Abstract Models of Memory Management*

Greg Morrisett
Carnegie Mellon
jgmorris@cs.cmu.edu

Matthias Felleisen
Rice University
matthias@cs.rice.edu

Robert Harper
Carnegie Mellon
rwh@cs.cmu.edu

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 specify in a compact manner a wide variety of memory management techniques, including standard trace-based garbage collection (*i.e.*, the family of copying and mark/sweep collection algorithms), 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 prove the idea that type inference can be used to collect some objects that are accessible but never used.

*This work was sponsored in part by the Advanced Research Projects Agency (ARPA), 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, Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and ARPA grant No. F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

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 written in an advanced programming language will crash due to dangling pointer problems while allocation, access, and deallocation are 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. 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 [18, 19] 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-$\mathbf{S}$ 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. We define a semantic notion of garbage collection for λgc and prove that there is no *optimal* collection strategy that is computable. In Section 3, we specify the "free-variable" garbage collection rule which models trace-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 Section 4, we formalize so-called "tag-free" collection algorithms for explicitly-typed, monomorphic languages such as Pascal and Algol [7, 29, 8]. We show how to *recover* 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.

In Section 5, 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 [3, 5, 14, 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 6 discusses related work and Section 7 closes with a summary.

Due to a lack of space, most of the proofs for lemmas are omitted in this paper. However, full details may be recovered from our companion technical report [20].

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 ($e_1\ e_2$).

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 and 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 letrec H 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 programs to be equivalent up to a consistent α-conversion of bound variables.

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 letrec H 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: letrec H' 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,

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

Figure 1 defines the set of evaluation contexts and instruction expressions for λgc. The definition of evaluation contexts (E) reflects the left-to-right, call-by-value 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.

The evaluation rules for λgc 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**.

[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 5.

Programs:

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

Evaluation Contexts and Instruction Expressions:

(contexts)	E	$\in$	*Ctxt*	$::=$	$[\,] \mid \langle E, e \rangle \mid \langle x, E \rangle \mid \pi_i\ E \mid E\ e \mid x\ E$
(instructions)	I	$\in$	*Instr*	$::=$	$h \mid \pi_i\ x \mid x\ y$

Rewriting Rules

(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 1: The Syntax and Operational Semantics of λgc

The irreducible 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:*

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

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 irreducible programs [11, 30].

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 2.2 (Kleene Equivalence) $(P_1, G_1) \simeq (P_2, G_2)$ *means* $P_1 \Downarrow_{G_1} \mathsf{letrec}\ H_1\ \mathsf{in}\ x$ *where* $H_1(x) = i$ *if and only if* $P_2 \Downarrow_{G_2} \mathsf{letrec}\ H_2\ \mathsf{in}\ y$ *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 2.3 (Garbage) *If* $P = \mathsf{letrec}\ H \uplus \{x = h\}\ \mathsf{in}\ e$, *then the binding* $x = h$ *is* garbage *with respect to* P *iff* $P \simeq \mathsf{letrec}\ H\ \mathsf{in}\ e$. *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.*

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

Proposition 2.4 (Garbage Undecidable) *Determining if a binding is garbage in an arbitrary closed λgc program is undecidable.*

Proof (sketch): We can reduce the halting problem to an optimal collector by taking an arbitrary program, adding a binding to the heap and modifying the program so that if it terminates, it accesses the extra binding. An optimal collector will collect the binding if and only if the original program does not terminate. □

3 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.

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:

$$\textbf{(fv)} \qquad \begin{array}{l} \mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \xmapsto{\text{fv}} \mathsf{letrec}\ H_1\ \mathsf{in}\ e \\ \quad \text{if } Dom(H_2) \cap FV(\mathsf{letrec}\ H_1\ \mathsf{in}\ e) = \emptyset \end{array}$$

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.

Lemma 3.1 (Postponement) *If* $P_1 \xmapsto{\text{fv}} P_2 \xmapsto{\text{R}} P_3$, *then there exists a* P_2' *such that* $P_1 \xmapsto{\text{R}} P_2' \xmapsto{\text{fv}} P_3$.

Proof (sketch): By cases on the elements of R. □

Lemma 3.2 (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 (sketch): 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 3.3 (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 let $P' = \mathsf{letrec}\ H_1\ \mathsf{in}\ e$ 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 3.4 *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:

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

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:

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

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 $H_1(x) = i$, $(H_1 \uplus H_2)(x) = i$. □

3.1 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:

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

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:

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

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

Theorem 3.5 *If* $P \xmapsto{\text{fva}} P'$, *then* $P \xmapsto{\text{fv}} P'$.

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

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

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

If we require that a collection algorithm produce a *closed* program, then **fva** is "optimal" in the following weak sense:

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.

3.2 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 3.6 (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 $1 \leq i < n$, $FV(H_i) \cap Dom(H_{i+1} \uplus H_{i+2} \uplus \cdots \uplus H_n) = \emptyset$. 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 3.7 (Generational Collection)
Let $H_1, \ldots, H_n$ be a generational partition of the heap of $P = \mathsf{letrec}\ H\ \mathsf{in}\ e$. Suppose $H_i = (H_i^1 \uplus H_i^2)$, and $Dom(H_i^2) \cap FV(\mathsf{letrec}\ H_i^1 \uplus H_{i+1} \uplus \cdots \uplus H_n\ \mathsf{in}\ e) = \emptyset$. Then $P \overset{\text{fv}}{\longmapsto} \mathsf{letrec}\ (H \setminus H_i^2)\ \mathsf{in}\ e$.

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

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

□

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

Theorem 3.8 *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 $P \overset{\text{R}}{\longmapsto} \mathsf{letrec}\ H \uplus H'\ \mathsf{in}\ e$, then $H_1, \ldots, H_n, H'$ is a generational partition of $H \uplus H'$.*

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.

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

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

The third reason generational collection is important is that empirical evidence shows that "objects tend to die young" [28]. 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 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 [26]. 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, 13, 27, 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. 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.

4.1 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 2). Heap values are not paired with their type, reflecting the fact that the memory is "almost tag free." Of course, some type information is recorded within heap values that are functions. In a low-level model that uses closures (an explicit value environment paired with code), this corresponds to maintaining a *type environment*, recording the types of the values in the closure's environment. The type environment for a closure can be computed at compile time and shared among multiple instances, just like the code.

The evaluation contexts (E) consist of a raw context (U) and a type (τ). Raw contexts are filled with instructions (I)

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:

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

Rewriting Rules:

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

Figure 2: The Syntax and Operational Semantics of λgc-mono

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 on the sub-expressions of the program's body. Allocation of integers, tuples and functions strips the type tag off 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 4.1 (λgc-mono Stuck Programs) *A program is stuck if it is of one of the following forms:*

$$\begin{array}{ll}
\mathsf{letrec}\ H\ \mathsf{in}\ E[\pi_i\ (x{:}\tau)] & (x \notin \mathit{Dom}(H)\ \mathit{or} \\
 & \ H(x) \neq \langle x_1, x_2\rangle) \\
\mathsf{letrec}\ H\ \mathsf{in}\ E[(x{:}\tau_1)\ (y{:}\tau_2)] & (x\ \mathit{or}\ y \notin \mathit{Dom}(H)\ \mathit{or} \\
 & \ H(x) \neq \lambda z{:}\tau.e)
\end{array}$$

The static semantics of λgc-mono consists of four judgements about program components. The first judgement, $\Gamma \rhd e$, is a binary relation between a type assignment Γ, mapping a finite set of variables to types, and a typed expression e. Figure 3 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 $\Gamma \rhd h : \tau$ which asserts that the heap value h has type τ under the assumptions in Γ. The judgement is defined using inference rules (not shown here) similar to the expression-level rules. The second is $\Gamma \rhd H : \Gamma'$, which asserts that the variables 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 requires "guessing" the types of the values in the heap and verifying these guesses simultaneously. The third judgement, $\rhd P$, asserts the well-typing of a complete program.

The calculus λgc-mono is *type sound* in that evaluation of well formed programs cannot get stuck [30]. A key to the proof of soundness is a *type preservation* lemma:

Lemma 4.2 (Type Preservation) *If* $\rhd P$ *and* $P \xmapsto{\text{RM}} P'$, *then* $\rhd P'$.

Theorem 4.3 (Type Soundness) *If* $\rhd P$ *then either* P *is an answer or else there exists some* P' *such that* $P \xmapsto{\text{RM}} P'$ *and* $\rhd P'$.

4.2 Using Types in Garbage Collection

Specifying a valid garbage collection rule that exploits types is straightforward. The key property that the rule must preserve is *type preservation.* That is, if $P \xmapsto{\text{gc}} P'$, 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:

$$(\textbf{mono})\quad \begin{array}{c}\mathsf{letrec}\ H_1 \uplus H_2\ \mathsf{in}\ e \xmapsto{\text{mono}} \mathsf{letrec}\ H_1\ \mathsf{in}\ e \\ \text{if } \rhd \mathsf{letrec}\ H_1\ \mathsf{in}\ e\end{array}$$

Theorem 4.4 *For all well formed λgc-mono programs P,* $(P, RM) \simeq (P, RM + \textbf{mono})$.

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.

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

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

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

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

Figure 3: The Static Semantics of λgc-mono

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

If $P = \mathsf{letrec}\ H\ \mathsf{in}\ e$ 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.

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

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 4.6 *If $\emptyset \triangleright H : \Gamma \uplus \{x{:}\tau\}$ then there exists an h such that $\{x = h\} \subseteq H$, $MTA(Tag[\tau](h)) \triangleright h : \tau$, and $MTA(Tag[\tau](h)) \subseteq \Gamma \uplus \{x{:}\tau\}$*

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:

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

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

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:

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

The following definition gives the primary invariants of the algorithm:

Definition 4.7 (mono-a Invariants) $\langle H_f, \Gamma_s, H_t, \Gamma_t\rangle$ *satisfies the* **mono-a** *invariants with respect to a program* $\mathsf{letrec}\ H\ \mathsf{in}\ e$ *and type assignment* Γ_0 *iff:*

1. $H = H_f \uplus H_t$
2. $\Gamma_s \uplus \Gamma_t \triangleright e$
3. $Dom(\Gamma_s) \subseteq Dom(H_f)$
4. $\Gamma_s \triangleright H_t : \Gamma_t$
5. $\Gamma_s \uplus \Gamma_t \subseteq \Gamma_0$

Intuitively, the invariants guarantee (1) that each binding is accounted for, (2) every binding needed for e is in the scan-set or to-heap, (3) the scan-set corresponds to bindings in the from-heap, (4) the scan-set holds all free variables in the to-heap, and (5) the scan-set and to-heap agree with Γ_0.

Lemma 4.8 *If $\emptyset \triangleright H : \Gamma_0$, T has the* **mono-a** *invariant properties with respect to P and Γ_0, and $T \stackrel{\text{mono}}{\Longrightarrow} T'$, then T'*

[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.

has the **mono-a** *invariant properties with respect to* P *and* Γ_0.

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

Theorem 4.9 *If* P *is a well formed program and* $P \overset{\text{mono-a}}{\longmapsto} P'$, *then* $P \overset{\text{mono}}{\longmapsto} P'$.

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 $P \overset{\text{mono-a}}{\longmapsto} P'$, then $\triangleright P'$.

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

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

then by Lemma 4.8 taking $\Gamma_s = \emptyset$, we know that $\emptyset \triangleright H' : \Gamma_t$ and $\Gamma_t \triangleright e$. Thus, by the **prog** rule of the static semantics, $\triangleright\ \mathsf{letrec}\ H'\ \mathsf{in}\ e$. □

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

Theorem 4.10 *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 $P = \mathsf{letrec}\ H\ \mathsf{in}\ e$ 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 4.8, 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. □

Extending the **mono-a** collection algorithm to work for a language with explicit polymorphism, where types are passed to polymorphic routines at run time as suggested by various language implementors [21, 1, 27, 15], is straightforward because enough information is preserved by evaluation to always reconstruct the type of a polymorphic object. In our technical report [20], we show how this may be accomplished.

5 Collecting Reachable Garbage Using Type Inference

Thus 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.

For 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.

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:

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

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 Figure 1 for the syntax and dynamic semantics of λgc and Figure 3 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 4. These rules define judgements of the form $\Gamma \vdash e : \tau$, where Γ is a type assignment, e is a λgc expression, and τ is a type.

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.

We will show that if we can find a typing for a program that assigns a heap location a type variable, then the contents of that location has no effect on the rest of 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 *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 a 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 functional type ($\rightarrow$) if, whenever such answers are applied to appropriately related answers, the resulting computations yield related results. Our proof only covers programs without cycles in their heaps, but it should be possible to extend our arguments to all programs (see below for more details).

The relations are defined in Figure 5 by induction on types. The definitions are parameterized by an arbitrary

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

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

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

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

Figure 4: Type Inference Rules for λgc

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.

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 answers 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 $\Theta(t)$. The relation is extended to the other types in a natural fashion. For example, if τ is an arrow type $\tau_1 \to \tau_2$, 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. 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 logical relation on answers 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 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 is the key to establishing our result: an expression is related to itself in the context of any two related heaps.

Lemma 5.1 *For all $\Theta : Tvar \to \mathcal{P}(Ans \times Ans)$, 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 (sketch): By induction on the derivation of $\Gamma \vdash e : \tau$. Here we give the interesting case (**fn**). Suppose $\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2$ and $\Theta \models H_1 \approx H_2 : \Gamma$. By **alloc**,

$$\mathsf{letrec}\ H_1\ \mathsf{in}\ \lambda x.e \overset{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H_1 \uplus \{y_1 = \lambda x.e\}\ \mathsf{in}\ y_1$$
$$\mathsf{letrec}\ H_2\ \mathsf{in}\ \lambda x.e \overset{\text{alloc}}{\longmapsto} \mathsf{letrec}\ H_2 \uplus \{y_2 = \lambda x.e\}\ \mathsf{in}\ y_2$$

for some fresh y_1 and y_2. We must show that the two resulting answers are appropriately related.

Let H_1' and H_2' be heaps such that $H_i \uplus \{y_i = \lambda x.e\} \subseteq H_i'$ for $i = 1, 2$, and

$$\Theta \models \mathsf{letrec}\ H_1'\ \mathsf{in}\ z_1 \approx \mathsf{letrec}\ H_2'\ \mathsf{in}\ z_2 : \tau_1$$

for some z_1 and z_2. We must show that

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

Taking $H_i'' = H_i' \uplus \{x = H_i'(z_i)\}$, this follows if

$$\Theta \models \mathsf{letrec}\ H_1''\ \mathsf{in}\ e \sim \mathsf{letrec}\ H_2''\ \mathsf{in}\ e : \tau_2.$$

By the induction hypothesis, it suffices to show that

$$\Theta \models H_1'' \approx H_2'' : \Gamma \uplus \{x : \tau_1\}.$$

since e has a smaller typing derivation than $\lambda x.e$. By the lemma's hypothesis, we know that $\Theta \models H_1 \approx H_2 : \Gamma$, and since heaps remain related under extensions, $\Theta \models H_1'' \approx H_2'' : \Gamma$. By assumption,

$$\Theta \models \mathsf{letrec}\ H_1''\ \mathsf{in}\ z_1 \approx \mathsf{letrec}\ H_2''\ \mathsf{in}\ z_2 : \tau_1.$$

Since $H_i'' = H_i' \uplus \{x = H_i'(z_i)\}$, it is easy to show that

$$\Theta \models \mathsf{letrec}\ H_1''\ \mathsf{in}\ x \approx \mathsf{letrec}\ H_2''\ \mathsf{in}\ x : \tau_1$$

Consequently, H_1'' and H_2'' have related heap-values for each variable in $\Gamma \uplus \{x : \tau_1\}$ and thus $\Theta \models H_1'' \approx H_2'' : \Gamma \uplus \{x : \tau_1\}$. The other cases follow in a similar manner. □

Our goal is to show that if Γ is a type assignment, e an expression, and H a heap 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 5.1 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

Computations:

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

Answers:

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

Figure 5: Relational Interpretation of Types

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

Finally, we can state and prove the following Inference GC specification: Given a cycle-free program, if we can find a typing that assigns a heap location a type variable, then that location can be bound to "0" without effecting evaluation.

Theorem 5.3 (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 5.2, 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 5.1 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 first hypothesis. By the definition of $\approx$ at int, $H_b(y) = i$ (or $H_a(x) = i$). Thus,

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

□

Using the Inference GC theorem, we can now show that the binding $x_3 = \langle x_2, x_2 \rangle$ in the program:

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

can be replaced by $x_3 = 0$. Taking $\Gamma_1 = \{x_2{:}t_2, x_3{:}t_3\}$, $\Gamma_2 = \{x_1{:}\mathsf{int}, x_4{:}\mathsf{int} \times t_3\}$, $H_1 = \{x_2 = 2, x_3 = \langle x_2, x_2 \rangle\}$, $H_2 = \{x_1 = 1, x_4 = \langle x_1, x_3 \rangle\}$ it is easy to show that the preconditions of the theorem hold. Consquently, replacing H_1 with $H_1' = \{x_2 = 0, x_3 = 0\}$ results in a Kleene equivalent program:

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

Now by invoking **fv**, we can collect the binding $x_2 = 0$:

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

and we know that the resulting program is Kleene equivalent to the original program.

6 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 model the type-based collectors of Sections 4 and 5. Nettles [22] 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.

Hudak gives a denotational model that tracks reference counts for a first-order language [16]. 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. Both Hudak and Chirimar *et al.* assume a weak approximation of garbage (reference counts). Barendsen and Smetsers give a Curry-like type system for functional languages extended with *uniqueness* information that guarantees an object is only "locally acccessible" [6]. This provides a compiler enough information to determine when certain objects may be garbage collected or over-written.

Tolmach [27] built a type-recovery collector for a variant of SML that passes type information to polymorphic routines during execution, effectively implementing a polymorphic version of our language and collector described in Section 4. 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].

Over the past few years, a number of papers on inference-based collection in monomorphic [7, 29, 8] and polymorphic [3, 13, 14, 12] languages appeared in the literature. 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 [5] 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 [14] 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's abstraction/parametricity theorem [24]. 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 give 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 [23, 25] and Launchbury [17] 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.* [4] have presented a purely syntactic theory of the call-by-need λ-calculus that is largely compatible with our work.

7 Summary and Future Work

Our paper provides a unifying 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 flexible, 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. In the future, we hope to extend the Inference GC theorem of Section 5 to cover programs with cyclic heaps. One approach is to interpret types as CPOs and prove that programs have a suitable *compactness* property: every cyclic heap is appropriately approximated by some finite "unwinding" of the heap. An alternative approach might be to provide a purely syntactic argument that locations assigned a type variable never arise in an "active" position during evaluation. We also hope to provide a proof of correctness for an algorithm that simultaneously infers types and performs garbage collection.

8 Acknowledgements

Thanks to Edo Biagoni, Andrzej Filinski, Mark Leone, Jack Morrisett, Chris Okasaki, Andrew Tolmach, 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, Orlando, FL, 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, 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.

[5] 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, Nice, France, June 1990.

[6] E. Barendsen and S. Smetsers. Conventional and uniqueness typing in graph rewrite systems. In *Proceedings of the 13th Conference on the Foundations of Software Technology and Theoretical Computer Science 1993, Bombay*, New York, NY, 1993. Springer-Verlag. Extended abstract.

[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, San Francisco, CA, 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, San Francisco, CA, 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, Orlando, FL, June 1994.

[13] 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, Toronto, Ontario, Canada, June 1991.

[14] 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, San Francisco, CA, June 1992.

[15] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the 22nd Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, Jan. 1995.

[16] 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, Cambridge, MA, 1986.

[17] 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.

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

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

[20] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. Technical Report CMU–CS–95–110, School of Computer Science, Carnegie Mellon University, Jan. 1995. Available as `ftp://reports.adm.cs.cmu.edu/usr/anon/1995/CMU-CS-95-110.ps`.

[21] 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.

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

[23] 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.

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

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

[26] 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, Palo Alto, CA, Oct. 1987.

[27] 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, Orlando, FL, June 1994.

[28] 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, PA, Apr. 1984.

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

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