Single and loving it:
Must-alias analysis for higher-order languages

Suresh Jagannathan* Peter Thiemann§ Stephen Weeks* Andrew Wright*

Abstract

In standard control-flow analyses for higher-order languages, a single abstract binding for a variable represents a set of exact bindings, and a single abstract reference cell represents a set of exact reference cells. While such analyses provide useful may-alias information, they are unable to answer must-alias questions about variables and cells, as these questions ask about equality of specific bindings and references.

In this paper, we present a novel program analysis for higher-order languages that answers must-alias questions. At every program point, the analysis associates with each variable and abstract cell a *cardinality*, which is either single or multiple. If variable x is single at program point p, then all bindings for x in the heap reachable from the environment at p hold the same value. If abstract cell r is single at p, then at most one exact cell corresponding to r is reachable from the environment at p.

Must-alias information facilitates various program optimizations such as lightweight closure conversion [19]. In addition, must-alias information permits analyses to perform *strong updates* [3] on abstract reference cells known to be single. Strong updates improve analysis precision for programs that make significant use of state.

A prototype implementation of our analysis yields encouraging results. Over a range of benchmarks, our analysis classifies a large majority of the variables as single.

1 Introduction

Standard control-flow analyses for higher-order languages [7, 18] compute an abstract value for every subexpression and variable of the program. This information is useful for a variety of optimizations because it answers *may-alias* questions—two variables may alias at a given program point if they *ever* have the same value when control reaches that point. However, some equally valuable optimizations pose *must-alias* questions—two variables must alias at a given program point if they *always* have the same value when control reaches that point.

One optimization that requires must-alias information is removing variables from closures, also known as lightweight closure conversion [19]. Consider the following program fragment:

```
λx.
    let y = λz. ... x ...
    in ... (e 1) ...
```

If a control-flow analysis indicates that expression e always evaluates to a closure constructed from λz, we can use a non-standard calling convention for λz. In particular, we can eliminate variable x from the closure for λz by passing it explicitly:

```
λx.
    let y = λxz. ... x ...
    in ... (e x 1) ...
```

Another optimization we can perform using this information is to inline λz at the call site (e 1). Both these transformations are correct only if x is guaranteed to have the same value at the call (e 1) as it does in the closure to which e evaluates. Hence, to enable these optimizations requires determining that x in the environment of the call must alias x in the closure returned by e. Determining this property for higher-order programs is non-trivial because first-class procedures allow multiple bindings for the same free variable to exist simultaneously. Indeed, few implementations of higher-order languages perform must-alias optimizations except in the simplest of cases.

Must-alias information is equally useful in optimizing programs that use state in non-trivial ways. Consider the following program fragment:

```
let y = deref r1 in
...
let z = deref r2
in ...
```

If r1 and r2 must-alias at the definition of z, they must be bound to the same exact cell. Provided that no assignments to this cell occur between the definitions of y and z, an optimization may replace the dereference of r2 with a more efficient reference to y:

```
let y = deref r1 in
    ...
let z = y
in ...
```

*NEC Research Institute, 4 Independence Way, Princeton, NJ 08540. {suresh,sweeks,wright}@research.nj.nec.com.

§Universität Tübingen, Sand 13, D-72076 Tübingen, Germany. thiemann@informatik.uni-tuebingen.de.

To answer must-alias questions on variables, we extend standard control-flow analysis to compute a per-program-point *variable cardinality map* which indicates whether all reachable environments binding a variable x hold the same value. If so, x is marked single at that point; otherwise, x is marked multiple. In the first example, if x is single (at the application), then the closure called at (e 1) contains the same value for x as the current environment, and x can be eliminated from the closure for λz. If x is multiple, the values for x could be different, and eliminating x from the closure could change the meaning of the program.

To answer must-alias questions on reference cells, we also compute a per-program-point *store cardinality map*. In standard control-flow analyses, an abstract cell represents a set of exact cells allocated at the same program point. The store cardinality map indicates, for a given program point p, whether an abstract cell reachable at p corresponds to at most one exact cell. If so, the cell is marked single at p; otherwise, it is marked multiple. In the second example above, if the abstract values of r1 and r2 refer to the same cell c, and if the store cardinality map indicates c is single at the point where z is bound, the dereference of r2 can be replaced with y provided c is not mutated before z is bound.

The single property for first-class reference cells can also significantly improve the precision of a control-flow analysis. In the following program, a standard control-flow analysis would compute an abstract value representing the set $\{-3, 0, 1, 2, \ldots\}$ because all possible values a cell may acquire are merged:

```
let     f = λz. setref z -3 in
letrec  g = λx.
                  let y = (ref x)
                  in (f y);
                     if x < 10 then (g (x+1));
                     (deref y)
in (g 0)
```

Our analysis uses single information for cells to produce a more precise answer. Even though the above program creates many exact cells which are simultaneously live, only one is reachable from the current environment at the setref. Hence the abstract cell (ref x) is single at this point. By using a *per-program-point store*, the analysis can perform a *strong update* [3] following the call to f, replacing the old contents of (ref x) with -3. Consequently, the analysis computes an abstract value representing $\{-3\}$ for this program. We expect precision of this kind will be particularly valuable for object-oriented languages such as Java that encourage composing programs from small imperative pieces.

To further improve the precision of cardinality maps, our analysis uses an abstract form of garbage collection that traces the sets of bindings and references reachable from a function and its continuation, and discards unreachable elements of the cardinality maps and abstract store. Intuitively, the analysis mimics the behavior of a tracing garbage collector on a stack machine by partitioning the set of reachable variables (and cells) at function calls and returns. At a call, the partition distinguishes bindings reachable through the callee's frame from those reachable through the caller's continuation. If the callee introduces a binding that is also reachable from its current frame, the corresponding variable must be multiple since more than one binding for the same variable is simultaneously reachable from within the callee. Thus, in the expression

```
letrec f = λza. ...(f (λx.(z x)) z) ...
in f (λx.x) 13
```

multiple bindings for z are simultaneously reachable in f's body through the arguments supplied in the recursive call. Consequently, z must be multiple in f. However, since only the most recent binding for a is reachable in f's body, a is single even though many bindings for it are introduced during the recursive evaluation.

At a return, the partition records the bindings (and cells) reachable through the return value. For example, if a binding in a caller's continuation is also returned by the callee, the corresponding variable must be multiple since more than one binding for the same variable is reachable after the call. Thus, in the expression

```
letrec f =
    λxy. let w = λz.x in
         let g = if p then (f (x+1) (y+1)) else λv.v in
         let a = ... x ... y ...
         in w
in ...
```

the recursive call to f returns a closure whose environment contains a binding for x. Since a binding for x (to a different value) is also available in the caller, x is multiple at the binding for a. The variable y which is not reachable through w is single at a.

The remainder of the paper is organized as follows. The next section introduces a simple higher-order language with references. We define an exact semantics for this language, and present a formal definition of *single* and *must-alias* with respect to this semantics. Section 3 presents our analysis framework. Section 4 formalizes our notion of abstract reachability and presents an algorithm for computing a succession of analyses with increasing precision. Section 5 introduces a number of examples highlighting different features of the framework. Section 6 describes an implementation of the analysis, preliminary experimental results, and running time complexity. Section 7 gives comparisons to related work.

2 Must-alias and single

We study a language with *expressions* $e \in Exp$ of the form

$$
\begin{array}{rcl}
e & ::= & x \\
 & | & \mathsf{let}\ x = \lambda y.e'\ \mathsf{in}\ e \\
 & | & \mathsf{let}\ x = y\ z\ \mathsf{in}\ e \\
 & | & \mathsf{let}\ x = \mathsf{ref}\ y\ \mathsf{in}\ e \\
 & | & \mathsf{let}\ x = \mathsf{deref}\ y\ \mathsf{in}\ e \\
 & | & \mathsf{setref}\ x\ y\ \mathsf{in}\ e
\end{array}
$$

where $x, y, z, w \in Var$ are variables.[1] In a let-expression that binds x in e, $\lambda y.e'$ constructs a lexically-scoped first-class procedure, $y\ z$ denotes call-by-value application, ref y constructs a first-class reference cell, and deref y extracts the value of the cell bound to y. The expression setref $x\ y$ in e assigns the value of y to the cell bound to x, and continues with e. Bound and free variables are defined as usual, and $FV(e)$ yields the set of free variables of an expression. Programs P are closed expressions.

In addition to the usual assumption that bound variables are distinct from free variables in distinct expressions, we also assume that all bound variables in a program are distinct. We use *ExpOcc*, *LamOcc*, *AppOcc*, and *RefOcc* to name the sets of specific *occurrences* of subterms of the

[1] We use A-normal form [6] in order to simplify the presentation.

forms e, $\lambda x.e$, $y\ z$, and $\mathsf{ref}\ y$, respectively, in the given program (occurrences can be defined formally using paths or unique expression labels). We use *VarOcc* to name the set of variables occurring in *binding* positions. The *last* variable of an expression, which yields the expression's value, is defined as follows:

$$\begin{array}{rcl} \mathit{last}(x) & = & x \\ \mathit{last}(\mathsf{let}\ x = \ldots\ \mathsf{in}\ e) & = & \mathit{last}(e) \\ \mathit{last}(\mathsf{setref}\ x\ y\ \mathsf{in}\ e) & = & \mathit{last}(e) \end{array}$$

Throughout the remainder of the paper, we use the following notation for sets and functions: $A \setminus B$ denotes set difference; $[x \mapsto y]$ is a partial function that has domain $\{x\}$ and maps x to y; $f[x \mapsto y]$ denotes the functional update or extension of f at x to y; $f|_A$ is the partial function formed by restricting the domain of f to the set A; $f_{\setminus A}$ is the partial function formed by excluding A from the domain of f; $f_1 \leq f_2$ if $dom(f_1) \subseteq dom(f_2)$ and $f_1(x) \leq f_2(x)$ for all $x \in dom(f_1)$, for the appropriate notion of $\leq$ on the functions' range.

The exact semantics is given by a CESK-style transition system [5] $(\mathit{State}, \longmapsto)$, where states are defined as follows.

$$\begin{array}{rcrcl} \Sigma & \in & \mathit{State} & = & \mathit{ExpOcc} \times \mathit{Env} \times \mathit{Store} \times \mathit{Cont} \\ v & \in & \mathit{Val} & = & (\mathit{LamOcc} \times \mathit{Env}) \cup (\mathit{RefOcc} \times \mathbf{N}) \\ \rho & \in & \mathit{Env} & = & \mathit{VarOcc} \rightharpoonup \mathit{Val} \\ \sigma & \in & \mathit{Store} & = & (\mathit{RefOcc} \times \mathbf{N}) \rightharpoonup \mathit{Val} \\ \kappa & \in & \mathit{Cont} & = & (\mathit{VarOcc} \times \mathit{ExpOcc} \times \mathit{Env})^* \end{array}$$

A state $[e, \rho, \sigma, \kappa]$ is a quadruple consisting of an expression e, an evaluation environment ρ for e, the current store σ, and e's continuation κ. Values v are either closures or locations. A closure $\langle \lambda x.e, \rho \rangle$ is a pair consisting of a λ-expression and environment, as usual. A location $\langle \mathsf{ref}\ x, i \rangle$ is a pair consisting of a ref-occurrence and a unique integer. Environments are partial functions that map variables to values, and stores are partial functions that map locations to values. A continuation $\langle x_1, e_1, \rho_1 \rangle \ldots \langle x_n, e_n, \rho_n \rangle$ is a sequence of frames. Figure 1 defines the single-step transition relation $\longmapsto$ between exact states. The evaluation relation $\longmapsto^*$ is the reflexive, transitive closure of $\longmapsto$, with evaluation of program P beginning in the state $\Sigma_0 = [P, \phi, \phi, []]$.

2.1 Must-alias

To formalize the must-alias question for variables and reference cells, we define a notion of *path* as follows:

$$p, q \ ::= \ x \ \mid \ p.x \ \mid \ p!$$

A path of the form x means an access to x from the current environment of some state. A path of the form $p.x$ means an access to x from a closure accessed from p. A path $p!$ means an access to the contents of a reference cell from p. More precisely, given a state $[e, \rho, \sigma, \kappa]$, the function $\mathcal{E}_{\rho\sigma}(\cdot)$ which interprets paths is defined as follows:

$$\begin{array}{rcl} \mathcal{E}_{\rho\sigma}(x) & = & \rho(x) \\ \mathcal{E}_{\rho\sigma}(p.x) & = & \rho'(x) \text{ if } \mathcal{E}_{\rho\sigma}(p) = \langle \lambda y.e', \rho' \rangle \text{ and } x \in FV(\lambda y.e') \\ \mathcal{E}_{\rho\sigma}(p!) & = & \sigma(\langle \mathsf{ref}\ x, i \rangle) \text{ if } \mathcal{E}_{\rho\sigma}(p) = \langle \mathsf{ref}\ x, i \rangle. \end{array}$$

Two paths *must-alias* at an expression e if they always yield the same value.

Definition 2.1 (Must-alias) *In program P, paths p and q* must-alias *at e, written $p \stackrel{e}{\equiv} q$, if $\mathcal{E}_{\rho\sigma}(p) = \mathcal{E}_{\rho\sigma}(q)$ for all states $[e, \rho, \sigma, \kappa]$ such that $\Sigma_0 \longmapsto^* [e, \rho, \sigma, \kappa]$.*

For example, when inlining $\lambda x'.e''$ with w free at call site $e = (\mathsf{let}\ x = y\ z\ \mathsf{in}\ e')$ where w is bound, we can bind free w references in e'' to w in the current environment provided $y.w \stackrel{e}{\equiv} w$.

2.2 Single

Our control-flow analysis can be used to answer must-alias questions for paths terminating in the same variable x when all bindings for x reachable from a state hold the same value. By reachable, we mean the natural notion that tracing garbage collectors use to mark or copy cells that must be retained. For a given store σ, let $\stackrel{\sigma}{\leadsto}$ (pronounced "reaches") be the least reflexive, transitive relation on values such that:

$$\begin{array}{rcl} \langle \mathsf{ref}\ x, i \rangle & \stackrel{\sigma}{\leadsto} & \sigma(\langle \mathsf{ref}\ x, i \rangle) \\ \langle \lambda x.e, \rho \rangle & \stackrel{\sigma}{\leadsto} & \rho(y) \quad \text{for all } y \in FV(\lambda x.e) \end{array}$$

Also define $\rho \stackrel{\sigma}{\leadsto} v$ if there exists $x \in dom(\rho)$ such that $\rho(x) \stackrel{\sigma}{\leadsto} v$. Based on this definition, we can formalize the single property as follows.

Definition 2.2 (Single) *At state $[e, \rho, \sigma, \kappa]$, we say that*

1. *variable x is* single *if, for all environments $\rho_1, \rho_2 \in \{\rho\} \cup \{\rho' \mid \rho \stackrel{\sigma}{\leadsto} \langle \lambda y.e', \rho' \rangle\}$, if $x \in dom(\rho_1)$ and $x \in dom(\rho_2)$ then $\rho_1(x) = \rho_2(x)$;*

2. *cell $\mathsf{ref}\ x$ is* single *if $\rho \stackrel{\sigma}{\leadsto} \langle \mathsf{ref}\ x, i \rangle$ and $\rho \stackrel{\sigma}{\leadsto} \langle \mathsf{ref}\ x, j \rangle$ implies $i = j$.*

Furthermore, we say x (resp. $\mathsf{ref}\ x$) is single at e if x (resp. $\mathsf{ref}\ x$) is single at $[e, \rho, \sigma, \kappa]$ whenever $\Sigma_0 \longmapsto^* [e, \rho, \sigma, \kappa]$.

Single is important for must-alias information because of the following lemma.

Lemma 2.3 *If x is single at e, then $p.x \stackrel{e}{\equiv} q.x \stackrel{e}{\equiv} x$ for all paths p and q.*

In the next section, we define an analysis that computes approximations to "x is single at e" and "$\mathsf{ref}\ x$ is single at e", for all x, $\mathsf{ref}\ x$, and e occurring in the program.

3 Framework

Given a program P, we define an *analysis* as a quadruple $\langle F, S, G, H \rangle$ consisting of an *abstract environment* F, a per-program-point *abstract store* S, and two per-program-point *cardinality maps* G and H (see Figure 2). For each variable x, $F(x)$ is an abstract value, i.e. a set of λ-occurrences and ref-occurrences approximating the values to which x can be bound. We use $o \in \mathit{LamOcc} \cup \mathit{RefOcc}$ as a metavariable for an element of an abstract value. For each expression occurrence e, $S(e)$ is an abstract store mapping an abstract cell to an abstract value describing the possible values that the cell can hold when evaluation reaches e. An abstract store may be undefined for cells that are unreachable. For each expression occurrence e, $G(e)$ is a cardinality map that conservatively approximates whether a variable is single when evaluation reaches e. Similarly, $H(e)$ is a cardinality map that conservatively approximates whether a cell is single when evaluation reaches e.

To formally specify the meaning of an analysis, we define several relations between objects of the exact semantics and components of the analysis. The relation $[e, \rho, \sigma, \kappa] \sqsubseteq$

$$
\begin{array}{rcll}
[\mathsf{let}\ x = \lambda w.e'\ \mathsf{in}\ e, \rho, \sigma, \kappa] & \longmapsto & [e, \rho[x \mapsto \langle \lambda w.e', \rho|_{FV(\lambda w.e')}\rangle], \sigma, \kappa] & \\
[\mathsf{let}\ x = y\ z\ \mathsf{in}\ e, \rho, \sigma, \kappa] & \longmapsto & [e', \rho'[w \mapsto \rho(z)], \sigma, \langle x, e, \rho\rangle : \kappa] & \text{where } \rho(y) = \langle \lambda w.e', \rho'\rangle \\
[w, \rho, \sigma, \langle x, e, \rho'\rangle : \kappa] & \longmapsto & [e, \rho'[x \mapsto \rho(w)], \sigma, \kappa] & \\
[\mathsf{let}\ x = \mathsf{ref}\ y\ \mathsf{in}\ e, \rho, \sigma, \kappa] & \longmapsto & [e, \rho[x \mapsto \langle \mathsf{ref}\ y, i\rangle], \sigma[\langle \mathsf{ref}\ y, i\rangle \mapsto \rho(y)], \kappa] & \text{where } \langle \mathsf{ref}\ y, i\rangle \notin dom(\sigma) \\
[\mathsf{let}\ x = \mathsf{deref}\ y\ \mathsf{in}\ e, \rho, \sigma, \kappa] & \longmapsto & [e, \rho[x \mapsto \sigma(\rho(y))], \sigma, \kappa] & \\
[\mathsf{setref}\ x\ y\ \mathsf{in}\ e, \rho, \sigma, \kappa] & \longmapsto & [e, \rho, \sigma[\rho(x) \mapsto \rho(y)], \kappa] &
\end{array}
$$

Figure 1: Exact semantics.

$$
\begin{array}{rcll}
\langle F, S, G, H\rangle & \in & \mathit{Analysis} & = \mathit{AEnv} \times \mathit{AStores} \times \mathit{VarCardMaps} \times \mathit{CellCardMaps} \\
F & \in & \mathit{AEnv} & = \mathit{VarOcc} \to \mathit{AVal} \\
a & \in & \mathit{AVal} & = \mathcal{P}(\mathit{LamOcc} \cup \mathit{RefOcc}) \\
S & \in & \mathit{AStores} & = \mathit{ExpOcc} \to \mathit{AStore} \\
s & \in & \mathit{AStore} & = \mathit{RefOcc} \rightharpoonup \mathit{AVal} \\
G & \in & \mathit{VarCardMaps} & = \mathit{ExpOcc} \to \mathit{VarCardMap} \\
g & \in & \mathit{VarCardMap} & = \mathit{VarOcc} \rightharpoonup \{\mathsf{single}, \mathsf{multiple}\} \\
H & \in & \mathit{CellCardMaps} & = \mathit{ExpOcc} \to \mathit{CellCardMap} \\
h & \in & \mathit{CellCardMap} & = \mathit{RefOcc} \rightharpoonup \{\mathsf{single}, \mathsf{multiple}\}
\end{array}
$$

Figure 2: Abstract domains.

$\langle F, S, G, H\rangle$ (pronounced "agrees") describes when an analysis conservatively approximates an exact state. Two secondary relations describe agreement between exact environments and abstract environments, and between exact values and abstract values.

Definition 3.1 (Agrees) *Relations $\Sigma \sqsubseteq \langle F, S, G, H\rangle$, $\rho \sqsubseteq F$ and $v \sqsubseteq_F a$ are the least relations such that:*

1. $[e, \rho, \sigma, \kappa] \sqsubseteq \langle F, S, G, H\rangle$ *if*
   (a) $\rho \sqsubseteq F$;
   (b) *If* $\rho \overset{\sigma}{\leadsto} \langle \mathsf{ref}\ x, i\rangle$ *then* $\sigma(\langle \mathsf{ref}\ x, i\rangle) \sqsubseteq_F S(e)(\mathsf{ref}\ x)$;
   (c) *If* $G(e)(x) = \mathsf{single}$ *then* x *is single at* $[e, \rho, \sigma, \kappa]$;
   (d) *If* $H(e)(\mathsf{ref}\ x) = \mathsf{single}$ *then* $(\mathsf{ref}\ x)$ *is single at* $[e, \rho, \sigma, \kappa]$.
2. $\rho \sqsubseteq F$ *iff* $\rho(x) \sqsubseteq_F F(x)$ *for all* $x \in dom(\rho)$.
3. $\langle \lambda x.e, \rho\rangle \sqsubseteq_F \lambda x.e$ *iff* $\rho \sqsubseteq F$.
4. $\langle \mathsf{ref}\ x, i\rangle \sqsubseteq_F \mathsf{ref}\ x$.
5. $v \sqsubseteq_F a$ *iff* $v \sqsubseteq_F o$ *for some* $o \in a$.

For an exact state to agree with an abstract state, the exact environment must agree with the abstract environment and the exact store σ must agree with the per-program-point store for all reachable locations. Also, if a variable or cell is marked single by a cardinality map, then it must be single in the exact state.[2]

An analysis is *sound* for program P if it conservatively approximates each of the exact states computed by the exact transition relation.

Definition 3.2 (Soundness) *Analysis $\langle F, S, G, H\rangle$ is* sound *for P if $\Sigma \sqsubseteq \langle F, S, G, H\rangle$ whenever $\Sigma_0 \longmapsto^* \Sigma$.*

[2]For proofs, we use a stronger version of $\sqsubseteq$ that relates the exact continuation κ to the abstract state.

The following theorem enables us to use a sound analysis to answer must-alias questions about variables.

Theorem 3.3 *If $\langle F, S, G, H\rangle$ is sound and $G(e)(x) = \mathsf{single}$ then $p.x \overset{e}{\equiv} q.x \overset{e}{\equiv} x$ for all paths p and q.*

Proof. Assume $\Sigma_0 \longmapsto^* [e, \rho, \sigma, \kappa]$. Since $\langle F, S, G, H\rangle$ is sound, $[e, \rho, \sigma, \kappa] \sqsubseteq \langle F, S, G, H\rangle$. By the definition of $\sqsubseteq$ and $G(e)(x) = \mathsf{single}$, x is single at $[e, \rho, \sigma, \kappa]$. By Lemma 2.3, $\mathcal{E}_{\rho\sigma}(p.x) = \mathcal{E}_{\rho\sigma}(q.x) = \mathcal{E}_{\rho\sigma}(x)$. □

To state a similar theorem about cells, we first define a function $\mathcal{A}_{Fs}(\cdot)$ that interprets paths in an abstract state:

$$
\begin{array}{rcl}
\mathcal{A}_{Fs}(x) & = & F(x) \\
\mathcal{A}_{Fs}(p.x) & = & F(x) \\
\mathcal{A}_{Fs}(p!) & = & \bigcup_{(\mathsf{ref}\ x) \in \mathcal{A}_{Fs}(p)} s(\mathsf{ref}\ x)
\end{array}
$$

The following theorem allows us to answer must-alias questions about cells.

Theorem 3.4 *If $\langle F, S, G, H\rangle$ is sound and $H(e)(\mathsf{ref}\ x) = \mathsf{single}$ and $\mathcal{A}_{FS(e)}(p) = \{\mathsf{ref}\ x\} = \mathcal{A}_{FS(e)}(q)$ then $p \overset{e}{\equiv} q$.*

Must-alias results for longer paths follow from the fact that $p \overset{e}{\equiv} q$ implies $p.x \overset{e}{\equiv} q.x$ and $p! \overset{e}{\equiv} q!$.

We now define a collection of *safety* constraints that are sufficient to guarantee an analysis $\langle F, S, G, H\rangle$ is sound. We present these constraints in three groups: constraints on the abstract environment F; constraints on the per-program-point variable cardinality map G; and constraints on the per-program-point cell cardinality map H and per-program-point abstract store S.

3.1 Environment constraints

Figure 3 presents the constraints for abstract environments. These are standard for a monovariant control-flow analysis [7, 12], except that we include conditional constraints and

For all $e \in \mathit{Init}_F$:

1. if e is $(\mathsf{let}\, x = \lambda w.e'' \,\mathsf{in}\, e')$ then $\{\lambda w.e''\} \leq F(x)$;
2. if e is $(\mathsf{let}\, x = \mathsf{ref}\, y \,\mathsf{in}\, e')$ then $\{\mathsf{ref}\, y\} \leq F(x)$ and $F(y) \leq S(e')(\mathsf{ref}\, y)$;
3. if e is $(\mathsf{let}\, x = \mathsf{deref}\, y \,\mathsf{in}\, e')$ then for all $(\mathsf{ref}\, z) \in F(y)$: $S(e)(\mathsf{ref}\, z) \leq F(x)$;
4. if e is $(\mathsf{setref}\, x\, y \,\mathsf{in}\, e')$ then for all $(\mathsf{ref}\, z) \in F(x)$: $F(y) \leq S(e')(\mathsf{ref}\, z)$;
5. if e is $(\mathsf{let}\, x = y\, z \,\mathsf{in}\, e')$ then for all $\lambda w.e'' \in F(y)$:
   (a) $F(z) \leq F(w)$;
   (b) if $\mathit{last}(e'') \in \mathit{Init}_F$ then $F(\mathit{last}(e'')) \leq F(x)$.

Figure 3: Environment constraints.

a per-program-point store. To avoid enforcing constraints on dead code, we only require constraints to hold on *initialized* expressions, where Init_F is the smallest set of subexpressions of P that includes P and is closed under the following rules:

1. If $(\mathsf{let}\, x = \lambda y.e'' \,\mathsf{in}\, e') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.
2. If $(\mathsf{let}\, x = \mathsf{ref}\, y \,\mathsf{in}\, e') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.
3. If $(\mathsf{let}\, x = \mathsf{deref}\, y \,\mathsf{in}\, e') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.
4. If $(\mathsf{setref}\, x\, y \,\mathsf{in}\, e') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.
5. If $(\mathsf{let}\, x = y\, z \,\mathsf{in}\, e') \in \mathit{Init}_F$ and $\lambda w.e'' \in F(y)$ then
   (a) $e'' \in \mathit{Init}_F$
   (b) if $\mathit{last}(e'') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.

In Figure 3, we use $\leq$ as the subset order on abstract values. Constraint 1 ensures that a λ-expression flows to all uses of the variable x to which it is bound. Constraints 2, 3, and 4 handle first-class references by propagating values as appropriate between variables and a per-program-point store. These constraints establish no relation between the stores at different program points; constraints between stores are described in Section 3.3. Constraint 5 ensures that, for all λ-expressions the function expression y could yield, (a) the formal parameter w of the λ-expression includes all possible values of the argument z, and (b) the result variable x for the call includes all possible values the function could return from the last expression in its body.

3.2 Cardinality constraints

Figure 4 presents the safety constraints for variable cardinality maps. Cardinality maps are ordered pointwise as partial functions, where $\mathsf{single} \leq \mathsf{multiple}$. The figure uses a commutative and associative operation on cardinality maps $g \bowtie g'$ defined as follows:

$$(g \bowtie g')(x) = \begin{cases} \mathsf{multiple} & \text{if } x \in (\mathit{dom}(g) \cap \mathit{dom}(g')) \\ g(x) & \text{if } x \in \mathit{dom}(g) \\ g'(x) & \text{if } x \in \mathit{dom}(g') \end{cases}$$

The constraints in Figure 4 propagate cardinalities from the program point preceding evaluation of an expression to the following program point. For example, constraints 1, 2 and 3 require the cardinality map for expressions following a variable binding for x to include a cardinality for x. The constraints use $\bowtie$ to enforce the fact that if a binding for x does not already exist, then x can be marked single. On the other hand, if a binding for x already exists, there are now multiple bindings, and x must be marked $\mathsf{multiple}$. Constraint 4 simply propagates the cardinality map to the next program point, because an assignment does not bind a variable.

Constraint 5, for function call, is the most interesting. It consists of two sub-constraints, which must hold for every function that can be called, as determined by the abstract environment. Constraint 5a propagates part of the cardinality map to the body of the function, and handles the new variable binding for w. Constraint 5b propagates the rest of the cardinality map to the return point, as well as a part of the cardinality map from the body of the called function and a binding for the variable x. In determining what parts of the cardinality map to propagate, constraints 5a and 5b use a notion of *abstract reachability* to restrict the domain of cardinality maps. For a given call, the sets V_p, V_s, and V_r determine how much cardinality information is "passed" to the function, how much is "saved" until the return point, and how much is "returned" by the function body, respectively. The sets are computed in terms of a pair of functions:

$$R_{\text{pass}}, R_{\text{return}} : \mathit{AppOcc} \times \mathit{LamOcc} \to \mathit{AVal}.$$

For a given call site and function called at that site, R_{pass} returns an approximation to the set of values reachable from the function or the argument. R_{return} approximates the set of values reachable from the result returned by the call or from values that were available to the call, and hence possibly modified through assignments. Given R_{pass} and R_{return}, V_p and V_r are computed using a function $\mathit{FreeVarsOf}$, defined as follows:

$$\mathit{FreeVarsOf}(a) = \{y \mid (\lambda x.e) \in a \text{ and } y \in FV(\lambda x.e)\}.$$

The set V_s is simply a set difference, and contains variables (abstractly) reachable at the call but which are *not* passed to the called function.

By making the sets V_p and V_r as small as possible, the analysis can mark more variables as single, and hence provide better must-alias information. For example, in constraint 5, suppose the bound variable w of the called function occurs in $\mathit{dom}(G(e)|_{V_p})$. This means that w occurs free through some closure that is deemed reachable by R_{pass} from the function or argument. Consequently, w must be marked $\mathsf{multiple}$ in the cardinality map at the entry to the function since this condition implies that multiple instances of w may be extant during evaluation of the function body. Similarly, upon return, the cardinality map will mark the variable x

For all $e \in \mathit{Init}_F$:

1. if e is $(\mathsf{let}\, x = \lambda y.e'' \,\mathsf{in}\, e')$ then $G(e) \bowtie [x \mapsto \mathsf{single}] \leq G(e')$;
2. if e is $(\mathsf{let}\, x = \mathsf{ref}\, y \,\mathsf{in}\, e')$ then $G(e) \bowtie [x \mapsto \mathsf{single}] \leq G(e')$;
3. if e is $(\mathsf{let}\, x = \mathsf{deref}\, y \,\mathsf{in}\, e')$ then $G(e) \bowtie [x \mapsto \mathsf{single}] \leq G(e')$;
4. if e is $(\mathsf{setref}\, x\, y \,\mathsf{in}\, e')$ then $G(e) \leq G(e')$;
5. if e is $(\mathsf{let}\, x = y\, z \,\mathsf{in}\, e')$ then for all $(\lambda w.e'') \in F(y)$:
   (a) $G(e)|_{V_p} \bowtie [w \mapsto \mathsf{single}] \leq G(e'')$;
   (b) if $\mathit{last}(e'') \in \mathit{Init}_F$ then $G(e)|_{V_s} \bowtie G(\mathit{last}(e''))|_{V_r} \bowtie [x \mapsto \mathsf{single}] \leq G(e')$;

   where
$$\begin{aligned} V_p &= \mathit{FreeVarsOf}(R_{\text{pass}}(e, \lambda w.e'')) \\ V_s &= \mathit{dom}(G(e)) \setminus V_p \\ V_r &= \mathit{FreeVarsOf}(R_{\text{return}}(e, \lambda w.e'')) \end{aligned}$$

Figure 4: Variable cardinality map constraints.

which binds the result as multiple if x was included in the cardinality map containing the variables saved across the call, or if x was included in the cardinality map returned by the call.

3.3 Store constraints

Analyses for mostly functional languages typically treat assignment by using one global abstract store. Using a global abstract store forces the analysis to merge together all values that might ever be assigned to a given cell [7, 8, 18]. In our framework, this amounts to requiring $S(e) = S(e')$ for all subexpressions e, e' of program P. While sound, this treatment of assignment leads to overly imprecise analysis of imperative programs. Consider the following program:

```
let x = λx'.x'   in
let y = λy'.y'   in
let z = ref x    in
    setref z y   in
let w = deref z
in  w
```

Using a global abstract store, the least abstract environment for this program maps w to the abstract value $\{\lambda x', \lambda y'\}$. However, classical intraprocedural dataflow analysis, adapted appropriately for our language, would easily determine that w can only be bound to $\lambda y'$. Such a precise result is only possible if the analysis keeps different approximations for the contents of the store at different program points, and is able to ignore old values in a cell when that cell is updated (e.g. the update of z above). Chase et al. refer to this property as *strong update* [3].

Strong updates are problematic in the presence of first-class cells. In order to perform a strong update on an abstract cell, there must be only one corresponding exact cell. The may-alias information that standard control-flow analyses determine for functional languages is not sufficient to determine this. However, by extending our analysis to compute cardinality information for reference cells, we can use the single property on abstract cells to perform strong updates.

Figure 5 defines a collection of cell cardinality constraints, analogous to those in Figure 4 for variables. The figure also includes constraints that propagate abstract stores between different program points. We define $\bowtie$ and $\leq$ on *CellCardMap* as on *VarCardMap*; $\sqcup$ as the least upper bound of two abstract stores; and *RefsOf* as follows:

$$\mathit{RefsOf}(a) = \{\mathsf{ref}\ x \mid (\mathsf{ref}\ x) \in a\}.$$

Constraints 1, 2, and 3 are analogous to those given in Figure 4. Constraint 4 performs a strong update when a single cell is assigned. If the abstract value for x holds a single abstract cell $(\mathsf{ref}\ z)$, and if that cell is single in the cardinality map, then the old abstract value for $(\mathsf{ref}\ z)$ need not be propagated to the abstract store for the following program point. If the abstract value for x is empty, then no store need be propagated at all. Otherwise, if the abstract value for x holds more than one abstract cell or if the abstract cell is multiple, the entire abstract store must be propagated to the following program point. In any case, environment constraint 4 (Figure 3) ensures that the abstract store following the assignment contains at least the abstract value of y.

Constraints 5a and 5b propagate cardinalities for cells and abstract stores in a similar manner to the variable cardinality constraints in Figure 4, using abstract reachability to eliminate unnecessary cells.

3.4 Safety

We call the pair of functions $\langle R_{\text{pass}}, R_{\text{return}}\rangle$ that parameterize the constraint system a *reachmap*, and define a notion of *safety* parameterized by a reachmap.

Definition 3.5 (R-safe) *Given $R \in \mathit{ReachMap}$, analysis $\langle F, S, G, H\rangle$ is R-safe if the environment constraints in Figure 3, the cardinality constraints in Figure 4, and the store constraints in Figure 5 hold.*

To improve precision, we would like the reachmap to be as small as possible; however, soundness imposes certain lower bounds. For now, an obviously sound choice is to assume everything is reachable. We refer to this reachmap as $R^\top$:

$$\begin{aligned} R^\top &= \langle R^\top_{\text{pass}}, R^\top_{\text{return}}\rangle \\ R^\top_{\text{pass}}(e, \lambda w.e'') &= \mathit{LamOcc} \cup \mathit{RefOcc} \\ R^\top_{\text{return}}(e, \lambda w.e'') &= \mathit{LamOcc} \cup \mathit{RefOcc} \end{aligned}$$

In Section 6, we describe an algorithm

$$\mathit{MinSafe} : \mathit{ReachMap} \to \mathit{AEnv} \times \mathit{AStores} \times \mathit{VarCardMaps}$$

For all $e \in Init_F$:

1. if e is $(\mathsf{let}\, x = \lambda y.e'' \,\mathsf{in}\, e')$ then $H(e) \leq H(e')$ and $S(e) \leq S(e')$;
2. if e is $(\mathsf{let}\, x = \mathsf{ref}\, y \,\mathsf{in}\, e')$ then $H(e) \bowtie [\mathsf{ref}\, y \mapsto \mathsf{single}] \leq H(e')$ and $S(e) \leq S(e')$;
3. if e is $(\mathsf{let}\, x = \mathsf{deref}\, y \,\mathsf{in}\, e')$ then $H(e) \leq H(e')$ and $S(e) \leq S(e')$;
4. if e is $(\mathsf{setref}\, x\, y \,\mathsf{in}\, e')$ then:
   (a) $H(e) \leq H(e')$;
   (b) if $F(x) = \{\mathsf{ref}\, z\}$ and $H(e)(\mathsf{ref}\, z) = \mathsf{single}$ then $S(e)_{\setminus\{\mathsf{ref}\, z\}} \leq S(e')$ else if $F(x) \neq \emptyset$ then $S(e) \leq S(e')$;
5. If e is $(\mathsf{let}\, x = y\, z \,\mathsf{in}\, e')$ then for all $(\lambda w.e'') \in F(y)$:
   (a) $H(e)|_{L_p} \leq H(e'')$ and $S(e)|_{L_p} \leq S(e'')$;
   (b) if $last(e'') \in Init_F$ then $H(e)|_{L_s} \bowtie H(last(e''))|_{L_r} \leq H(e')$ and $S(e)|_{L_s} \sqcup S(last(e''))|_{L_r} \leq S(e')$;

   where
$$\begin{aligned} L_p &= RefsOf(R_{\text{pass}}(e, \lambda w.e'')) \\ L_s &= dom(H(e)) \setminus L_p \\ L_r &= RefsOf(R_{\text{return}}(e, \lambda w.e'')) \end{aligned}$$

Figure 5: Store constraints.

that, given an appropriate reachmap R, computes a minimum R-safe analysis using standard constraint-solving techniques [13].

3.5 Example

To illustrate how the reachmap is used by the analysis, consider analyzing the following program fragment using $R^\top$:

```
let f = λx.(let g = λy.x in
            let h = (g 13) in h) in
let j = λz.z in
let k = (f j)
in k
```

At the call to f,

$$R^\top_{\text{pass}}((\mathsf{f\ j}), \lambda\mathsf{x}) = R^\top_{\text{return}}((\mathsf{f\ j}), \lambda\mathsf{x}) = \{\lambda\mathsf{x}, \lambda\mathsf{y}, \lambda\mathsf{z}\}.$$

Thus, $V_p = V_r = \{\mathsf{x}\}$ since $\mathsf{x} \in FV(\lambda\mathsf{y})$. By rule 5a in Figure 4, x is therefore multiple in the variable cardinality map computed for f's body. Hence, using the results of this analysis under $R^\top$ we cannot conclude that g.x must alias x at the binding for h. This imprecision would prevent an optimization such as lightweight closure-conversion from being applied at g's call site in f. However, by using a more precise reachmap with

$$R_{\text{pass}}((\mathsf{f\ j}), \lambda\mathsf{x}) = R_{\text{return}}((\mathsf{f\ j}), \lambda\mathsf{x}) = \{\lambda\mathsf{x}, \lambda\mathsf{z}\},$$

our analysis will indeed allow us to conclude g.x must alias x at h's definition.

4 Abstract reachability and correctness

While it is certainly correct, $MinSafe(R^\top)$ is not very precise because it never removes entries from the cardinality maps or the abstract store. Using a more precise reachmap can improve precision because the cardinality constraints mark a bound variable single only if it is not reachable. We now show how to compute a succession of reachmaps and analyses of increasing precision.

We first define the abstract analogue of exact reachability. For abstract environment F and abstract store s, let $\overset{F\,s}{\leadsto}$ be the least reflexive, transitive relation on elements of abstract values such that:

$$\begin{aligned} &\mathsf{ref}\, x \overset{F\,s}{\leadsto} o \text{ for all } o \in s(\mathsf{ref}\, x) \\ &\lambda x.e \overset{F\,s}{\leadsto} o \text{ for all } o \in F(y),\ y \in FV(\lambda x.e) \end{aligned}$$

This definition ensures that an abstract value is reachable if its exact analog is reachable.

Lemma 4.1 *If $[e, \rho, \sigma, \kappa] \sqsubseteq \langle F, S, G, H\rangle$ and $\rho \overset{\sigma}{\leadsto} v$ and $v \overset{\sigma}{\leadsto} v'$ and $v \sqsubseteq_F o$ then there exists o' such that $v' \sqsubseteq_F o'$ and $o \overset{F\,S(e)}{\leadsto} o'$.*

Given an abstract environment and an abstract store, we can use abstract reachability to compute a new reachmap. Consider an application $e = (\mathsf{let}\, x = y\, z \,\mathsf{in}\, e')$ and suppose that y can evaluate to $\lambda w.e''$. Recall that $R_{\text{pass}}(e, \lambda w.e'')$ approximates the set of values reachable from z and $\lambda w.e''$ at this call site. $R_{\text{return}}(e, \lambda w.e'')$ approximates the set of values reachable from the return value or from one of the locations that existed before the call and which may have been modified during the evaluation of e''. The following definition formalizes these intuitions.

Definition 4.2 (Collect) *$Collect(F, S) = \langle R_p, R_r\rangle$ where, for all $e = (\mathsf{let}\, x = y\, z \,\mathsf{in}\, e')$ and $\lambda w.e'' \in F(y)$:*

$$\begin{aligned} R_p(e, \lambda w.e'') &= \{o' \mid o \in (F(z) \cup \{\lambda w.e''\}) \text{ and } o \overset{F\,s_0}{\leadsto} o'\} \\ R_r(e, \lambda w.e'') &= \{o' \mid o \in (F(last(e'')) \cup R_p(e, \lambda w.e'')) \\ &\qquad \text{and } o \overset{F\,s_1}{\leadsto} o'\} \end{aligned}$$

where $s_0 = S(e)$ and $s_1 = S(last(e''))$.

Provided the reachmap R used in an R-safe analysis contains every value reachable according to *Collect*, the analysis is sound.

Theorem 4.3 (Analysis correctness) *If $\langle F, S, G, H\rangle$ is R-safe and $Collect(F, S) \leq R$ then $\langle F, S, G, H\rangle$ is sound.*

Proof. Let $A = \langle F, S, G, H\rangle$. We define a stronger version of $\sqsubseteq$ on states, $\Sigma \sqsubseteq \langle R, A\rangle$, that uses the reachmap and includes a requirement that the continuation agrees, $\kappa \sqsubseteq \langle R, A\rangle$. Using this stronger version we can prove that $\Sigma_0 \sqsubseteq \langle R, A\rangle$ and that if $\Sigma \sqsubseteq \langle R, A\rangle$, A is safe, and $\Sigma \longmapsto \Sigma'$ then $\Sigma' \sqsubseteq \langle R, A\rangle$. Soundness follows by induction on the number of steps in the transition. □

Together with the fact that smaller reachmaps yield more precise analyses, this theorem indicates that the most precise, safe analysis that we can compute is one in which $\langle R_{\text{pass}}, R_{\text{return}}\rangle = \mathit{Collect}(F, S)$. However, simply plugging this definition of R_{pass} and R_{return} into Figures 4 and 5 is unsound since the functional underlying the resulting constraint system is not monotonic. This is because we compute a solution to the constraint system by starting with an empty abstract environment and empty per-program-point store. We add elements to the environment and store as dictated by the constraints until all constraints are satisfied. As the abstract environment grows, the sets computed by R_{pass} and R_{return} grow, hence V_p, V_r, L_p, and L_r grow. However, V_s and L_s *shrink*, due to the use of set difference in their definition.

Fortunately, as observed in the previous section, we have one choice for the reachmap which is trivially sound: $R^{\top}$. By holding the reachmap fixed, we can compute a safe analysis which we can then use to compute a better reachmap. Hence we can use *MinSafe* and *Collect* to compute a succession of reachmaps and analyses of increasing precision. Define:

$$\begin{array}{rcl} R_0 & = & R^{\top} \\ R_{i+1} & = & \mathit{Collect}(F_i, S_i) \text{ where } \langle F_i, S_i, G_i, H_i\rangle = A_i \\ A_i & = & \mathit{MinSafe}(R_i) \end{array}$$

This algorithm initially assumes that every value is reachable at calls and returns, and computes a minimum $R^{\top}$-safe analysis. Next, it uses the results of the analysis to compute a new reachmap R_1, which can then be used to compute a minimum R_1-safe analysis, etc. The correctness of this algorithm follows from two facts. First, we show that the analysis becomes more precise at each step. By finiteness of A_i, the iteration will eventually terminate with

$$\begin{array}{rcl} \langle F_\infty, S_\infty, G_\infty, H_\infty\rangle & = & \mathit{MinSafe}(R_\infty) \\ \text{where } R_\infty & = & \mathit{Collect}(F_\infty, S_\infty). \end{array}$$

Second, we show that the analysis is safe at the end of each step. Hence, the algorithm yields sound results *even if it is not run to the fixed point.*

Theorem 4.4 (Algorithm correctness) *For all i:*

1. $A_{i+1} \leq A_i$,
2. $\mathit{Collect}(F_i, S_i) \leq R_i$, *and*
3. A_i *is* R_i*-safe,*

where $\langle F_i, S_i, G_i, H_i\rangle = A_i$.

Proof. That A_i is R_i-safe follows from the definition of *MinSafe*. The other conditions are proved by induction on i. The base case ($i = 0$) follows from the definition of *MinSafe* and the fact that $R_0 = R^{\top}$. For the inductive case, assume that $\mathit{Collect}(F_i, S_i) \leq R_i$. By monotonicity of *MinSafe* (which critically depends on using *Init* in formulating the constraints)

$$A_{i+1} = \mathit{MinSafe}(\mathit{Collect}(F_i, S_i)) \leq \mathit{MinSafe}(R_i) = A_i.$$

By monotonicity of *Collect*,

$$\mathit{Collect}(F_{i+1}, S_{i+1}) \leq \mathit{Collect}(F_i, S_i) = R_{i+1}.$$

□

5 Examples

In this section, we present a series of small examples that illustrate different properties of our analysis. Our description is based on the results computed by $\mathit{MinSafe}(R_\infty)$.

5.1 Alias constraints

Consider the following program:

```
let z = 13 in
let f = (λx. let y = λw.z in y) in
let g = (f 13) in
let a = (g z)
in a
```

Variable z is captured in the closure returned by the call (f 13). Prior to the call, the variable cardinality map binds z and f to single. To analyze the call, we partition the map according to the free variables reachable through f and 13. Since z occurs free in f, we have $V_p = \{\mathsf{z}\}$ and $V_s = \{\mathsf{f}\}$. We compute the cardinality map for the return point by joining the set V_r of variables reachable through the return value (λw.z) with the set V_s of variables accessible from the continuation. The variables reachable through the return value are $V_r = \{\mathsf{z}\}$. Thus, although z is captured in the returned closure, it is still single after the call. Hence, the analysis concludes $\mathsf{g.z} \overset{\mathsf{a}}{\equiv} \mathsf{z}$.

Higher-order languages allow multiple bindings for the same variable to exist simultaneously through first-class procedures. The analysis records such variables as multiple in the variable cardinality map at those program points where multiple bindings for the variable are accessible. Figure 6 defines a higher-order program that creates overlapping multiple bindings for the same variable; we add labels to relevant program points for clarity. Salient elements of the abstract environment and variable cardinality map are also shown. Note that the call to f binds x to f. Moreover, the outer call to x in f's body rebinds x to g. Thus, by constraint rule 5a in Figure 4, x must be multiple in g since the inner call to f (at label 6) supplies a new value for x, the procedure bound to g, that is closed over the current value of x. Thus, by rule 5b, x must be multiple in f as well. Since multiple bindings for y are not accessible simultaneously, y remains single.

5.2 Sensitivity to program transformations

Our framework is sensitive to program transformations that change how variables are bound, since such changes affect abstract reachability. Continuation-passing style (CPS) is an example of such a transformation.

Figure 7 presents a program written in direct style, and its CPS counterpart. In the direct-style program, z is reachable and single at the call to f. Moreover, since only a closed procedure is passed to f, V_p is empty and V_s contains z. Since the value returned by f is a closed procedure, V_r will not contain z, and so z remains single. In other words, although many instances of z may be simultaneously live, only one instance is ever reachable during any given call to f since z is never captured by a closure that flows back into the procedure.

```
let¹ f = λx y. let³ g = λa b.
                          (if p
                             then let⁴ h = (x a b) in h⁵
                             else b)
               in let⁶ j = (x g λv.0) in j⁷
in let² k = (f f λz.z) in k
```

$$\begin{array}{ll}
F(\mathsf{x}) = \{\lambda\mathsf{xy}, \lambda\mathsf{ab}\} & G(1) = [\,] \\
F(\mathsf{a}) = \{\lambda\mathsf{ab}\} & G(3) = [\mathsf{x} \mapsto \mathsf{multiple}, \mathsf{y} \mapsto \mathsf{single}] \\
F(\mathsf{b}) = \{\lambda\mathsf{v}\} & G(5) = [\mathsf{x} \mapsto \mathsf{multiple}, \mathsf{a} \mapsto \mathsf{single}, \mathsf{b} \mapsto \mathsf{single}, \mathsf{h} \mapsto \mathsf{single}] \\
F(\mathsf{y}) = \{\lambda\mathsf{v}, \lambda\mathsf{z}\} & G(7) = [\mathsf{x} \mapsto \mathsf{multiple}, \mathsf{y} \mapsto \mathsf{single}, \mathsf{g} \mapsto \mathsf{single}, \mathsf{j} \mapsto \mathsf{single}] \\
F(\mathsf{h}) = \{\lambda\mathsf{v}\} & \\
F(\mathsf{k}) = \{\lambda\mathsf{v}\} & G(2)|_{V_p} = [\,] \\
 & G(2)|_{V_s} = [\mathsf{f} \mapsto \mathsf{single}] \\
 & G(4)|_{V_p} = [\mathsf{x} \mapsto \mathsf{multiple}] \\
 & G(4)|_{V_s} = [\mathsf{a} \mapsto \mathsf{single}, \mathsf{b} \mapsto \mathsf{single}] \\
 & G(6)|_{V_p} = [\mathsf{x} \mapsto \mathsf{multiple}] \\
 & G(6)|_{V_s} = [\mathsf{y} \mapsto \mathsf{single}, \mathsf{g} \mapsto \mathsf{single}]
\end{array}$$

Figure 6: Flow and variable cardinality map for a higher-order program.

```
letrec f = λz. let h = if p
                        then f λx.x
                        else z
                   g = λx. (z x)
               in (g h)
```

```
letrec f = λk z. let k₁ = λv.
                            let g = λk₂ x. (z k₂ x)
                            in (g k v)
                 in if p
                    then (f k₁ (λk₃ x.(k₃ x)))
                    else (k₁ z)
```

Figure 7: CPS-style transformations can affect the precision of the analysis.

The situation is quite different in the CPS'ed program. After CPS, z is also reachable through k_1 because it is captured by the continuation. It must therefore be included in V_p at the call to f. Hence, by constraint rule 5a, z must be multiple within both k_1 and f.

5.3 Store constraints

The analysis propagates cardinality information for reference cells in much the same way as it does for variables, except that *strong updates* are permitted on the per-program point store as described by condition 4b in Figure 5. The following example demonstrates when the analysis may perform a strong update on a cell:

```
let f = (λx.λy.let z = ref x in (y z)) in
let g = (f 13) in
let h = (g λa.a)
in setref h 14
```

The strong update rule permits an analysis to discard the previous abstract value stored in h when analyzing the assignment setref h 14, provided that h's abstract value is a unique cell c and the store cardinality map indicates that c is single. Analogous to the treatment of variables, a cell is multiple at a program point if more than one instance of the cell is simultaneously reachable from that point. Although z, the variable bound to the cell returned by the call to g, escapes from its binding context (via the call (y z)), it is not captured by a procedure or any another cell reachable from the closure bound to g. Consequently, the cell bound to h is single. Since the flow map indicates that h's abstract value is {ref x}, an analysis can perform a strong update on the store at the program point following the assignment, disregarding the cell's original contents.

The less precise analysis $\mathit{MinSafe}(R^\top)$ treats ref x as multiple and hence cannot perform a strong update at the assignment. This is because prior to the call to g,

$$(\mathsf{ref\ x}) \in \mathit{RefsOf}(R^\top_{\mathrm{pass}}((\mathsf{g}\ \lambda\mathsf{a.a}),\ \lambda\mathsf{y})).$$

Using constraint rule 2 in Figure 5, the evaluation of g's body returns a new store cardinality map with ref x multiple.

In contrast, the following example illustrates when a strong update is not permissible even under our more precise notion of abstract reachability:

```
let f = (λz.let h = ref z in λx.h) in
let g = f 0 in
let h = f g
in . . .
```

The cell bound to h is reachable from the closure returned by the call to f. Thus, this cell is included in the store cardinality map during the evaluation of call f g. By constraint 2 in Figure 5, ref z must be therefore be multiple. Since ref z is multiple upon return from the call (by constraint 5b in Figure 5), strong updates cannot be performed on the abstract cell returned from any call to h.

6 Implementation

We have constructed a prototype implementation of our analysis for a subset of Scheme. Currently, the prototype only handles programs with fixed arity procedures, and performs single analysis on variables and explicit reference cells (accessed with `make-cell`, `cell-ref`, and `cell-set!`). The prototype does not yet compute single information for cons cells and vector elements. We describe the principles underlying the implementation, report some preliminary experimental results, and derive upper bounds for its complexity.

6.1 Description

We solve the constraints based on implementation techniques of Palsberg and Schwartzbach [13]. The $Init_F$ predicate is implemented by a flag for each expression e. This flag either indicates that the expression e is initialized, or it is a list of thunks that suspend the generation of those constraints that depend on $e \in Init_F$.

An abstract value a is represented by a vector of flags, indexed by *LamOcc* and *RefOcc*, and a list $a_1, \ldots, a_n$ of abstract values for which constraints $a \leq a_i$ have been generated. Again, the flag at index $o \in LamOcc \cup RefOcc$ either indicates that $o \in a$, or it is a list of thunks that generate constraints in the case that o is added to a. This mechanism implements constraints of the form "for all $o \in a$: $a_1 \leq a_2$" as in [13], but it generalizes to the conditional generation of arbitrary constraints (cf. item 5 of Figure 3).

An abstract store $S(e)$ is represented by a vector of abstract values and a list of procedures. The procedures implement the constraints $S(e) \leq S(e')$ in Figure 5. They are called for $S(e)$ whenever the domain of $S(e)$ is extended and usually generate a constraint on abstract values of the form $S(e)(r) \leq S(e')(r)$ for some $r \in RefOcc$.

A cardinality map is represented by a finite mapping and a list of procedures. All elements from the list of procedures are called whenever the cardinality map is updated. These procedures implement constraints like $H(e) \leq H(e')$ in Figure 5. An entry in a cardinality map is either single or multiple or a list of thunks to be executed whenever the cardinality map becomes defined at this point. This mechanism is used to implement constraints involving the operator $\bowtie$ and the strong update in item 4b of Figure 5.

6.2 Experiments

We used a set of benchmark programs ranging in size from 200 to 3000 lines to evaluate the effectiveness of our analysis. Figure 8 gives a short description of the benchmarks, and Table 1 shows the results of the must-alias analysis. The table reports the number of variables in the original program. The transformation to A-normal form introduces additional variables (up to 15 times the number of original variables) but the analysis classifies these variables as single everywhere. This is of course the expected result, but it is not entirely trivial, since the additional variables can be stored in the continuation of an application.

Each program was analyzed once with $R^\top$ and once to the fixed point; two iterations usually sufficed to reach the fixed point. The column single (resp. multiple) counts the number of variables that were classified as single (resp. multiple) at every program point. The column "both" counts the number of variables that have single at some program points and multiple at others. On average, the analysis detects that more than 93% of the variables are always single. Not surprisingly, the programs where all variables are single with just $R^\top$ (**Boyer**, **GPScheme**, **Nucleic-2**) are essentially first-order programs. The increase in the number of variables classified as "both" when moving from $R^\top$ to the fixed point (**Graphs**, **Splay**)is not an indication of nonmonotonicity. Rather, some variables that were identified as multiple by $R^\top$ were marked as "both" by the more precise analysis.

Boyer	is a term-rewriting theorem prover;
GPScheme	is an imperative implementation of genetic programming (from the Scheme repository);
Graphs	counts the number of directed graphs with a distinguished root and k nodes of out-degree at most two;
Lattice	enumerates the maps between two lattices;
Matrix	tests whether a random matrix $M \in \{+1, -1\}^{n \times n}$ is maximal under permutation and negation of rows and columns;
Maze	generates a random maze and finds a path;
Nbody	implements the Greengard multipole algorithm for computing gravitational forces on point masses distributed uniformly in a cube;
Nucleic-2	is a slightly modified version of a program to compute the spatial arrangement of atoms in nucleic acids;
Splay	implements splay trees;
Unify	is an imperative unification algorithm.

Figure 8: Benchmarks.

Table 2 shows the results of the store analysis. There are no results for **Graphs** and **Nucleic-2**, because they do not use explicit references. While **Nbody**, **Nucleic-2**, and **Matrix** use mutable vectors and records, our implementation currently does not compute single information for such objects. The second column of the table reports the number of abstract references. The third through fifth columns contain a summary of the results for a solution of the constraints with $R^\top$ while columns six through eight summarize the results for the complete analysis run to the fixed point. The third (resp. fourth) column reports the number of abstract references that are always classified as single (resp. multiple). The fifth column reports the number of abstract references that have both, single and multiple, occurrences. The same applies to columns six through eight.

Since the analysis is run after assignment conversion, the analysis has to "rediscover" that many cells are actually converted variables. This happens accurately for **Boyer**, **GPScheme**, and **Lattice**. All other programs use first-class references directly. The results are encouraging since few occurrences of abstract references are multiple.

In the future, we intend to extend the implementation to trace pairs, vectors, records, and variable-arity procedures. We expect further improvement of the accuracy of the results. Specifically, we expect it will be possible to analyze the single property for pairs and records and perform more strong updates.

6.3 Complexity

We now consider the asymptotic running time of the implementation. Let l be the number of λ-expressions, a be the number of applications, r be the number of ref-expressions,

program	#vars	$R^{\top}$ single	multiple	both	fixed point single	multiple	both
Boyer	45	45	0	0	45	0	0
GPScheme	125	125	0	0	125	0	0
Graphs	148	117	30	1	139	5	4
Lattice	65	50	14	1	65	0	0
Matrix	232	192	26	14	221	3	8
Maze	186	174	12	0	186	0	0
Nbody	419	303	112	4	416	0	3
Nucleic-2	180	180	0	0	180	0	0
Splay	271	217	48	6	252	0	19
Unify	46	43	3	0	43	3	0

Table 1: Results for must-alias analysis on variables.

program	#refs	$R^{\top}$ single	multiple	both	fixed point single	multiple	both
Boyer	6	4	2	0	6	0	0
GPScheme	20	2	18	0	20	0	0
Lattice	1	0	1	0	1	0	0
Matrix	1	1	0	0	1	0	0
Nbody	5	3	2	0	5	0	0
Splay	6	3	3	0	3	0	3
Unify	4	1	3	0	4	0	0

Table 2: Results for must-alias analysis on cells.

and v be the number of variables in a program P. We use these measures to express detailed running times for *MinSafe* and *Collect*.

Since *MinSafe* is computed by solving four sets of constraints ($Init_F$ and Figures 3, 4, and 5), we sum over the costs of solving each set of constraints. To bound the running time for solving a set of constraints, we multiply the number of constraints by the maximum amount of time it takes to satisfy any constraint. For example, the standard 0CFA control flow analysis consists of a collection of al constraints (every application could call every procedure), each of which could take time l to satisfy (l procedures could be propagated across the constraint). The running time is thus al^2, i.e., 0CFA is cubic in program size.

The number of $Init_F$ constraints is at most al by item 5 in the definition of $Init_F$. Each flag of the $Init_F$ constraints is propagated at most once. Hence, the total time to satisfy these initial constraints is al. The environment constraints in Figure 3 yield a similar analysis as that of 0CFA, except that $l + r$ values can be propagated across each constraint. Hence the environment constraints lead to a running time of $al(l + r)$. The time to solve the variable cardinality constraints in Figure 4 is dominated by rule 5, which can create alv constraints, each of which takes constant time to solve. The constraints in Figure 5 are also dominated by rule 5, which can cause the creation of alr constraints, since each store constraint can cause r constraints. Each constraint for the cell cardinality map can be solved in constant time. However, the constraints on the per-program-point store dominate the cost, because $l + r$ values can be propagated across each constraint. Hence the running time of *MinSafe* is bounded by

$$alv + alr(l + r),$$

i.e., *MinSafe* is quartic in program size.

The computation of *Collect* proceeds in two phases. First, for each application e we compute $\overset{F\,S(e)}{\rightsquigarrow}$, and for each $\lambda w.e$ we compute $\overset{F\,s}{\rightsquigarrow}$, where $s = S(last(e))$. Each is a standard transitive closure computation, and requires $(l + r)^3$, hence the total time is $(a + l)(l + r)^3$. Using the results of the transitive closures, for e and $\lambda w.e''$ in the domain of R_{pass} and R_{return}, we compute a union of at most $l + r$ sets, each of which can contain $l + r$ values. This requires $(l + r)^2$ time. Hence the asymptotic running time of *Collect* is bounded by

$$(a + l)(l + r)^3 + al(l + r)^2,$$

i.e., *Collect* is quartic in program size.

Both *MinSafe* and *Collect* include a quadratic component al that reflects a worst-case assumption about the precision of the control-flow analysis. This assumption is extremely pessimistic—practical implementations of control-flow analyses often exhibit nearly linear behavior. Hence we expect tuned implementations of our must-alias analysis to exhibit no worse than quadratic behavior. Further, the work to compute abstract reachability directly depends on the precision of the flow analysis. Hence the transitive closure computation in *Collect* will be faster with a more precise, polyvariant control-flow analysis.

The quadratic aspect of maintaining variables and reference cells at each program point is unavoidable. However, the bounds for this component are also extremely pessimistic, as the cardinality maps and abstract store will contain only those variables and cells that are abstractly reachable at each point.

6.4 Functional subset

By restricting our attention to the purely functional subset of our language, we significantly reduce the worst-case complexity of the algorithm. First, there is no need for a double fixed-point calculation, since their is no mutual dependency between environment constraints and the variable cardinality constraints. We can solve the environment constraints first, use the results to compute *Collect*, and then solve the cardinality constraints last. Second, the environment constraints *Collect* can be computed faster—in time al^2 and l^3 respectively. Hence the entire algorithm requires only cubic time.

7 Related work

There is a large body of work on may-alias analysis of dynamically allocated objects for higher-order languages. Indeed, standard control-flow analysis can be regarded as a simple may-alias analysis. Jones and Muchnick [10] used an allocation point to name all objects allocated at that point in a may alias analysis. They also introduced a per-program-point approximation to the heap [9, 10].

Several researchers have used notions of reachability in abstract heaps for storage optimizations, although none have considered its applicability for must-alias analysis. Ruggieri and Murtagh [15] compute sets of objects passed to and returned from procedures for lifetime analysis to enable stack allocation of objects on frames with greater lifetimes. Serrano and Feeley [16] present a similar idea for a higher-order language. Deutsch [4] uses abstract reachability to enable stack allocation and to determine when objects can be destructively updated. Like our work, Sestoft [17] uses reachability in order to determine when multiple instances of a variable could exist simultaneously. When only a single instance exists, the variable can be "globalized" and thus not be stored in a stack frame or closure record. Our notion of single is different from his insofar as we allow multiple instances to exist (possibly reachable from the stack), as long as only one is reachable from the current environment. Furthermore, Sestoft does not consider the applicability of his analysis to locations.

Chase et.al. [3] use a per-program point "storage shape graph" to analyze a language with mutable cons cells. They introduce the notion of "strong update" in this context. Unlike our system, they do not use a notion of reachability to improve precision, nor do they attempt to formalize the correctness of their analysis. Indeed, their analysis rules are extremely complex and informal—an important goal of this paper was to formally justify some of their work.

Altucher and Landi [1] present a must-alias analysis for C in order to improve def-use information. They claim their work is "the first attempt to specify must-alias information for names which represent a set of dynamically allocated locations". They use an extension of the Jones and Muchnick notion of object naming. Their notion of must-alias is incomparable to ours because an object name (i.e. allocation point) in their approach refers only to the most recent object allocated at that point. Hence, they still may be able to determine when multiple instances of an object are reachable, while we can not. Conversely, we may be able to determine a must alias that is not on the most recently allocated object from a given program point. Because their framework and source language is so different from ours, it is unclear if their approach could be adapted to a higher-order languages with first-class references.

Most compilers for higher-order languages perform local analyses to determine when variables in a closure are available at a call site [2, 11, 14]. When identified, these variables can be eliminated from the closure and accessed directly. Steckler and Wand [19] formalize the notion of availability of variables, and define a global analysis to compute availability as a postpass to control-flow analysis. They also show how to use the information to perform lightweight closure conversion. We believe their analysis is strictly less precise than ours because it does not incorporate notions of abstract reachability and it uses a weaker per-program-point notion of invariance set. For example, a closure which escapes from the binding context of its free variables and subsequently returns back into scope is not eligible for lightweight closure conversion even if there are no multiple instances of the closure's free variables extant at the return point.

Shivers [18] introduced the term "environment problem" to refer to the difficulties caused by multiple simultaneous instances of variables in performing optimization of higher-order languages based on control-flow analysis. To handle the environment problem, he defines an informal extension to his framework called "reflow analysis." He presents a special case of closure optimization called "lambda propagation" that can detect when all of the free variables of a lambda are available at a call site and avoid construction of the closure.

References

[1] Altucher, R. Z., and Landi, W. An extended form of must alias analysis for dynamic allocation. In *ACM Symposium on Principles of Programming Languages* (Jan. 1995), pp. 74–84.

[2] Appel, A. W. *Compiling with Continuations.* Cambridge University Press, 1992.

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

[4] Deutsch, A. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *ACM Symposium on Principles of Programming Languages* (Jan. 1990), pp. 157–168.

[5] Felleisen, M. *The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages.* PhD thesis, Indiana University, 1987.

[6] Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. The essence of compiling with continuations. In *ACM Conference on Programming Language Design and Implementation* (1993).

[7] Heintze, N. Set-based analysis of ML programs. In *ACM Conference on LISP and Functional Programming* (1994), pp. 306–317.

[8] Jagannathan, S., and Wright, A. K. Flow-directed inlining. In *ACM Conference on Programming Language Design and Implementation* (May 1996).

[9] Jones, N. D., and Muchnick, S. S. *Flow Analysis and Optimization of LISP-like Structures.* Prentice–Hall, 1981, ch. 4, pp. 102–131.

[10] Jones, N. D., and Muchnick, S. S. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *ACM Symposium on Principles of Programming Languages* (Jan. 1982), pp. 66–74.

[11] Kranz, D., Kelsey, R. A., Rees, J., Hudak, P., Philbin, J., and Adams, N. I. ORBIT: An optimizing compiler for Scheme. In *ACM SIGPLAN Symposium on Compiler Construction* (1986).

[12] Palsberg, J. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems 17*, 1 (Jan. 1995), 47–62.

[13] Palsberg, J., and Schwartzbach, M. I. Safety analysis versus type inference. *Information and Computation 118* (1995), 128–141.

[14] Rozas, G. J. Taming the Y operator. In *ACM Conference on LISP and Functional Programming* (San Francisco, California, June 1992), pp. 226–234.

[15] Ruggieri, C., and Murtagh, T. P. Lifetime analysis of dynamically allocated objects. In *ACM Symposium on Principles of Programming Languages* (Jan. 1988), pp. 285–293.

[16] Serrano, M., and Feeley, M. Storage use analysis and its applications. In *International Conference on Functional Programming* (May 1996).

[17] Sestoft, P. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, Department of Computer Science, University of Copenhagen, 1991.

[18] Shivers, O. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1991.

[19] Steckler, P. A., and Wand, M. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems 19*, 1 (Jan. 1997), 48–86.

A Additional rules for if and letrec

We provide a brief summary of the necessary extensions to add conditionals and recursive procedures to the framework. We extend the language with two new constructs, and extend *last* in the obvious way:

$$\begin{array}{rcl} e & ::= & \mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e' \\ & | & \mathsf{letrec}\ f = \lambda w.e''\ \mathsf{in}\ e' \end{array}$$

$$\begin{array}{l} \mathit{last}(\mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e') = \mathit{last}(e') \\ \mathit{last}(\mathsf{letrec}\ f = \lambda w.e''\ \mathsf{in}\ e') = \mathit{last}(e') \end{array}$$

Exact states remain the same, except that we add a new kind of value for recursive closures:

$$v \in \mathit{Val} = \ldots \cup (\mathit{VarOcc} \times \mathit{LamOcc} \times \mathit{Env})$$

We extend the exact transition relation with two rules to handle if, one rule to handle letrec, and one rule to handle application of recursive closures.

$$\begin{array}{l} [\mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e', \rho, \sigma, \kappa] \\ \quad \longmapsto [e_1, \rho, \sigma, \langle x, e', \rho\rangle : \kappa] \quad \text{when } \rho(x) = \mathtt{true} \\ [\mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e', \rho, \sigma, \kappa] \\ \quad \longmapsto [e_2, \rho, \sigma, \langle x, e', \rho\rangle : \kappa] \quad \text{when } \rho(x) = \mathtt{false} \\ [\mathsf{letrec}\ f = \lambda w.e''\ \mathsf{in}\ e', \rho, \sigma, \kappa] \\ \quad \longmapsto [e', \rho[f \mapsto \langle f, \lambda w.e'', \rho|_{FV(\lambda w.e'')}\rangle], \sigma, \kappa] \\ [\mathsf{let}\ x = y\ z\ \mathsf{in}\ e', \rho, \sigma, \kappa] \\ \quad \longmapsto [e'', \rho'[w \mapsto \rho(z)][f \mapsto \langle f, \lambda w.e'', \rho'\rangle], \sigma, \langle x, e', \rho\rangle : \kappa] \\ \quad \text{where } \rho(y) = \langle f, \lambda w.e'', \rho'\rangle \end{array}$$

We extend the constraints for Init_F in a straightforward manner.

1. if $(\mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e') \in \mathit{Init}_F$ then
   (a) if $\mathtt{true} \in F(x)$ then $e_1 \in \mathit{Init}_F$
   (b) if $\mathtt{false} \in F(x)$ then $e_2 \in \mathit{Init}_F$
   (c) if $\mathit{last}(e_1) \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$
   (d) if $\mathit{last}(e_2) \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$
2. if $(\mathsf{letrec}\ f = \lambda w.e''\ \mathsf{in}\ e') \in \mathit{Init}_F$ then $e' \in \mathit{Init}_F$.

We combine the extensions of Figures 3, 4, and 5 as a single set of constraints. As before, the constraints need hold only if $e \in \mathit{Init}_F$.

1. if e is $\mathsf{let}\ x = (\mathsf{if}\ y\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ \mathsf{in}\ e'$then
   (a) if $\mathtt{true} \in F(y)$ then
      i. $F(\mathit{last}(e_1)) \leq F(x)$
      ii. $G(\mathit{last}(e_1)) \bowtie [x \mapsto \mathsf{single}] \leq G(e')$
      iii. $S(\mathit{last}(e_1)) \leq S(e')$
      iv. $H(\mathit{last}(e_1)) \leq H(e')$
   (b) if $\mathtt{false} \in F(y)$ then
      i. $F(\mathit{last}(e_2)) \leq F(x)$
      ii. $G(\mathit{last}(e_2)) \bowtie [x \mapsto \mathsf{single}] \leq G(e')$
      iii. $S(\mathit{last}(e_2)) \leq S(e')$
      iv. $H(\mathit{last}(e_2)) \leq H(e')$
2. if e is $\mathsf{letrec}\ f = \lambda w.e''\ \mathsf{in}\ e'$ then
   (a) $\{\lambda w.e''\} \leq F(f)$
   (b) $G(e) \bowtie [f \mapsto \mathsf{single}] \leq G(e')$
   (c) $S(e) \leq S(e')$
   (d) $H(e) \leq H(e')$.