widehat{D})$ maps a primitive to a sound abstraction.

Figure 4 defines the transition $\hat{\varsigma} \approx\!> \hat{\varsigma}'$. The first transition rule (arg. eval.) looks up the procedure for the expression f, evaluates the arguments $e_1, \ldots, e_n$ and moves forward. The next rule (conditional) makes a best-effort attempt to avoid forking on conditional evaluation. The subsequent rule (let-binding)

$$(\llbracket\texttt{(}f\ e_1 \cdots e_n\texttt{)}\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{\boldsymbol{r}}, \widehat{ve}, \widehat{\mu}, \widehat{succ}(\widehat{\varsigma}, \widehat{t}))$$
$$\text{where} \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}(f, \widehat{\beta}, \widehat{ve}) \\ \widehat{d}_i = \widehat{\mathcal{A}}(e_i, \widehat{\beta}, \widehat{ve}) \\ \widehat{r}_i = \textbf{if } e_i \in \mathit{VAR} \textbf{ then } (e_i, \widehat{t}) \textbf{ else } \bot \end{cases} \quad \text{(arg. eval.)}$$

$$(\llbracket\texttt{(if } e_c\ e_t\ e_f\texttt{)}\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (\widehat{proc}, \langle\rangle, \widehat{ve}, \widehat{\mu}, \widehat{succ}(\widehat{\varsigma}, \widehat{t}))$$
$$\text{where } \widehat{proc} \in \begin{cases} \widehat{\mathcal{A}}(e_t, \widehat{\beta}, \widehat{ve}) & \texttt{\#f} \notin \widehat{\mathcal{A}}(e_c, \widehat{\beta}, \widehat{ve}) \\ \widehat{\mathcal{A}}(e_f, \widehat{\beta}, \widehat{ve}) & \{\texttt{\#f}\} = \widehat{\mathcal{A}}(e_c, \widehat{\beta}, \widehat{ve}) \\ \widehat{\mathcal{A}}(e_t, \widehat{\beta}, \widehat{ve}) \sqcup \widehat{\mathcal{A}}(e_f, \widehat{\beta}, \widehat{ve}) & \text{otherwise} \end{cases} \quad \text{(conditional)}$$

$$(\llbracket\texttt{(let ((}v\ e\texttt{))}\ call\texttt{)}\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (call, \widehat{\beta}[v \mapsto \widehat{t}], \widehat{ve}', \widehat{\mu}', \widehat{succ}(\widehat{\varsigma}, \widehat{t}))$$
$$\text{where} \begin{cases} \widehat{ve}' = \widehat{ve} \sqcup [(v, \widehat{t}) \mapsto \widehat{\mathcal{A}}(e, \widehat{\beta}, \widehat{ve})] \\ \widehat{\mu}' = \widehat{\mu} \oplus (\lambda\_.0)[(v, \widehat{t}) \mapsto 1] \end{cases} \quad \text{(let-binding)}$$

$$(\llbracket\texttt{(letrec ((}v\ e\texttt{)}^*\texttt{)}\ call\texttt{)}\rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (call, \widehat{\beta}', \widehat{ve}', \widehat{\mu}', \widehat{t}')$$
$$\text{where} \begin{cases} \widehat{t}' = \widehat{succ}(\widehat{\varsigma}, \widehat{t}) \\ \widehat{\beta}' = \widehat{\beta}[v_i \mapsto \widehat{t}'] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \widehat{t}') \mapsto \widehat{\mathcal{A}}(lam_i, \widehat{\beta}', \widehat{ve})] \\ \widehat{\mu}' = \widehat{\mu} \oplus (\lambda\_.0)[(v_i, \widehat{t}') \mapsto 1] \end{cases} \quad \text{(letrec-binding)}$$

$$((\llbracket\texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ call\texttt{)}\rrbracket, \widehat{\beta}), \widehat{\boldsymbol{d}}, \widehat{\boldsymbol{r}}, \widehat{ve}, \widehat{\mu}, \widehat{t}) \approx\!> (call, \widehat{\beta}', \widehat{ve}', \widehat{\mu}', \widehat{succ}(\widehat{\varsigma}, \widehat{t}))$$
$$\text{where} \begin{cases} \widehat{\beta}' = \widehat{\beta}[v_i \mapsto \widehat{t}] \\ \widehat{ve}' = \widehat{ve} \sqcup [(v_i, \widehat{t}) \mapsto \widehat{d}_i] \\ \widehat{\mu}' = \widehat{\mu}[(v_i, \widehat{t}) \mapsto \widehat{\mu}(v_i, \widehat{t}) \oplus \textbf{if } (v_i, \widehat{t}) = \widehat{r}_i \textbf{ then } 0 \textbf{ else } 1]. \end{cases} \quad \text{(proc. app.)}$$

Fig. 4. The ΓCFA abstract transition $\widehat{\varsigma} \approx\!> \widehat{\varsigma}'$

covers the $\widehat{Eval}$-to-$\widehat{Eval}$ transition for `let` constructs. The (letrec-binding) rule is similar, but it implements recursive environment structure by evaluating the λ terms within the *next* environment $\widehat{\beta}'$. The final rule (proc. app.) covers $\widehat{Apply}$-to-$\widehat{Eval}$ transitions for the application of a procedure.

In an improvement upon previous work [9], we include machinery to detect when a binding remains invariant across a call. The sole purpose of passing a vector of references (*i.e.*, bindings) is to determine when a variable is being rebound to itself. In the (proc. app.) rule, when it's found that a binding is being rebound to itself, its abstract cardinality—the number of concrete bindings it represents—does not increase.

The root of the abstract graph for a program *call* is the initial machine state, an $\widehat{Eval}$ state with an empty environment and a counter that maps everything to 0: $(call, \bot, \bot, (\lambda\_.0), \widehat{t}_0)$.

Note that using a CPS-based representation renders all the rules of our semantics axioms: none of the rules in Figure 4 are inference rules with antecedents. Thus, a CPS semantics really captures the notion of a "machine," where each transition depends on a local, bounded amount of computation and context.

Finally, note what happens when we cast our fairly standard higher-order control-flow analysis as an abstract small-step semantics: it maps a program into a finite state-graph. . . which is exactly what a model-checker needs. Before invoking a model checker, however, we'll first turn our attention to techniques to "sharpen" our abstract state graph, reducing the degree of approximation inherent in its finite structure.

4 Governors

Conditional abstract garbage collection attempts to discard even some "reachable" abstract objects by proving that an unsatisfiable condition guards their use. This requires a syntactic function that yields the sequence of the conditions that hold upon reaching an expression. For example, in the expression:

```
(let ((a 3))
  (if (= a b) e1 e2))
```

The *binding* `(↦ a 3)` and the *condition* `(= a b)` govern the use of the expression `e1`, whereas `(↦ a 3)` and `(not (= a b))` govern the use of `e2`. Formally, given a term t and a subterm $s \in t$, the governors of s within t are the conditions in the vector $\mathcal{G}(t, s)$, where $\mathcal{G}$ is defined in Figure 5.

$$
\begin{aligned}
\mathcal{G}(v, s) &= \langle\rangle \\
\mathcal{G}(\mathit{const}, s) &= \langle\rangle \\
\mathcal{G}([\![\texttt{(}\lambda\ \texttt{(}v_1 \cdots v_n\texttt{)}\ \mathit{call}\texttt{)}]\!], s) &= \mathcal{G}(\mathit{call}, s) \\
\mathcal{G}([\![\texttt{(}e_1 \cdots e_n\texttt{)}]\!], s) &= \begin{cases} \mathcal{G}(e_i, s) & s \in e_i \\ \langle\rangle & \text{otherwise} \end{cases} \\
\mathcal{G}([\![\texttt{(let ((}v\ e\texttt{))}\ \mathit{call}\texttt{)}]\!], s) &= \begin{cases} \mathcal{G}(e, s) & s \in e \\ \langle[\![\texttt{(}\mapsto\ v\ \ e\texttt{)}]\!]\rangle \mathbin{\S} \mathcal{G}(\mathit{call}, s) & s \in \mathit{call} \\ \langle\rangle & \text{otherwise} \end{cases} \\
\mathcal{G}([\![\texttt{(letrec ((}v\ \mathit{lam}\texttt{)}^*\texttt{)}\ \mathit{call}\texttt{)}]\!], s) &= \begin{cases} \langle[\![\texttt{(}\mapsto\ v_i\ \ \mathit{lam}_i\texttt{)}]\!]\rangle \mathbin{\S} \mathcal{G}(\mathit{lam}_i, s) & s \in \mathit{lam}_i \\ \langle[\![\texttt{(}\mapsto\ v_i\ \ \mathit{lam}_i\texttt{)}]\!]\rangle \mathbin{\S} \mathcal{G}(\mathit{call}, s) & s \in \mathit{call} \\ \langle\rangle & \text{otherwise} \end{cases} \\
\mathcal{G}([\![\texttt{(if}\ e_c\ e_t\ e_f\texttt{)}]\!], s) &= \begin{cases} \langle e_c\rangle \mathbin{\S} \mathcal{G}(e_t, s) & s \in e_t \\ \langle[\![\texttt{(not}\ e_c\texttt{)}]\!]\rangle \mathbin{\S} \mathcal{G}(e_f, s) & s \in e_f \\ \langle\rangle & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 5. The governor function. (We write $v_1 \mathbin{\S} v_2$ to concatenate two vectors.)

5 Conditional abstract garbage collection

In previous work [9], we based abstract garbage collection on the same notion as concrete garbage collection: *reachability.* That is, if object a is reachable, and

a points to b, then object b is also considered reachable. Reachability, however, is overly conservative, as it might keep objects uncollected when they will never again be *used*.

Consider the following thunk-creating function, `f`:

```
(define (f a b c d)
  (λ () (if (equal? a b) c d)))
```

Analyzing the expression `(f x x y z)` produces an abstract closure containing entries for the variables `a`, `b`, `c` and `d` in its environment. So, all four bindings would be considered *reachable* from this closure. In reality, however, it is impossible to reach the binding to the variable `d`, since the predicate `(not (equal? a b))` governs its use, and because the predicate is provably unsatisfiable from the information `x` = `a` = `b`. To lessen such problems, we annotate object-to-object links with governing conditions in the abstract heap $\widehat{ve}$; these conditions must be satisfiable for a binding to be potentially usable.

To build this stronger GC, we first need the concept of the set of bindings *touched* by a value. The touching function accepts an environment, a counter and a value, and it returns the bindings directly touched by that value:

$$\widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(lam, \widehat{\beta}) = \{(v, \widehat{\beta}(v)) : v \in \mathit{free}(lam) \text{ and } (\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \langle\rangle)\ \mathit{MaySat}\ \mathcal{G}(lam, v)\},$$

where the *MaySat* (may satisfy) relation includes a binding only if all of its governors could be satisfiable.

The *MaySat* relation is a subset of $(\widehat{BEnv} \times \widehat{VEnv} \times \widehat{Count} \times \widehat{Gov}^*) \times \widehat{Gov}^*$. The notion that a compound environment $(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})$ may satisfy a vector of governors $\boldsymbol{g'}$ is defined recursively:

$$\frac{(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})\ \mathit{MaySat}\ g'_1 \qquad (\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g}\ \S\ \langle g'_1\rangle)\ \mathit{MaySat}\ \langle g'_2, \ldots, g'_n\rangle}{(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})\ \mathit{MaySat}\ \langle g'_1, \ldots, g'_n\rangle}$$

The base case, $(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})\ \mathit{MaySat}\ \langle\rangle$, holds trivially.

Clearly, we can specify a number of rules to describe the *MaySat* relation on a single governor. The less obvious rules are below. For any case not covered, the *MaySat* relation can always conservatively report "yes," Were the relation *MaySat* to always report "yes," the GC would become reachability-based.

Binding governors are trivially satisfied, and they also yield an equivalence:

$$\frac{(\mapsto\ v\ \ e) \in \boldsymbol{g}}{(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})\ \mathit{MaySat}\ (\equiv\ v\ \ e)}$$

Surprisingly, with the use of abstract counting, we can also attempt to prove complete equality ($\equiv$) for function values by checking (efficiently) to see if two closures happen to describe the same function:

$$\frac{\widehat{\mathcal{A}}(v_1, \widehat{\beta}, \widehat{ve}) = \widehat{\mathcal{A}}(v_2, \widehat{\beta}, \widehat{ve}) = (lam, \widehat{\beta}') \qquad \forall v \in \mathit{free}(lam) : \widehat{\mu}(v, \widehat{\beta}'(v)) = 1}{(\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \boldsymbol{g})\ \mathit{MaySat}\ (\equiv\ v_1\ \ v_2)}$$

We may also choose to invoke an external theorem prover in an attempt to demonstrate the *MaySat* relation. In other work [8], we explored the sound integration of abstract interpretation and theorem proving. The issues and solutions encountered there are adaptable to this context as well.

At this point, we can define the remainder of the garbage-collection machinery. Basic values touch nothing; for denotables, we extend touching:

$$\widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} = \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{proc}_1) \cup \cdots \cup \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{proc}_n).$$

We can then extend the notion of touching to states:

$$\widehat{\mathcal{T}}(call, \widehat{\beta}, \widehat{ve}, \widehat{\mu}, \widehat{t}) = \left\{ (v, \widehat{\beta}(v)) : \begin{array}{l} v \in \mathit{free}(call), \text{ and} \\ (\widehat{\beta}, \widehat{ve}, \widehat{\mu}, \langle\rangle) \ \mathit{MaySat}\ \mathcal{G}(call, v) \end{array} \right\}$$

$$\widehat{\mathcal{T}}(\widehat{proc}, \widehat{\boldsymbol{d}}, \widehat{\boldsymbol{r}}, \widehat{ve}, \widehat{\mu}, \widehat{t}) = \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{proc}) \cup \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{d}_1) \cup \cdots \cup \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{d}_n).$$

These functions return the root set from which garbage collection begins. Note that the touching function does *not* return the references supplied, $\widehat{\boldsymbol{r}}$. These references are never used to index into the abstract heap $\widehat{ve}$, and so do not constitute a reachable use.

The resource we care about is the set of reachable bindings (not values), so the following relation links binding to binding, skipping over intervening values:

$$\widehat{b}_{\text{toucher}} \overset{\widehat{\mu}}{\underset{\widehat{ve}}{\hat{\rightsquigarrow}}} \widehat{b}_{\text{touched}} \text{ iff } \widehat{b}_{\text{touched}} \in \widehat{\mathcal{T}}^{\widehat{\mu}}_{\widehat{ve}}(\widehat{ve}(\widehat{b}_{\text{toucher}})).$$

The abstract reachable-bindings function, $\widehat{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Bind})$ computes the bindings reachable from a state:

$$\widehat{\mathcal{R}}(\widehat{\varsigma}) = \{\widehat{b} : \widehat{b}_{\text{root}} \in \widehat{\mathcal{T}}(\widehat{\varsigma}) \text{ and } \widehat{b}_{\text{root}} \left(\overset{\widehat{\mu}_{\widehat{\varsigma}}}{\underset{\widehat{ve}_{\widehat{\varsigma}}}{\hat{\rightsquigarrow}}}\right)^{*} \widehat{b}\}.$$

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

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

Less formally, abstract garbage collection restricts the global variable environment and the counter to those bindings which are reachable from that state.[1]

For any state, we can make a garbage-collecting transition instead of a regular transition:

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

Unlike the flow-analytic version of ΓCFA, there is no advantage for precision in delaying a collection, so every transition now collects.[2] Figure 6 provides a visual representation of the abstract heap both without governors (traditional ΓCFA) and with governors (our enhanced ΓCFA).

[1] When an entry in a counter $\widehat{\mu}$ is restricted, it maps to 0 rather than the value $\perp$.

[2] Some optimizations, such as Super-β copy propagation, require that the flow analysis preserves information about dead bindings as long as possible. If counting can prove a dead binding equivalent to a live binding, it is sometimes efficient to replace the live variable with the otherwise dead variable.

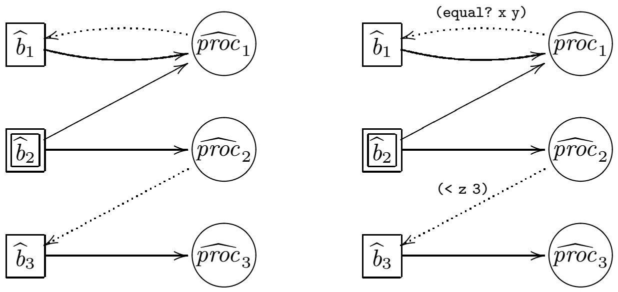


Fig. 6. Two illustrations of an abstract environment ($\widehat{ve}$). Abstract bindings (in boxes) behave like addresses. Abstract values are in circles. Solid arrows denote that a binding yields a particular value in this abstract machine state's total environment $\widehat{ve}$. Dotted arrows denote that a value touches ($\widehat{\mathcal{T}}$) a particular binding. The labels on dotted arrows denote the guards which must be satisfiable in order for the binding to be semantically touchable. The image on the left denotes a heap without governors; the image on the right includes sample governors which must be satisfied for a value to touch a binding.

6 Termination

Naïvely exploring the entire abstract transition graph, while sound, is not the best approach to running the analysis. At the very least, the state-space should be explored in depth-first order; each time a new state $\hat{\varsigma}$ is encountered, the analysis should check to see whether there exists previously-visited state $\hat{\varsigma}'$ such that $\hat{\varsigma} \sqsubseteq \hat{\varsigma}'$. If so, this branch terminates soundly.

Even this approach, however, misses opportunties to cut off forking due to conditionals such as `if`. Instead, the search can use *two* work lists: a *normal* work list, and a *join-point* work list. In the normal phase, the search pulls from the normal work list. When queueing subsequent states, a state applying a join-point continuation[3] goes in the join-point work list. After exhausting the normal work list, the search runs garbage collection on all states in the join-point list. After this, the search is free to merge (through $\sqcup : \mathit{Apply} \times \mathit{Apply} \to \mathit{Apply}$) those states currently at the same continuation. Aggressive merging lowers precision in exchange for speed, whereas less enthusiastic merging leads to higher precision but more time. After this, the join-point and normal lists are swapped, and the exploration continues.

7 A small example

In this section, we will trace through a small example that very simply demonstrates how abstract garbage collection leads to increased flow-sensitivity even in a context-insensitive analysis. Flow-sensitivity, in turn, is important when

[3] Join-point continuations are easily annotated during CPS conversion.

verifying the safety of programs that must obey an ordering in their use of an API. We opt for a very simple specification: that calls to `lock` and `unlock` are in the right order and never nested.

Take the following program:

```
(define (lockloop n)
  (if (= n 0) (begin (lock mutex) (lockloop 1))
              (begin (unlock mutex) (lockloop 0))))
(lockloop 0)
```

Clearly, this program will forever alternate between locking and unlocking `mutex`. But can we model check the computation's abstract state space to verify that it correctly observes the lock/unlock protocol? Unfortunately, if we proceed with an ordinary 0CFA-level abstract interpretation, we're told that this code could potentially lock `mutex` twice. Here's what happens:

1. The flow set for `n` grows to $\{0\}$.
2. The true conditional arm is taken.
3. `mutex` is locked.
4. `lockloop` is called recursively.
5. The flow set for `n` grows to $\{0, 1\}$.
6. Both conditional arms are taken.
7. The analysis tries to re-lock `mutex`.
8. Lock-order-safety verification fails.

The problem we're encountering is that in a traditional abstract interpretation, the flow sets increase monotonically. With abstract garbage collection enabled, however, flow sets can contract, and we get the following scenario:

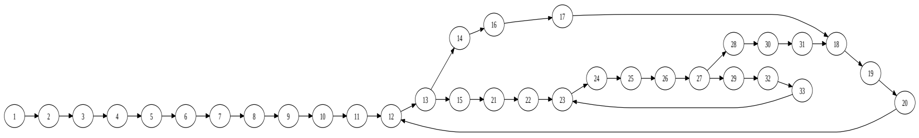
1. The flow set for `n` grows to $\{0\}$.
2. The true conditional arm is taken.
3. `mutex` is locked.
4. The flow set for `n` is GC'd.
5. `lockloop` is called recursively.
6. The flow set for `n` grows to $\{1\}$.
7. The false conditional arm is taken.
8. `mutex` is unlocked.
9. The flow set for `n` is GC'd.
10. `lockloop` is called recursively.
11. The flow set for `n` grows to $\{0\}$.
12. Lock-order verification succeeds.

With abstract garbage collection enabled, this small example is verified to be safe with respect to proper locking behavior even with 0CFA-level precision. Figure 7 depicts the abstract transition graphs generated both without, and then with, abstract garbage collection enabled. As before, the simplification makes it possible *visually* to reconstruct the control flow of the code from the garbage-collected graph.

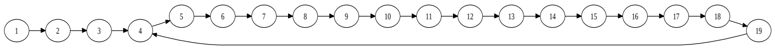
Note that we have *not* verified the (enormous) state space produced by interleaving execution steps of the locking thread with execution steps of some other thread in some concurrent semantics, which, of course, is the context in which we usually care about locks. We have simply verified that a single sequential computation manipulates a resource such as a lock or a file descriptor according to the requirements of some prescribed use protocol.

8 A higher-order example

Garbage collection also plays a critical role in taming higher-orderness during model checking. Consider the following code, which demonstrates this point:



(without GC)



(with GC)

Fig. 7. Abstract state transition graphs without, and then with, abstract garbage collection for the infinite lock-unlock loop example.

```
(define mylock   (identity lock))
(define myunlock (identity unlock))
(mylock mutex)   (myunlock mutex)
```

Once again, running the 0CFA-level interpretation without garbage collection fails to verify. Running it again, but with garbage collection, succeeds.

As before, the problem is flow-set merging. Both `lock` and `unlock` are seen flowing out of the identity function `id` when `myunlock` is bound. Hence, the flow set for `myunlock` includes both `lock` and `unlock`. Thus, it appears to the program that "`lock lock`" is a possible sequence.

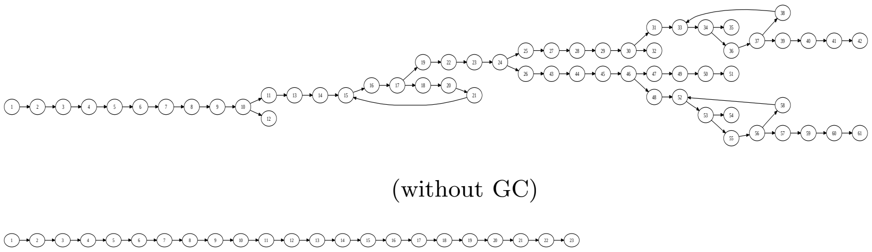
With garbage collection enabled, the flow set for the return value of `id` is collected before the second call, thereby keeping the flow set of `myunlock` to strictly `unlock`. Consequently, the only lock sequence exhibited is "`lock unlock`."

Figure 8 contains the abstract transition graphs both with and without garbage collection for a 0CFA-level contour set. Once again, the collected graph has exactly the linear progression of states we expect from this example. The uncollected graph is even more unwieldy than expected. This happens because continuations (unseen in the direct-style code) also merge in the abstract, and this leads to further losses in precision and speed. In the garbage-collected version, however, flow sets for continuations are also collected.

When a sequence of locks must be taken in order to use a resource, handling higher-orderness precisely is even more important, for then code patterns such as the following become commonplace and natural to the functional programmer:

```
(map lock lock-list)
...
(map unlock lock-list)
```

Fortunately, with ΓCFA, the flow sets for `f` don't merge between invocations of `map`, as they ordinarily would without garbage collection.



(without GC)

(with GC)

Fig. 8. Abstract state transition graphs without and then with abstract garbage collection for the higher-order lock-unlock example.

9 Understanding abstract garbage collection

It's worth exploring the subtle interaction between flow sensitivity, continuations and abstract GC with a quick case-wise analysis. Programmers hoping to have their programs validated by this technology should know when abstract GC wins, and when it loses.

Suppose the call site `(f ...` e`)` with continuation argument e invokes the function `(`λ `(... k) ...)` during abstract interpretation. Let's also assume a 0CFA contour set for now. We can divide this situation into three possible cases.

The first case is when this function is being called recursively as a self-tail call. That is, a frame for this λ term is live and topmost on the stack, the continuation e is the variable `k`, and the function `f` evaluates to this λ term. Because this is a tail call, the flow set for the variable `k` is going to merge with itself. In other words, no precision is lost for this continuation. As a result, no additional branching results when this function returns to the values that `k` represents. This is important, because iteration constructs such as `for` loops and `while` loops transform to this kind of tail recursion in CPS. The extra intelligence we have added about re-binding a variable to itself prevents counting precision from degrading in this case, too.

The second case is when this λ term is being called recursively (perhaps indirectly or mutually) but not as a tail-call. That is, a frame for this λ term is live on the stack. This liveness makes the binding for `k` uncollectable. As a result, the flow set for the return point e will merge into the flow set for the continuation `k`, which already contains return points for the external call to this λ term. Consequently, when interpretation returns from this λ term, it will return to external callers from internal or indirectly recursive call sites. If the precision loss is an issue, switching to a 1CFA contour set or to polymorphic-splitting [13] removes some of this kind of merging.

The third case is when this λ term is not live on the stack; that is, an external call to this λ term. In this case, the binding to the continuation variable `k` is collectable. Consequently, before merging the flow set for the return point e into the flow set for the continuation `k`, the flow set for the continuation `k` is reset to empty. So, in the abstract interpretation, this λ term returns only to the return points in the flow set for the return point e.

This behavior is a major departure from ordinary flow-sensitive 0CFA analyses, where a function spuriously returns to the return points of all previous callers. The net effect of this behavior is to augment the degree of polyvariance achieved for any given contour set. Perhaps most importantly, we can make promises to the programmer that if they use strict tail-recursion and imperative iteration constructs such as `while` and `for`, they will be rewarded during abstract interpretation.

10 Implementation

We have an implementation of ΓCFA for Scheme, written in Haskell. This is the implementation that we used to analyse the lock protocols of the examples in the previous two sections. The implementation also produces warnings for possible list-access violations, *e.g.* taking the `car` of the empty list. In addition, it performs shape analysis on linked lists, reporting back locations through which improper (*i.e.*, non-`nil`-terminated) lists may pass. At present, the implementation does not utilize an external theorem prover for the *MaySat* relation. We are currently working with our colleagues at Georgia Tech to integrate the ACL2 theorem prover into the system.

11 Related work

The analysis of recursive, higher-order functions in the λ calculus has a rich history dating back to Church's original work. In recent years, software verification and model-checking have made strides with tools such as SLAM [2] and TERMINATOR [3]. TERMINATOR, in fact, can reason about function pointers, which are a strictly weaker, environmentless cousin to the higher-order closures we deal with here. Fusing Leuschel *et al.*'s recent work [7] on symbolic closures with our own presents a promising avenue for future research.

ΓCFA is embedded within the Cousots' framework of abstract interpretation [4, 5]. It falls into the family of *sound*, *context-sensitive*, *flow-sensitive*, *non-monotonic* model checkers for higher-order programs. ΓCFA differs from other approaches in that it is geared specifically toward controlling spurious branches that result from control structures such as continuations and higher-order functions. We believe it is possible to adapt the notion of abstract garbage collection to abstract-interpretation-based checkers.

References

1. Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

2. Ball, T., and Rajamani, S. K. The SLAM project: Debugging system software via static analysis. In *Proceedings of the 29th Annual ACM SIGPLAN Conference on the Principles of Programming Languages* (Portland, Oregon, January 2002), pp. 1–3.
3. Cook, B., Podelski, A., and Rybalchenko, A. Termination proofs for systems code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)* (Ottawa, Canada, June 2006).
4. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California, Jan. 1977), vol. 4, pp. 238–252.
5. Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *ACM SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas, Jan. 1979), vol. 6, pp. 269–282.
6. Danvy, O. A first-order one-pass CPS transformation. *Theoretical Computer Science 308*, 1-3 (2003), 239–257.
7. Leuschel, M., and Bendisposto, J. Animating and model checking b specifications with higher-order recursive functions. In *Rigorous Methods for Software Construction and Analysis* (2006), J.-R. Abrial and U. Glässer, Eds., no. 06191 in Dagstuhl Seminar Proceedings.
8. Might, M. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM SIGPLAN Conference on the Principles of Programming Languages* (Nice, France, January 2007).
9. Might, M., and Shivers, O. Improving Flow Analysis via *Γ*CFA: Abstract Garbage Collection and Counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming (ICFP 2006)* (Portland, Oregon, September 2006).
10. Sabry, A., and Felleisen, M. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation 6*, 3/4 (1993), 289–360.
11. Shivers, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, June 1988), pp. 164–174.
12. Steele Jr., G. L. RABBIT: a compiler for SCHEME. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
13. Wright, A. K., and Jagannathan, S. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*, 1 (January 1998), 166–207.