Abstract Models of Memory Management

Greg Morrisett Matthias Felleisen Robert Harper

January 1995
CMU–CS–95–110

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU–CS–FOX–95–01

This research was sponsored by the Defense Advanced Research Projects Agency, CSTO, under the title "The Fox Project: Advanced Development of Systems Software", ARPA Order No. 8313, issued by ESD/AVS under Contract No. F19628-91-C-0168.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: garbage collection,allocation,type theory,polymorphism

Abstract

Most specifications of garbage collectors concentrate on the low-level algorithmic details of *how* to find and preserve accessible objects. Often, they focus on bit-level manipulations such as "scanning stack frames," "marking objects," "tagging data," *etc.* While these details are important in some contexts, they often obscure the more fundamental aspects of memory management: *what* objects are garbage and *why*?

We develop a series of calculi that are just low-level enough that we can express allocation and garbage collection, yet are sufficiently abstract that we may formally prove the correctness of various memory management strategies. By making the heap of a program syntactically apparent, we can specify memory actions as rewriting rules that allocate values on the heap and automatically dereference pointers to such objects when needed. This formulation permits the specification of garbage collection as a relation that removes portions of the heap without affecting the outcome of the evaluation.

Our high-level approach allows us to compactly specify and prove correct a wide variety of memory management techniques, including standard trace-based garbage collectors (*i.e.*, the family of copying and mark/sweep collectors), generational collection, and type-based tag-free collection. Furthermore, since the definition of garbage is based on the *semantics* of the underlying language instead of the conservative approximation of inaccessibility, we are able to specify and formally prove the idea that type inference can be used to collect some objects that are accessible but never used.

1 Memory Safety

Advanced programming languages manage memory allocation and deallocation automatically. Automatic memory managers, or garbage collectors, significantly facilitate the programming process because programmers can rely on the language implementation for the delicate tasks of finding and freeing unneeded objects. Indeed, the presence of a garbage collector ensures *memory safety* in the same way that a type system guarantees *type safety*: no program in an advanced programming language will crash due to dangling pointer problems while pointer allocation, access, and deallocation is transparent. However, in contrast to type systems, memory management strategies and particularly garbage collectors rarely come with a compact formulation and a formal proof of soundness. Indeed, since garbage collectors work on the machine representations of abstract values, the very idea of providing a proof of memory safety sounds unrealistic given the lack of simple models of memory operations.

The recently developed syntactic approaches to the specification of language semantics by Felleisen and Hieb [11] and Mason and Talcott [23, 24] are the first execution models that are intensional enough to permit the specification of memory management actions and yet are sufficiently abstract to permit compact proofs of important properties. Starting from the λ_v-**S** calculus of Felleisen and Hieb, we design compact specifications of a number of memory management ideas and prove several correctness theorems.

The basic idea underlying the development of our garbage collection calculi is the representation of a program's run-time memory as a global series of syntactic declarations. The program evaluation rules allocate large objects in the global declaration, which represents the heap, and automatically dereference pointers to such objects when needed. As a result, garbage collection can be specified as any relation that removes portions of the current heap without affecting the result of a program's execution.

In Section 2, we present a small functional programming language, λgc, with a rewriting semantics that makes allocation explicit. In Section 3, we define a semantic notion of garbage collection for λgc and show that there is no *optimal* collection strategy that is computable. In Section 4, we specify the "free-variable" garbage collection rule which models tracing-based collectors including mark/sweep and copying collectors. We prove that the free-variable rule is correct and provide two "implementations" at the syntactic level: the first corresponds to a copying collector, the second to a generational one. In the context of generational collection, we also illustrate how mutable reference cells render the original generational collector unsound and how the problem can be fixed at an abstract level.

In Section 5, we show how the use of abstract syntax facilitated our work in the first few sections. Specifically, the abstract syntax of λgc implicitly carries the *shape* (size and pointer information) of allocated objects, which permits specifications of GC algorithms that ignore the problem of traversing the memory graph in the absence of guiding information. To illustrate how to refine λgc just enough to address this problem, we formalize so-called "tag-free" collection algorithms for explicitly-typed, monomorphic languages such as Pascal and Algol [7, 36, 8]. We show how to *reconstruct* necessary shape information about values from types during garbage collection. We are able to prove the correctness of the garbage collection algorithm by using a well known *type preservation* argument. We then sketch how the type-based garbage collection algorithms can be extended to a language with a polymorphic type system in the style of the Girard-Reynold's System F [13, 14, 30]. Our formulation leads to a fairly simple proof of the correctness of Tolmach's garbage collector for the Gallium ML compiler [34].

In Section 6, we justify our semantic definition of garbage by showing that Milner-style type inference can be used to prove that an object is semantically garbage even though the object is still reachable. While previous authors have sketched this idea [6, 3, 16, 12], we are the first to present a formal proof of this result. The proof is obtained by casting the well-known interpretation of types as logical relations into our framework.

Section 7 discusses related work and Section 8 closes with a summary.

Programs:

$$
\begin{array}{lrcll}
\text{(variables)} & x, y, z & \in & \mathit{Var} & \\
\text{(integers)} & i & \in & \mathit{Int} & ::= \; \cdots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \cdots \\
\text{(expressions)} & e & \in & \mathit{Exp} & ::= \; x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_1 \; e \mid \pi_2 \; e \mid \lambda x.e \mid e_1 \; e_2 \\
\text{(heap values)} & h & \in & \mathit{Hval} & ::= \; i \mid \langle x_1, x_2 \rangle \mid \lambda x.e \\
\text{(heaps)} & H & \in & \mathit{Heap} & ::= \; \{x_1 = h_1, \ldots, x_n = h_n\} \\
\text{(programs)} & P & \in & \mathit{Prog} & ::= \; \mathsf{letrec} \; H \; \mathsf{in} \; e \\
\text{(answers)} & A & \in & \mathit{Ans} & ::= \; \mathsf{letrec} \; H \; \mathsf{in} \; x
\end{array}
$$

Evaluation Contexts and Instruction Expressions:

$$
\begin{array}{lrcll}
\text{(contexts)} & E & \in & \mathit{Ctxt} & ::= \; [\,] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i \; E \mid E \; e \mid x \; E \\
\text{(instructions)} & I & \in & \mathit{Instr} & ::= \; h \mid \pi_i \; x \mid x \; y
\end{array}
$$

Figure 1: The Syntax of λgc

---

2 Modeling Allocation: λgc

Syntax: The syntax of λgc (see Figure 1) is that of a conventional, higher-order, applicative programming language based on the λ-calculus. Following the tradition of functional programming, a λgc program (P) consists of some mutually recursive definitions (H) and an expression (e). The global definitions are useful for defining mutually recursive procedures, but their primary purpose here is to represent the run-time *heap* of a program. In general, there can be cycles in a heap so we use letrec instead of let as the binding form. Expressions are either variables (x), integers (i), pairs ($\langle e_1, e_2 \rangle$), projections (π_i), abstractions ($\lambda x.e$), or applications ($e_1 \; e_2$).

$$
\begin{array}{rcl}
FV(i) & = & \emptyset \\
FV(x) & = & \{x\} \\
FV(\langle e_1, e_2 \rangle) & = & FV(e_1) \cup FV(e_2) \\
FV(\pi_i \; e) & = & FV(e) \\
FV(\lambda x.e) & = & FV(e) \setminus \{x\} \\
FV(e_1 \; e_2) & = & FV(e_1) \cup FV(e_2)
\end{array}
$$

$$
\begin{array}{rcl}
FV(\{x_1 = h_1, \ldots, x_n = h_n\}) & = & (FV(h_1) \cup \ldots \cup FV(h_n)) \setminus \{x_1, \ldots, x_n\} \\
FV(\mathsf{letrec} \; H \; \mathsf{in} \; e) & = & (FV(H) \cup FV(e)) \setminus \mathit{Dom}(H)
\end{array}
$$

Figure 2: Calculating Free Variables

Formally, the heap is a series of pairs, called *bindings*, consisting of variables and heap values. Heap values are a semantically significant subset of expressions. The order of the bindings is irrelevant. In addition, each variable must be bound to at most one heap value in a heap. Hence, we treat heaps as sets and, when convenient, as finite functions. We write $Dom(H)$ to denote the bound variables of H and $Rng(H)$ to denote the heap values bound in H. We sometimes refer to variables bound in the heap as *locations* or *pointers*.

The language contains two binding constructs: $\lambda x.e$ binds x in e and $\mathsf{letrec}\ H\ \mathsf{in}\ e$ binds the variables in $Dom(H)$ to the expressions $Rng(H)$ in both the values in $Rng(H)$ and in e. Following convention, we consider expressions to be equivalent up to a consistent α-conversion of bound variables. The free variables of expressions, heaps, and programs are determined by the definitions of Figure 2.

Considering programs equivalent modulo α-conversion and the treatment of heaps as sets instead of sequences hides many of the complexities of memory management. In particular, programs are automatically considered equivalent if the heap is re-arranged and locations are re-named as long as the "graph" of the program is preserved. This abstraction allows us to focus on the issues of determining what bindings in the heap are garbage without specifying how such bindings are represented in a real machine.

Mathematical Notation: We use $X \uplus X'$ to denote the union of two disjoint sets, X and X'. We use $H \uplus H'$ to denote the union of two heaps whose domains are disjoint. We use $\{e_1/x\}e_2$ to denote capture-avoiding substitution of the expression e_1 for the free variable x in the expression e_2. We use $X \setminus X'$ to denote $\{x \in X \mid x \notin X'\}$.

Semantics: The rewriting semantics for λgc is an adaptation of the standard reduction function of the λ_v-S calculus [11]. Roughly speaking, this kind of semantics describes an abstract machine whose states are programs and whose instructions are relations between programs. The desired final state of this abstract machine is an answer program (A) whose body is a pointer to some value, such as an integer, in the heap.

Each rewriting step of a program $\mathsf{letrec}\ H\ \mathsf{in}\ e$ proceeds according to a simple algorithm. If the body of the program, e, is not a variable, it is partitioned into an evaluation context E (an expression with a hole `[ ]` in the place of a sub-expression), which represents the control state, and an instruction expression I, which roughly corresponds to a program counter: $e = E[I]$. The instruction expression determines the next expression e' and any changes to the heap resulting in a new heap H'. Putting the pieces together yields the next program in the evaluation sequence: $\mathsf{letrec}\ H'\ \mathsf{in}\ E[e']$. Each instruction determines one transition rule of the abstract machine. Formally, a rule denotes a relation between programs. A set of rules denotes the union of the respective relations.

We use the following conventions: Let G be a set of program relations, and let P and P' be programs. Then,

> $P \stackrel{\mathrm{G}}{\longmapsto} P'$ means P rewrites to P' according to one of the rules in G and $\stackrel{\mathrm{G}}{\longmapsto}^*$ is the reflexive, transitive closure of $\stackrel{\mathrm{G}}{\longmapsto}$.
>
> $G + r$ is the union of G with the rule r.
>
> A program P is *canonical* with respect to G iff there is no rule in G and no P' such that $P \stackrel{\mathrm{G}}{\longmapsto} P'$.
>
> $P \Downarrow_G P'$ means that $P \stackrel{\mathrm{G}}{\longmapsto}^* P'$ and P' is canonical with respect to G.
>
> $P \Uparrow_G$ means that there exists an infinite sequence of programs P_i such that $P \stackrel{\mathrm{G}}{\longmapsto} P_1 \stackrel{\mathrm{G}}{\longmapsto} P_2 \stackrel{\mathrm{G}}{\longmapsto} \cdots$.

Figure 1 defines the set of evaluation contexts and instruction expressions for λgc. The definition of evaluation contexts (E) reflect the left-to-right evaluation order of the language. All terms to the left of the path from the root to the hole are variables; the terms on the right are arbitrary. Instruction expressions (I) consist of *heap* values (h), applications of (pointers to heap-allocated) procedures to (pointers to heap-allocated) values, and projections of (pointers to heap-allocated) tuples.

(**alloc**)	$\mathsf{letrec}\ H\ \mathsf{in}\ E[h] \xmapsto{\text{alloc}} \mathsf{letrec}\ H \uplus \{x = h\}\ \mathsf{in}\ E[x]$	
(**proj**)	$\mathsf{letrec}\ H\ \mathsf{in}\ E[\pi_i\ x] \xmapsto{\pi_i} \mathsf{letrec}\ H\ \mathsf{in}\ E[x_i]$	$(H(x) = \langle x_1, x_2\rangle$ and $i = 1, 2)$
(**app**)	$\mathsf{letrec}\ H\ \mathsf{in}\ E[x\ y] \xmapsto{\text{app}} \mathsf{letrec}\ H \uplus \{z = H(y)\}\ \mathsf{in}\ E[e]$	$(H(x) = \lambda z.e)$

Figure 3: Rewriting Rules for λgc

The evaluation rules for λgc (see Figure 3) reflect the intentions behind our choice of instruction expressions. The transition rule **alloc** models the allocation of values in the heap by binding the value to a new variable and using this variable in its place in the program. Note that the "$\uplus$" notation carries the implicit requirement that the newly allocated variable x cannot be in the domain of the heap H. The transition **proj** specifies how a projection instruction extracts the appropriate component from a pointer to a heap-allocated pair. Similarly, **app** is a transliteration of the conventional β-value rule into our modified setting. It binds the formal parameter of a heap-allocated procedure to the value of the pointer given as the actual argument, and places the expression part of the procedure into the evaluation context. Multiple applications of the same procedure require α-conversion to ensure that the formal parameter does not conflict with bindings already in the heap[1]. We use R to abbreviate the union of the rules **alloc**, **proj**, and **app**.

As a simple example of evaluation under R, consider rewriting the program $P = \mathsf{letrec}\ \{x = 1\}\ \mathsf{in}\ (\lambda y.\pi_1\ y)\langle x, x\rangle$:

$$\begin{array}{lll}
\mathsf{letrec}\ \{x = 1\}\ \mathsf{in}\ (\lambda y.\pi_1\ y)\langle x, x\rangle & \xmapsto{\text{alloc}} & \mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y\}\ \mathsf{in}\ z\ \langle x, x\rangle \\
 & \xmapsto{\text{alloc}} & \mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle\}\ \mathsf{in}\ z\ w \\
 & \xmapsto{\text{app}} & \mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle, y = \langle x, x\rangle\}\ \mathsf{in}\ \pi_1\ y \\
 & \xmapsto{\pi_1} & \mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle, y = \langle x, x\rangle\}\ \mathsf{in}\ x
\end{array}$$

We can break P into an evaluation context $E_0 = [\ ]\ \langle x, x\rangle$ and instruction $I_0 = \lambda y.\pi_1\ y$. Since I_0 is a heap value, **alloc** applies. We choose a new variable z, bind the heap value to z, and replace it with z in the hole of E_0, resulting in the program $\mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y\}\ \mathsf{in}\ z\ \langle x, x\rangle$. This program can be broken into evaluation context $E_1 = z\ [\ ]$ and instruction $I_1 = \langle x, x\rangle$. I_1 is also a heap value, so **alloc** applies, we choose a new variable w, bind the heap value to w, and replace the instruction with w, resulting in the program $\mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle\}\ \mathsf{in}\ z\ w$. This program can be broken into an empty context ($E_2 = [\ ]$) and instruction $I_2 = z\ w$. Since z is bound to a procedure and w is bound in the heap, **app** applies. The heap value that w is bound to ($\langle x, x\rangle$) is bound to the formal argument of the procedure (y) in the heap and the body of the procedure replaces the instruction, resulting in the program $\mathsf{letrec}\ \{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle, y = \langle x, x\rangle\}\ \mathsf{in}\ \pi_1\ y$. This program can be broken into an empty evaluation context ($E_3 = [\ ]$) and instruction $I_3 = \pi_1\ y$. Since y is bound to a pair, the **proj** rule applies and the instruction is replaced with the first

[1] An alternative rule for application substitutes the actual argument (y) for the formal (z) within e and performs no allocation. This rule is essentially equivalent to **app**, but the definition above simplifies the proofs of Section 6.

component of the pair, namely x, resulting in the answer program letrec $\{x = 1, z = \lambda y.\pi_1\ y, w = \langle x, x\rangle, y = \langle x, x\rangle\}$ in x.

The canonical programs of λgc are either answers or stuck programs. The latter correspond to machine states that result from the misapplication of primitive program operations or unbound variables.

Definition 2.1 (Stuck Programs) *A program is* stuck *if it is of one of the following forms:*

- letrec H in E[$\pi_i\ x$] $\quad (x \notin Dom(H)$ *or* $H(x) \neq \langle x_1, x_2\rangle)$ *or*
- letrec H in E[$x\ y$] $\quad (x$ *or* $y \notin Dom(H)$ *or* $H(x) \neq \lambda z.e)$.

All programs either diverge or evaluate to an answer or a stuck program. Put differently, the evaluation process defines a partial function from λgc programs to canonical programs [11, 37].

Theorem 2.2 *If P is a closed program, then there exists at most one P' such that $P \Downarrow_R P'$.*

Proof (sketch): The theorem relies on the following lemma which shows that at each evaluation step, the instruction expression is uniquely determined.

Lemma 2.3 (Unique Decomposition) *If $P =$ letrec H in e is a λgc program, then either P is stuck, P is an answer, or else there exists a unique evaluation context E and instruction I such that $e = E$[I].*

Proof (sketch): We show by induction on the structure of e that either e is a variable or there exists a unique E and I, such that $e = E$[I]. Here, we give one case of the induction as an example: suppose $e = (e_1\ e_2)$. There are four cases to consider. If e_1 is not a variable, then by induction, there exists an E', I, such that $e_1 = E'$[I], in which case we take $E = E'\ e_2$. If e_1 is a variable, then $e_1 = x$ for some x and $H(x) = \lambda z.e'$ else e would be stuck. Suppose e_2 is not a variable. Then by induction, $e_2 = E'$[I] and we can take $E = x\ E'$. Otherwise, $e_2 = y$ and we can take $E =$ [ ] and $I = (e_1\ e_2)$. Other cases follow similarly. □

The rest follows from the lemma in a straightforward manner, given that P is closed. □

3 A Semantic Definition of Garbage

Since the semantics of λgc makes the allocation of values explicit, including the implicit pointer dereferencing in the language, we can also define what it means to garbage collect a value in the heap and then analyze some basic properties. A binding $x = h$ in the heap of a program is garbage if removing the binding has no "observable" effect on running the program. In our case, we consider only integer results and non-termination to be observable.

Definition 3.1 (Kleene Equivalence) *$(P_1, G_1) \simeq (P_2, G_2)$ means $P_1 \Downarrow_{G_1}$ letrec H_1 in x where $H_1(x) = i$ if and only if $P_2 \Downarrow_{G_2}$ letrec H_2 in y and $H_2(y) = i$. If $G_1 = G_2 = R$, then we simply write $P_1 \simeq P_2$.*

A binding is *garbage* if removing it results in a program that is Kleene equivalent to the original program:

Definition 3.2 (Garbage) *Given a program $P =$ letrec $H \uplus \{x = h\}$ in e, the binding $x = h$ is* garbage *with respect to P iff $P \simeq$ letrec H in e.*

A *collection* of a program is the same program with some garbage bindings removed. An optimal collection of a program is a program with as many garbage bindings removed as possible.

Definition 3.3 (Collection, Optimal Collection) *The program* letrec H in e *is a* collection *of* $P =$ letrec $H \uplus H'$ in e *iff all bindings* $x = h$ *in* H' *are garbage with respect to* P. *The program* letrec H in e *is an* optimal collection *of* P *iff no binding in* H *is garbage with respect to* P.

Unfortunately, there can be no optimal garbage collector because determining whether a binding is garbage or not is undecidable.

Proposition 3.4 (Garbage Undecidable) *Determining if a binding is garbage in an arbitrary closed* λgc *program is undecidable.*

Proof: Since λgc is essentially an untyped λ-calculus, determining whether an arbitrary closed program $P =$ letrec H in e halts with an answer or not is undecidable. Assume that GC is an algorithm for determining whether or not some binding in the heap is garbage. Then, in particular, GC can tell whether or not $x = \langle x_1, x_2 \rangle$ is garbage in the program:

$$P' = \textsf{letrec } H \uplus \{x_1 = 1, x_2 = 2, x = \langle x_1, x_2 \rangle\} \textsf{ in } (\lambda y.\pi_1\ x)\ e$$

where x does not occur in $Dom(H)$. We will show that GC claims that the binding for x is garbage if and only if P terminates with an answer, which proves that GC cannot exist.

Assume GC claims the binding for x is garbage. Then, the program letrec H in $(\lambda y.\pi_1\ x)e$ is Kleene equivalent to P', which clearly implies that P' either diverges or gets stuck, because this program will always either diverge or get stuck. To see this, note that evaluation will become stuck when x is dereferenced by the projection operation. Given an infinite sequence of reduction steps for P', we can easily produce an infinite sequence of reduction steps for P, since this sequence must occur while evaluating e. Similarly, given a sequence of reduction steps which takes P' to a stuck program, we can produce a sequence of reduction steps that causes P to become stuck, because P' must become stuck while evaluating e, since x is still bound in P'. Consequently, if GC claims x is garbage, then P either diverges or gets stuck.

Now assume GC claims that the binding for x is not garbage. This implies the existence of a heap value h and variable z such that

$$P' \overset{\mathrm{R}}{\longmapsto}{}^{*} \textsf{letrec } H' \uplus \{x_1 = 1, x_2 = 2, x = \langle x_1, x_2 \rangle, z = h\} \textsf{ in } (\lambda y.\pi_1\ x)\ z$$

Otherwise, x would never be involved in a reduction step. Given this finite reduction sequence for P', it follows that $P \overset{\mathrm{R}}{\longmapsto}{}^{*}$letrec $H' \uplus \{z = h\}$ in z In short, the evaluation of P terminates with an answer given the assumption that x is not garbage.

Since the two cases show that GC claims x is not garbage if and only if P terminates with an answer, we have established the proposition. $\square$

4 Reachability-Based Garbage Collection

Since computing an optimal collection is undecidable, a garbage collection algorithm must conservatively approximate the set of garbage bindings. Most garbage collectors compute the *reachable* set of bindings in a program, given the variables in use in the current instruction expression and control state. All reachable bindings are preserved; the others are eliminated.

In Section 4.1 we present a general specification for reachability-based garbage collection and prove that it only collects true garbage. In Section 4.2 we give a high-level description of an algorithm based on tracing collection and prove that the algorithm satisfies our specification. In Section 4.3 we specialize our algorithm to one that takes advantage of a generational partition and in Section 4.4, we show how generational collection works in the presence of assignment.

4.1 The Free-Variable Rule

Following Felleisen and Hieb [11], reachability in λgc is formalized by considering free variables. The following "free-variable" GC rule describes bindings as garbage if there are no references to these bindings in the other bindings, nor in the currently evaluating expression:

$$(\mathbf{fv}) \qquad \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \overset{\text{fv}}{\longmapsto} \mathsf{letrec}\ H_1\ \mathsf{in}\ e \quad (Dom(H_2) \cap FV(\mathsf{letrec}\ H_1\ \mathsf{in}\ e) = \emptyset)$$

The **fv** rule is correct in that it only removes garbage, and thus computes valid collections. The keys to the proof of correctness of **fv** are a *postponement* lemma and a *diamond* lemma. The statements of these lemmas can be summarized by the diagrams of Figure 4 respectively, where solid arrows denote relations that are assumed to exist and dashed arrows denote relations that can be derived from the assumed relations.

$$\begin{tikzcd} P_1 \arrow[r, "\text{fv}"] \arrow[d, dashed, "\text{R}"'] & P_2 \arrow[d, "\text{R}"] \\ P_2' \arrow[r, dashed, "\text{fv}"'] & P_3 \end{tikzcd} \qquad \begin{tikzcd} P_1 \arrow[r, "\text{fv}"] \arrow[d, "\text{R}"'] & P_2 \arrow[d, dashed, "\text{R}"] \\ P_2' \arrow[r, dashed, "\text{fv}"'] & P_3 \end{tikzcd}$$

Figure 4: Postponement and Diamond Properties

Lemma 4.1 (Postponement) *If $P_1 \overset{\text{fv}}{\longmapsto} P_2 \overset{\text{R}}{\longmapsto} P_3$, then there exists a P_2' such that $P_1 \overset{\text{R}}{\longmapsto} P_2' \overset{\text{fv}}{\longmapsto} P_3$.*

Proof: The proof proceeds by case analysis on the elements of R:

alloc: Assume $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[h] \overset{\text{fv}}{\longmapsto} \mathsf{letrec}\ H_1\ \mathsf{in}\ E[h] \overset{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus \{x = h\}\ \mathsf{in}\ E[x]$. We only need to show that **fv** applies after performing the allocation:

$$\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[h] \overset{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus H_2 \uplus \{x = h\}\ \mathsf{in}\ E[x]$$

But this is clear since allocation only adds x to the locations of the resulting program, and x cannot be in $Dom(H_2)$ by the side-condition of the **alloc** rule. Consequently:

$$\mathsf{letrec}\ H_1 \uplus H_2 \uplus \{x = h\}\ \mathsf{in}\ E[x] \overset{\text{fv}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus \{x = h\}\ \mathsf{in}\ E[x]$$

proj: Assume $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[\pi_i\ x] \xmapsto{\text{fv}} \mathsf{letrec}\ H_1\ \mathsf{in}\ E[\pi_i\ x] \xmapsto{\pi_i} \mathsf{letrec}\ H_1\ \mathsf{in}\ E[x_i]$ Since **proj** applies, x must be bound to some pair $\langle x_1, x_2\rangle$ and furthermore, x must be in H_1 else **fv** does not apply. Thus we may swap the steps:

$$\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[\pi_i\ x] \xmapsto{\pi_i} \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[x_i] \xmapsto{\text{fv}} \mathsf{letrec}\ H_1\ \mathsf{in}\ E[x_i]$$

app: Assume $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[x\ y] \xmapsto{\text{fv}} \mathsf{letrec}\ H_1\ \mathsf{in}\ E[x\ y] \xmapsto{\text{app}} \mathsf{letrec}\ H_1 \uplus \{z = h\}\ \mathsf{in}\ E[e]$ Since **app** applies, x must be bound to $\lambda z.e$, y must be bound to h, and z cannot occur in the domain of H_1. Furthermore, both x and y must be bound in H_1. Performing the application first yields:

$$\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[x\ y] \xmapsto{\text{app}} \mathsf{letrec}\ H_1 \uplus H_2 \uplus \{z = h\}\ \mathsf{in}\ E[e]$$

The resulting program is well formed only if z is not already bound in H_2. This can be ensured by α-converting the program if necessary. Let x' be a variable that occurs in e. Then either x' is z or x' occurs free in $\lambda z.e$. In either case, x' is necessarily bound in $H_1 \uplus \{z = h\}$ in order for the **fv** rule to apply, thus x' is not bound in H_2. Consequently, we can take the following step:

$$\mathsf{letrec}\ H_1 \uplus H_2 \uplus \{z = h\}\ \mathsf{in}\ E[e] \xmapsto{\text{fv}} \mathsf{letrec}\ H_1 \uplus \{z = h\}\ \mathsf{in}\ E[e]$$

□

Lemma 4.2 (Diamond) *If $P_1 \xmapsto{\text{fv}} P_2$ and $P_1 \xmapsto{\text{R}} P_2'$, then there exists a P_3 such that $P_2 \xmapsto{\text{R}} P_3$ and $P_2' \xmapsto{\text{fv}} P_3$.*

Proof: Assume $P_1 = \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ E[I]$, $P_2 = \mathsf{letrec}\ H_1\ \mathsf{in}\ E[I]$, and $P_1 \xmapsto{\text{fv}} P_2$. We can easily show by case analysis on the elements of R that if $P_1 \xmapsto{\text{R}} P_2'$ where $P_2' = \mathsf{letrec}\ H_1 \uplus H_2 \uplus H_3\ \mathsf{in}\ E[e]$, for some H_3 and e, then $P_2' \xmapsto{\text{fv}} P_3$ and $P_2 \xmapsto{\text{R}} P_3$ where $P_3 = \mathsf{letrec}\ H_1 \uplus H_3\ \mathsf{in}\ E[e]$. □

With the Postponement and Diamond Lemmas in hand, it is straightforward to show that **fv** is a correct GC rule.

Theorem 4.3 (Correctness of fv) *If $P \xmapsto{\text{fv}} P'$, then P' is a collection of P.*

Proof: Let $P = \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e$ and $P' = \mathsf{letrec}\ H_1\ \mathsf{in}\ e$ such that $P \xmapsto{\text{fv}} P'$. We must show P evaluates to an integer value iff P' evaluates to the same integer. Suppose $P' \Downarrow_R \mathsf{letrec}\ H\ \mathsf{in}\ x$ and $H(x) = i$. By induction on the number of rewriting steps using the Postponement Lemma, we can show that $P \Downarrow_R \mathsf{letrec}\ H \uplus H_2\ \mathsf{in}\ x$ and clearly $(H \uplus H_2)(x) = H(x) = i$. Now suppose $P \Downarrow_R \mathsf{letrec}\ H\ \mathsf{in}\ x$ and $H(x) = i$. By induction on the number of rewriting steps using the Diamond Lemma, we know that there exists an H' such that $P' \Downarrow_R \mathsf{letrec}\ H'\ \mathsf{in}\ x$ and $\mathsf{letrec}\ H\ \mathsf{in}\ x \xmapsto{\text{fv}} \mathsf{letrec}\ H'\ \mathsf{in}\ x$. Thus, x must be bound in H' and since **fv** only drops bindings, $H'(x) = i$. □

This theorem shows that a single application of **fv** results in a Kleene equivalent program. A real implementation interleaves garbage collection with evaluation. The following theorem shows that adding **fv** to R preserves evaluation.

Theorem 4.4 *For all programs P, $(P, R) \simeq (P, R + \mathbf{fv})$.*

Proof: Clearly any evaluation under R can be simulated by $R+\mathbf{fv}$ simply by not performing any **fv** steps. Thus, if $P \Downarrow_R A$ then $P \Downarrow_{R+\mathbf{fv}} A$. Now suppose $P \Downarrow_{R+\mathbf{fv}}$ $\mathsf{letrec}\ H_1\ \mathsf{in}\ x_1$ and $H_1(x_1) = i$. Then there exists a finite rewriting sequence using $R + \mathbf{fv}$ as follows:

$$P \overset{\text{R+fv}}{\longmapsto} P_1 \overset{\text{R+fv}}{\longmapsto} P_2 \overset{\text{R+fv}}{\longmapsto} \cdots \overset{\text{R+fv}}{\longmapsto} \mathsf{letrec}\ H_1\ \mathsf{in}\ x_1$$

We can show by induction on the number of rewriting steps in this sequence, using the Postponement Lemma, that all **fv** steps can be performed at the end of the evaluation sequence. This provides us with an alternative evaluation sequence where all the R steps are performed at the beginning:

$$P \overset{\text{R}}{\longmapsto} P_1' \overset{\text{R}}{\longmapsto} P_2' \overset{\text{R}}{\longmapsto} \cdots \overset{\text{R}}{\longmapsto} P_n' \overset{\text{fv}}{\longmapsto} P_{n+1} \overset{\text{fv}}{\longmapsto} P_{n+2} \overset{\text{fv}}{\longmapsto} \cdots \overset{\text{fv}}{\longmapsto} \mathsf{letrec}\ H_1\ \mathsf{in}\ x_1$$

Since **fv** does not affect the expression part of a program and only removes bindings from the heap, $P_n' =$ $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ x$ for some H_2. Thus, $P \Downarrow_R$ $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ x$. Since $H_1(x) = i$, $(H_1 \uplus H_2)(x) = i$. □

4.2 The Free-Variable Tracing Algorithm

The free-variable GC rule is a *specification* of a garbage collection algorithm. It assumes some mechanism for partitioning the set of bindings into two disjoint pieces such that one set of bindings is unreachable from the second set of bindings and the body of the program. Real garbage collection algorithms need a deterministic mechanism for generating this partitioning. It is possible to formulate an abstract version of such a mechanism, the free-variable tracing algorithm, by lifting the ideas of mark-sweep and copying collectors to the level of program syntax.

We adopt the terminology of copying collection in the description of the free-variable tracing algorithm. We use two heaps and a set: a "from-heap" (H_f), a "scan-set" (S), and a "to-heap" (H_t). The from-heap is the set of bindings in the current program and the to-heap will become the set of bindings preserved by the algorithm. The scan-set records the set of variables reachable from the to-heap that have not yet been moved from the from-heap to the to-heap. The scan-set is often referred to as the "frontier."

The body of the algorithm proceeds as follows: A variable x is removed from S such that H_f has a binding for x. If no such locations are in S, the algorithm terminates. Otherwise, it *scans* the heap value h to which x is bound in the from-set H_f, looking for free variables. For each $y \in FV(h)$, it checks to see if y has already been forwarded to the to-set H_t. Only if y is not bound in H_t does it add the variable to the scan-set S. This ensures that a variable moves at most once from the from-heap to the scan-set.

Formulating the free-variable tracing algorithm as a rewriting system is easy. It requires only one rule that relates triples of from-sets, scan-sets, and to-sets:

$$\langle H_f \uplus \{x = h\}, S \uplus \{x\}, H_t\rangle \implies \langle H_f, S \cup (FV(h) \setminus (Dom(H_t) \uplus \{x\})), H_t \uplus \{x = h\}\rangle$$

Initially the free variables of the evaluation context and instruction expression, which correspond to the "roots" of a computation, are placed in S. Computing the free variables of the context represents the scanning of the "stack" of a conventional implementation while computing the free variables of the instruction expression corresponds to scanning the "registers." The initial tuple is re-written until we reach a state where no variable in the scan-set is bound in the from-heap. At this point, we have forwarded enough bindings to the to-heap. This leads to the following free-variable tracing algorithm rule:

$$(\mathbf{fva}) \qquad \begin{array}{c} \mathsf{letrec}\ H\ \mathsf{in}\ e \overset{\text{fva}}{\longmapsto} \mathsf{letrec}\ H'\ \mathsf{in}\ e \\ \text{if} \\ \langle H, FV(e), \emptyset\rangle \implies^* \langle H'', S, H'\rangle \text{ and } Dom(H'') \cap S = \emptyset \end{array}$$

Clearly, the algorithm always terminates since the size of the from-heap strictly decreases with each step. Furthermore, this new rewriting rule is a subset of the rule **fv**, which implies the correctness of the algorithm.

Theorem 4.5 *If* $P \overset{\text{fva}}{\longmapsto} P'$*, then* $P \overset{\text{fv}}{\longmapsto} P'$*.*

Proof: Let $P = \textsf{letrec}\ H\ \textsf{in}\ e$ be a λgc program. The first step is to prove the basic invariants of the garbage collection rewriting system. If $\langle H, FV(e), \emptyset\rangle \Longrightarrow^* \langle H_f, S, H_t\rangle$, then $H_f \uplus H_t = H$ and $FV(\textsf{letrec}\ H_t\ \textsf{in}\ e) = S$. Now let $P' = \textsf{letrec}\ H_1\ \textsf{in}\ e$ and suppose $P \overset{\text{fva}}{\longmapsto} P'$. Then,

$$\langle H_1 \uplus H_2, FV(e), \emptyset\rangle \Longrightarrow^* \langle H_2, S, H_1\rangle.$$

and $Dom(H_2) \cap S = \emptyset$. By the invariants, $FV(\textsf{letrec}\ H_1\ \textsf{in}\ e) = S$, so $Dom(H_2) \cap FV(P') = \emptyset$. Consequently, $P \overset{\text{fv}}{\longmapsto} P'$. □

If we require that a collection algorithm produce a *closed* program, then **fva** is an optimal collection algorithm. That is, if P is a closed program and $P \overset{\text{fva}}{\longmapsto} P'$, then P' has the fewest bindings needed to keep the program closed without affecting evaluation. Assuming each step in the free-variable tracing algorithm takes time proportional to the size (in symbols) of the heap object forwarded to the to-heap, the time cost of the algorithm is proportional to the amount of data preserved, not the total amount of data in the original heap.

4.3 Generational Garbage Collection

The free-variable tracing algorithm examines *all* of the reachable bindings in the heap to determine that a set of bindings may be removed. By carefully partitioning the heap into smaller heaps, a garbage collector can scan less than the whole heap and still free significant amounts of memory. A *generational partition* of a program's heap is a sequence of sub-heaps ordered in such a way that "older" generations never have pointers to "younger" generations.

Definition 4.6 (Generational Partition) *A* generational partition *of a heap* H *is a sequence of heaps* $H_1, H_2, \ldots, H_n$ *such that* $H = H_1 \uplus H_2 \uplus \cdots \uplus H_n$ *and for all* i *such that* $1 \le i < n$, $FV(H_i) \cap Dom(H_{i+1} \uplus H_{i+2} \uplus \cdots \uplus H_n) = \emptyset$*. The* H_i *are referred to as generations and* H_i *is said to be an older generation than* H_j *if* $i < j$*.*

Given a generational partition of a program's heap, a free-variable based garbage collector can eliminate a set of bindings in younger generations without looking at any older generations.

Theorem 4.7 (Generational Collection) *Let* $H_1, \ldots, H_i, \ldots, H_n$ *be a generational partition of the heap of* $P = \textsf{letrec}\ H\ \textsf{in}\ e$*. Suppose* $H_i = (H_i^1 \uplus H_i^2)$*, and* $Dom(H_i^2) \cap FV(\textsf{letrec}\ H_i^1 \uplus H_{i+1} \uplus \cdots \uplus H_n\ \textsf{in}\ e) = \emptyset$*. Then* $P \overset{\text{fv}}{\longmapsto} \textsf{letrec}\ (H \setminus H_i^2)\ \textsf{in}\ e$*.*

Proof: We must show that $Dom(H_i^2) \cap FV(\textsf{letrec}\ (H \setminus H_i^2)\ \textsf{in}\ e) = \emptyset$. Since $H_1, \cdots, H_n$ is a generational partition of H, for all j, $1 \le j < i$, $FV(H_j) \cap Dom(H_{j+1} \uplus \cdots \uplus H_n) = \emptyset$. Hence, $FV(H_1 \uplus \cdots \uplus H_{i-1}) \cap Dom(H_i^2) = \emptyset$. Now,

$$\begin{array}{rl}
& FV(\textsf{letrec}\ H \setminus H_i^2\ \textsf{in}\ e) \cap Dom(H_i^2) \\
= & (FV(H \setminus H_i^2) \cup FV(e)) \cap Dom(H_i^2) \\
= & (FV(H_1 \uplus \cdots \uplus H_{i-1}) \cup FV(H_i^1 \uplus \cdots \uplus H_n) \cup FV(e)) \cap Dom(H_i^2) \\
= & (FV(H_1 \uplus \cdots \uplus H_{i-1}) \cap Dom(H_i^2)) \cup ((FV(H_i^1 \uplus \cdots \uplus H_n) \cup FV(e)) \cap Dom(H_i^2)) \\
= & \emptyset \cup ((FV(H_i^1 \uplus \cdots \uplus H_n) \cup FV(e)) \cap Dom(H_i^2)) \\
= & FV(\textsf{letrec}\ H_i^1 \uplus \cdots \uplus H_n\ \textsf{in}\ e) \cap Dom(H_i^2) \\
= & \emptyset
\end{array}$$

$\square$

Generational collection is important for three practical reasons: First, evaluation of closed, pure λgc programs makes it easy to maintain generational partitions.

Theorem 4.8 (Generational Preservation for R) *Let $P = \mathsf{letrec}\ H\ \mathsf{in}\ e$ be a closed program. If $H_1, \ldots, H_n$ is a generational partition of H and $P \stackrel{\mathrm{R}}{\longmapsto} \mathsf{letrec}\ H \uplus H'\ \mathsf{in}\ e$, then $H_1, \ldots, H_n, H'$ is a generational partition of $H \uplus H'$.*

Proof: The **proj** rule does not modify the heap, so it clearly preserves the generational partition. If the **alloc** rule is applied, then $e = E[h]$ for some E and h and $H' = \{x = h\}$. Since P is closed, x cannot occur free in H. Thus for all i, $1 \leq i \leq n$, $FV(H_i) \cap Dom(H_{i+1} \uplus H_{i+2} \uplus \cdots \uplus H_n \uplus H') = \emptyset$. If the **app** rule is applied, then $e = E[x\ y]$ for some E, x, and y such that x is bound to $\lambda z.e'$ and y is bound to some h in H. Thus, $H' = \{z = h\}$ and since P is closed, z cannot occur free in H. Thus for all i, $1 \leq i \leq n$, $FV(H_i) \cap Dom(H_{i+1} \uplus H_{i+2} \uplus \cdots \uplus H_n \uplus H') = \emptyset$. $\square$

The second reason generational collection is important is that, given a generational partition, we can directly use the free-variable tracing algorithm to generate a collection of a program. The following rule invokes the free-variable algorithm on the program $\mathsf{letrec}\ H_2\ \mathsf{in}\ e$ where H_1, H_2 is a generational partition of the original program's heap. The resulting heap is glued onto H_1 to produce a collection of the program.

$$\textbf{(gen)} \qquad \begin{array}{c} \mathsf{letrec}\ H\ \mathsf{in}\ e \stackrel{\mathrm{gen}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus H_2'\ \mathsf{in}\ e \\ \text{if } \mathsf{letrec}\ H_2\ \mathsf{in}\ e \stackrel{\mathrm{fva}}{\longmapsto} \mathsf{letrec}\ H_2'\ \mathsf{in}\ e \\ \text{and } H_1, H_2 \text{ form a generational partition of } H \end{array}$$

The rule's soundness follows directly from the Generational Collection theorem, together with the soundness of the free-variable tracing algorithm.

Theorem 4.9 *If $P \stackrel{\mathrm{gen}}{\longmapsto} P'$, then $P \stackrel{\mathrm{fv}}{\longmapsto} P'$.*

The third reason generational collection is important is that empirical evidence shows that "objects tend to die young" [35]. That is, recently allocated bindings are more likely to become garbage in a small number of evaluation steps. Thus, if we place recently allocated bindings in younger generations we can concentrate our collection efforts on these generations, ignoring older generations, and still eliminate most of the garbage.

4.4 Assignment and Generational Garbage Collection

The addition of certain language features or implementation techniques such as assignment or lazy evaluation tends to break the Generational Preservation theorem. For instance, consider the addition of an assignment operator ($:=$) to λgc:

$$\begin{array}{lclcl} \text{(expressions)} & e & \in & \mathit{Exp} & ::= & \cdots \mid e_1 := e_2 \\ \text{(contexts)} & E & \in & \mathit{Ctxt} & ::= & \cdots \mid E := e \mid x := E \\ \text{(instructions)} & I & \in & \mathit{Instr} & ::= & \cdots \mid x := y \end{array}$$

$$\textbf{(set)} \qquad \mathsf{letrec}\ H \uplus \{x = h, y = h'\}\ \mathsf{in}\ E[x := y] \stackrel{\mathrm{set}}{\longmapsto} \mathsf{letrec}\ H \uplus \{x = h', y = h'\}\ \mathsf{in}\ E[x]$$

resulting in a language λgc-set. Evaluation of λgc-set programs under $R+$**set** does not satisfy the Generational Preservation theorem. To see this, consider the program:

$$\mathsf{letrec}\ \{x_0 = 0, x_1 = 1, x_2 = \langle x_0, x_0\rangle, x_3 = \langle x_2, x_2\rangle\}\ \mathsf{in}\ \pi_1\ (\pi_1\ (x_1 := x_3))$$

where the heap can be generationally partitioned into $H_1 = \{x_0 = 0, x_1 = 1\}$ and $H_2 = \{x_2 = \langle x_0, x_0\rangle, x_3 = \langle x_2, x_2\rangle\}$. Performing the assignment results in the program:

$$\textsf{letrec}\ \{x_0 = 0, x_1 = \langle x_2, x_2\rangle, x_2 = \langle x_0, x_0\rangle, x_3 = \langle x_2, x_2\rangle\}\ \textsf{in}\ \pi_1\ (\pi_1\ x_1)$$

but taking $H_1 = \{x_0 = 0, x_1 = \langle x_2, x_2\rangle\}$ and $H_2 = \{x_2 = \langle x_0, x_0\rangle, x_3 = \langle x_2, x_2\rangle\}$ no longer yields a generational partition. If we attempted to do a generational collection on H_2 using this partition, then both x_2 and x_3 would be collected since there is no reference to them in the expression part of the program. This would result in the program:

$$\textsf{letrec}\ \{x_0 = 0, x_1 = \langle x_2, x_2\rangle\}\ \textsf{in}\ \pi_1\ (\pi_1\ x_1)$$

which after one **proj** step would be stuck:

$$\textsf{letrec}\ \{x_0 = 0, x_1 = \langle x_2, x_2\rangle\}\ \textsf{in}\ \pi_1\ x_2$$

because x_2 is unbound. Yet, if no collection was performed, the computation would not get stuck and would return an answer of 0.

It is possible to maintain a generational partition for λgc-set during evaluation. We simply keep track of all "older" bindings that are updated and move them from the older generation to a younger generation. The following theorem formalizes this idea for two generations. The generalization to multiple generations is straightforward.

Theorem 4.10 (Generational Preservation for $R + \textbf{set}$) *If $P =$ letrec $H_1 \uplus H_2$ in e is a closed λgc-set program and $P \overset{\text{R+set}}{\longmapsto}{}^* P'$ where $P' =$ letrec $H_1 \uplus H_2' \uplus H_3$ in e' and $Dom(H_2) = Dom(H_2')$, then $H_1, (H_2' \uplus H_3)$ is a generational partition of P'.*

The theorem states that if we evaluate P for some number of steps resulting in P', where $H_1 \uplus H_2$ is conceptually the older generation, H_2 is the subset of the older generation that is modified via **set** to become H_2', and H_3 is the younger generation of newly allocated bindings, then H_1 and $(H_2' \uplus H_3)$ form a generational partition. There are a variety of techniques for determining which values in the older generation must be moved to the younger generation. We refer the interested reader to Hosking *et al.* [19].

5 Garbage Collection via Type Recovery

The delimiters and other tokens of the abstract syntax mark or "tag" heap values with enough information that we can distinguish pairs from functions, pointers from integers, *etc.*. This allows us to navigate through the memory unambiguously, but placing tags on heap values and stripping them off to perform a computation can impose a heavy overhead on the running time and space requirements of programs [33]. An alternative to tagging is the use of types to determine the shape of an object. If types are determined at compile time and evaluation maintains enough information that the types of reachable objects can always be recovered, then there is no need to tag values.

A number of researchers have made attempts to explore this alternative [7, 8, 3, 15, 26, 34, 2], but none of them presented concise characterizations of the underlying techniques with correctness proofs. In this section, we present the basic idea behind type-recovery based garbage collection. In the following subsection, we give a simple overview of the techniques used to record and reconstruct the types of objects in collectors such as Britton's [8] and Tolmach's [34]. We then introduce λgc-mono, an explicitly typed, monomorphic variant of λgc. We show how to adapt the free-variable tracing algorithm to recover types of objects in the heap and to use these types in the traversal of heap objects instead of abstract syntax. The proof of correctness for this garbage collection algorithm is given by extending the proof of soundness of the type system for λgc-mono. The last subsection extends the work to an explicitly typed, polymorphic language.

5.1 An Overview of Type Recovery

The type of a data structure such as a tuple or integer determines the types of the components of the data structure (if any), but the type of a function does not contain the types of the free variables of the function. If the compiler associates the types of all free variables with the code of a function, it becomes possible to recover the types of all reachable objects in the heap, given the set of reachable objects from the current evaluation context and instruction. This provides a potential for space savings compared to tagging all values, because type information, like code, can be shared among different applications of the same function.

For example, in a low-level model that uses closures (an explicit environment mapping variables to values paired with an expression) instead of substitution, the expression part or "code" of the closure can be shared. The type information needed to find and preserve the free variables of the closure is simply a type environment that lists the types of each value in the closure's environment. This information can also be shared because the types of the free variables are the same for each value environment paired with a given expression.

As noted earlier, determining the free variables of the evaluation context and instruction corresponds to scanning a stack and registers in a conventional implementation. Thus, the compiler must provide a type map for each stack frame. Like the type environment associated with a closure, this information can be recorded at compile time and a pointer to the information can be pushed on the stack as a new stack frame is allocated. The type information is "collected" as stack frames are popped.

In our high-level framework, we use a global heap instead of closures, rely upon α-conversion to generate unique names, and use evaluation contexts instead of an explicit stack to model evaluation. Consequently, the set of free variables changes as expressions are evaluated and α-conversion is performed. Thus, the type maps associated with functions and evaluation contexts must also change. Our approach is to represent these type maps implicitly by initially tagging each expression with its type and by allowing the substitution that occurs as part of α-conversion to replace a variable with a variable, but leave the type tag intact. The type map information associated with a closure or evaluation context can then be extracted by a simple function similar to the FV function of Figure 2. As data structures are allocated, the type information is stripped, leaving the heap essentially tag free and ensuring that we do not use the abstract syntax to navigate the heap. However, the type information remains intact within functions, corresponding to the explicit type maps needed for closures.

5.2 λgc-mono

λgc-mono is an explicitly typed, monomorphic variant of λgc. The set of types (τ) of λgc-mono contains the conventional basic types and constructed types for typing a functional programming language like λgc. The expressions of λgc-mono are the same as for λgc, except that each raw expression (u) is paired with some type information (see Figure 5). Heap values are not paired with their type, reflecting the fact that the memory is "almost tag free."

The evaluation contexts (E) consist of a raw context (U) and a type (τ). Raw contexts are filled with instructions (I) which are a subset of the raw expressions (u). The evaluation rules for λgc-mono, named RM, are variants of the rules for λgc that largely ignore the type information

Types:

$$\text{(types)} \quad \tau \in \mathit{Type} ::= \mathsf{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

Programs:

$$\begin{array}{llcll}
\text{(expressions)} & e & \in & \mathit{TExp} & ::= & (u : \tau) \\
 & u & \in & \mathit{UExp} & ::= & x \mid i \mid \langle e_1, e_2 \rangle \mid \pi_i\ e \mid \lambda x{:}\tau.e \mid e_1\ e_2 \\
\text{(heap values)} & h & \in & \mathit{Hval} & ::= & i \mid \langle x_1, x_2 \rangle \mid \lambda x{:}\tau.e \\
\text{(heaps)} & H & \in & \mathit{Heap} & ::= & \{x_1 = h_1, \ldots, x_n = h_n\} \\
\text{(programs)} & P & \in & \mathit{Prog} & ::= & \mathsf{letrec}\ H\ \mathsf{in}\ e \\
\text{(answers)} & A & \in & \mathit{Ans} & ::= & \mathsf{letrec}\ H\ \mathsf{in}\ (x : \tau)
\end{array}$$

Evaluation Contexts and Instructions:

$$\begin{array}{llcll}
\text{(contexts)} & E & \in & \mathit{TCtxt} & ::= & (U : \tau) \\
 & U & \in & \mathit{UCtxt} & ::= & [\,] \mid \langle E, e \rangle \mid \langle (x{:}\tau), E \rangle \mid \pi_i\ E \mid E\ e \mid (x{:}\tau)\ E \\
\text{(instructions)} & I & \in & \mathit{Instr} & ::= & i \mid \langle (x_1{:}\tau_1), (x_2{:}\tau_2) \rangle \mid \lambda x{:}\tau.e \mid \pi_1\ (x{:}\tau) \mid \pi_2\ (x{:}\tau) \mid (x{:}\tau_1)\ (y{:}\tau_2)
\end{array}$$

Figure 5: The Syntax of λgc-mono

on the sub-expressions of the program's body.

$$\begin{array}{ll}
\textbf{(alloc-int)} & \mathsf{letrec}\ H\ \mathsf{in}\ E[i] \stackrel{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H \uplus \{x = i\}\ \mathsf{in}\ E[x] \\
\textbf{(alloc-pair)} & \mathsf{letrec}\ H\ \mathsf{in}\ E[\langle x_1{:}\tau_1, x_2{:}\tau_2 \rangle] \stackrel{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H \uplus \{x = \langle x_1, x_2 \rangle\}\ \mathsf{in}\ E[x] \\
\textbf{(alloc-fn)} & \mathsf{letrec}\ H\ \mathsf{in}\ E[\lambda y{:}\tau.e] \stackrel{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H \uplus \{x = \lambda y{:}\tau.e\}\ \mathsf{in}\ E[x] \\
\textbf{(proj)} & \mathsf{letrec}\ H\ \mathsf{in}\ E[\pi_i\ (x{:}\tau)] \stackrel{\pi_i}{\longmapsto} \mathsf{letrec}\ H\ \mathsf{in}\ E[x_i] \quad (H(x) = \langle x_1, x_2 \rangle, i = 1, 2) \\
\textbf{(app)} & \mathsf{letrec}\ H\ \mathsf{in}\ E[(x{:}\tau_1)\ (y{:}\tau_2)] \stackrel{\text{app}}{\longmapsto} \mathsf{letrec}\ H \uplus \{z = H(y)\}\ \mathsf{in}\ E[u] \quad (H(x) = \lambda z{:}\tau'.(u{:}\tau''))
\end{array}$$

Allocation of integers, tuples and functions strips the type tag off of the heap value before placing it in the heap. Allocation of tuples also removes the tags from the components of the tuple. The removal of type information corresponds to the passage of a value from code to data and does not necessarily reflect a runtime cost. Projection and application are essentially the same as for λgc. Note that substitution of a result expression for an instruction occurs "in place," and hence the type of the instruction is ascribed to the expression.

The notion of a stuck state is adapted in accordance with the type structure of the language.

Definition 5.1 (λgc-mono Stuck Programs) *A program is stuck if it is of one of the following forms:*

- $\mathsf{letrec}\ H\ \mathsf{in}\ E[\pi_i\ (x{:}\tau)] \quad (x \notin \mathit{Dom}(H)$ *or* $H(x) \neq \langle x_1, x_2 \rangle)$*;*
- $\mathsf{letrec}\ H\ \mathsf{in}\ E[(x{:}\tau_1)\ (y{:}\tau_2)] \quad (x \notin \mathit{Dom}(H)$ *or* $H(x) \neq \lambda z{:}\tau.e$ *or* $y \notin \mathit{Dom}(H))$*.*

5.3 Static Semantics

The static semantics of λgc-mono consists of four judgements about program components. The first judgement, $\Gamma \triangleright e$, is a binary relation between a type assignment Γ, mapping a finite set of variables to types, and a typed expression e. Figure 6 (Expressions) contains the fairly conventional inference rules for expressions that generate the static semantics. Typing heaps and complete programs requires three additional judgements. The first is $\Gamma \triangleright h : \tau$ which asserts that the heap value h has type τ under the assumptions in Γ. The second is $\Gamma \triangleright H : \Gamma'$, which asserts that the variables

Expressions:

$$(\textbf{var})\ \frac{}{\Gamma \uplus \{x{:}\tau\} \triangleright (x : \tau)} \qquad (\textbf{int})\ \frac{}{\Gamma \triangleright (i : \mathsf{int})}$$

$$(\textbf{pair})\ \frac{\Gamma \triangleright (u_1{:}\tau_1) \qquad \Gamma \triangleright (u_2{:}\tau_2)}{\Gamma \triangleright (\langle (u_2{:}\tau_1), (u_2{:}\tau_2)\rangle : \tau_1 \times \tau_2)} \qquad (\textbf{proj})\ \frac{\Gamma \triangleright (u : \tau_1 \times \tau_2)}{\Gamma \triangleright (\pi_i\ (u : \tau_1 \times \tau_2) : \tau_i)} \quad (i = 1, 2)$$

$$(\textbf{fn})\ \frac{\Gamma \uplus \{x{:}\tau_1\} \triangleright (u : \tau_2)}{\Gamma \triangleright (\lambda x{:}\tau_1.(u{:}\tau_2) : \tau_1 \to \tau_2)} \qquad (\textbf{app})\ \frac{\Gamma \triangleright (u_1 : \tau_1 \to \tau_2) \qquad \Gamma \triangleright (u_2{:}\tau_1)}{\Gamma \triangleright ((u_1 : \tau_1 \to \tau_2)\ (u_2{:}\tau_1) : \tau_2)}$$

Heaps and Programs:

$$(\textbf{h-int})\ \frac{}{\Gamma \triangleright i : \mathsf{int}}$$

$$(\textbf{h-pair})\ \frac{\Gamma \triangleright (x_1{:}\tau_1) \qquad \Gamma \triangleright (x_2{:}\tau_2)}{\Gamma \triangleright \langle x_1, x_2\rangle : \tau_1 \times \tau_2} \qquad (\textbf{h-fn})\ \frac{\Gamma \triangleright (\lambda x{:}\tau_1.e : \tau_1 \to \tau_2)}{\Gamma \triangleright \lambda x{:}\tau_1.e : \tau_1 \to \tau_2)}$$

$$(\textbf{heap})\ \frac{\forall\, x \in \mathit{Dom}(\Gamma')\,.\,\Gamma \uplus \Gamma' \triangleright H(x) : \Gamma'(x)}{\Gamma \triangleright H : \Gamma'} \qquad (\textbf{prog})\ \frac{\emptyset \triangleright H : \Gamma \qquad \Gamma \triangleright e}{\triangleright\, \mathsf{letrec}\ H\ \mathsf{in}\ e}$$

Figure 6: The Static Semantics of λgc-mono

given type τ in Γ' are bound to heap values in H of the appropriate type, under the assumptions in Γ. The judgement's definition via the **heap** rule, similar to Harper's store typing [18] and the typing for Wright & Felleisen's Reference ML [37], requires "guessing" the types of the values in the heap and verifying these guesses simultaneously due to potential cycles. The third judgement, $\triangleright P$, asserts the well-typing of a complete program. Figure 6 (Heaps and Programs) contains the necessary inference rules for all of these judgements.

Definition 5.2 (Well Formed λgc-mono Programs) *A λgc-mono program P is well formed if $\triangleright P$ is derivable.*

The calculus λgc-mono is *type sound* in that evaluation of well formed programs cannot get stuck [37]. Our proof of soundness uses a *subject-reduction* based argument in the style of Wright and Felleisen [37].

Theorem 5.3 (Type Soundness) *If $\triangleright P$ then either P is an answer or else there exists some P' such that $P \overset{\mathrm{RM}}{\longmapsto} P'$ and $\triangleright P'$.*

Proof (sketch): The first two lemmas (below) are needed to prove properties about the allocation rule. The Weakening lemma allows us to throw extra information into a type assignment and still type check an expression; the Allocation lemma allows us to replace an expression with a variable, as long as that location is placed in the type assignment with the proper type.

Lemma 5.4 (Weakening) *If $\Gamma \triangleright e$ and $x \notin FV(e)$, then $\Gamma \uplus \{x{:}\tau\} \triangleright e$.*

Lemma 5.5 (Allocation) *If $\Gamma \triangleright (u : \tau)$ and $\Gamma \triangleright E[u]$ and $x \notin Dom(\Gamma)$, then $\Gamma \uplus \{x{:}\tau\} \triangleright E[(x{:}\tau)]$.*

The Unique Decomposition lemma allows us to deconstruct a program into a unique evaluation context and instruction expression.

Lemma 5.6 (Unique Decomposition) *If $P = \mathsf{letrec}\ H\ \mathsf{in}\ e$ has no free variables, then either P is stuck, P is an answer, or else there exists unique E and I such that $e = E[I]$.*

Lemma 5.7 (Stuck Programs Untypeable) *If P is a stuck program, then $\not\triangleright P$.*

Finally, the key lemma is Type Preservation, which asserts that if a program is well typed and it takes a step, then the resulting program is also well typed.

Lemma 5.8 (Type Preservation) *If $\triangleright P$ and $P \overset{\mathrm{RM}}{\longmapsto} P'$, then $\triangleright P'$.*

Proof (sketch): Follows from the Unique Decomposition, Weakening, and Allocation lemmas, together with an examination of each of the rules of RM. □

Given these lemmas, the proof of type soundness is straightforward. Suppose $\triangleright P$ and P does not diverge. Then there exists some P' such that $P \overset{\mathrm{RM}}{\longmapsto}{}^* P'$ and P' is canonical with respect to RM. Further suppose that $P' = \mathsf{letrec}\ H\ \mathsf{in}\ e$ for some H and e. If $e = (x{:}\tau)$, then by Type Preservation, $\triangleright\ \mathsf{letrec}\ H\ \mathsf{in}\ x{:}\tau$ and the theorem is satisfied. Otherwise, by the Unique Decomposition lemma, P' must be stuck and thus $\triangleright P'$ is not derivable, contradicting type preservation. □

5.4 Using Types in Garbage Collection

Specifying a valid garbage collection rule that exploits types is now straightforward. The key property that the rule must preserve is *type preservation.* That is, if $P \overset{\mathrm{gc}}{\longmapsto} P'$, and P is well typed, then P' must be well typed. One way to achieve this goal is to make it into a side-condition of the GC rule:

$$(\mathbf{mono}) \qquad \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \overset{\mathrm{mono}}{\longmapsto} \mathsf{letrec}\ H_1\ \mathsf{in}\ e \quad \text{if} \quad \triangleright\ \mathsf{letrec}\ H_1\ \mathsf{in}\ e$$

Theorem 5.9 *For all well formed λgc-mono programs P, $(P, RM) \simeq (P, RM + \mathbf{mono})$.*

Proof: Since **mono** preserves types, the type soundness proof trivially extends to the dynamic semantics with **mono**. This implies that, since P is well formed, P cannot get stuck under either system. Since bindings are only dropped and not updated by **mono**, the results of evaluating under either system must be the same. □

Like the free-variable rule of λgc, **mono** needs to be refined before it can serve as the basis for an implementation. The refinement is similar to the one from **fv** to **fva** but uses the types embedded in programs to traverse heap values. The basis for the collection algorithm is a function that determines a minimal, with respect to set-inclusion, type assignment Γ for any expression e such that $\Gamma \triangleright e$:

$$\begin{array}{rcl}
MTA(x{:}\tau) & = & \{x{:}\tau\} \\
MTA(i{:}\tau) & = & \emptyset \\
MTA(\langle e_1, e_2\rangle : \tau_1 \times \tau_2) & = & MTA(e_1) \cup MTA(e_2) \\
MTA(\pi_i\ e : \tau) & = & MTA(e) \\
MTA(\lambda x{:}\tau_1.e : \tau_1 \to \tau_2) & = & MTA(e) \setminus \{x{:}\tau_1\} \\
MTA(e_1\ e_2 : \tau) & = & MTA(e_1) \cup MTA(e_2)
\end{array}$$

Lemma 5.10 *If $\Gamma \triangleright e$, then $MTA(e) \triangleright e$ and $MTA(e) \subseteq \Gamma$.*

If $P = \textsf{letrec}\ H\ \textsf{in}\ e$ and P is closed, then $MTA(e)$ determines the types of the locations in the heap H that are immediately reachable from the expression e. A garbage collector can use this type information together with the following *Tag* function to traverse the reachable heap values based on their types instead of their abstract syntax.

$$\begin{array}{lcl} Tag[\textsf{int}](i) & = & (i{:}\textsf{int}) \\ Tag[\tau_1 \times \tau_2](\langle x_1, x_2\rangle) & = & (\langle (x_1{:}\tau_1), (x_2{:}\tau_2)\rangle : \tau_1 \times \tau_2) \\ Tag[\tau_1 \to \tau_2](\lambda x{:}\tau_1.e) & = & ((\lambda x{:}\tau_1.e) : \tau_1 \to \tau_2) \end{array}$$

Tag is a *curried* function that takes a type and then takes a heap value of that type, and annotates that heap value with enough information to turn it back into an expression. It is important to note that *Tag* pattern-matches and operates according to the type argument given and not the abstract syntax of the heap value given. *MTA* can be used on the resulting expression to find the minimal type assignment for the heap value. This provides us with the free locations and their types for the heap value. The following lemma summarizes the relationship between *Tag* and *MTA*:

Lemma 5.11 *If* $\emptyset \triangleright H : \Gamma \uplus \{x{:}\tau\}$ *then there exists an* h *such that*

1. $\{x = h\} \subseteq H$
2. $MTA(Tag[\tau](h)) \triangleright h : \tau$
3. $MTA(Tag[\tau](h)) \subseteq \Gamma \uplus \{x{:}\tau\}$

Equipped with *MTA*, we can now redefine the free-variable tracing algorithm so that it uses *Tag* to traverse heap values. The algorithm is specified in the same manner as **fva**, *i.e.*, as a rewriting system among tuples of the form $\langle H_f, \Gamma_s, H_t, \Gamma_t\rangle$ where H_f is the from-heap, Γ_s is a type assignment corresponding to the scan-set, H_t is the to-heap, and Γ_t is a type assignment for the to-heap:

$$\langle H_f \uplus \{x = h\}, \Gamma_s \uplus \{x{:}\tau\}, H_t, \Gamma_t\rangle \stackrel{\text{mono}}{\Longrightarrow} \langle H_f, \Gamma_s', H_t \uplus \{x = h\}, \Gamma_t'\rangle$$

$$\begin{array}{lrcl} \text{where} & \Gamma_s' & = & (\Gamma_s \cup MTA(Tag[\tau](h)) \setminus \Gamma_t' \\ & \Gamma_t' & = & \Gamma_t \uplus \{x{:}\tau\} \end{array}$$

The algorithm is initialized by taking the program heap H as the initial from-heap and the minimal type assignment of the program expression as Γ_s. At each step in the algorithm, Γ_s describes the types of all locations that are immediately reachable from e or H_t, but have not yet been forwarded to H_t. Γ_t describes the types of all locations that have been forwarded to H_t.[2]

When a variable x is found in Γ_s with type τ and x is bound in H_f to the heap value h, the collector forwards the binding $x = h$ to H_t and adds $x{:}\tau$ to Γ_t. It then uses $Tag[\tau]$ to traverse h, placing the necessary type information on the components so that *MTA* can determine the heap value's minimal type assignment. This step provides the locations (and their types) that are immediately reachable from h. Finally, the collector adds each of these locations to Γ_s unless they have already been forwarded to H_t.

Using this algorithm instead of an *a priori* partitioning of the heaps, **mono-a** becomes a high-level specification of a collector whose traversal of heap values uses types instead of tag information:

$$(\textbf{mono-a}) \quad \begin{array}{c} \textsf{letrec}\ H\ \textsf{in}\ e \stackrel{\text{mono-a}}{\longmapsto} \textsf{letrec}\ H'\ \textsf{in}\ e \\ \text{if} \\ \langle H, MTA(e), \emptyset, \emptyset\rangle \stackrel{\text{mono}}{\Longrightarrow}{}^{*} \langle H'', \emptyset, H', \Gamma\rangle \end{array}$$

The following definition gives the primary invariants of the algorithm:

[2]The garbage collection rewriting system only maintains Γ_t to simplify the presentation and proof; an implementation will not have to construct Γ_t.

Definition 5.12 (mono-a Invariants) $\langle H_f, \Gamma_s, H_t, \Gamma_t\rangle$ *has the* **mono-a** *invariants with respect to a program* letrec H in e *and type assignment* Γ_0 *iff:*

1.	$H = H_f \uplus H_t$	*(each binding is either in the from-heap or to-heap.)*
2.	$\Gamma_s \uplus \Gamma_t \rhd e$	*(every binding needed for e is in the scan-set or to-heap.)*
3.	$Dom(\Gamma_s) \subseteq Dom(H_f)$	*(the scan-set corresponds to bindings in the from-heap.)*
4.	$\Gamma_s \rhd H_t : \Gamma_t$	*(the scan-set holds all free variables in the to-heap.)*
5.	$\Gamma_s \uplus \Gamma_t \subseteq \Gamma_0$	*(the scan-set and to-heap agree with* Γ_0*.)*

Lemma 5.13 *If* $\emptyset \rhd H : \Gamma_0$*,* T *has the* **mono-a** *invariant properties with respect to* P *and* Γ_0*, and* $T \stackrel{\text{mono}}{\Longrightarrow} T'$*, then* T' *has the* **mono-a** *invariant properties with respect to* P *and* Γ_0*.*

Proof: Assume $P =$ letrec H in e and $T = \langle H_f \uplus \{x = h\}, \Gamma_s \uplus \{x{:}\tau\}, H_t, \Gamma_t\rangle$ and this rewrites to $T' = \langle H_f, \Gamma'_s, H_t \uplus \{x = h\}, \Gamma'_t\rangle$ where $\Gamma'_t = \Gamma_t \uplus \{x{:}\tau\}$ and $\Gamma'_s = (\Gamma_s \cup MTA(Tag[\tau](h))) \setminus \Gamma'_t$. We must show:

1. $H = H_f \uplus H_t \uplus \{x = h\}$: Follows directly from the first invariant for T.

2. $\Gamma'_s \uplus \Gamma'_t \rhd e$: Follows directly from the second invariant for T together with weakening.

3. $Dom(\Gamma'_s) \subseteq Dom(H_f)$: This follows from the third invariant if $(MTA(Tag[\tau](h)) \setminus \Gamma'_t) \subseteq Dom(H_f)$. This in turn follows from $\rhd H : \Gamma_0$, $\{x{:}\tau\} \subseteq \Gamma_0$, invariant 5 for T, and Lemma 5.11.

4. $\Gamma'_s \rhd H_t \uplus \{x = h\} : \Gamma'_t$: This follows from the fourth invariant of T and the **heap** rule of the static semantics if we can show $(\Gamma_s \cup (MTA(Tag[\tau](h)) \setminus \Gamma'_t)) \uplus \Gamma'_t \rhd h : \tau$. This in turn follows from $\{x{:}\tau\} \subseteq \Gamma_0$ and Lemma 5.11.

5. $\Gamma'_s \uplus \Gamma'_t \subseteq \Gamma_0$: This follows from the fifth invariant of T if $MTA(Tag[\tau](h)) \subseteq \Gamma_0$. This in turn follows from $\{x{:}\tau\} \subseteq \Gamma_0$ and Lemma 5.11.

□

Lemma 5.14 *If* $\emptyset \rhd H : \Gamma_0$*,* $\Gamma_0 \rhd e$*, and* $\langle H, MTA(e), \emptyset, \emptyset\rangle \stackrel{\text{mono}}{\Longrightarrow} {}^*T$*, then* T *has the* **mono-a** *invariant properties with respect to* H *and* Γ_0*.*

Proof: By induction on the length of the rewriting sequence $\langle H, MTA(e), \emptyset, \emptyset\rangle \stackrel{\text{mono}}{\Longrightarrow}{}^*T$. The base case directly follows from Lemma 5.10. The inductive step directly follows from Lemma 5.13.
□

The correctness of the algorithmic type-based garbage collection rule can now easily be verified.

Theorem 5.15 *If* P *is a well formed program and* $P \stackrel{\text{mono-a}}{\longmapsto} P'$*, then* $P \stackrel{\text{mono}}{\longmapsto} P'$*.*

Proof: We must verify that the **mono-a** garbage collection rule preserves typability in order for **mono** to apply. That is, we must show that if P is a well formed program, and $P \stackrel{\text{mono-a}}{\longmapsto} P'$, then $\rhd P'$.

Let $P =$ letrec H in e and suppose $\rhd P$. Then, for some Γ_0, $\emptyset \rhd H : \Gamma_0$ and $\Gamma_0 \rhd e$. If

$$\langle H, MTA(e), \emptyset, \emptyset\rangle \stackrel{\text{mono}}{\Longrightarrow}{}^*\langle H'', \emptyset, H', \Gamma_t\rangle,$$

then by Lemma 5.14 taking $\Gamma_s = \emptyset$, we know that $\emptyset \rhd H' : \Gamma_t$ (invariant 4) and $\Gamma_t \rhd e$ (invariant 2) . Thus, by the **prog** rule of the static semantics, $\rhd$ letrec H' in e. □

Furthermore, the algorithm never gets stuck, so it always applies:

Theorem 5.16 *If P is a well formed program, then there exists a program P' such that $P \overset{\text{mono-a}}{\longmapsto} P'$.*

Proof: If the algorithm takes a step, the size of the from-heap strictly decreases, so we know the collection either terminates or gets stuck. Here we show that the collection cannot get stuck, so it must terminate. Suppose $P = \textsf{letrec } H \textsf{ in } e$ and $\langle H, MTA(e), \emptyset, \emptyset \rangle \overset{\text{mono}}{\Longrightarrow}{}^{*} \langle H_f, \Gamma_s \uplus \{x{:}\tau\}, H_t, \Gamma_t \rangle$. By Lemma 5.14, we know that $x \in Dom(H_f)$, since the domain of the scan type-assignment is a subset of the from-heap. Consequently, there exists an h such that $H_f = H'_f \uplus \{x = h\}$ and $\langle H_f, \Gamma_s \uplus \{x{:}\tau\}, H_t, \Gamma_t \rangle \overset{\text{mono}}{\Longrightarrow}{}^{*} \langle H'_f, \Gamma'_s, H_t \uplus \{x = h\}, \Gamma'_t \rangle$ with appropriate Γ'_t and Γ'_s. □

The **mono-a** rule does not perform type *inference* but rather type *recovery.* That is, the types of values in the heap are recovered without unification. This was accomplished by making the types of subexpressions explicit in the program. We have traded tags on data structures such as pairs, with tags on sub-expressions of functions. However, the tags can be shared among functions that use the same code, differing only in their environment. Furthermore, some space savings can be realized for *parametric* data structures such as lists. For example, if we have $x : (\textsf{int} \times \textsf{int})\ \textsf{list}$, as a subexpression, x may be bound to a large list, but each of the components of the list have the same type and consequently share the same "type tag" in an expression. A true comparison between tag-oriented and type-oriented garbage collection will have to be based on implementations.

5.5 Explicit Polymorphism

Extending type-recovery based garbage collection to an explicitly polymorphic language where types are passed to polymorphic routines at run-time is straightforward. The language of the extended calculus, λgc-poly, is an adaptation of the Girard-Reynolds polymorphic λ-calculus. The extended language of types contains type variables and quantified types:

$$\begin{array}{llll} \text{(type variables)} & t \in & Tvar & \\ \text{(types)} & \tau \in & Type & ::= \ \cdots \mid t \mid \forall t.\tau \end{array}$$

The marker $\forall$ binds a type variable t in a type expression τ. Types that only differ in their choice of bound type variables are considered to be equivalent. $FTV(\tau)$ denotes the free type variables of type τ.

The syntactic categories for λgc-poly are the same as for λgc-mono, except for the following additions:

$$\begin{array}{lllll} u & \in & \text{untyped expr} & ::= & \cdots \mid \Lambda t.e \mid e\,[\tau] \\ h & \in & \text{heap values} & ::= & \cdots \mid \Lambda t.e \end{array}$$

The Λ-expression, or type abstraction, is a construct for introducing and scoping polymorphism; type application $(e\,[\tau])$ utilizes type abstractions at concrete types.

The set of evaluation contexts for λgc-poly is the same as for λgc-mono, modulo the extended syntax. In particular, there is no evaluation context within Λ-abstractions, because type abstracted expressions are allocated heap values. The one-step rewriting rules are the same as for λgc-mono with the addition of two new rules, one for allocating Λ-abstractions and one for governing the application of Λ-abstractions:

$$\begin{array}{lll} \textbf{(alloc-t)} & \textsf{letrec } H \textsf{ in } E[\Lambda t.e] \overset{\text{alloc}}{\longmapsto} \textsf{letrec } H \uplus \{x = \Lambda t.e\} \textsf{ in } E[x] & \\ \textbf{(type-app)} & \textsf{letrec } H \textsf{ in } E[(x{:}\tau)\,[\tau']] \overset{\text{t-app}}{\longmapsto} \textsf{letrec } H \textsf{ in } E[\{\tau'/t\}u] & (H(x) = \Lambda t.(u : \tau'')) \end{array}$$

As with other heap values, type abstracted expressions are allocated on the heap. Type application $((x{:}\tau)\,[\tau'])$ replaces all free occurrences of the bound type variable in a Λ-expression with a type. A compiler for this language might represent Λ-expressions in the same way that it represents

λ-abstractions and treat type application the same as term application. Since types are passed to type abstractions during evaluation, we say that this semantics is a *type-passing* interpretation. We refer to the extended set of rules as RP.

The static semantics for λgc-poly defines the following judgements:

1.	$\Delta \rhd \tau$	(type τ is well formed.)
2.	$\Delta \rhd \Gamma$	(type assignment Γ is well formed.)
3.	$\Delta; \Gamma \rhd e$	(expression e is well formed.)
4.	$\Delta; \Gamma \rhd h : \tau$	(heap value h is well formed.)
5.	$\Delta; \Gamma \rhd H : \Gamma'$	(heap H is well formed.)
6.	$\rhd P$	(program P is well formed.)

where Δ is a set of type variables and Γ is a type assignment. The first judgement asserts that the type τ is well formed with respect to Δ in that the free type variables of τ occur in Δ. The second judgement asserts that Γ is well formed with respect to Δ, in that all of the types in the range of Γ are well formed with respect to Δ. The third judgement asserts that an expression is well formed. The judgement is defined in the same fashion as the expression-level typing judgement for λgc-mono, except that Γ is required to be well formed with respect to Δ, and the resulting type must also be well formed with respect to Δ. The following rules are added for Λ-abstractions and type applications respectively:

$$\frac{\Delta \rhd \Gamma \qquad \Delta \uplus \{t\}; \Gamma \rhd (u{:}\tau)}{\Delta; \Gamma \rhd (\Lambda t.(u{:}\tau) : \forall t.\tau)} \ (t \notin \Delta) \qquad \frac{\Delta; \Gamma \rhd (u : \forall t.\tau') \qquad \Delta \rhd \tau}{\Delta; \Gamma \rhd ((u : \forall t.\tau')\,[\tau] : \{\tau/t\}\tau')}$$

Similarly, a rule is added to typecheck a Λ-abstraction as a heap value:

$$\frac{\Delta \rhd \Gamma \qquad \Delta \uplus \{t\}; \Gamma \rhd (u{:}\tau)}{\Delta; \Gamma \rhd \Lambda t.(u{:}\tau) : \forall t.\tau}$$

The last two judgements assert that heaps and programs are well formed and are defined by the following inference rules:

$$\frac{\forall\, x \in \mathit{Dom}(\Gamma')\,.\,\Delta; \Gamma \uplus \Gamma' \rhd H(x) : \Gamma'(x)}{\Delta; \Gamma \rhd H : \Gamma'} \qquad \frac{\emptyset; \emptyset \rhd H : \Gamma \qquad \Gamma \rhd e}{\rhd \mathsf{letrec}\ H\ \mathsf{in}\ e}$$

A program is well formed if the heap and expression each type check with an empty Δ. Consequently, there can be no free type variables in a well formed program.

As for λgc-mono, evaluation preserves typing, so if a program is well formed and then takes some number of steps, the resulting program will have no free type variables. Furthermore, the type system for λgc-poly is sound:

Theorem 5.17 (Type Soundness) *If $\rhd P$ then either P is an answer or else there exists a P' such that $P \overset{\mathrm{RP}}{\longmapsto} P'$ and $\rhd P'$.*

The type-based free-variable algorithm can be adapted for λgc-poly as follows. First, MTA and *Tag* are extended to work with the new language constructs:

$$\begin{aligned} MTA(\Lambda t.e) &= MTA(e) \\ MTA(e\,[\tau]) &= MTA(e) \end{aligned}$$

$$\mathit{Tag}[\forall t.\tau](\Lambda t.e) = (\Lambda t.e : \forall t.\tau)$$

The garbage collection algorithm itself remains unchanged:

$$\langle H_f \uplus \{x = h\}, \Gamma_s \uplus \{x{:}\tau\}, H_t, \Gamma_t\rangle \stackrel{\text{poly}}{\Longrightarrow} \langle H_f, \Gamma'_s, H_t \uplus \{x = h\}, \Gamma'_t\rangle$$

$$\text{where} \qquad \begin{array}{lcl} \Gamma'_s & = & (\Gamma_s \cup \mathit{MTA}(\mathit{Tag}[\tau](h)) \setminus \Gamma'_t \\ \Gamma'_t & = & \Gamma_t \uplus \{x{:}\tau\} \end{array}$$

To prove the algorithm does not get stuck, we must show that we never find a variable in Γ_s that is assigned an unknown type (t). But this follows from the well-typedness of the original program. Consequently, it is straightforward to adapt the proofs of termination and correctness of **mono-a** to show that the rule **poly-a** is correct and always applies.

6 Collecting Reachable Garbage Using Type Inference

So far, we have only considered specifications and algorithms for collecting unreachable bindings. In this section, we show that by using type inference during the garbage collection process, some bindings that *are* reachable can still be safely collected. That is, type inference can be used to prove that an object is garbage even though it is reachable.

As a simple example, consider the following λgc program:

$$\mathsf{letrec}\ \{x_1 = 1, x_2 = 2, x_3 = \langle x_2, x_2\rangle, x_4 = \langle x_1, x_3\rangle\}\ \mathsf{in}\ \pi_1\ x_4$$

Every binding in the heap is accessible from the program's expression ($\pi_1\ x_4$), so the free-variable based collection rules can collect nothing. But clearly the program will never dereference x_3 nor x_2. The inference-based collection scheme described in this section will allow us to conclude that replacing the binding $x_3 = \langle x_2, x_2\rangle$ with $x_3 = 0$ (or any other binding) will have no observable effect on evaluation. That is, the inference collection scheme shows that

$$\mathsf{letrec}\ \{x_1 = 1, x_2 = 2, x_3 = 0, x_4 = \langle x_1, x_3\rangle\}\ \mathsf{in}\ \pi_1\ x_4$$

is Kleene equivalent to the original program. Now by applying the free-variable rule, we can conclude that the binding $x_2 = 2$ can be safely collected.

In subsection 6.1, we introduce type inference for λgc based on the presentation of Mitchell [25, Section 4.5]. In Section 6.2 we show that if type inference can assign a binding an unconstrained type and the rest of the program still has a valid typing, then the binding can effectively be collected. Our proof of this theorem is based on the method of *logical relations*, a standard proof technique from type theory. (See Mitchell [25, Section 3] for an overview of logical relations and various references.)

6.1 λgc and Type Inference

We start by considering the original λgc language as an *implicitly* typed, monomorphic language, where the types of the language are the same as for λgc-mono except for the addition of type variables:

$$(\text{types}) \quad \tau \ \in \ \mathit{Type} \ ::= \ t \mid \mathsf{int} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

By implicitly typed, we mean that the terms of the language are not decorated with types as in λgc-mono. We add type variables to the set of types so that each well-formed expression has a *principal* or most general type (explained below).

Type inference is the process of decorating λgc programs with types so that the resulting program type checks under the λgc-mono rules. (Refer to Figures 1 and 3 for the syntax and dynamic semantics of λgc and Figure 6 for the typing rules for λgc-mono.) Alternatively, we may directly specify a set of typing rules for λgc programs by taking the typing rules for λgc-mono and erasing the type information from the terms, resulting in the inference system of Figure 7. These rules define judgements of the form $\Gamma \vdash e : \tau$, where Γ is a type assignment and e is a λgc expression. We use "$\vdash$" instead of "$\triangleright$" to keep from confusing these typing judgements with the λgc-mono and λgc-poly judgements.

Expressions:

$$(\textbf{var})\ \frac{}{\Gamma \uplus \{x{:}\tau\} \vdash x : \tau} \qquad (\textbf{int})\ \frac{}{\Gamma \vdash i : \mathsf{int}}$$

$$(\textbf{tuple})\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2} \qquad (\textbf{proj})\ \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i} \quad (i = 1, 2)$$

$$(\textbf{fn})\ \frac{\Gamma \uplus \{x{:}\tau_1\} \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \qquad (\textbf{app})\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Heaps and Programs:

$$(\textbf{heap})\ \frac{\forall\, x \in Dom(\Gamma')\,.\,\Gamma \uplus \Gamma' \vdash H(x) : \Gamma'(x)}{\Gamma \vdash H : \Gamma'} \qquad (\textbf{prog})\ \frac{\emptyset \vdash H : \Gamma \qquad \Gamma \vdash e : \tau}{\vdash \mathsf{letrec}\ H\ \mathsf{in}\ e : \tau}$$

Figure 7: Type Inference Rules for λgc

A given λgc expression can have multiple typing derivations according to these rules and consequently multiple typings, but an expression's typings may be ordered so that there is a most general, or *principal* typing and every other typing is an *instance* of this principal typing and is thus derivable. The ordering is defined in terms of a substitution:

Definition 6.1 (Substitution) *A substitution S is a function from type variables to type expressions.*

We write $S\tau$ to denote the type obtained by replacing each type variable t in τ with $S(t)$ and we write $S\Gamma$ to denote the type assignment $\{x{:}S\tau \mid x{:}\tau \in \Gamma\}$.

Definition 6.2 (Instance, Subsumes, Principal Typing) *A typing $\Gamma' \vdash e : \tau'$ is an* instance *of the typing $\Gamma \vdash e : \tau$ if there exists a substitution S such that $S\Gamma \subseteq \Gamma'$ and $S\tau = \tau'$. In such a case, we say that $\langle \Gamma, \tau \rangle$* subsumes *$\langle \Gamma', \tau' \rangle$. A typing $\Gamma \vdash e : \tau$ is* principal *if for all other typings $\Gamma' \vdash e : \tau'$, $\langle \Gamma, \tau \rangle$ subsumes $\langle \Gamma', \tau' \rangle$.*

A consequence of the definition of a principal typing is that any instance of the principal typing is derivable:

Lemma 6.3 *If $\langle \Gamma, \tau \rangle$ is a principal typing for e, then for all substitutions S, $S\Gamma \vdash e : S\tau$.*

It is straightforward to see that principal typings for λgc expressions are unique (up to α-conversion of type variables). Consequently, we can search for *the* principal typing of a λgc program in order to determine if there is any typing derivation of a program. The critical component of a type inference algorithm is an algorithm for *unifying* types and type assignments:

Definition 6.4 (Unifier, Most General Unifier) *A* unifier *of a set of type equations E is a substitution S such that for all $\tau_1 = \tau_2 \in E$, $S\tau_1$ is syntactically equal to $S\tau_2$. A unifier S of E is more general than S_1 if there exists a unifier S_2 such that $S_1 = S_2 \circ S$.*

Lemma 6.5 (Unification) *[31] Let E be a set of type equations. There is an algorithm $\mathit{Unify}(E)$ that computes a most general unifier of E if one exists and fails otherwise.*

If Γ_1 and Γ_2 are type assignments, then the unification algorithm can be used to compute a most general substitution S such that $S\Gamma_1 \cup S\Gamma_2$ is well formed, if such a substitution exists. (Recall that a well formed type assignment maps term variables to at most one type.) We simply compute $\mathit{Unify}(E)$ where $E = \{\tau_1 = \tau_2 \mid x{:}\tau_1 \in \Gamma_1 \text{ and } x{:}\tau_2 \in \Gamma_2\}$. We write $\mathit{Unify}(\Gamma_1, \Gamma_2)$ for the substitution obtained in this manner.

$$
\begin{array}{rcll}
PT(x) & = & \langle\{x : t\}, t\rangle & (t \text{ fresh}) \\
PT(i) & = & \langle\emptyset, \mathsf{int}\rangle & \\
PT(\langle e_1, e_2\rangle) & = & \mathsf{let}\ \langle\Gamma_1, \tau_1\rangle\ =\ PT(e_1) & \\
 & & \mathsf{in\ let}\ \langle\Gamma_2, \tau_2\rangle\ =\ PT(e_2) & \\
 & & \quad \mathsf{in\ let}\ S\ =\ \mathit{Unify}(\Gamma_1, \Gamma_2) & \\
 & & \qquad \mathsf{in}\langle S\Gamma_1 \cup S\Gamma_2, S\tau_1 \times S\tau_2\rangle & \\
PT(\pi_i\ e) & = & \mathsf{let}\ \langle\Gamma, \tau\rangle\ =\ PT(e) & \\
 & & \mathsf{in\ let}\ S\ =\ \mathit{Unify}(\{x{:}\tau\}, \{x{:}t_1 \times t_2\}) & (t_1, t_2 \text{ fresh}) \\
 & & \quad \mathsf{in}\ \langle S\Gamma, St_i\rangle & \\
PT(\lambda x.e) & = & \mathsf{let}\ \langle\Gamma, \tau_1 \times \tau_2\rangle\ =\ PT(\langle x, e\rangle) & \\
 & & \mathsf{in}\ \langle\Gamma \setminus \{x{:}\tau_1\}, \tau_1 \to \tau_2\rangle & \\
PT(e_1\ e_2) & = & \mathsf{let}\ \langle\Gamma_1, \tau_1\rangle\ =\ PT(e_1)\ \mathsf{in\ let}\ \langle\Gamma_2, \tau_2\rangle\ =\ PT(e_2) & \\
 & & \mathsf{in\ let}\ S\ =\ \mathit{Unify}(\Gamma_1 \uplus \{x{:}\tau_1\}, \Gamma_2 \uplus \{x{:}\tau_2 \to t\}) & (x, t \text{ fresh}) \\
 & & \quad \mathsf{in}\ \langle S\Gamma_1 \cup S\Gamma_2, St\rangle & \\
 & & & \\
PTA(e) & = & \mathsf{let}\ \langle\Gamma, \tau\rangle\ =\ PT(e)\ \mathsf{in}\ \Gamma & \\
PTT(e) & & \mathsf{let}\ \langle\Gamma, \tau\rangle\ =\ PT(e)\ \mathsf{in}\ \tau &
\end{array}
$$

Figure 8: An Algorithm for Computing the Principal Typing of an Expression

Given a λgc expression e, the algorithm $PT(e)$ in Figure 8 computes the most general type assignment and type for that expression. The algorithm is similar to the MTA algorithm of Section 5.4, but uses the $\mathit{Unify}(-)$ routine to compute most general substitutions that allow two type assignments to be merged. The variable case returns a type assignment mapping the variable to a fresh type variable and returns that type variable as the type of the expression. The integer case returns an empty type assignment and the integer type. The tuple case computes the principal typings of the components of the tuple, and then computes a unifying substitution of their respective type assignments. This step is necessary because the same variable could occur within two different components, and the types of these two occurrences must unify. The substitution is applied to each component's principal type assignment and the union of the resulting assignments is returned as the principal assignment of the tuple. The type is calculated by applying the substitution to each of the components' principal types and then forming a tuple type. The type of an abstraction $\lambda x.e$ is calculated by finding the principal typing of the expression $\langle x, e\rangle$. The type of the resulting expression contains the domain type of the abstraction paired with the range type. The pairing is needed since x may not occur in e and thus may not occur in the resulting type assignment Γ. The principal type assignment of the abstraction is simply Γ minus the occurrence of $x{:}\tau_1$.

The following lemma establishes the key properties of $PT(e)$:

Lemma 6.6 *If $PT(e) = \langle \Gamma, \tau \rangle$, then $\Gamma \vdash e : \tau$. Furthermore, if $\Gamma' \vdash e : \tau'$ for some Γ' and τ', then $PT(e)$ exists and $PT(e)$ subsumes $\langle \Gamma', \tau' \rangle$.*

The algorithm for determining the principal typing of an expression can be easily extended to find a principal typing for a heap and a program. The former returns a principal pair of type assignments $\langle \Gamma, \Gamma_H \rangle$ for H, such that $\Gamma \vdash H : \Gamma_H$ and the latter returns the principal type assignment and type for the program:

$$
\begin{array}{rcl}
PT(\{x_1 = h_1, \ldots, x_n = h_n\}) & = & \textsf{let } \langle \Gamma_1, \tau_1 \rangle \;=\; PT(h_1) \quad \cdots \quad \langle \Gamma_n, \tau_n \rangle \;=\; PT(e_n) \\
& & \textsf{in let } \Gamma = \{x_1{:}\tau_1, \cdots, x_n{:}\tau_n\} \\
& & \quad \textsf{in let } S \;=\; \mathit{Unify}(\Gamma, \Gamma_1, \cdots, \Gamma_n) \\
& & \quad\quad \textsf{in } \langle (S\Gamma_1 \cup \cdots \cup S\Gamma_n) \setminus S\Gamma, S\Gamma \rangle
\end{array}
$$

$$
\begin{array}{rcl}
PT(\textsf{letrec } H \textsf{ in } e) & = & \textsf{let } \langle \Gamma, \Gamma_H \rangle = PT(H) \\
& & \textsf{in let } \langle \Gamma_e, \tau \rangle = \; PT(e) \\
& & \quad \textsf{in let } S \;=\; \mathit{Unify}(\Gamma \uplus \Gamma_H, \Gamma_e) \\
& & \quad\quad \textsf{in } \langle S\Gamma \cup (S\Gamma_e \setminus S\Gamma_H), S\tau \rangle
\end{array}
$$

The following lemma shows that $PT(P)$ computes the principal type for a program and that any typing for the program is an instance of the principal typing.

Lemma 6.7 *If $PT(P) = \langle \Gamma, \tau \rangle$, then $\Gamma \vdash P : \tau$. Furthermore, if $\Gamma' \vdash P : \tau'$, then $\langle \Gamma, \tau \rangle$ subsumes $\langle \Gamma', \tau' \rangle$.*

The algorithm is complete in that if there exists some typing for the program, then the algorithm will not fail to find the principal typing of the program.

Lemma 6.8 *If $\Gamma \vdash P : \tau$, then $PT(P)$ exists.*

Finally, soundness for the static semantics follows by showing that principal typings of a program are preserved by evaluation.

Theorem 6.9 (Type Soundness) *If $\vdash P$ then either P is an answer or else there exists a P' such that $P \stackrel{\mathrm{R}}{\longmapsto} P'$ and $\vdash P'$.*

6.2 The Inference GC Specification

We will show that if we can find a typing for a program that assigns a location bound in the heap a type variable, then that location may be bound to *any* value without affecting evaluation. Consequently, any pointers contained in the location's binding do not need to be scanned and traced during garbage collection. The intuition behind the theorem is that a location's type is unconstrained only if the location is not applied nor projected nor *used* in some manner that would constrain the type. Consequently, we can replace the binding in the heap with any binding we choose. In particular, if the location is bound to a large heap value, we can bind the location to an integer or dummy piece of code without affecting evaluation. This replacement allows us to collect any bindings that used to be reachable through this binding without knowing anything about the shape of the original heap value.

The proof of the theorem relies upon the semantic interpretation of types as logical relations. In our case, the relations are a type-indexed family of binary relations relating programs to programs, answers to answers, and heaps to heaps. The relations are contrived so that, if two programs are related, then they are Kleene equivalent, so one program converges to an answer iff the other converges to a related answer and related answers at base type (int) yield equal values. Roughly speaking, the relations are logically extended to relate answers of higher type ($\rightarrow$) if, whenever such answers are applied to appropriately related answers, the resulting computations yield related results.

The relations are defined in Figure 9 by induction on types. The definitions are parameterized by an arbitrary relational interpretation of type variables, Θ. If t is a type variable, then $\Theta(t)$ determines some fixed, but arbitrary relation between answer programs. This is consistent with the idea that well-typed programs have an implicit "$\forall$" quantifier for the type variables in a program. The parameterization of the interpretation of type variables makes it straightforward to extend the definition of the relations to account for predicative polymorphism.

Computations:

$$\begin{array}{lll}
\Theta \models P_1 \sim P_2 : \tau & \text{iff} & \vdash P_1 : \tau \text{ and } \vdash P_2 : \tau \text{ and} \\
& & P_1 \Downarrow_R A_1 \text{ implies } P_2 \Downarrow_R A_2 \text{ and } \Theta \models A_1 \approx A_2 : \tau \\
& & \qquad\qquad \text{and} \\
& & P_2 \Downarrow_R A_2 \text{ implies } P_1 \Downarrow_R A_1 \text{ and } \Theta \models A_1 \approx A_2 : \tau
\end{array}$$

Answers:

$$\begin{array}{lll}
\Theta \models A_1 \approx A_2 : t & \text{iff} & \langle A_1, A_2 \rangle \in \Theta(t) \\
\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ x_1 \approx \mathsf{letrec}\ H_2\ \mathsf{in}\ x_2 : \mathsf{int} & \text{iff} & H_1(x_1) = H_2(x_2) = i \\
\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ x_1 \approx \mathsf{letrec}\ H_2\ \mathsf{in}\ x_2 : \tau_1 \times \tau_2 & \text{iff} & \Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ \pi_1\ x_1 \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ \pi_1\ x_2 : \tau_1 \\
& & \text{and} \\
& & \Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ \pi_2\ x_1 \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ \pi_2\ x_2 : \tau_2 \\
\\
\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ x_1 \approx \mathsf{letrec}\ H_2\ \mathsf{in}\ x_2 : \tau_1 \rightarrow \tau_2 & \text{iff} & \forall H_1', H_2', y_1, y_2. \\
& & \Theta \models \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ y_1 \approx \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ y_2 : \tau_1 \\
& & \text{implies} \\
& & \Theta \models \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ x_1\ y_1 \sim \\
& & \qquad\qquad \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ x_2\ y_2 : \tau_2
\end{array}$$

Heaps:

$$\begin{array}{lll}
\Theta \models H_1 \approx H_2 : \Gamma & \text{iff} & \emptyset \vdash H_1 : \Gamma \text{ and } \emptyset \vdash H_2 : \Gamma \text{ and} \\
& & \forall x \in \mathit{Dom}(\Gamma). \Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_2\ \mathsf{in}\ x : \Gamma(x)
\end{array}$$

Figure 9: Relational Interpretation of Types

Two well-typed programs P_1 and P_2 are related at a type τ, written $\Theta \models P_1 \sim P_2 : \tau$ iff, whenever one of the programs terminates with an answer, then the other program terminates with a related answer at type τ.

Two answer programs A_1 and A_2 are related at type τ, written $\Theta \models A_1 \approx A_2 : \tau$, as follows: If τ is a type variable t, then the answers are related iff they are in the relation $\Theta(t)$. If τ is an integer, then the answers are related iff, when we lookup the answer variables in their respective heaps, they are bound to the same integer value. If τ is a pair, then the answers are related iff, whenever we project a component, the resulting programs are related. Finally, if τ is an arrow type $\tau_1 \rightarrow \tau_2$, the answers are related iff, whenever we apply the answer variables to related arguments at type τ_1, we get related programs at type τ_2. Even though the relations between programs and answers are defined in terms of one another, the relations are well-founded because the size of the type index always decreases when one relation refers to another.

The definition of the relations ensures that related programs remain related even if more bindings are added to the programs' heaps.

Lemma 6.10 *If* $\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ e_1 \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ e_2 : \tau$ *and* $\vdash \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ e : \tau$ *and* $\vdash \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ e : \tau$*, then* $\Theta \models \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ e \sim \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ e : \tau$*.*

Since evaluation only adds new bindings and leaves existing bindings intact, it is clear that evaluation preserves the relations. If the language permitted assignment, then this property would not necessarily hold.

For the statement of the following lemma, we need to extend the answer relation to heaps. Two heaps, H_1 and H_2, are related at a context Γ, written $\Theta \models H_1 \approx H_2 : \Gamma$, if they can be α-converted so that, for all variables x in Γ, the answers $\mathsf{letrec}\ H_1\ \mathsf{in}\ x$ and $\mathsf{letrec}\ H_2\ \mathsf{in}\ x$ are related at $\Gamma(x)$.

The following lemma establishes our primary result: an expression is related to itself in the context of any two related heaps.

Lemma 6.11 *For all* $\Theta : Tvar \to (Ans \leftrightarrow Ans)$*, if* $\Gamma \vdash e : \tau$ *and* $\Theta \models H_1 \approx H_2 : \Gamma$*, then* $\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ e \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ e : \tau$*.*

Proof: By induction on the derivation of $\Gamma \vdash e : \tau$.

var: $(\Gamma \uplus \{x{:}\tau\} \vdash x : \tau)$ By the definition of $\Theta \models H_1 \approx H_2 : \Gamma$.

int: $(\Gamma \vdash i : \mathsf{int})$ $\mathsf{letrec}\ H_j\ \mathsf{in}\ i \xmapsto{\mathrm{R}} \mathsf{letrec}\ H_j \uplus \{x_j = i\}\ \mathsf{in}\ x_j$ for $j = 1, 2$ and these two answers are related by the definition of $\Theta \models A_1 \approx A_2 : \mathsf{int}$.

tuple: $(\Gamma \vdash \langle e_1, e_2\rangle : \tau_1 \times \tau_2)$ Suppose $\mathsf{letrec}\ H_1\ \mathsf{in}\ \langle e_1, e_2\rangle$ does not diverge. Then by examination of the rewriting rules, it is clear that:

$$\begin{array}{l}\mathsf{letrec}\ H_1\ \mathsf{in}\ \langle e_1, e_2\rangle \xmapsto{\mathrm{R}}{}^{*}\ \ \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ \langle x_1, e_2\rangle \xmapsto{\mathrm{R}}{}^{*}\\ \quad \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ \langle x_1, x_2\rangle \xmapsto{\mathrm{R}}\\ \quad \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\} \uplus \{x = \langle x_1, x_2\rangle\}\ \mathsf{in}\ x\end{array}$$

Thus, $\mathsf{letrec}\ H_1\ \mathsf{in}\ e_1 \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ x_1$. By the induction hypothesis applied to the first hypothesis of the **pair** rule, we conclude that $\mathsf{letrec}\ H_2\ \mathsf{in}\ e_1 \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ x_1'$ and these two answers are related. Consequently, $\mathsf{letrec}\ H_2\ \mathsf{in}\ \langle e_1, e_2\rangle \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ \langle x_1', e_2\rangle$.

From the rewriting sequence above, we can conclude that $\mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ e_2 \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_2$. By hypothesis, $\Theta \models H_1 \approx H_2 : \Gamma$ and thus by Lemma 6.10 $\Theta \models H_1 \uplus H_1' \uplus \{x_1 = h_1\} \approx H_2 \uplus H_2' \uplus \{x_2 = h_2\} : \Gamma$, so the induction hypothesis applies to the second hypothesis of the **pair** rule and we can conclude that $\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ e_2 \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_2'$ and these two answers are related. Consequently, $\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ \langle x_1', e_2\rangle \xmapsto{\mathrm{R}}{}^{*} \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ \langle x_1', x_2'\rangle$. Since related answers remain related under heap extensions,

$$\begin{array}{l}\Theta \models \mathsf{letrec}\ H_1 \uplus H_1' \uplus H_1'' \uplus \{x_1 = h_1, x_2 = h_2, x = \langle x_1, x_2, \}\rangle\ \mathsf{in}\ x_i \approx\\ \qquad \mathsf{letrec}\ H_2 \uplus H_2' \uplus H_2'' \uplus \{x_1' = h_1', x_2' = h_2', x' = \langle x_1', x_2', \}\rangle\ \mathsf{in}\ x_i' : \tau_i \qquad (i = 1, 2)\end{array}$$

Consequently,

$$\begin{array}{l}\Theta \models \mathsf{letrec}\ H_1 \uplus H_1' \uplus H_1'' \uplus \{x_1 = h_1, x_2 = h_2, x = \langle x_1, x_2, \}\rangle\ \mathsf{in}\ \pi_i\ x \sim\\ \qquad \mathsf{letrec}\ H_2 \uplus H_2' \uplus H_2'' \uplus \{x_1' = h_1', x_2' = h_2', x' = \langle x_1', x_2', \}\rangle\ \mathsf{in}\ \pi_i\ x' : \tau_i \qquad (i = 1, 2)\end{array}$$

Thus, $\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ \langle e_1, e_2\rangle \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ \langle e_1, e_2\rangle : \tau_1 \times \tau_2$.

proj: $(\Gamma \vdash \pi_i\ e : \tau_i)$ Suppose $\mathsf{letrec}\ H_1\ \mathsf{in}\ \pi_i\ e$ converges. Then,

$$\mathsf{letrec}\ H_1\ \mathsf{in}\ \pi_i\ e \stackrel{\mathrm{R}}{\longmapsto}{}^* \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ \pi_i\ x \stackrel{\mathrm{R}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ y$$

for some H_1', x, and y. Thus, $\mathsf{letrec}\ H_1\ \mathsf{in}\ e \stackrel{\mathrm{R}}{\longmapsto}{}^* \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ x$. By the induction hypothesis, $\mathsf{letrec}\ H_2\ \mathsf{in}\ e \stackrel{\mathrm{R}}{\longmapsto}{}^* \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ x'$ and $\Theta \models \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ x' : \tau_1 \times \tau_2$. By the definition of this relation, we know that

$$\Theta \models \mathsf{letrec}\ H_1 \uplus H_1'\ \mathsf{in}\ \pi_i\ x \sim \mathsf{letrec}\ H_2 \uplus H_2'\ \mathsf{in}\ \pi_i\ x' : \tau_1 \times \tau_2$$

for $i = 1, 2$. Thus, $\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ \pi_i\ e \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ \pi_i\ e : \tau_1 \times \tau_2$.

fn: $(\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2)$ $\mathsf{letrec}\ H_i\ \mathsf{in}\ \lambda x.e \stackrel{\mathrm{R}}{\longmapsto} \mathsf{letrec}\ H_i \uplus \{y_i = \lambda x.e\}\ \mathsf{in}\ y_i$ for $i = 1, 2$. We must show that these two answers are related. Suppose:

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e\} \uplus H_1'\ \mathsf{in}\ z_1 \approx \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e\} \uplus H_2'\ \mathsf{in}\ z_2 : \tau_1$$

for some H_1', H_2', z_1, and z_2. We need to show that:

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e\} \uplus H_1'\ \mathsf{in}\ y_1\ z_1 \sim \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e\} \uplus H_2'\ \mathsf{in}\ y_2\ z_2 : \tau_2$$

This follows if:

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1'\ \mathsf{in}\ e \sim \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2'\ \mathsf{in}\ e : \tau_2$$

where $h_i = (H_i \uplus \{y_i = \lambda x.e\} \uplus H_i')(z_i)$ for $i = 1, 2$. By the induction hypothesis, it suffices to show that:

$$\Theta \models H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1' \approx H_1 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2' : \Gamma \uplus \{x : \tau_1\}$$

By the lemma's hypothesis, we know that $\Theta \models H_1 \approx H_2 : \Gamma$ and thus by Lemma 6.10

$$\Theta \models H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1' \approx H_2 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2' : \Gamma.$$

Since

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1'\ \mathsf{in}\ z_1 \approx \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2'\ \mathsf{in}\ z_2 : \tau_1$$

and h_1 and h_2 are bound to z_1 and z_2 in these respective heaps, it follows that:

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1'\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2'\ \mathsf{in}\ x : \tau_1$$

Consequently,

$$\Theta \models H_1 \uplus \{y_1 = \lambda x.e, x = h_1\} \uplus H_1' \approx H_1 \uplus \{y_2 = \lambda x.e, x = h_2\} \uplus H_2' : \Gamma \uplus \{x : \tau_1\}$$

holds and, working backwards we know that

$$\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e\} \uplus H_1'\ \mathsf{in}\ y_1\ z_1 \sim \mathsf{letrec}\ H_1 \uplus \{y_2 = \lambda x.e\} \uplus H_2'\ \mathsf{in}\ y_2\ z_2 : \tau_2$$

and thus $\Theta \models \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e\}\ \mathsf{in}\ y_1 \approx \mathsf{letrec}\ H_2 \uplus \{y_2 = \lambda x.e\}\ \mathsf{in}\ y_2 : \tau_1 \to \tau_2$.

app: $(\Gamma \vdash e_1\ e_2 : \tau_2)$ Suppose $\mathsf{letrec}\ H_1\ \mathsf{in}\ e_1\ e_2$ does not diverge. Then by examination of the rewriting rules, it is clear that:

$$\begin{array}{l} \mathsf{letrec}\ H_1\ \mathsf{in}\ e_1\ e_2 \stackrel{\mathrm{R}}{\longmapsto}{}^* \quad \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ x_1\ e_2 \stackrel{\mathrm{R}}{\longmapsto}{}^* \\ \quad \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_1\ x_2 \end{array}$$

Thus, $\mathsf{letrec}\ H_1\ \mathsf{in}\ e_1 \stackrel{\mathrm{R}}{\longmapsto}{}^* \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ x_1$. By the induction hypothesis applied to the first hypothesis of the **app** rule, we conclude that $\mathsf{letrec}\ H_2\ \mathsf{in}\ e_1 \stackrel{\mathrm{R}}{\longmapsto}{}^* \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ x_1'$

and these two answers are related at $\tau_1 \to \tau_2$. Consequently, $\mathsf{letrec}\ H_2\ \mathsf{in}\ e_1\ e_2 \overset{\mathrm{R}}{\longmapsto}{}^{*}\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ x_1'\ e_2$.

From the rewriting sequence above, we can conclude that $\mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\}\ \mathsf{in}\ e_2 \overset{\mathrm{R}}{\longmapsto}{}^{*}\mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_2$. By hypothesis, $\Theta \models H_1 \approx H_2 : \Gamma$ and thus $\Theta \models H_1 \uplus H_1' \uplus \{x_1 = h_1\} \approx H_2 \uplus H_2' \uplus \{x_2 = h_2\} : \Gamma$, so the induction hypothesis applies to the second hypothesis of the **app** rule and we can conclude that $\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ e_2 \overset{\mathrm{R}}{\longmapsto}{}^{*}\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_2'$ and these two answers are related at τ_1. Consequently, $\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\}\ \mathsf{in}\ x_1'\ e_2 \overset{\mathrm{R}}{\longmapsto}{}^{*}\mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_1'\ x_2'$.
Since

$$\begin{aligned}\Theta \models\ & \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_2 \approx \\ & \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_2' : \tau_1,\end{aligned}$$

and

$$\begin{aligned}\Theta \models\ & \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_1 \approx \\ & \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_1' : \tau_1 \to \tau_2\end{aligned}$$

by the definition of $\approx$ at arrow-types, it follows that

$$\begin{aligned}\Theta \models\ & \mathsf{letrec}\ H_1 \uplus H_1' \uplus \{x_1 = h_1\} \uplus H_1'' \uplus \{x_2 = h_2\}\ \mathsf{in}\ x_1\ x_2 \sim \\ & \mathsf{letrec}\ H_2 \uplus H_2' \uplus \{x_1' = h_1'\} \uplus H_2'' \uplus \{x_2' = h_2'\}\ \mathsf{in}\ x_1'\ x_2' : \tau_2\end{aligned}$$

Thus, $\Theta \models \mathsf{letrec}\ H_1\ \mathsf{in}\ e_1\ e_2 \sim \mathsf{letrec}\ H_2\ \mathsf{in}\ e_1\ e_2 : \tau_2$.

□

Our goal is to show that if we are given a context Γ, expression e, and heap H such that $\Gamma \vdash e : \tau$ and $\emptyset \vdash H : \Gamma$, then $\mathsf{letrec}\ H'\ \mathsf{in}\ e$ is Kleene equivalent to $\mathsf{letrec}\ H\ \mathsf{in}\ e$ where H' is defined as follows:

$$H' = \{x = H(x) \mid \Gamma(x) \notin Tvar\} \uplus \{x = 0 \mid \Gamma(x) \in Tvar\}$$

This follows from Lemma 6.11 if we can show that, taking $\Theta_0(t)$ to be the everywhere-defined relation on answer programs, $\Theta_0 \models H \approx H' : \Gamma$. This in turn follows if we can show that $\Theta_0 \models H \approx H : \Gamma$ (H is related to itself), since $\Theta_0(t)$ relates every program and $H'(x)$ differs from $H(x)$ only when $\Gamma(x) = t$.

Unfortunately, we cannot directly show that a well formed heap is related to itself! The problem is that if we attempt to argue by induction on the derivation of $\emptyset \vdash H : \Gamma$, the uses of the **heap** rule require that we assume what we are trying to prove. The same problem is encountered when using logical relations to reason about conventional calculi with recursion or iteration operators.

If we forbid cycles in the heap, then we can transform the derivation of the heap's well formedness into a derivation that only uses a let-style rule instead of the recursive letrec-style **heap** rule:

$$(\textbf{let-heap})\ \frac{\Gamma_1 \uplus \Gamma_2 \vdash h : \tau \qquad \Gamma_1 \vdash H : \Gamma_2}{\Gamma_1 \vdash H \uplus \{x = h\} : \Gamma_2 \uplus \{x{:}\tau\}}$$

Consequently, if a heap is cycle free, we may show by induction on the derivation using the **let-heap** rule that the heap is related to itself.

Lemma 6.12 *If $\Gamma_1 \vdash H : \Gamma_2$, $\Theta \models H_1 \approx H_2 : \Gamma_1$ and H is cycle free, then $\Theta \models H_1 \uplus H \approx H_2 \uplus H : \Gamma_1 \uplus \Gamma_2$.*

Proof: Since H is cycle free, there exists some derivation of $\Gamma_1 \vdash H : \Gamma$ using only the **let-heap** rule. The proof proceeds by induction on the height of this derivation. If H is empty, then the lemma holds trivially by the assumptions about H_1 and H_2. Suppose $\Gamma_1 \vdash H : \Gamma_2$

and $\Gamma_1 \uplus \Gamma_2 \vdash h : \tau$. By induction, $\Theta \models H_1 \uplus H \approx H_2 \uplus H : \Gamma_1 \uplus \Gamma_2$. By Lemma 6.11, $\Theta \models \mathsf{letrec}\ H_1 \uplus H\ \mathsf{in}\ h \sim \mathsf{letrec}\ H_2 \uplus H\ \mathsf{in}\ h : \tau$. Thus,

$$\Theta \models \mathsf{letrec}\ H_1 \uplus H \uplus \{x = h\}\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_2 \uplus H \uplus \{x = h\}\ \mathsf{in}\ x : \tau.$$

Consequently, $\Theta \models H_1 \uplus H \uplus \{x = h\} \approx H_2 \uplus H \uplus \{x = h\} : \Gamma_1 \uplus \Gamma_2 \uplus \{x{:}\tau\}$. □

Finally, we can state and prove the following Inference GC specification:

Theorem 6.13 (Inference GC) *Let $\Gamma_1 = \{x_1{:}t_1, \ldots, x_n{:}t_n\}$ and $H_1 = \{x_1 = h_1, \ldots, x_n = h_n\}$ and $H_1' = \{x_1 = 0, \ldots, x_n = 0\}$. If*

1. *$\Gamma_1 \uplus \Gamma_2 \vdash e : \tau$ ($\tau \notin Tvar$), and*
2. *$\Gamma_1 \vdash H_2 : \Gamma_2$, and*
3. *$\exists S.\emptyset \vdash H_1 : S\Gamma_1$, and*
4. *H_2 is cycle free,*

then $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \simeq \mathsf{letrec}\ H_1' \uplus H_2\ \mathsf{in}\ e$.

Proof: Taking $\Theta_0(t)$ to be the everywhere defined relation, $\Theta_0 \models H_1 \approx H_1' : \Gamma_1$ holds trivially. By Lemma 6.12, since H_2 is cycle free and $\Gamma_1 \vdash H_2 : \Gamma_2$, we know that $\Theta_0 \models H_1 \uplus H_2 \approx H_1' \uplus H_2 : \Gamma_1 \uplus \Gamma_2$. Since $\Gamma_1 \uplus \Gamma_2 \vdash e : \tau$, we know by Lemma 6.11 that $\Theta_0 \models \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \sim \mathsf{letrec}\ H_1' \uplus H_2\ \mathsf{in}\ e : \tau$. Thus, $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \Downarrow \mathsf{letrec}\ H_a\ \mathsf{in}\ x$ iff $\mathsf{letrec}\ H_1' \uplus H_2\ \mathsf{in}\ e \Downarrow \mathsf{letrec}\ H_b\ \mathsf{in}\ y$ and $\Theta_0 \models \mathsf{letrec}\ H_a\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_b\ \mathsf{in}\ y : \tau$. Suppose $H_a(x) = i$ (or $H_b(y) = i$). Then τ must be int since $\tau \notin Tvar$. By the definition of $\approx$ at int, $H_b(y) = i$ (or $H_a(x) = i$). Thus, $\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \simeq \mathsf{letrec}\ H_1' \uplus H_2\ \mathsf{in}\ e$. □

It should be possible to extend our argument to cyclic heaps if we can show that every cyclic heap is appropriately approximated by some finite "unwinding" of the heap. One approach is to use a fixed-point semantics for λgc where types are interpreted as CPOs formed by suitably lifting answer programs. The meaning of an answer program $\mathsf{letrec}\ H\ \mathsf{in}\ x$ would essentially be the least fixed-point of the chain of meanings of the approximations $\mathsf{letrec}\ H^0\ \mathsf{in}\ x$,$\mathsf{letrec}\ H^1\ \mathsf{in}\ x$,$\mathsf{letrec}\ H^2\ \mathsf{in}\ x$,... where H^i denotes the i^{th} unwinding of the heap H. See Gunter [17, Chapter 4.4] for an example of the treatment of unwinding and approximation in the context of a conventional λ-calculus.

7 Related Work

The literature on garbage collection in sequential programming languages *per se* contains few papers that attempt to provide a compact characterization of algorithms or correctness proofs. Demers *et al.* [10] give a model of memory parameterized by an abstract notion of a "points-to" relation. As a result, they can characterize *reachability*-based algorithms including mark-sweep, copying, generational, "conservative," and other sophisticated forms of garbage collection. However, their model is intentionally divorced from the programming language and cannot take advantage of any *semantic* properties of evaluation, such as type preservation. Consequently, their framework cannot easily model the type-based collectors of Sections 5 and 6. Nettles [27] provides a concrete specification of a copying garbage collection algorithm using the Larch specification language. Our specification of the free-variable tracing algorithm is essentially a high-level, one-line description of his specification. Like Demers *et al.*, he uses a traditional definition of garbage based on unreachability in the memory graph and purposefully distances the specification from any language evaluation details.

Hudak gives a denotational model that tracks reference counts for a first-order language [20]. He presents an abstraction of the model and gives an algorithm for computing approximations of reference counts statically. Chirimar, Gunter, and Riecke give a framework for proving invariants regarding memory management for a language with a *linear* type system [9]. Their low-level semantics specifies explicit memory management based on reference counting. The goal of the work was to determine whether, using the linear type system, in-place update would be a sound optimization. Both Hudak and Chirimar *et al.* assume a fairly weak approximation of garbage (reference counts).

Tolmach [34] built a type-recovery collector for a variant of SML that passes type information to polymorphic routines during execution, effectively implementing our λgc-poly language and the corresponding collector of Subsection 5.5. Aditya and Caro gave a type-recovery algorithm for an implementation of Id that uses a technique that appears to be equivalent to type passing [1] and Aditya, Flood, and Hicks extended this work to garbage collection for Id [2].

There have been a variety of papers regarding inference-based collection for monomorphic [7, 36, 8] and polymorphic languages [3, 15, 16, 12]. Appel [3] argued informally that "tag-free" collection is possible for polymorphic languages such as SML by a combination of recording information statically and performing what amounts to type inference during the collection process, though the connections between inference and collection were not made clear. Baker [6] recognized that Milner-style type inference can be used to prove that reachable objects can be safely collected, but did not give a formal account of this result. Goldberg and Gloger [16] recognized that it is not possible to reconstruct the concrete types of all reachable values in an implementation of an ML-style language that does not pass types to polymorphic routines. They gave an informal argument based on traversal of stack frames to show that such values are semantically garbage. Fradet [12] gave another argument based on Reynolds' abstraction/parametricity theorem [29]. Fradet's formulation is closer to ours than Goldberg and Gloger's, since he represented the evaluation "stack" as a source-language term. However, none of these papers gives a complete formulation of the underlying dynamic and static semantics of the language and thus the proofs of correctness are necessarily *ad hoc*.

Finally, Purushothaman and Seaman [28, 32] and Launchbury [21] have proposed "natural" semantics for *call-by-need* (*lazy*) languages where the semantic objects include an explicit heap. This allows sharing and memoization of computations to be expressed in the semantics. More recently, Ariola *et al.* [5] (see also [4, 22]) have presented a purely syntactic theory of the call-by-need λ-calculus that is largely compatible with our work.

8 Summary

Our paper provides a unifying semantic framework for a variety of garbage collection ideas including standard copying and mark-sweep collection, generational collection, tag-free collection, and inference-based collection. By making allocation and the heap explicit, we are able to reason about memory management using traditional λ-calculus techniques. In particular, we are able to make strong connections between garbage collection and type theory. By abstracting away such low-level details as evaluation stacks, registers, and addresses, we are able to formulate complicated collection algorithms in a compact manner and yet give a formal proof of correctness. The framework is reasonably robust, as demonstrated by the breadth of topics covered in this paper, and we expect that the framework can be extended to deal with other work on garbage collection.

9 Acknowledgements

Thanks to Edo Biagoni, Andrzej Filinski, Mark Leone, Scott Nettles, Chris Okasaki, and Jeannette Wing for proofreading earlier drafts of this document and for providing valuable insight. This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.

References

[1] S. Aditya and A. Caro. Compiler-directed type reconstruction for polymorphic languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, Copenhagen, Denmark, June 1993.

[2] S. Aditya, C. Flood, and J. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 12–23, June 1994.

[3] A. W. Appel. Runtime tags aren't necessary. *Journal of Lisp and Symbolic Computation*, 2:153–162, 1989.

[4] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. Technical Report CIS–TR–94–23, University of Oregon, Oct. 1994.

[5] Z. M. Ariola, M. Felleisen, J. Maraist, M. Odersky, and P. Wadler. A call-by-need lambda calculus. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 1995.

[6] H. Baker. Unify and conquer (garbage, updating, aliasing ...) in functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 218–226, 1990.

[7] P. Branquart and J. Lewi. A scheme for storage allocation and garbage collection for Algol-68. In *Algol-68 Implementation*. North-Holland Publishing Company, Amsterdam, 1970.

[8] D. E. Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[9] J. Chirimar, C. A. Gunter, and J. G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 139–150, June 1992.

[10] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, Jan. 1990.

[11] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. Technical Report 89-100, Rice University, June 1989. Also appears in: *Theoretical Computer Science,* **102**, 1992.

[12] P. Fradet. Collecting more garbage. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 24–33, June 1994.

[13] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application a l'elimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium, edited by J.E. Fenstad. North-Holland, Amsterdam*, pages 63–92, 1971.

[14] J.-Y. Girard. *Interprétation Fonctionnelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur.* PhD thesis, Université Paris VII, 1972.

[15] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 165–176, June 1991.

[16] B. Goldberg and M. Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 53–65, June 1992.

[17] C. A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

[18] R. Harper. A simplified account of polymorphic references. Technical Report CMU–CS–93–169, School of Computer Science, Carnegie Mellon University, June 1993.

[19] A. L. Hosking, J. E. B. Moss, and D. Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 92–109, Oct. 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.

[20] P. Hudak. A semantic model of reference counting and its abstraction. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 351–363, 1986.

[21] J. Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the 20th Annual ACM Symposium on Principles of Programming Languages*, Charleston, SC, Jan. 1993.

[22] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus (unabridged). Technical Report 28/94, Universität Karlsruhe, Fakultät für Informatik, Oct. 1994.

[23] I. Mason and C. Talcott. Reasoning about programs with effects. In *Proceedings of Programming Language Implementation and Logic Programming*, LNCS 582. Springer-Verlag, 1990.

[24] I. Mason and C. Talcott. Equivalences in functional languages with effects. *Journal of Functional Programming*, 2(1), 1991.

[25] J. C. Mitchell. Type systems for programming languages. Technical Report STAN–CS–89–1277, Department of Computer Science, Stanford University, 1989.

[26] R. Morrison, A. Dearle, R. C. H. Connor, and A. L. Brown. An ad hoc approach to the implementation of polymorphism. *ACM Trans. Prog. Lang. Syst.*, 13(3):342–371, July 1991.

[27] S. Nettles. A Larch specification of copying garbage collection. Technical Report CMU–CS–92–219, School of Computer Science, Carnegie Mellon University, Dec. 1992.

[28] Purushothaman and J. Seaman. An adequate operational semantics of sharing in lazy evaluation. In *Proceedings of the 4th European Symposium on Programming*, LNCS 582. Springer-Verlag, 1992.

[29] J. Reynolds. Types, abstraction, and parametric polymorphism. In *Proceedings of Information Processing 83*, pages 513–523, 1983.

[30] J. C. Reynolds. Towards a theory of type structure. In *Proceedings, Colloque sur la Programmation. Lecture Notes in Computer Science*, volume 19, pages 408–425. Springer-Verlag, Berlin, 1974.

[31] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery*, 12, 1965.

[32] J. M. Seaman. *An Operational Semantics of Lazy Evaluation for Analysis.* PhD thesis, Pennsylvania State University, 1993.

[33] P. Steenkiste and J. Hennessey. Tags and type checking in LISP: Hardware and software approaches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 50–59, Oct. 1987.

[34] A. Tolmach. Tag-free garbage collection using explicit type parameters. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, June 1994.

[35] D. Ungar. Generational scavenging: A non-disruptive high performance storage management reclamation algorithm. In *ACM SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 15–167, Pittsburgh, Pennsylvania, Apr. 1984.

[36] P. Wodon. Methods of garbage collection for Algol-68. In *Algol-68 Implementation*. North-Holland Publishing Company, Amsterdam, 1970.

[37] A. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, Apr. 1991.