Semantics of Memory Management for Polymorphic Languages

Greg Morrisett Robert Harper

September, 1996
CMU-CS-96-176

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Also published as Fox Memorandum CMU–CS–FOX–96–04.

This research was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software", ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050, and in part by the National Science Foundation under Grant No. CCR-9502674.

Keywords: type theory, operational semantics, garbage collection, memory management

Abstract

We present a static and dynamic semantics for an abstract machine that evaluates expressions of a polymorphic programming language. Unlike traditional semantics, our abstract machine exposes many important issues of memory management, such as value sharing and control representation. We prove the soundness of the static semantics with respect to the dynamic semantics using traditional techniques. We then show how these same techniques may be used to establish the soundness of various memory management strategies, including type-based, tag-free garbage collection; tail-call elimination; and environment strengthening.

1 Introduction

Type theory and operational semantics are remarkably effective tools for programming language design and implementation [28, 13]. An important and influential example is provided by *The Definition of Standard ML* (SML) [28]. The *static semantics* of SML is specified as a collection of *elaboration rules* that defines the context-sensitive constraints on the formation of programs. The *dynamic semantics* is specified as a collection of *evaluation rules* that defines the operational semantics of a program. The static and dynamic semantics are related by a *type soundness* theorem stating that certain forms of run-time error cannot arise in the evaluation of a well-formed program. The methodology of *The Definition of Standard ML* has been refined in a number of subsequent studies of the type theory and operational semantics of SML and related languages.

Of particular interest for purposes of this paper is the variety of methods for defining the operational semantics of deterministic, sequential languages. Two main approaches have emerged, one based on *evaluation relations*, the other based on *transition systems*. The evaluation-based approach is typified by Kahn's *natural semantics* [16] and is used extensively in *The Definition of Standard ML*. The transition-based approach is typified by Plotkin's *structured operational semantics* [33], but also includes approaches based on abstract machines [18] and program rewriting [21, 44]. Both approaches share the goal of achieving a "fully abstract" semantics that suppresses irrelevant details, avoids over-specification, and facilitates reasoning about programs. Experience has shown that these goals are difficult to achieve in a single framework.

As a case in point, we consider the memory allocation behavior of programs. A significant advantage of high-level programming languages, such as SML, is that the details of memory management are inaccessible to the programmer. For example, in SML it is impossible to determine whether or not a pair of values is allocated in the heap or in registers. This is not an oversight! Rather, the intention is to free the programmer from the details of memory management, and to allow the compiler to make representation choices based on contingencies not entirely within the programmer's control. (See Appel's critique for a discussion of this and related points [5].)

Applying the full abstraction criterion discussed above, the operational semantics of such languages should abstract away the details of memory management from the definition of the language. Indeed, the dynamic semantics given in *The Definition of Standard ML* avoids explicit treatment of memory allocation insofar as it is observable through the use of reference types. The semantics freely forms tuples, environments, closures, and recursive data structures without regard to their representation in memory. Consequently, no accounting of memory sharing is provided by the semantics.

For many purposes, such as reasoning about the extensional behavior of programs, this approach is ideal. Yet, issues of memory management cannot be entirely overlooked. For example, an important use of operational semantics is to serve as a guide to the compiler writer, who must make data structure representation decisions that critically affect the performance of compiled code. In this case it is essential to make storage allocation decisions explicit in the semantics. Otherwise, important notions, such as "space safety" [4, 37], "tail recursion" [17], and "garbage collection" [42], remain vague notions outside of the scope of a rigorous semantics.

In this paper we propose to explore the use of operational semantics to define not only the high-level execution behavior of programs, but also their low-level allocation behavior. We consider as a case study an explicitly-typed, polymorphic programming language with unbounded recursion, product (tuple) types, and a natural numbers type. This language is sufficiently rich to encompass important issues, including allocation of types at run-time, allocation of aggregate data structures, inductively defined data structures, and the representation of types as data structures. Yet, it is sufficiently simple to admit a rigorous treatment of its memory allocation behavior and sharing of storage among complex values. To do so, we give an operational semantics for the language formulated as a transition system between states of an abstract machine. The machine state includes a *heap*, containing allocated types and data; an *environment*, containing types and bindings for variables; a *stack*, containing control information; and an *expression* to be evaluated. The operational semantics is related to the type system by a soundness theorem characterizing the shapes of values of each type.

To illustrate the use of the framework, we consider in detail several critical storage management problems. We give a detailed treatment of *tag-free copying garbage collection*. The collector is presented as a transition system that faithfully captures the behavior of a copying garbage collector, including the use of type information to "parse" and "trace" heap values during collection. We provide the first proof of correctness for such a collector, a significant advance on current practice. In addition, we discuss two other forms of garbage collection: *tail recursion elimination*, which reduces the space required by the control stack, and *black holing*, which reduces the space required by environments. All of these memory management techniques are used within the TIL/ML compiler [38], and thus the material presented here provides a faithful model of this particular implementation. Nevertheless, the framework we propose is general enough to model a variety of language implementations.

The rest of this paper is organized as follows: In Section 2, we present the syntax and static semantics of our core polymorphic language, $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$. In Section 3, we present an abstract machine for evaluating $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ expressions. The section provides both a static and dynamic semantics for the abstract machine and a proof of type soundness.

In Section 4, we consider the issue of heap garbage and a specification for a general-purpose heap-garbage collector. We show the soundness of a particular class of collectors, namely those based on inaccessibility of heap objects. In Section 5, we show how to implement a particular heap-garbage collection algorithm, namely the *type-based, tag-free* garbage collector used by Tolmach [40], which is closely related to the *mostly tag-free* collector used by the TIL/ML compiler [38]. We prove the correctness of the algorithm using syntactic techniques similar to those used to prove type soundness for the abstract machine.

In Section 6, we consider other kinds of garbage in the abstract machine, notably stack garbage and environment garbage. We show how the addition of a tail-call facility can be used to eliminate a certain class of stack garbage, and how the addition of environment strengthening rules can be used to eliminate a certain class of environment garbage. Again, correctness of these memory management techniques can be shown through the syntactic methods employed in previous sections.

Finally, we discuss related work in Section 7, and we summarize and conclude in Section 8.

2 The $\lambda_{\rm gc}^{\rightarrow\forall}$ Language

In this section, we present $\lambda_{\rm gc}^{\rightarrow\forall}$, a call-by-value variant of the Girard-Reynolds polymorphic λ-calculus [24, 34]. In the following section, we define an abstract machine for evaluating $\lambda_{\rm gc}^{\rightarrow\forall}$ expressions. The abstract machine makes explicit many operational details that are pertinent to memory management, such as the heap, the control stack, and the environment.

Perhaps the most novel aspect of $\lambda_{\rm gc}^{\rightarrow\forall}$ is that, unlike traditional models of typed-languages, type information is maintained throughout evaluation in order to support type-based, tag-free garbage collection as implemented by Tolmach [40] and in the TIL/ML compiler [38]. Therefore, modeling allocation, sharing, and garbage collection of *types* is just as important as modeling memory management for *values*.

To simplify the abstract machine, the expressions and types of $\lambda_{\rm gc}^{\rightarrow\forall}$ are restricted to *named form*, also known as *A Normal Form* [36]. The restriction to named form amounts to the requirement that the result of every step of evaluation or allocation be bound to a variable, which is then used to refer to the result of this computation. Every expression and every type of the second-order λ-calculus can be put into named form by simply introducing `let` and `let type` expressions appropriately.

$$
\begin{array}{lll}
\text{(type variable)} & t & \\
\text{(type)} & \sigma & ::= \; t \mid \mathsf{nat} \mid \mathsf{unit} \mid \sigma_1 \times \sigma_2 \mid \sigma_1 \to \sigma_2 \mid \forall t.\sigma \\
 & & \\
\text{(named form type)} & \tau & ::= \; t \mid \mathsf{let}\ t = \nu\ \mathsf{in}\ \tau \\
\text{(type binding)} & \nu & ::= \; \mathsf{nat} \mid \mathsf{unit} \mid t_1 \times t_2 \mid t_1 \to t_2 \mid \forall t.\tau \\
 & & \\
\text{(value variable)} & x & \\
\text{(named form expression)} & e & ::= \; x \mid \mathtt{let}\ x{:}t = b\ \mathtt{in}\ e \mid \mathtt{let\ type}\ t = \nu\ \mathtt{in}\ e \\
\text{(expression binding)} & b & ::= \; a \mid c \\
\text{(allocation binding)} & a & ::= \; \mathsf{0} \mid \mathtt{succ}\ x \mid \langle\rangle \mid \langle x_1, x_2 \rangle \mid \mathtt{fix}\ x{:}t(x_1{:}t_1).e \mid \Lambda t.\, e \\
\text{(computation binding)} & c & ::= \; \mathtt{case}(x, e_0, \lambda x_1{:}t_1.e_1) \mid \pi_i\, x \mid x_1\, x_2 \mid x\,[t]
\end{array}
$$

Figure 1: Syntax of the $\lambda_{\mathrm{gc}}^{\to\forall}$Language

For example, one named form representation of the expression $(\lambda x{:}\mathsf{nat}.\,\mathtt{succ}\ \ x)\,\mathsf{0}$ is:

$$
\begin{array}{l}
\mathtt{let\ type}\ t = \mathsf{nat}\ \mathtt{in} \\
\mathtt{let\ type}\ t_1 = t \to t\ \mathtt{in} \\
\mathtt{let}\ x_1{:}t_1 = \lambda x{:}t.\,(\mathtt{let}\ x_2{:}t = \mathtt{succ}\ x\ \mathtt{in}\ x_2)\ \mathtt{in} \\
\mathtt{let}\ x_3{:}t = \mathsf{0}\ \mathtt{in} \\
\mathtt{let}\ x_4{:}t = x_1\, x_3 \\
\mathtt{in}\ x_4
\end{array}
$$

The restriction to named form is largely a matter of technical convenience. The penalty is that the typing rules are somewhat more complicated since we must expand bindings of type variables during type checking (see Section 2.2). The advantage is that we can easily recover the type of an expression from the types attached to the bindings of its sub-expressions.

2.1 Syntax of $\lambda_{\mathrm{gc}}^{\to\forall}$

The syntax of the $\lambda_{\mathrm{gc}}^{\to\forall}$ language is defined in Figure 1. Types include type variables, nat, unit, binary products, arrow types, and type abstractions. Instead of using types directly to decorate $\lambda_{\mathrm{gc}}^{\to\forall}$ terms, we use a named form representation of types in order to make allocation and sharing of type information explicit. A named form type is either a type variable or a let that binds a named form type *binding* (ν) to a variable in the scope of a named form type. Named form type bindings include primitive constructors (e.g., nat), compound constructors where the components are type variables (e.g., $t_1 \times t_2$ and $t_1 \to t_2$), or a type abstraction where the body of the abstraction is a named form type.

The expressions of $\lambda_{\mathrm{gc}}^{\to\forall}$ are also required to be in named form. They are variables, `let` expressions binding a named form binding to a variable in the scope of a named form expression, and `let type` expressions binding a named form type binding to a type variable in the scope of a named form expression.

A named form expression binding is either an allocation binding or a computation binding. Allocation bindings correspond to values that will be allocated on the heap. These consist of primitive values (e.g., `0`), compound values where the components are variables (e.g., `succ` x and $\langle x_1, x_2\rangle$), and recursive abstractions (`fix` $x{:}t(x_1{:}t_1).e$). We use $\lambda x_1{:}t_1.e$ to abbreviate a recursive abstraction `fix` $x{:}t(x_1{:}t_1).e$ where x does not occur free in e. Computation bindings correspond to computational steps to be taken during evaluation. These consist of a `case` expression for testing natural numbers, projection for pairs, and application for both term and type abstractions.

We have chosen to allocate all values to simplify the presentation. However, it is straightforward to modify the language to support unallocated naturals, for instance, by defining a syntactic class of "small" values and by allowing small values to occur within bindings. This corresponds to the use of machine registers, rather than memory locations, to store values.

The binding conventions for the language are familiar: t is bound in σ for $\forall t.\sigma$, t is bound in τ for $\mathsf{let}\ t = \nu\ \mathsf{in}\ \tau$, t is bound in τ for $\forall t.\tau$, t is bound in e for `let type` $t = \nu$ `in` e, x is bound in e for `let` $x{:}t = b$ `in` e, x and x_1 are bound in e for `fix` $x{:}t(x_1{:}t_1).e$, and t is bound in e for $\Lambda t.\, e$.

All syntactic objects are identified up to a systematic renaming (α-conversion) of bound variables. We use $FV(X)$ to denote the free value variables of a syntactic object X, and $FTV(X)$ to denote the free type variables of X. Capture-avoiding substitution is defined as usual, given the binding conventions listed above.

2.2 Typing Rules for $\lambda_{\mathrm{gc}}^{\to\forall}$

The typing rules for $\lambda_{\mathrm{gc}}^{\to\forall}$ are defined relative to contexts declaring type variables and value variables.

$$\begin{array}{lcll} \text{(variable type assignment)} & \Gamma & ::= & \emptyset \mid \Gamma[x{:}\sigma] \\ \text{(type variable context)} & \Delta & ::= & \emptyset \mid \Delta[t] \mid \Delta[t = \sigma] \end{array}$$

We consider type assignments as finite maps from value variables to types. Hence, the order of bindings in a type assignment is considered irrelevant, and a variable may not be declared more than once in a single type assignment.

There are two forms of type variable declarations, *abstract* declarations ($[t]$) and *transparent* declarations ($[t = \sigma]$). Abstract declarations are used when processing a polymorphic abstraction (e.g., $\Lambda t.\, e$), whereas transparent declarations are used when processing a `let type` binding (e.g., $\mathsf{let}\ t = \nu\ \mathsf{in}\ \tau$ and `let type` $t = \nu$ `in` e). We define $Abstr(\Delta)$ to be the set of abstract bindings in the context Δ, $Transp(\Delta)$ to be the domain of the set of transparent bindings in Δ, and $Dom(\Delta) = Abstr(\Delta) \cup Transp(\Delta)$. More precisely,

$$\begin{aligned} Abstr(\emptyset) &= \emptyset \\ Abstr(\Delta[t]) &= Abstr(\Delta) \cup \{t\} \\ Abstr(\Delta[t = \sigma]) &= Abstr(\Delta) \end{aligned}$$

$$\begin{aligned} Transp(\emptyset) &= \emptyset \\ Transp(\Delta[t]) &= Transp(\Delta) \\ Transp(\Delta[t = \sigma]) &= Transp(\Delta) \cup \{t\} \end{aligned}$$

The well-formed type variable contexts are defined as follows: The empty context ($\emptyset$) is well-formed; The context $\Delta[t]$ is well-formed iff Δ is well-formed and $t \notin Dom(\Delta)$; The context $\Delta[t = \sigma]$ is well-formed iff Δ is well-formed, $t \notin Dom(\Delta)$, and the free type variables of σ are a subset of $Abstr(\Delta)$. Hence, a free type variable occurring in a transparent binding must be previously declared as an abstract type variable. A type assignment Γ is well-formed with respect to a context Δ iff $FTV(\Gamma) \subseteq Abstr(\Delta)$. Finally, we consider type variable contexts equivalent up to any re-ordering of the bindings that respects the dependencies of transparent bindings on abstract bindings.

Judgments of the typing rules are listed in Figure 2, and the axioms and inference rules that may be used to derive these judgments are given in Figures 3 and 4.

Intuitively, the first two judgments ($\Delta \vdash \tau \Downarrow \sigma$ and $\Delta \vdash \nu \Downarrow \sigma$) substitute transparent type bindings in Δ, for free type variables in τ/ν, and eliminate any nested let-expressions within the named form type to obtain the equivalent conventional type σ. Judgments 3 and 4 are derived from conventional typing rules for the polymorphic λ-calculus.

Throughout, we assume that all judgment components are well-formed. For example, in order to derive $\Delta; \Gamma \vdash e : \sigma$, we assume that Δ is well-formed and Γ is well-formed with respect to Δ. Hence, many side-conditions, such as the requirement that a variable not be bound twice in a context, are left implicit. From the well-formedness condition and the rules, we may derive the following properties of the typing judgments.

Lemma 2.1

1. *If $\Delta \vdash \tau \Downarrow \sigma$, then $FTV(\tau) \subseteq Dom(\Delta)$ and $FTV(\sigma) \subseteq Abstr(\Delta)$.*

1.	$\Delta \vdash \tau \Downarrow \sigma$	named form type τ reduces to σ	(see Figure 3)
2.	$\Delta \vdash \nu \Downarrow \sigma$	type binding ν reduces to σ	(see Figure 3)
3.	$\Delta; \Gamma \vdash e : \sigma$	expression e has type σ	(see Figure 4)
4.	$\Delta; \Gamma \vdash b : \sigma$	binding b has type σ	(see Figure 4)

Figure 2: Typing Judgments for the $\lambda_{\text{gc}}^{\rightarrow\forall}$ Language

2. *If* $\Delta \vdash \nu \Downarrow \sigma$*, then* $FTV(\nu) \subseteq Dom(\Delta)$ *and* $FTV(\sigma) \subseteq Abstr(\Delta)$*.*

3. *If* $\Delta; \Gamma \vdash e : \sigma$*, then* $FV(e) \subseteq Dom(\Gamma)$*,* $FTV(e) \subseteq Dom(\Delta)$*, and* $FTV(\sigma) \subseteq Abstr(\Delta)$*.*

4. *If* $\Delta; \Gamma \vdash b : \sigma$*, then* $FV(e) \subseteq Dom(\Gamma)$*,* $FTV(b) \subseteq Dom(\Delta)$*, and* $FTV(\sigma) \subseteq Abstr(\Delta)$*.*

3 The $\lambda_{\text{gc}}^{\rightarrow\forall}$ Abstract Machine

The dynamic semantics of the $\lambda_{\text{gc}}^{\rightarrow\forall}$ language is given by a transition system between states of an abstract machine. The abstract machine is derived from the CESK machine of Felleisen and Friedman [20]. States of the machine are a quadruple (H, S, E, e) where H is a *heap*, S is a *stack*, E is an *environment*, and e is an expression of the $\lambda_{\text{gc}}^{\rightarrow\forall}$ language. The heap consists of a *type heap* containing allocated types, and a *value heap* containing allocated values. The environment consists of a *type environment* providing bindings for type variables, and a *value environment* providing types and values for ordinary variables. The stack consists of a composition of *frames*, each of which is a *closure* consisting of an environment and a λ-term.

This organization faithfully reflects a conventional implementation of the $\lambda_{\text{gc}}^{\rightarrow\forall}$ language, except that it abstracts from the allocation of environments. (For a treatment of this topic, see Minamide, Morrisett, and Harper's account of closure conversion for a typed language [29].) In particular, this organization is a fairly accurate model of the run-time data structures used by the TIL/ML compiler [38].

To establish soundness of the type system we, define a syntactic typing discipline for the states of the abstract machine, and prove progress and preservation lemmas for it. Although they do not arise in the simple language considered here, cyclic data structures are compatible with the syntactic type discipline and present no

$\boxed{1. \quad \Delta \vdash \tau \Downarrow \sigma}$

$$\textbf{(opaque)}\ \ \Delta[t] \vdash t \Downarrow t \qquad\qquad \textbf{(transp)}\ \ \Delta[t = \sigma] \vdash t \Downarrow \sigma$$

$$\textbf{(look-opaque)}\ \ \frac{\Delta \vdash t \Downarrow \sigma}{\Delta[t'] \vdash t \Downarrow \sigma}\ \ (t \neq t') \qquad \textbf{(look-transp)}\ \ \frac{\Delta \vdash t \Downarrow \sigma}{\Delta[t' = \sigma'] \vdash t \Downarrow \sigma}\ \ (t \neq t')$$

$$\textbf{(type-def)}\ \ \frac{\Delta \vdash \nu \Downarrow \sigma' \qquad \Delta[t = \sigma'] \vdash \tau \Downarrow \sigma}{\Delta \vdash \mathsf{let}\ t = \nu\ \mathsf{in}\ \tau \Downarrow \sigma}$$

$\boxed{2. \quad \Delta \vdash \nu \Downarrow \sigma}$

$$\textbf{(nat)}\ \ \Delta \vdash \mathsf{nat} \Downarrow \mathsf{nat} \qquad\qquad \textbf{(unit)}\ \ \Delta \vdash \mathsf{unit} \Downarrow \mathsf{unit}$$

$$\textbf{(prod)}\ \ \frac{\Delta \vdash t_1 \Downarrow \sigma_1 \qquad \Delta \vdash t_2 \Downarrow \sigma_2}{\Delta \vdash t_1 \times t_2 \Downarrow \sigma_1 \times \sigma_2}$$

$$\textbf{(arrow)}\ \ \frac{\Delta \vdash t_1 \Downarrow \sigma_1 \qquad \Delta \vdash t_2 \Downarrow \sigma_2}{\Delta \vdash t_1 \to t_2 \Downarrow \sigma_1 \to \sigma_2}$$

$$\textbf{(all)}\ \ \frac{\Delta[t] \vdash \tau \Downarrow \sigma}{\Delta \vdash \forall t.\tau \Downarrow \forall t.\sigma}$$

Figure 3: Named Form Type Reduction

$\boxed{3.\quad \Delta;\Gamma \vdash e : \sigma}$

$$\textbf{(var)}\ \ \Delta;\Gamma[x{:}\sigma] \vdash x : \sigma$$

$$\textbf{(let-exp)}\ \frac{\Delta;\Gamma \vdash b : \sigma' \qquad \Delta \vdash t \Downarrow \sigma' \qquad \Delta;\Gamma[x{:}\sigma'] \vdash e : \sigma}{\Delta;\Gamma \vdash \texttt{let}\ \ x{:}t = b\ \ \texttt{in}\ \ e : \sigma} \qquad \textbf{(let-type)}\ \frac{\Delta \vdash \nu \Downarrow \sigma' \qquad \Delta[t = \sigma'];\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \texttt{let type}\ \ t = \nu\ \ \texttt{in}\ \ e : \sigma}$$

$\boxed{4.\quad \Delta;\Gamma \vdash b : \sigma}$

$$\textbf{(nat-I1)}\ \ \Delta;\Gamma \vdash \mathsf{0} : \mathsf{nat} \qquad \textbf{(nat-I2)}\ \frac{\Delta;\Gamma \vdash x : \mathsf{nat}}{\Delta;\Gamma \vdash \mathsf{succ}\ \ x : \mathsf{nat}}$$

$$\textbf{(nat-E)}\ \frac{\Delta;\Gamma \vdash x : \mathsf{nat} \qquad \Delta;\Gamma \vdash e_0 : \sigma \qquad \Delta \vdash t_1 \Downarrow \mathsf{nat} \qquad \Delta;\Gamma[x_1{:}\mathsf{nat}] \vdash e_1 : \sigma}{\Delta;\Gamma \vdash \mathsf{case}(x, e_0, \lambda x_1{:}t_1.e_1) : \sigma} \qquad \textbf{(unit-I)}\ \ \Delta;\Gamma \vdash \langle\rangle : \mathsf{unit}$$

$$\textbf{(prod-I)}\ \frac{\Delta;\Gamma \vdash x_1 : \sigma_1 \qquad \Delta;\Gamma \vdash x_2 : \sigma_2}{\Delta;\Gamma \vdash \langle x_1, x_2\rangle : \sigma_1 \times \sigma_2}$$

$$\textbf{(prod-E)}\ \frac{\Delta;\Gamma \vdash x : \sigma_1 \times \sigma_2}{\Delta;\Gamma \vdash \pi_i\, x : \sigma_i} \quad (i = 1,2)$$

$$\textbf{(arrow-I)}\ \frac{\Delta \vdash t \Downarrow \sigma_1 \to \sigma_2 \qquad \Delta \vdash t_1 \Downarrow \sigma_1 \qquad \Delta;\Gamma[x{:}\sigma_1 \to \sigma_2, x_1{:}\sigma_1] \vdash e : \sigma_2}{\Delta;\Gamma \vdash \texttt{fix}\ \ x{:}t(x_1{:}t_1).e : \sigma_1 \to \sigma_2} \qquad \textbf{(arrow-E)}\ \frac{\Delta;\Gamma \vdash x : \sigma_1 \to \sigma \qquad \Delta;\Gamma \vdash x_1 : \sigma_1}{\Delta;\Gamma \vdash x\, x_1 : \sigma}$$

$$\textbf{(all-I)}\ \frac{\Delta[t];\Gamma \vdash e : \sigma}{\Delta;\Gamma \vdash \Lambda t.\, e : \forall t.\sigma} \qquad \textbf{(all-E)}\ \frac{\Delta;\Gamma \vdash x_1 : \forall t.\sigma \qquad \Delta \vdash t_1 \Downarrow \sigma'}{\Delta;\Gamma \vdash x_1\,[t_1] : \{\sigma'/t\}\sigma}$$

Figure 4: Expression Typing

(pointer)	p		
(type environment)	TE	$::=$	$\emptyset \mid TE[t \mapsto p]$
(type heap value)	μ	$::=$	$\mathsf{nat} \mid \mathsf{unit} \mid p_1 \times p_2 \mid p_1 \rightarrow p_2 \mid \langle\!\langle TE, \forall t.\tau \rangle\!\rangle$
(type heap)	TH	$::=$	$\emptyset \mid TH[p \mapsto \mu]$
(location)	l		
(value environment)	VE	$::=$	$\emptyset \mid VE[x{:}t \mapsto l]$
(heap value)	h	$::=$	$\mathsf{0} \mid \mathtt{succ}\ l \mid \langle\rangle \mid \langle l_1, l_2 \rangle \mid \langle\!\langle E, \mathtt{fix}\ x{:}t(x_1{:}t_1).e \rangle\!\rangle \mid \langle\!\langle E, \Lambda t.\, e \rangle\!\rangle$
(value heap)	VH	$::=$	$\emptyset \mid VH[l \mapsto h]$
(environment)	E	$::=$	(TE, VE)
(heap)	H	$::=$	(TH, VH)
(stack)	S	$::=$	$[]_\sigma \mid S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle$
(program)	P	$::=$	(H, S, E, e)
(answer)	A	$::=$	$(H, []_\sigma, E, x)$

Figure 5: Syntax of the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$Machine

difficulties for extending the proofs of the main results. (This is in contrast to the complex fixed point constructions used elsewhere [39].)

The remainder of this section is organized as follows: In Section 3.1, we present the syntax of the constructs that make up the abstract machine. In Section 3.2, we present the static semantics for the abstract machine. In Section 3.3, we present the transition system for the abstract machine, and in Section 3.4, we present a proof of soundness of the static semantics with respect to this transition system.

3.1 Syntax of the Abstract Machine

The syntax of the states of the abstract machine is given in Figure 5. Each state or *program* P is a 4-tuple, (H, S, E, e), where H is a *heap*, S is a *stack*, E is an *environment*, and e is a $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ expression.

Environments contain a type environment mapping type variables to pointers (to heap-allocated types), and a value environment mapping value variables to type variables and locations (of heap-allocated values). It is important to emphasize that a value environment maps a value variable to both a type and a location.

Heaps consist of a type heap, mapping type pointers to type heap values (e.g.,

$p \mapsto \mu$), and a value heap, mapping value locations to heap values (e.g., $l \mapsto h$). Type heap values include base constructors (e.g., nat); n-ary constructors, where the component types are pointers to other heap-allocated types (e.g., $p_1 \rightarrow p_2$); and type closures ($\langle\langle TE, \forall t.\tau \rangle\rangle$). Type closures contain a named form representation of a polymorphic type ($\forall t.\tau$) and a type environment mapping the free type variables of the polymorphic type to heap-allocated type values. Heap values include primitive values (e.g., 0); constructed values, where the components are locations of other heap-allocated values (e.g., $\langle l_1, l_2 \rangle$); or value closures. Value closures ($\langle\langle E, e \rangle\rangle$) consist of an environment and a named form representation of either a type- or value-abstraction. The environment of a closure provides bindings for the free type and value variables of the closure's abstraction.

Stacks are either empty ($[]_\sigma$) or else a composition of a stack and a stack frame. Stack frames ($\langle\langle E, \lambda x{:}t.e \rangle\rangle$) are represented as closures with an environment and a value abstraction. The environment provides bindings for the free type and value variables of the abstraction. Intuitively, by composing the closures that make up the stack frame, we obtain the current "continuation" for the abstract machine. If we chose to restrict $\lambda_{\text{gc}}^{\rightarrow\forall}$ expressions to continuation-passing style (CPS) as in the SML/NJ compiler [7], there would be no need for a stack in the abstract machine. However, many implementations do not use a CPS representation, and memory management of the stack is a key issue for these systems. We therefore choose to work with the more general framework at the price of a slightly more complicated abstract machine. In Section 6.1 we discuss in further detail the connection between a CPS-based implementation and our abstract machine.

Answer programs represent terminal states of the abstract machine, and are thus a subset of programs where the expression portion is simply a value variable and the stack is empty.

The binding conventions governing these constructs are as follows:

- All type variables in the domain of TE are bound in $\forall t.\tau$ for $\langle\langle TE, \forall t.\tau \rangle\rangle$.
- All type pointers in the domain of TH are bound in μ for $TH[p \mapsto \mu]$.
- For any closure $\langle\langle (TE, VE), e \rangle\rangle$, where e is either a fix-expression, lambda-expression, or type-abstraction, all type variables in the domain of TE are bound in VE and e, and all value variables in the domain of VE are bound in e.
- All locations in the domain of VH are bound in h for $VH[l \mapsto h]$.
- For a program $((TH, VH), S, (TE, VE), e)$:
  1. All pointers in the domain of TH are bound in VH, S, and TE,

2. All locations in the domain of VH are bound in S and VE,
3. All type variables in the domain of TE are bound in VE and e, and
4. All value variables in the domain of VE are bound in e.

All syntactic forms are identified up to systematic renaming of bound variables. We write $FL(X)$ to denote the free locations of a syntactic object X, and $FP(X)$ to denote the free pointers of X.

No value variable or type variable may be bound more than once in an environment, nor may any location or pointer be bound more than once in a heap. Type environments and value environments are considered equivalent up to any re-ordering. Type heaps and value heaps are considered equivalent up to any re-ordering that respects previously bound pointers or locations. Hence, we may treat type environments as finite maps from type variables to pointers, value environments as finite maps from value variables to type variables and locations, type heaps as finite maps from pointers to type heap values, and value heaps as finite maps from locations to heap values.

When convenient, we use the following syntactic conventions:

- $E[t \mapsto p]$ abbreviates $(TE[t \mapsto p], VE)$ when $E = (TE, VE)$.
- $E[x{:}t \mapsto l]$ abbreviates $(TE, VE[x{:}t \mapsto l])$ when $E = (TE, VE)$.
- $E(t)$ abbreviates p when $E = (TE[t \mapsto p], VE)$.
- $E(x)$ abbreviates l when $E = (TE, VE[x{:}t \mapsto l])$.
- $H[p \mapsto \mu]$ abbreviates $(TH[p \mapsto \mu], VH)$ when $H = (TH, VH)$.
- $H[l \mapsto h]$ abbreviates $(TH, VH[l \mapsto h])$ when $H = (TH, VH)$.
- $H(p)$ abbreviates μ when $H = (TH[p \mapsto \mu], VH)$.
- $H(l)$ abbreviates h when $H = (TH, VH[l \mapsto h])$.
- $S_1 \bar{\circ} S_2$ abbreviates S_1 when $S_2 = []_\sigma$.
- $S_1 \bar{\circ} S_2$ abbreviates $(S_1 \bar{\circ} S) \circ \langle\langle \lambda x{:}t.e \rangle\rangle$ when $S_2 = S \circ \langle\langle E, \lambda x{:}t.e \rangle\rangle$.
- $F_1 \uplus F_2$ abbreviates the the union of F_1 and F_2 when F_1 and F_2 are each finite maps from X to Y such that $Dom(F_1) \cap Dom(F_2) = \emptyset$.
- $H_1 \uplus H_2$ abbreviates $(TH_1 \uplus TH_2, VH_1 \uplus VH_2)$ when $H_1 = (TH_1, VH_1)$ and $H_2 = (TH_2, VH_2)$.
- $Dom(H)$ abbreviates $Dom(TH) \cup Dom(VH)$ when $H = (TH, VH)$.

3.2 Typing Rules for the Abstract Machine

The typing rules for the $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ abstract machine are defined using the following two forms of context:

$$\begin{array}{lrcl} \text{(type pointer context)} & \Phi & ::= & \emptyset \mid \Phi[p = \sigma] \\ \text{(location type assignment)} & \Psi & ::= & \emptyset \mid \Psi[l{:}\sigma] \end{array}$$

Informally, type pointer contexts are the analog of type variable contexts, and location type assignments are the analog of variable type assignments. However, type pointer contexts have no dependencies, so we may consider both type pointer contexts and location type assignments as partial functions. Furthermore, the types in the image of either a location type assignment or type pointer context are required to be closed. That is, if $\Psi(l) = \sigma$ or $\Phi(p) = \sigma$, then σ has no free type variables or type pointers.

The typing judgments for the abstract machine are given in Figure 6. The axioms and inference rules for deriving these judgments are given in Figures 10 through 12 of Appendix A.

Intuitively, judgments 5 through 7 extend the reduction judgments 1 and 2 on named form types and bindings to allocated types, type environments, and type heaps. For example, the judgment $\Phi \vdash TE \Downarrow \Delta$ applies Φ (which maps pointers to types) to the range of TE (which maps type variables to pointers) to obtain a type variable context Δ (mapping type variables to types). Similarly, judgments 8 through 10 extend the typing judgments 3 and 4 for the language to value environments, heap values, and value heaps. Judgment 11 assigns an arrow type $\sigma_1 \rightarrow \sigma_2$ to a stack, meaning that the continuation of the machine is expecting the expression to evaluate to a σ_1 value, and the rest of the computation will then produce a σ_2 value. Empty stacks are explicitly tagged with the result type of the computation. Finally, Judgment 12 determines that a program is well-formed with type σ if:

1. the type heap reduces to Φ,
2. the value heap is described by Ψ under the assumptions of Φ,
3. the stack has type $\sigma' \rightarrow \sigma$, under Φ and Ψ,
4. the type environment reduces to Δ under Φ,
5. the value environment is described by Γ under Δ and Ψ, and
6. the expression has type σ' under Δ and Γ.

The following technical lemma summarizes some properties of the type system.

5.	$\Phi \vdash TE \Downarrow \Delta$	type environment TE reduces to Δ	(see Figure A)
6.	$\Phi \vdash \mu \Downarrow \sigma$	allocated type μ reduces to σ	(see Figure A)
7.	$\vdash TH \Downarrow \Phi$	type heap TH reduces to Φ	(see Figure A)
8.	$\Psi; \Delta \vdash VE : \Gamma$	value environment VE is described by Γ	(see Figure A)
9.	$\Phi; \Psi \vdash h : \sigma$	allocated value h has type σ	(see Figure A)
10.	$\Phi \vdash VH : \Psi$	value heap VH is described by Ψ	(see Figure A)
11.	$\Phi; \Psi \vdash S : \sigma_1 \to \sigma_2$	S takes a σ_1 and produces a σ_2 value	(see Figure A)
12.	$\vdash P : \sigma$	program P has type σ	(see Figure A)

Figure 6: Typing Judgments for the $\lambda_{\text{gc}}^{\to\forall}$ Abstract Machine

Lemma 3.1

1. *If $\Phi \vdash TE \Downarrow \Delta$, then $Dom(TE) = Dom(\Delta)$, $Abstr(\Delta) = \emptyset$, and $Rng(TE) \subseteq Dom(\Phi)$.*

2. *If $\Phi \vdash \mu \Downarrow \sigma$, then $FP(\mu) \subseteq Dom(\Phi)$ and $FTV(\sigma) = \emptyset$.*

3. *If $\vdash TH \Downarrow \Phi$, then $FP(TH) = \emptyset$.*

4. *If $\Psi; \Delta \vdash VE : \Gamma$, then $Dom(VE) = Dom(\Gamma)$, $Abstr(\Delta) = \emptyset$, and if $x{:}t \mapsto l$ is in VE, then t is in $Dom(\Delta)$, and $\Delta(t) = \Psi(l)$.*

5. *If $\Phi; \Psi \vdash h : \sigma$, then $FTV(\sigma) = \emptyset$.*

6. *If $\Phi \vdash VH : \Psi$, then $Dom(VH) = Dom(\Psi)$, and $FTV(\Psi) = \emptyset$.*

7. *If $\Phi; \Psi \vdash S : \sigma_1 \to \sigma_2$, then $FTV(\sigma_1) = FTV(\sigma_2) = \emptyset$.*

8. *If $\vdash P : \sigma$, then $FTV(\sigma) = \emptyset$.*

3.3 Transition System for the Abstract Machine

Execution of the abstract machine is defined by a transition system, a binary relation between machine states (programs). The individual steps of the transition system are given in Figure 8. An informal description of these rules is given below:

$$\begin{array}{rcl}
\hat{TE}(\mathsf{nat}) & = & \mathsf{nat} \\
\hat{TE}(\mathsf{unit}) & = & \mathsf{unit} \\
\hat{TE}(t_1 \times t_2) & = & TE(t_1) \times TE(t_2) \\
\hat{TE}(t_1 \to t_2) & = & TE(t_1) \to TE(t_2) \\
\hat{TE}(\forall t.\tau) & = & \langle\!\langle TE, \forall t.\tau \rangle\!\rangle
\end{array}$$

$$\begin{array}{rcl}
\hat{E}(\mathsf{0}) & = & \mathsf{0} \\
\hat{E}(\texttt{succ}\ x) & = & \texttt{succ}\ E(x) \\
\hat{E}(\langle\rangle) & = & \langle\rangle \\
\hat{E}(\langle x_1, x_2\rangle) & = & \langle E(x_1), E(x_2)\rangle \\
\hat{E}(\texttt{fix}\ x{:}t(x_1{:}t_1).e) & = & \langle\!\langle E, \texttt{fix}\ x{:}t(x_1{:}t_1).e \rangle\!\rangle \\
\hat{E}(\Lambda t.\, e) & = & \langle\!\langle E, \Lambda t.\, e \rangle\!\rangle
\end{array}$$

Figure 7: Environment Substitution

- **return**: The current expression is a variable x and the stack is non-empty. The right-most stack frame is popped from the stack. The frame's environment, extended with the old environment's binding for x, replaces the current environment. The body of the abstraction of the frame replaces the current expression.

- **talloc**: The current expression is `let type` $t = \nu$ `in` e. The type environment is substituted for the free type variables in ν, yielding a type heap value $\hat{TE}(\nu)$. This type heap value is bound in the heap to a new pointer p, and the type variable t is bound in the type environment to the pointer p. The body of the `let type` replaces the current expression.

- **valloc**: The current expression is `let` $x{:}t = a$ `in` e. The environment is substituted for the free type and value variables in a, yielding a heap value $\hat{E}(a)$. This heap value is bound in the heap to a new location l, and the variable x is bound in the environment to the type t and the location l. The body of the `let` replaces the current expression.

- **c-zero**: The current expression is `let` $x{:}t = \texttt{case}(x', e_0, \lambda x_1{:}t_1.e_1)$ `in` e, and the variable x' is bound to a location which in turn is bound to the heap value 0. The e_0 clause is thus selected as the current expression. The body of the `let` and the current environment are pushed on the stack as a closure to be evaluated after evaluation of e_0 is complete.

(return) $(H, S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle, E', x') \longmapsto (H, S, E[x{:}t \mapsto E'(x')], e)$

(talloc) $(H, S, (TE, VE), \texttt{let type } t = \nu \texttt{ in } e) \longmapsto$
$(H[p \mapsto \hat{TE}(\nu)], S, (TE[t \mapsto p], VE), e)$

(valloc) $(H, S, E, \texttt{let } x{:}t = a \texttt{ in } e) \longmapsto (H[l \mapsto \hat{E}(a)], S, E[x{:}t \mapsto l], e)$

(c-zero)
$$\frac{H(E(x')) = \mathtt{0}}{\begin{array}{l}(H, S, E, \texttt{let } x{:}t = \texttt{case}(x', e_0, \lambda x_1{:}t_1.e_1) \texttt{ in } e) \longmapsto \\ \quad (H, S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle, E, e_0)\end{array}}$$

(c-succ)
$$\frac{H(E(x')) = \texttt{succ } l}{\begin{array}{l}(H, S, E, \texttt{let } x{:}t = \texttt{case}(x', e_0, \lambda x_1{:}t_1.e_1) \texttt{ in } e) \longmapsto \\ \quad (H, S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle, E[x_1{:}t_1 \mapsto l], e_1)\end{array}}$$

(proj)
$$\frac{H(E(x')) = \langle l_1, l_2 \rangle}{(H, S, E, \texttt{let } x{:}t = \pi_i\, x' \texttt{ in } e) \longmapsto (H, S, E[x{:}t \mapsto l_i], e)} (i = 1, 2)$$

(app)
$$\frac{H(E(x_1)) = \langle\!\langle E', \texttt{fix } x_1'{:}t_1'(x_2'{:}t_2').e' \rangle\!\rangle}{\begin{array}{l}(H, S, E, \texttt{let } x{:}t = x_1\, x_2 \texttt{ in } e) \longmapsto \\ \quad (H, S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle, E'[x_1'{:}t_1' \mapsto E(x_1), x_2'{:}t_2' \mapsto E(x_2)], e')\end{array}}$$

(tapp)
$$\frac{H(E(x_1)) = \langle\!\langle E', \Lambda t'.\, e' \rangle\!\rangle}{\begin{array}{l}(H, S, E, \texttt{let } x{:}t = x_1\, [t_1] \texttt{ in } e) \longmapsto \\ \quad (H, S \circ \langle\!\langle E, \lambda x{:}t.e \rangle\!\rangle, E'[t' \mapsto E(t_1)], e')\end{array}}$$

Figure 8: Transition Rules of the Abstract Machine

$$\text{(result)} \quad r \quad ::= \quad n \mid \langle\rangle \mid \langle r_1, r_2\rangle \mid \texttt{*fix*} \mid \texttt{*tabs*} \mid \bot \mid \texttt{*wrong*}$$

$$\begin{array}{l}
\mathit{print}[\mathsf{nat}](\mathit{VH}[l \mapsto \mathsf{0}], l) = 0 \\
\mathit{print}[\mathsf{nat}](\mathit{VH}[l \mapsto \texttt{succ}\ l'], l) = 1 + \mathit{print}[\mathsf{nat}](\mathit{VH}, l') \\
\mathit{print}[\mathsf{unit}](\mathit{VH}[l \mapsto \langle\rangle], l) = \langle\rangle \\
\mathit{print}[\sigma_1 \times \sigma_2](\mathit{VH}[l \mapsto \langle l_1, l_2\rangle], l) = \langle \mathit{print}[\sigma_1](\mathit{VH}, l_1), \mathit{print}[\sigma_2](\mathit{VH}, l_2)\rangle \\
\mathit{print}[\sigma_1 \rightarrow \sigma_2](\mathit{VH}[l \mapsto \langle\!\langle E, \texttt{fix}\ x{:}t(x_1{:}t_2).e\rangle\!\rangle], l) = \texttt{*fix*} \\
\mathit{print}[\forall t.\sigma](\mathit{VH}[l \mapsto \langle\!\langle E, \Lambda t.\, e\rangle\!\rangle], l) = \texttt{*tabs*}
\end{array}$$

$$\mathit{print}_{\mathrm{prog}}((\mathit{TH}, \mathit{VH}), []_\sigma, (\mathit{TE}, \mathit{VE}), x) = \mathit{print}[\sigma](\mathit{VH}, \mathit{VE}(x))$$

Figure 9: Printing a Result

- **c-succ**: The current expression is `let` $x{:}t = \texttt{case}(x', e_0, \lambda x_1{:}t_1.e_1)$ `in` e, and the variable x' is bound to a location which in turn is bound to the heap value `succ` l. The e_1 clause is thus selected as the current expression, and a new binding, mapping the x_1 to the type t_1 and the location l is entered into the environment. The body of the `let` and the current environment are pushed on the stack as a closure to be evaluated after evaluation of e_1 is complete.

- **proj**: The current expression is `let` $x{:}t = \pi_i\, x'$ `in` e, and the variable x' is bound to a location which in turn is bound to a pair $\langle l_1, l_2\rangle$. The appropriate component is selected (l_1 or l_2 depending on i), and bound to x in the environment (with the type t). The computation continues with the body of the `let`, e.

- **app**: The current expression is `let` $x{:}t = x_1\, x_2$ `in` e, and the variable x_1 is bound to a location which in turn is bound to a `fix`-closure. The body of the `let` and the current environment are pushed onto the stack. The environment of the closure is extended to map x_1' and x_2' to the location of the closure and the argument's location respectively, and this new environment is taken as the current environment of the machine. The body of the `fix`-abstraction is taken as the current expression to be evaluated. Notice in particular that evaluation of the code of the closure takes place under the environment of the closure, and not the current environment.

- **tapp**: The current expression is `let` $x{:}t = x_1\, [t_1]$ `in` e, and the variable x_1 is bound to a location which in turn is bound to a Λ-closure. The body of the `let` and the current environment are pushed onto the stack. The type environment of the closure is extended to map the bound type variable t' to the pointer to which the argument type is bound. This new environment is taken as the current environment of the machine and the body of the Λ-abstraction is taken as the current expression to be evaluated.

We assume when adding a new binding to an environment or heap that the bound pointer/location/variable is fresh.

Two of the rules — the **talloc** and **valloc** rules — make use of the auxiliary operations, $\hat{TE}$ and $\hat{E}$, defined in Figure 7. These functions "substitute" the environment for the free variables within the binding. However, for $\forall$-types, fix, and Λ-expressions, the substitution is delayed by forming a closure consisting of the environment and the binding.

As an example, consider the evaluation of the named form expression e_0 where:

$$\begin{array}{lcl}
e_0 & = & \texttt{let type } t = \mathsf{nat} \texttt{ in } e_1 \\
e_1 & = & \texttt{let type } t_1 = t \to t \texttt{ in } e_2 \\
e_2 & = & \texttt{let } x_1{:}t_1 = b_0 \texttt{ in } e_4 \\
b_0 & = & \texttt{fix } x'{:}t_1(x{:}t).e_3 \\
e_3 & = & \texttt{let } x_2{:}t = \mathsf{succ}\ x \texttt{ in } x_2 \\
e_4 & = & \texttt{let } x_3{:}t = \mathsf{0} \texttt{ in } e_5 \\
e_5 & = & \texttt{let } x_4{:}t = x_1\, x_3 \texttt{ in } x_4
\end{array}$$

$$\begin{array}{lcl}
TH_0 & = & [p \mapsto \mathsf{nat}, p_1 \mapsto (p \to p)] \\
TE_0 & = & [t \mapsto p, t_1 \mapsto p_1] \\
h_0 & = & \langle\!\langle (TE_0, \emptyset), b_0 \rangle\!\rangle \\
E_0 & = & (TE_0, [x_1{:}t_1 \mapsto l_1, x_3{:}t \mapsto l_3]) \\
S_0 & = & []_{\mathsf{nat}} \circ \langle\!\langle E_0, \lambda x_4{:}t.x_4 \rangle\!\rangle \\
VH_0 & = & [l_1 \mapsto h_0, l_3 \mapsto \mathsf{0}, l_2 \mapsto \mathtt{succ}\ l_3]
\end{array}$$

$$\begin{array}{ll}
 & ((\emptyset,\emptyset), []_{\mathsf{nat}}, (\emptyset,\emptyset), e_0) \\
\overset{\textbf{talloc}}{\longmapsto} & (([p \mapsto \mathsf{nat}],\emptyset), []_{\mathsf{nat}}, ([t \mapsto p],\emptyset), e_1) \\
\overset{\textbf{talloc}}{\longmapsto} & ((TH_0,\emptyset), []_{\mathsf{nat}}, (TE_0,\emptyset), e_2) \\
\overset{\textbf{valloc}}{\longmapsto} & ((TH_0,[l_1 \mapsto h_0], []_{\mathsf{nat}}, (TE_0,[x_1{:}t_1 \mapsto l_1]), e_4) \\
\overset{\textbf{valloc}}{\longmapsto} & ((TH_0,[l_1 \mapsto h_0, l_3 \mapsto \mathsf{0}], []_{\mathsf{nat}}, E_0, e_5) \\
\overset{\textbf{app}}{\longmapsto} & ((TH_0,[l_1 \mapsto h_0, l_3 \mapsto \mathsf{0}]), S_0, (TE_0,[x'{:}t_1 \mapsto l_1, x{:}t \mapsto l_3]), e_3) \\
\overset{\textbf{valloc}}{\longmapsto} & ((TH_0, VH_0), S_0, (TE_0,[x'{:}t_1 \mapsto l_1, x{:}t \mapsto l_3, x_2{:}t \mapsto l_2]), x_2) \\
\overset{\textbf{return}}{\longmapsto} & ((TH_0, VH_0), []_{\mathsf{nat}}, (TE_0,[x_1{:}t_1 \mapsto l_1, x_3{:}t \mapsto l_3, x_4{:}t \mapsto l_2]), x_4)
\end{array}$$

Applying $print_{\text{prog}}$ to the terminal state yields:

$$print[\mathsf{nat}](VH_0, l_2) = 1 + print[\mathsf{nat}]([l_1 \mapsto h_0, l_3 \mapsto \mathsf{0}], l_3) = 1 + 0 = 1$$

We define the relation $\overset{\mathbf{R}}{\longmapsto}$ to be the union of the relations defined by the transition rules, and write $P \overset{\mathbf{R}}{\longmapsto} P'$ if (P, P') is in $\overset{\mathbf{R}}{\longmapsto}$. That is, $P \overset{\mathbf{R}}{\longmapsto} P'$ iff $P \longmapsto P'$ via **return**, **talloc**, **valloc**, **c-zero**, **c-succ**, **proj**, **app**, or **tapp**. We remark that, since at most one rule applies for a given program, $\overset{\mathbf{R}}{\longmapsto}$ defines a partial function from closed programs to programs. We take $\overset{\mathbf{R}}{\longmapsto}^*$ to be the reflexive, transitive closure of $\overset{\mathbf{R}}{\longmapsto}$. We say a program P *diverges* if there exists an infinite sequence of programs $P_1, P_2, P_3, \cdots$ such that $P \overset{\mathbf{R}}{\longmapsto} P_1 \overset{\mathbf{R}}{\longmapsto} P_2 \overset{\mathbf{R}}{\longmapsto} P_3 \overset{\mathbf{R}}{\longmapsto} \cdots$.

In Figure 9, we define a partial function from answers to results, where results are either a natural number, pair of results, the token `*fix*`, or the token `*tabs*`. The other results, $\bot$ and `*wrong*`, are used in our definition of evaluation below.

Definition 3.2 (Stuck Program) *A program P is stuck if either P is an answer and $print_{\text{prog}}(P)$ is undefined, or else P is not an answer and there exists no P' such that $P \overset{\mathbf{R}}{\longmapsto} P'$.*

We define evaluation as the following relation between programs and results:

Definition 3.3 (Evaluation Relation)

1. *$P \Downarrow r$ iff there exists an A such that $P \overset{\mathbf{R}}{\longmapsto}^* A$ and $print_{\text{prog}}(A) = r$.*
2. *$P \Downarrow \bot$ iff P diverges.*
3. *$P \Downarrow$ `*wrong*` iff there exists a stuck P' such that $P \overset{\mathbf{R}}{\longmapsto}^* P'$.*

From the fact that at most one $\overset{\mathbf{R}}{\longmapsto}$ rule can apply for a given closed program, it is clear that there is one and only one r such that $P \Downarrow r$. Hence, we may treat evaluation as a total function from closed programs to results.

Definition 3.4 (Evaluation Function) *$eval(P) = r$ iff $P \Downarrow r$.*

Finally, we will need a suitable notion of observational equivalence for programs. We say two programs are equivalent iff they evaluate to the same results.

Definition 3.5 (Kleene Equivalence) *$P_1 \simeq P_2$ iff $eval(P_1) = eval(P_2)$.*

3.4 Type Soundness

We prove the soundness of the type system (with respect to execution by the abstract machine) by establishing that the transition system preserves typability, and that well-typed programs are either answers, or admit a further transition. (This viewpoint is inspired by Wright and Felleisen [44].) We state the important lemmas and give the proof of soundness here. Proofs of the most important lemmas, Preservation and Progress, may be found in Appendix B.

Lemma 3.6 (*TE* Substitution) *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash TE \Downarrow \Delta$, *and* $\Delta \vdash \nu \Downarrow \sigma$, *then* $\Phi \vdash \hat{TE}(\nu) \Downarrow \sigma$.

Lemma 3.7 (*E* Substitution) *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, $\Phi \vdash TE \Downarrow \Delta$, $\Psi; \Delta \vdash VE : \Gamma$, *and* $\Delta; \Gamma \vdash a : \sigma$, *then* $\Phi; \Psi \vdash \hat{E}(a) : \sigma$ *where* $E = (TE, VE)$.

Lemma 3.8 *Let* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta')$ *be a well-formed context.*

1. *If* $\Delta[t]\Delta' \vdash \tau \Downarrow \sigma_1$, *then* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta') \vdash \tau \Downarrow \{\sigma_2/t\}\sigma_1$.
2. *If* $\Delta[t]\Delta' \vdash \nu \Downarrow \sigma_1$, *then* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta') \vdash \nu \Downarrow \{\sigma_2/t\}\sigma_1$.

Proof: Simultaneously, by induction on τ and ν. □

Lemma 3.9 *Let* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta')$ *be a well-formed context.*

1. *If* $\Delta[t]\Delta'; \Gamma \vdash e : \sigma_1$, *then* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta'); (\{\sigma_2/t\}\Gamma) \vdash e : \{\sigma_2/t\}\sigma_1$.
2. *If* $\Delta[t]\Delta'; \Gamma \vdash b : \sigma_1$, *then* $\Delta[t = \sigma_2](\{\sigma_2/t\}\Delta'); (\{\sigma_2/t\}\Gamma) \vdash b : \{\sigma_2/t\}\sigma_1$.

Proof: Simultaneously, by induction on e and b, using Lemma 3.8 □

Lemma 3.10 (Preservation) *If* $\vdash P : \sigma$ *and* $P \xmapsto{\mathbf{R}} P'$, *then* $\vdash P' : \sigma$.

Lemma 3.11 (Canonical Forms) *If* $\Phi; \Psi \vdash h : \sigma$, *then:*

1. *if* $\sigma = \mathsf{nat}$, *then* h *is either* $\mathsf{0}$ *or* $\mathsf{succ}\ l$ *for some* l.
2. *if* $\sigma = \mathsf{unit}$, *then* h *is* $\langle\rangle$.
3. *if* $\sigma = \sigma_1 \times \sigma_2$, *then* h *is* $\langle l_1, l_2\rangle$ *for some* l_1 *and* l_2.
4. *if* $\sigma = \sigma_1 \to \sigma_2$, *then* h *is* $\langle\langle E, \mathtt{fix}\ x{:}t(x_1{:}t_1).e\rangle\rangle$ *for some* E, x, t, x_1, t_1, *and* e.

5. *if* $\sigma = \forall t.\tau$, *then* h *is* $\langle\langle E, \Lambda t.\, e\rangle\rangle$ *for some* E *and* e.

Proof: By inspection of the heap value typing rules. □

Lemma 3.12 (Progress) *If* $\vdash P : \sigma$, *then either* P *is an answer or else there exists a* P' *such that* $P \overset{\mathbf{R}}{\longmapsto} P'$.

Lemma 3.13 *If* $\vdash TH : \Phi$, $\Phi \vdash VH[l \mapsto h] : \Psi[l : \sigma]$, *then there exists an* r *such that* $print[\sigma](VH[l \mapsto h], l) = r$.

Corollary 3.14 *If* $\vdash A : \sigma$, *then there exists an* r *such that* $print_{\text{prog}}(A) = r$.

Theorem 3.15 (Soundness) *If* $\vdash P : \sigma$, *then* $eval(P) \neq$ `*wrong*`.

Proof: If P diverges then $eval(P) = \bot$. If P does not diverge, then there exists some P' such that $P \overset{\mathbf{R}}{\longmapsto}^* P'$ and there is no P'' where $P' \overset{\mathbf{R}}{\longmapsto} P''$. By induction on the number of rewriting steps taking P to P' using the Preservation Lemma, we can show that $\vdash P' : \sigma$. By the Progress Lemma, P' must be an answer. By Corollary 3.14, there exists an r such that $print_{\text{prog}}(P') = r$. Since `*wrong*` is not in the image of $print_{\text{prog}}$, $r \neq$ `*wrong*` and hence $eval(P) \neq$ `*wrong*`. □

4 Heap Garbage

In this section, we consider a general definition of "garbage" as a heap object that is not needed by the program in order to produce the same result. (Notions of "garbage" for the stack and environment are considered in Section 6 below.)

Definition 4.1 (Heap Garbage) *Let* $P = (H \uplus H', S, E, e)$ *be a well-formed program and take* $P' = (H, S, E, e)$. *We say that* H' *is garbage with respect to* P *iff* P' *is well-formed and* $P \simeq P'$.

This definition of garbage allows us to eliminate any portion of the heap as long as we do not change the observable behavior of the program. Notice that a heap object is not regarded as garbage if it is required for the well-formedness of P, even if the computation could proceed to a final answer without referring to that object. It is possible to drop the well-formedness condition and consider a more semantic definition of garbage, at the expense of considerable technical complication [31]. We prefer the more restrictive definition because it is simpler and closer to practice.

Implementors use one of a variety of techniques to determine which portions of the heap can be collected. One of the most important techniques is based on the idea of accessibility. Formally, we can drop a heap binding if the resulting program has no free reference to the binding:

$$\textbf{(GC)}\ \frac{FP(H,S,E,e)=\emptyset \qquad FL(H,S,E,e)=\emptyset}{(H \uplus H', S, E, e) \overset{\mathbf{GC}}{\longmapsto} (H, S, E, e)}$$

The idea is that $\overset{\mathbf{GC}}{\longmapsto}$ models a garbage collector that drops zero or more bindings from the heap, but ensures that none of the dropped bindings are accessible. Note that for a well-formed program, the $\overset{\mathbf{GC}}{\longmapsto}$ rule is always enabled, since we can always drop an empty heap. Furthermore, the composition of two $\overset{\mathbf{GC}}{\longmapsto}$ steps can always be simulated by a single $\overset{\mathbf{GC}}{\longmapsto}$ step.

A key property of $\overset{\mathbf{GC}}{\longmapsto}$ is that it preserves typing in the same fashion as the other rewriting rules.

Lemma 4.2

1. *If* $\vdash (H[p \mapsto \mu], S, E, e) : \sigma$ *and* $FP(H,S,E,e) = \emptyset$*, then* $\vdash (H,S,E,e) : \sigma$.

2. *If* $\vdash (H[l \mapsto h], S, E, e) : \sigma$ *and* $FL(H,S,E,e) = \emptyset$*, then* $\vdash (H,S,E,e) : \sigma$.

Proof (sketch): We argue the case for part 1, as the case for part 2 is similar. Suppose $H = (TH, VH)$ and $E = (TE, VE)$. Since $\vdash (H[p \mapsto \mu], S, E, e) : \sigma$, there exists Φ, σ_0, Ψ, σ', Δ, and Γ such that $\vdash TH[p \mapsto \mu] \Downarrow \Phi[p = \sigma_0]$, $\Phi[p = \sigma_0] \vdash VH : \Psi$, $\Phi[p = \sigma_0]; \Psi \vdash S : \sigma' \to \sigma$, $\Phi[p = \sigma_0] \vdash TE \Downarrow \Delta$, $\Psi; \Delta \vdash VE : \Gamma$, and $\Delta; \Gamma \vdash e : \sigma'$. Since $FP(H,S,E,e) = \emptyset$, $FP(H) = \emptyset$, $FP(S) \subseteq Dom(TH)$, $FP(E) \subseteq Dom(TH)$. Hence we may construct derivations of $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, $\Phi; \Psi \vdash S : \sigma' \to \sigma$, and $\Phi \vdash TE \Downarrow \Delta$. Therefore, $\vdash (H,S,E,e) : \sigma$. □

Lemma 4.3 (GC Preservation) *If* $\vdash P_1 : \sigma$ *and* $P_1 \overset{\mathbf{GC}}{\longmapsto} P_2$*, then* $\vdash P_2 : \sigma$.

Proof: By induction on the number of bindings dropped, using Lemma 4.2. □

From this lemma and our original Progress result (see 3.12), we can conclude that a well-typed program can never become stuck with respect to the $\overset{\mathbf{R}}{\longmapsto}$ rules, even if the program takes a $\overset{\mathbf{GC}}{\longmapsto}$ step.

Corollary 4.4 (GC Progress) *If* $\vdash P_1 : \sigma$*, and* $P_1 \overset{\mathbf{GC}}{\longmapsto} P_2$*, then either* P_2 *is a printable answer or else there exists a* P_3 *such that* $P_2 \overset{\mathbf{R}}{\longmapsto} P_3$.

Consequently, no matter how we add $\overset{\textbf{GC}}{\longmapsto}$ to our evaluation relation, a well-typed program will never become stuck.

We would like to add $\overset{\textbf{GC}}{\longmapsto}$ to the rewriting rules for our abstract machine to model an implementation that interleaves evaluation with garbage collection. Since the null collection (in which no bindings are eliminated) is always possible, this introduces the potential for non-termination by infinite repetition of $\overset{\textbf{GC}}{\longmapsto}$ steps. In practice garbage collection steps occur only after some number of standard evaluation steps have occurred (e.g., at the beginning of each basic block), ruling out infinite repetition of vacuous collection steps. We adopt this restriction by defining the $\overset{\textbf{GCR}}{\longmapsto}$ relation to be the composition of the $\overset{\textbf{GC}}{\longmapsto}$ and $\overset{\textbf{R}}{\longmapsto}$ relations, and consider computation as a sequence of $\overset{\textbf{GCR}}{\longmapsto}$ steps. That is, if $P_1 \overset{\textbf{GCR}}{\longmapsto} P_2$, then there exists a P' such that $P_1 \overset{\textbf{GC}}{\longmapsto} P' \overset{\textbf{R}}{\longmapsto} P_2$. Note that by GC Progress, the fact that $\overset{\textbf{R}}{\longmapsto}$ is a partial function, and the fact that $\overset{\textbf{GC}}{\longmapsto}$ does not affect the expression of a program, any $\overset{\textbf{R}}{\longmapsto}$ step that can be taken before an arbitrary $\overset{\textbf{GC}}{\longmapsto}$ step, can still be taken after the $\overset{\textbf{GC}}{\longmapsto}$ step. We say a program P is stuck with respect to $\overset{\textbf{GCR}}{\longmapsto}$ if either P is an answer and $\mathit{print}_{\mathrm{prog}}(P)$ is undefined or else P is not an answer and there is no P' such that $P \overset{\textbf{GCR}}{\longmapsto} P'$.

Definition 4.5 (GC Evaluation Relation)

1. *$P \Downarrow_{\textbf{GCR}} r$ iff there exists an A such that $P \overset{\textbf{GCR}}{\longmapsto}{}^* A$ and $\mathit{print}_{\mathrm{prog}}(A) = r$.*

2. *$P \Downarrow_{\textbf{GCR}} \bot$ iff P diverges with respect to $\overset{\textbf{GCR}}{\longmapsto}$.*

3. *$P \Downarrow_{\textbf{GCR}}$ `*wrong*` iff there exists a stuck P' such that $P \overset{\textbf{GCR}}{\longmapsto}{}^* P'$.*

We would like to show that the addition of the $\overset{\textbf{GC}}{\longmapsto}$ steps to our evaluator does not affect the evaluation result of a given program. That is, we would like to show that $\Downarrow_{\textbf{GCR}}$ is a total function mapping closed programs to results, and is in fact the same total function as our original *eval*. (Note that this result would fail if we allowed infinite repetition of garbage collection steps.)

We begin by showing that printing is unaffected by garbage collection. We then show that, whenever a $\overset{\textbf{GC}}{\longmapsto}$ step is followed by an $\overset{\textbf{R}}{\longmapsto}$ step, we can postpone the garbage collection until after the $\overset{\textbf{R}}{\longmapsto}$ step has been taken. From this, it follows that we can simulate any $\overset{\textbf{GCR}}{\longmapsto}$ evaluation sequence with an $\overset{\textbf{R}}{\longmapsto}$ evaluation sequence.

Lemma 4.6 (GC Answer) *If $A \overset{\textbf{GC}}{\longmapsto} A'$ and $\mathit{print}_{\mathrm{prog}}(A') = r$, then $\mathit{print}_{\mathrm{prog}}(A) = r$.*

Proof (sketch): Since $\overset{\mathbf{GC}}{\longmapsto}$ only drops bindings and does not change bindings in the heap, and since $\overset{\mathbf{GC}}{\longmapsto}$ preserves typing, $\mathit{print}_{\mathrm{prog}}(A') = r$ implies $\mathit{print}_{\mathrm{prog}}(A) = r$. □

Lemma 4.7 (GC Postponement) *If* $\vdash P_1 : \sigma$ *and* $P_1 \overset{\mathbf{GC}}{\longmapsto} P_2 \overset{\mathbf{R}}{\longmapsto} P_3$, *then there exists a* P_2' *such that* $P_1 \overset{\mathbf{R}}{\longmapsto} P_2' \overset{\mathbf{GC}}{\longmapsto} P_3$.

Proof: Suppose $P_1 = (H_1 \uplus H_2, S, E, e)$, $P_2 = (H_1, S, E, e)$, and $P_3 = (H', S', E', e')$. We can show via case analysis on the rewriting rule taking P_2 to P_3, that $P_1 \overset{\mathbf{R}}{\longmapsto} (H' \uplus H_2, S', E', e')$. By Preservation, we know that $\vdash P_3 : \sigma$ and thus P_3 is closed, and thus, $(H' \uplus H_2, S', E', e') \overset{\mathbf{GC}}{\longmapsto} (H', S', E', e')$. □

Corollary 4.8 *For all* $n \geq 0$, *if* $P_0 \overset{\mathbf{GCR}}{\longmapsto}{}^n P_n$, *then there exists a* P_n' *such that* $P_0 \overset{\mathbf{R}}{\longmapsto}{}^n P_n'$ *and* $P_n' \overset{\mathbf{GC}}{\longmapsto} P_n$.

Proof: By induction on n using the Postponement lemma. □

Theorem 4.9 (GC Correctness) *If* $\vdash P : \sigma$, *then* $P \Downarrow r$ *iff* $P \Downarrow_{\mathbf{GCR}} r$.

Proof: We can simulate any $\overset{\mathbf{R}}{\longmapsto}$ step with a $\overset{\mathbf{GCR}}{\longmapsto}$ step by simply performing an empty garbage collection. That is, if $P \overset{\mathbf{R}}{\longmapsto} P'$, then $P \overset{\mathbf{GC}}{\longmapsto} P \overset{\mathbf{R}}{\longmapsto} P'$ and thus $P \overset{\mathbf{GCR}}{\longmapsto} P'$. Consequently, if $P \Downarrow r$, then $P \Downarrow_{\mathbf{GCR}} r$.

Suppose $P \Downarrow_{\mathbf{GCR}} r$. By GC Soundness, $r \neq$ `*wrong*`. If $r \neq \bot$, then there exists an n and A such that $P \overset{\mathbf{GCR}}{\longmapsto}{}^n A$ and $\mathit{print}_{\mathrm{prog}}(A) = r$. By Corollary 4.8, there exists an A' such that $P \overset{\mathbf{R}}{\longmapsto}{}^n A' \overset{\mathbf{GC}}{\longmapsto} A$. By Lemma 4.6, $\mathit{print}_{\mathrm{prog}}(A') = \mathit{print}_{\mathrm{prog}}(A) = r$.

If $r = \bot$ then there exists an infinite sequence $P', P_1, P_1', P_2, P_2', \cdots$ such that

$$P \overset{\mathbf{GC}}{\longmapsto} P' \overset{\mathbf{R}}{\longmapsto} P_1 \overset{\mathbf{GC}}{\longmapsto} P_1' \overset{\mathbf{R}}{\longmapsto} P_2 \overset{\mathbf{GC}}{\longmapsto} P_2' \overset{\mathbf{R}}{\longmapsto} \cdots.$$

By Corollary 4.8, we can construct an infinite sequence $P'', P_1'', P_2'', \cdots$ such that

$$P \overset{\mathbf{R}}{\longmapsto} P'' \overset{\mathbf{R}}{\longmapsto} P_1'' \overset{\mathbf{R}}{\longmapsto} P_2'' \overset{\mathbf{R}}{\longmapsto} \cdots$$

Thus, $P \Downarrow \bot$. Consequently, if $P \Downarrow_{\mathbf{GCR}} r$, then $P \Downarrow r$. □

Of course, $\overset{\mathbf{GC}}{\longmapsto}$ is too high-level to be taken as a primitive instruction, because the side-conditions require a global constraint — no free references to pointers or locations — and checking this constraint requires examining every variable in the program's state. Some mechanism is needed to determine efficiently which pointers and locations can be safely garbage collected. The following section addresses this issue.

5 Type-Based Tag-Free Heap Collection

In this section, we formulate a garbage collection rewriting rule that models type-based, tag-free copying collection in the style of Tolmach [40] and the TIL/ML compiler [38]. The key idea is to preserve all heap objects that can be reached (either directly or indirectly) from the current environment and stack. We use the type information recorded in environments during evaluation to determine the shape of heap objects. This allows us to extract locations (and their types) from heap objects without having to use any extra tags on the heap objects themselves.

We formalize the garbage collection process as a rewriting system. GC states are 4-tuples of the form (H_f, Q, L, H_t), where H_f and H_t are heaps, Q is a set of pointers, and L is a set of location and pointer pairs:

$$\begin{array}{llll} \text{(GC states)} & X & ::= & (H_f, Q, L, H_t) \\ \text{(type pointer sets)} & Q & ::= & \{p_1, \cdots, p_n\} \quad (n \geq 0) \\ \text{(typed location sets)} & L & ::= & \{l_1{:}p_1, \cdots, l_n{:}p_n\} \quad (n \geq 0) \end{array}$$

In the terminology of copying collectors, H_f is the "from-space", H_t is the "to-space", and L and Q together constitute the "scan-set" or "frontier". Throughout, Q and L contain those pointers and locations that are immediately accessible from the current environment, stack, or to-space but have not yet been forwarded from the from-space to the to-space. In addition, L tracks (pointers to) types of these accessible locations. From this extra type information, we can determine the "shape" of value objects referenced in the scan-set. For instance, if $l{:}p$ is in the scan-set and p is bound to a type heap value $p_1 \times p_2$, then we know that l must be bound to a heap value of the form $\langle l_1, l_2 \rangle$ and furthermore, l_i is described by p_i for $i = 1, 2$.

The basic rewriting rules of the GC algorithm are as follows:

$$\textbf{(gc-1)}\quad (H_f[p \mapsto \mu], Q \uplus \{p\}, L, H_t) \Longrightarrow (H_f, Q \cup FP(\mu), L, H_t[p \mapsto \mu])$$

$$\textbf{(gc-2)}\quad \frac{p \notin Dom(H_f)}{(H_f, Q \uplus \{p\}, L, H_t) \Longrightarrow (H_f, Q, L, H_t)}$$

$$\textbf{(gc-3)}\quad \frac{\mathcal{F}[(H_f \uplus H_t)(p)](p, h) = (Q', L')}{(H_f[l \mapsto h], Q, L \uplus \{l{:}p\}, H_t) \Longrightarrow (H_f, Q \cup Q', L \cup L', H_t[l \mapsto h])}$$

$$\textbf{(gc-4)}\quad \frac{l \notin Dom(H_f)}{(H_f, Q, L \uplus \{l{:}p\}, H_t) \Longrightarrow (H_f, Q, L, H_t)}$$

where

$$\begin{array}{rcl}
\mathcal{F}[\mathsf{nat}] & = & \lambda(p,h).\mathtt{case}\ h\ \mathtt{of} \\
 & & \qquad \mathtt{0} \Rightarrow (\emptyset, \emptyset) \\
 & & \qquad \mid\ \mathtt{succ}\ l \Rightarrow (\{p\}, \{l{:}p\}) \\
\mathcal{F}[\mathsf{unit}] & = & \lambda(p, \langle\rangle).(\emptyset, \emptyset) \\
\mathcal{F}[p_1 \times p_2] & = & \lambda(p, \langle l_1, l_2\rangle).(\{p_1, p_2\}, \{l_1{:}p_1, l_2{:}p_2\}) \\
\mathcal{F}[p_1 \to p_2] & = & \lambda(p, \langle\!\langle E, \mathtt{fix}\ x{:}t(x_1{:}t_1).e\rangle\!\rangle).\mathcal{F}_{\mathrm{env}}(E) \\
\mathcal{F}[\langle\!\langle TE, \forall t.\tau\rangle\!\rangle] & = & \lambda(p, \langle\!\langle E, \Lambda t.\,e\rangle\!\rangle).\mathcal{F}_{\mathrm{env}}(E)
\end{array}$$

and

$$\mathcal{F}_{\mathrm{env}}(TE, [x_1{:}t_1 \mapsto l_1, \cdots, x_n{:}t_n \mapsto l_n]) = (Rng(TE), \{l_1{:}TE(t_1), \cdots, l_n{:}TE(t_n)\})$$

The **gc-1** rule forwards a type binding $p \mapsto \mu$ from the from-space to the to-space when p is in the scan-set. All of the free pointers of the allocated type μ are added to the scan-set ensuring that these bindings are eventually forwarded to the to-space. Notice that calculating the free pointers of an allocated type requires tags on the allocated types so that we can tell, for instance, nat from $p_1 \to p_2$. The **gc-2** rule skips over a pointer in the scan-set when we determine that the pointer's binding has already been forwarded to the to-space.

The **gc-3** rule, like the **gc-1** rule, forwards a value binding $l \mapsto h$ from the from-space to the to-space when $l{:}p$ is in the scan-set. All of the free pointers and free locations of h, along with pointers to types describing these locations, are added to the scan-set. Unlike the first rule, we use the type information p recorded with a location to determine the shape of the corresponding heap value, and to extract the free pointers and locations. In particular, the function $\mathcal{F}$ takes an allocated type (μ), and returns a function which when given a pointer to that allocated type (p) and a heap value of the appropriate type (h), returns appropriate Q and L sets for the heap value. In other words, $\mathcal{F}$ calculates the free pointers and locations of the heap value, as well as the types of those locations. Furthermore, $\mathcal{F}$ requires no run-time tags on h to distinguish, for instance, naturals from pairs, or pairs from closures. We need some tag information in the case that $\mu = \texttt{nat}$ to determine whether the heap value is 0 or succ l. However, such a tag is needed anyway to support evaluation of case expressions.

The **gc-4** rule, like the **gc-2** rule, skips over a location/pointer pair in the scan-set when we determine that the location's binding has already been forwarded to the to-space.

The Tag-Free GC algorithm is initialized and finalized as follows:

$$\textbf{(tf-gc)}\ \frac{\mathcal{F}_{\text{env}}(E) = (Q_1, L_1) \qquad \mathcal{F}_{\text{stack}}(S) = (Q_2, L_2) \qquad (H, Q_1 \cup Q_2, L_1 \cup L_2, \emptyset) \Longrightarrow^* (H_f, \emptyset, \emptyset, H_t)}{(H, S, E, e) \overset{\textbf{tf-gc}}{\longmapsto} (H_t, S, E, e)}$$

where

$$\begin{array}{rcl} \mathcal{F}_{\text{stack}}([]_\sigma) & = & (\emptyset, \emptyset) \\ \mathcal{F}_{\text{stack}}(S \circ \langle\langle E, \lambda x{:}t.e\rangle\rangle) & = & (Q_a \cup Q_b, L_a \cup L_b) \\ & & (\text{where } (Q_a, L_a) = \mathcal{F}_{\text{stack}}(S) \text{ and } (Q_b, L_b) = \mathcal{F}_{\text{env}}(E)) \end{array}$$

The algorithm begins by calculating the free pointers and free locations (along with their types) of the current environment and stack. This set of pointers and typed locations is taken as the initial scan-set, and the heap of the program is taken as the initial from-space. Then, the algorithm repeatedly applies the gc rewriting rules until the scan-set is empty. At this point, we take the to-space as the "new" heap of the program.

Two important properties of the GC system are readily apparent. First, the GC system only drops bindings from the heap — it does not introduce new bindings nor change existing bindings. Hence, at each stage of the computation, the original heap can be recovered by taking the union of the from- and to-spaces. Second, the rewriting system must terminate, since (a) each heap binding is moved at most once from the from-space to the to-space, and (b) at each step either an element is discarded from the scan-set, or else a binding is moved from the from-space to the to-space.

With these properties in mind, to prove the correctness of the collection algorithm it suffices to show that the Tag-Free GC system does not get stuck (i.e., it is always possible to empty the scan-set), and the resulting program is closed. If the system always results in a closed program, then the system can be simulated by the original $\overset{\textbf{GC}}{\longmapsto}$ rewriting rule, which we have already proven to be a correct specification of garbage collection.

The critical step in showing that the Tag-Free GC algorithm can run to completion is proving a variant of the Canonical Forms lemma. In particular, we must show that when we add a location/pointer pair $l{:}p$ to the scan-set, then p is bound to a type that describes the heap value to which l is bound. This ensures that the $\mathcal{F}$ function is in fact defined on this location/pointer pair. Furthermore, we must show that this property holds of any new bindings we add to the scan-set.

Lemma 5.1 (Unexpanded Canonical Forms) *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, *and* $\Phi(p) = \Psi(l)$, *then:*

1. *if* $TH(p) = \mathsf{nat}$, *then* $VH(l) = \mathsf{0}$ *or* $VH(l) = \mathsf{succ}\ l'$ *for some* l'. *Furthermore,* $\Psi(l') = \mathsf{nat}$.

2. *if* $TH(p) = \mathsf{unit}$, *then* $VH(l) = \langle\rangle$.

3. *if* $TH(p) = p_1 \times p_2$, *then* $VH(l) = \langle l_1, l_2\rangle$ *for some* l_1 *and* l_2. *Furthermore,* $\Phi(p_i) = \Psi(l_i)$ *for* $i = 1, 2$.

4. *if* $TH(p) = p_1 \rightarrow p_2$, *then* $VH(l) = \langle\!\langle (TE, VE), \mathtt{fix}\ x_1{:}t_1(x_2{:}t_2).e\rangle\!\rangle$ *for some* TE, VE, x_1, t_1, x_2, t_2, *and* e. *Furthermore, for all* $x{:}t \mapsto l'$ *in* VE, $\Phi(TE(t)) = \Psi(l')$.

5. *if* $TH(p) = \langle\!\langle TE', \forall t.\tau\rangle\!\rangle$, *then* $VH(l) = \langle\!\langle (TE, VE), \Lambda t.\, e\rangle\!\rangle$ *for some* TE, VE, *and* e. *Furthermore, for all* $x{:}t \mapsto l'$ *in* VE, $\Phi(TE(t)) = \Psi(l')$.

Proof (sketch): Again, by inspection of the typing rules. For example, consider case 4: We know that $TH(p) = p_1 \rightarrow p_2$. So, by inspection of the type heap value reduction rules, only **th-arrow** applies. Thus, we know that there exist σ_1 and σ_2 such that $\Phi \vdash p_1 \rightarrow p_2 \Downarrow \sigma_1 \rightarrow \sigma_2$ by the **th-arrow** rule. Therefore, $\Phi(p) = \sigma_1 \rightarrow \sigma_2$, and by assumption, $\Psi(l) = \sigma_1 \rightarrow \sigma_2$. By inspection of the heap typing rules, $\Phi; \Psi \vdash VH(l) : \sigma_1 \rightarrow \sigma_2$ can only hold via the **vh-arrow** rule. Thus, $VH(l) = \langle\!\langle (TE, VE), \mathtt{fix}\ x_1{:}t_1(x_2{:}t_2).e\rangle\!\rangle$ for some TE, VE, x_1, t_1, x_2, t_2, and e. Furthermore, we know there exist Δ and Γ such that $\Phi \vdash TE \Downarrow \Delta$ and $\Delta; \Psi \vdash VE : \Gamma$. By the **cons-ve** rule, we can conclude that for all $x{:}t \mapsto l$ in VE, $\Phi(TE(t)) = \Psi(l)$. □

Lemma 5.2 *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, $\Phi(p) = \Psi(l)$, $\Phi \vdash TE \Downarrow \Delta$, *and* $\Psi; \Delta \vdash VE : \Gamma$, *then for some* Q *and* L,

1. $\mathcal{F}_{\text{env}}(TE, VE) = (Q, L)$,

2. $FP(TE, VE) = Q$,

3. $FL(TE, VE) = Dom(L)$,
4. *for all* $l'{:}p' \in L$, $\Phi(p') = \Psi(l')$.

Corollary 5.3 *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, *and* $\Phi(p) = \Psi(l)$, *then for some* Q *and* L*:*

1. $\mathcal{F}[TH(p)](p, VH(l)) = (Q, L)$,
2. $FP(VH(l)) = Q$,
3. $FL(VH(l)) = Dom(L)$,
4. *for all* $l'{:}p' \in L$, $\Phi(p') = \Psi(l')$.

Proof: The result follows directly from the definition of $\mathcal{F}$, the Unexpanded Canonical Forms Lemma, and Lemma 5.2. □

Corollary 5.4 *If* $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, *and* $\Phi; \Psi \vdash S : \sigma_1 \to \sigma_2$, *then for some* Q *and* L,

1. $\mathcal{F}_{\text{stack}}(S) = (Q, L)$,
2. $FP(S) = Q$,
3. $FL(S) = Dom(L)$,
4. *for all* $l'{:}p' \in L$, $\Phi(p') = \Psi(l')$.

Proof: The result follows directly from the definition of $\mathcal{F}_{\text{stack}}$ and Lemma 5.2. □

Next, we formulate a set of invariants that a GC state has throughout the rewriting system.

Definition 5.5 (Well-Formed GC State) *Let* $P = (H, S, E, e)$ *be a well-typed program such that* $H = (TH, VH)$, $\vdash TH \Downarrow \Phi$, *and* $\Phi \vdash VH : \Psi$. *We say a GC state* (H_f, Q, L, H_t) *is well-formed with respect to* P *iff:*

1. $H = H_f \uplus H_t$,
2. $FP(H_t, S, E, e) \subseteq Q$,
3. $FL(H_t, S, E, e) \subseteq Dom(L)$,
4. *for all* $l{:}p$ *in* L, $\Phi(p) = \Psi(l)$,

5. *$Q \subseteq Dom(TH)$,*

6. *for all $l{:}p$ in L, $l \in Dom(VH)$ and $p \in Dom(TH)$.*

Roughly speaking, the first requirement is that the from- and to-spaces, when taken together, constitute the original program's heap. Thus, no bindings are ever lost or created by the system. The second and third invariants tell us that Q holds the free pointers of the to-space, while L holds the free locations of the to-space. The fourth invariant ensures that for all location/pointer pairs $l{:}p \in L$, p is bound to a type which, when normalized, is the same type assigned to the location l in the proof that the original program is well-formed. Finally, the fifth and sixth invariants ensure that all pointers and locations in the scan set are drawn from those pointers and locations bound in the program's heap.

Next, we show that the GC transition system preserves well-formedness of GC states, and then show that well-formedness is sufficient to guarantee that a GC state is either terminal (i.e., the scan set is empty) or else there exists a transition to another well-formed GC state. These lemmas are the direct analogs of the Preservation and Progress Lemmas for the proof of type soundness for the abstract machine (see Section 3.4). Proofs of these Lemmas may be found in Appendix B.

Lemma 5.6 (GC State Preservation) *If X is well-formed with respect to P and $X \Longrightarrow X'$, then X' is well-formed with respect to P.*

Lemma 5.7 (GC State Progress) *If $X = (H_f, Q, L, H_t)$ is well-formed with respect to P, then either Q and L are empty or else there exists a GC state X' such that $X \Longrightarrow X'$.*

Finally, correctness of the GC transition system is established by showing that the initial GC state is well-formed, and that well-formedness of a terminal state is a sufficient condition to guarantee that the to-space contains all bindings needed to keep the program closed.

Theorem 5.8 (Tag-Free GC Correctness) *If $\vdash P : \sigma$, then there exists a P' such that $P \overset{\textbf{tf-gc}}{\longmapsto} P'$. Furthermore, $P \overset{\textbf{GC}}{\longmapsto} P'$.*

Proof: Let $P = (H, S, E, e)$ where $H = (TH, VH)$ and $E = (TE, VE)$. Since $\vdash P : \sigma$, there exists Φ, Ψ, Δ, Γ, and σ' such that $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, $\Phi; \Psi \vdash S : \sigma' \to \sigma$, $\Phi \vdash TE \Downarrow \Delta$, and $\Psi; \Delta \vdash VE : \Gamma$.

By Lemma 5.2, we know that there exists Q_1 and L_1 such that $\mathcal{F}_{\text{env}}(E) = (Q_1, L_1)$, $FP(E) = Q_1$, $FL(E) = Dom(L_1)$, and for all $l{:}p$ in L_1, $\Phi(p) = \Psi(l)$. By Lemma 5.4, we know that there exists Q_2 and L_2 such that $\mathcal{F}_{\text{stack}}(S) = (Q_2, L_2)$,

$FP(S) = Q_2$, $FL(S) = Dom(L_2)$, and for all $l{:}p$ in L_2, $\Phi(p) = \Psi(l)$. Therefore, taking $Q = Q_1 \cup Q_2$ and $L = L_1 \cup L_2$, we know that $FP(\emptyset, S, E, e) = Q$, $FL(\emptyset, S, E, e) = Dom(L)$, and for all $l{:}p$ in L, $\Phi(p) = \Psi(l)$.

Therefore, the initial GC state $X = (H, Q, L, \emptyset)$ is well-formed with respect to P. Since the GC rewriting system cannot diverge, there exists some GC state $X' = (H_f, Q', L', H_t)$ such that $X \Longrightarrow^* X'$ and no step can be taken from X'. By induction on the length of this rewriting sequence using the GC State Preservation Lemma, X' is well-formed with respect to P. Since no transition exists from X', we know via GC State Progress that both Q and L are empty. Hence, $P \overset{\textbf{tf-gc}}{\longmapsto} P'$ where $P' = (H_t, S, E, e)$.

Now, since X' is well-formed with respect to P, we know that $FP(P') \subseteq Q' = \emptyset$ and $FL(P') \subseteq Dom(L') = \emptyset$. Consequently, P' is closed and thus $P \overset{\textbf{GC}}{\longmapsto} P'$. $\square$

6 Other Kinds of Garbage

In the previous sections, we showed how to specify a certain class of heap garbage and how to collect this garbage without effecting the observational behavior of programs. However, for our abstract machine, garbage is not limited to the heap. In this section, we consider two additional forms of garbage, *stack garbage* and *environment garbage*, each of which may be the source of "space leaks" in a program. We show how an implementation may avoid these leaks.

In the most general sense, a frame on the stack is garbage if removing that frame results in a Kleene-equivalent program (i.e., an observationally equivalent program). Likewise, a binding in an environment is garbage if removing that binding results in a Kleene-equivalent program. The following definitions make these notions of garbage precise.

Definition 6.1 (Stack Garbage) *Let $P = (H, S_1 \bar{\circ} S_2 \bar{\circ} S_3, E, e)$ be a well-formed program and take $P' = (H, S_1 \bar{\circ} S_3, E, e)$. We say that the sub-stack S_2 is garbage with respect to P iff P' is well-formed and $P \simeq P'$.*

Definition 6.2 (Environment Garbage) *Let P be a well-formed program with environment E occurring somewhere in P (i.e., either the program environment, or the environment of a closure), and let P' be the program obtained by replacing E with $E \setminus E'$, where $E' \subseteq E$. We say that the sub-environment E' is garbage with respect to P iff P' is a well-formed program and $P \simeq P'$.*

It may not be practical to re-claim space for stack frames or environment entries except at certain points during evaluation. Hence, most implementations restrict their attention to a certain class of garbage stack frames or environment bindings. In the remainder of this section, we examine two specific approaches for collecting certain classes of stack and environment garbage, namely *tail-call elimination* for stack garbage collection, and *environment strengthening* (also known as "black-holing") for environment garbage collection.

6.1 Tail-Call Collection

Tail-call elimination is a space optimization used in many implementations to avoid unnecessary accumulation of control information[1]. The goal of tail-call elimination is to ensure that tail-recursive procedures execute in iterative fashion, with no space requirements not imposed by the code itself. In our framework the tail-call optimization can be phrased as the elimination of *identity continuations* on the stack. The idea is formalized by the following transition rule:

$$(\overset{\mathbf{ID}}{\longmapsto})\quad (H, S_1 \circ \langle\!\langle E', \lambda x{:}t.x \rangle\!\rangle \,\bar{\circ}\, S_2, E, e) \;\overset{\mathbf{ID}}{\longmapsto}\; (H, S_1 \,\bar{\circ}\, S_2, E, e)$$

The $\overset{\mathbf{ID}}{\longmapsto}$ rule eliminates any stack frame if the code of the frame is syntactically equivalent to the identity function. The following lemmas show that $\overset{\mathbf{ID}}{\longmapsto}$ preserves types and hence, cannot cause a well-typed program to become stuck.

Lemma 6.3 (ID Preservation) *If* $\vdash P : \sigma$ *and* $P \overset{\mathbf{ID}}{\longmapsto} P'$, *then* $\vdash P' : \sigma$.

Proof: P must be of the form $(H, S_1 \circ \langle\!\langle E_1, \lambda x{:}t.x \rangle\!\rangle \,\bar{\circ}\, S_2, E, e)$, where $H = (TH, VH)$, $E_1 = (TE_1, VE_1)$, and $E = (TE, VE)$. Since $\vdash P : \sigma$, there exists a Φ, Ψ, Δ, Γ, and σ' such that (a) $\vdash TH \Downarrow \Phi$, (b) $\Phi \vdash VH : \Psi$, (c) $\Phi; \Psi \vdash S_1 \,\bar{\circ}\, \langle\!\langle E_1, \lambda x{:}t.x \rangle\!\rangle \circ S_2 : \sigma' \to \sigma$, (d) $\Phi \vdash TE \Downarrow \Delta$, (e) $\Psi; \Delta \vdash VE : \Gamma$, and (f) $\Delta; \Gamma \vdash e : \sigma'$. It suffices to show that $\Phi; \Psi \vdash S_1 \,\bar{\circ}\, S_2 : \sigma' \to \sigma$.

Now (c) can only hold via **cons-s**. Thus, by induction on the size of S_2, we can show that there exists a σ_1 such that (g) $\Phi; \Psi \vdash S_1 \circ \langle\!\langle E_1, \lambda x{:}t.x \rangle\!\rangle : \sigma_1 \to \sigma$ and (h) $\Phi; \Psi \vdash S_2 : \sigma' \to \sigma_1$. It suffices to show that $\Phi; \Psi \vdash S_1 : \sigma_1 \to \sigma$, for then, by induction on the size of S_2 we may show that $\Phi; \Psi \vdash S_1 \,\bar{\circ}\, S_2 : \sigma' \to \sigma$.

Since (g) can only hold via **cons-s**, there must exist Δ_1, Γ_1, and σ_2 such that (i) $\Phi; \Psi \vdash S_1 : \sigma_2 \to \sigma$, (j) $\Phi \vdash TE_1 \Downarrow \Delta_1$, (k) $\Delta_1 \vdash t \Downarrow \sigma_1$, (l) $\Psi; \Delta_1 \vdash VE_1 : \Gamma_1$, and (m) $\Delta_1; \Gamma_1[x{:}\sigma_1] \vdash x : \sigma_2$. But (m) can only hold via the **var** rule and thus $\sigma_1 = \sigma_2$. Consequently, $\Phi; \Psi \vdash S_1 : \sigma_1 \to \sigma$. □

[1]Tail-call elimination may also improve the running time of programs by avoiding unnecessary stack manipulations, and by decreasing the time taken to extract pointers and locations for heap garbage collection.

Corollary 6.4 (ID Progress) *If* $\vdash P_1 : \sigma$ *and* $P_1 \overset{\mathbf{ID}}{\longmapsto} P_2$, *then either* P_2 *is a printable answer or else there exists a* P_3 *such that* $P_2 \longmapsto P_3$.

As with the $\overset{\mathbf{GC}}{\longmapsto}$ rule of Section 4, we wish to show that adding $\overset{\mathbf{ID}}{\longmapsto}$ to the rewriting rules for our abstract machine does not effect the observable behavior of program evaluation. Let $\overset{\mathbf{IDR}}{\longmapsto}$ be the union of the $\overset{\mathbf{R}}{\longmapsto}$ and $\overset{\mathbf{ID}}{\longmapsto}$ relations.

Definition 6.5 (ID Evaluation Relation)

1. $P \Downarrow_{\mathbf{IDR}} r$ *iff there exists an* A *such that* $P \overset{\mathbf{IDR}}{\longmapsto}{}^{*} A$ *and* $\mathit{print}_{\mathrm{prog}}(A) = r$.
2. $P \Downarrow_{\mathbf{IDR}} \bot$ *iff* P *diverges with respect to* $\overset{\mathbf{IDR}}{\longmapsto}$.
3. $P \Downarrow_{\mathbf{IDR}}$ `*wrong*` *iff there exists a stuck* P' *such that* $P \overset{\mathbf{IDR}}{\longmapsto}{}^{*} P'$.

We would like to show that $\Downarrow_{\mathbf{IDR}}$ is the same total function as our original *eval*. As with the $\Downarrow_{\mathbf{GCR}}$ relation, we show that, whenever a $\overset{\mathbf{ID}}{\longmapsto}$ step is followed by an $\overset{\mathbf{R}}{\longmapsto}$ step, we can postpone the identity frame collection until after the $\overset{\mathbf{R}}{\longmapsto}$ step has been taken. From this, it follows that we can simulate any $\overset{\mathbf{IDR}}{\longmapsto}$ evaluation sequence with an $\overset{\mathbf{R}}{\longmapsto}$ evaluation sequence.

Lemma 6.6 (ID Postponement) *If* $\vdash P_1 : \sigma$, $P_1 \overset{\mathbf{ID}}{\longmapsto} P_2 \overset{\mathbf{R}}{\longmapsto} P_3$, *then there exists a* P_2' *such that* $P_1 \overset{\mathbf{R}}{\longmapsto}{}^{*} P_2 \overset{\mathbf{ID}}{\longmapsto}{}^{*} P_3$.

Proof (sketch): The argument proceeds by case analysis on the $\overset{\mathbf{R}}{\longmapsto}$ rewriting rule taking P_2 to P_3. The most interesting case is when P_2 steps to P_3 via the **return** rule. For this case, there are two possible sub-cases depending upon whether or not the identity frame eliminated by the $\overset{\mathbf{ID}}{\longmapsto}$ transition is the right-most stack frame.

sub-case return-a: P_1 is of the form (H, S, E, x), where

$$S = S_1 \circ \langle\!\langle E_1, \lambda x_1{:}t_1.e_1\rangle\!\rangle \circ \langle\!\langle E', \lambda x'{:}t'.x'\rangle\!\rangle,$$

and

$$P_1 \overset{\mathbf{ID}}{\longmapsto} (H, S_1 \circ \langle\!\langle E_1, \lambda x_1{:}t_1.e_1\rangle\!\rangle, E, x) \overset{\mathbf{return}}{\longmapsto} (H, S_1, E_1[x_1{:}t_1 \mapsto E(x)], e_1)$$

But then

$$\begin{aligned} P_1 &\overset{\mathbf{return}}{\longmapsto} (H, S_1 \circ \langle\!\langle E_1, \lambda x_1{:}t_1.e_1\rangle\!\rangle, E'[x'{:}t' \mapsto E(x)], x') \\ &\overset{\mathbf{return}}{\longmapsto} (H, S_1, E_1[x_1{:}t_1 \mapsto E'[x'{:}t' \mapsto E(x)](x')], e_1) \end{aligned}$$

Thus, $P_1 \overset{\mathbf{R}}{\longmapsto}^* P_3 \overset{\mathbf{ID}}{\longmapsto}^* P_3$.

sub-case return-b: P_1 is of the form (H, S, E, x), where

$$S = S_1 \circ \langle\!\langle E', \lambda x'{:}t'.x' \rangle\!\rangle \circ S_2 \circ \langle\!\langle E_1, \lambda x_1{:}t_1.e_1 \rangle\!\rangle$$

and

$$\begin{array}{lll} P_1 & \overset{\mathbf{ID}}{\longmapsto} & (H, S_1 \circ S_2 \circ \langle\!\langle E_1, \lambda x_1{:}t_1.e_1 \rangle\!\rangle, E, x) \\ & \overset{\mathbf{return}}{\longmapsto} & (H, S_1 \circ S_2, E_1[x_1{:}t_1 \mapsto E(x)], e_1) \end{array}$$

But then

$$\begin{array}{lll} P_1 & \overset{\mathbf{return}}{\longmapsto} & (H, S_1 \circ \langle\!\langle E', \lambda x'{:}t'.x' \rangle\!\rangle \circ S_2, E_1[x_1{:}t_1 \mapsto E(x)], e_1) \\ & \overset{\mathbf{ID}}{\longmapsto} & (H, S_1 \circ S_2, E_1[x_1{:}t_1 \mapsto E(x)], e_1) \end{array}$$

Thus, there exists a P_2' such that $P_1 \overset{\mathbf{R}}{\longmapsto}^* P_2' \overset{\mathbf{ID}}{\longmapsto}^* P_3$. □

Corollary 6.7 *For all $n \geq 0$, if $P_0 \overset{\mathbf{IDR}}{\longmapsto}^n P_n$, then there exists a P_n' such that $P_0 \overset{\mathbf{R}}{\longmapsto}^* P_n'$ and $P_n' \overset{\mathbf{ID}}{\longmapsto}^* P_n$.*

Proof: By induction on n using the $\overset{\mathbf{ID}}{\longmapsto}$ Postponement lemma. □

Lemma 6.8 (ID Answer) *If $\vdash P : \sigma$, and $P \overset{\mathbf{ID}}{\longmapsto}^* A$ for some A, then $P \overset{\mathbf{R}}{\longmapsto}^* A$.*

Theorem 6.9 (ID Correctness) *If $\vdash P : \sigma$, then $P \Downarrow r$ iff $P \Downarrow_{\mathbf{GCR}} r$.*

Proof: We can simulate any $\overset{\mathbf{R}}{\longmapsto}$ step with a $\overset{\mathbf{IDR}}{\longmapsto}$ step by simply never performing an $\overset{\mathbf{ID}}{\longmapsto}$ transition. Consequently, if $P \Downarrow r$, then $P \Downarrow_{\mathbf{IDR}} r$.

Suppose $P \Downarrow_{\mathbf{IDR}} r$. By GC Soundness, $r \neq$ `*wrong*`. If $r \neq \bot$, then there exists an n and A such that $P \overset{\mathbf{IDR}}{\longmapsto}^n A$ and $\mathit{print}_{\mathrm{prog}}(A) = r$. By Corollary 6.7, there exists an P' such that $P \overset{\mathbf{R}}{\longmapsto}^* P' \overset{\mathbf{ID}}{\longmapsto}^* A$. By Lemma 6.8, $P' \overset{\mathbf{R}}{\longmapsto}^* A$.

If $r = \bot$ then there exists an infinite sequence $P', P_1, P_1', P_2, P_2', \cdots$ such that

$$P \overset{\mathbf{ID}}{\longmapsto}^* P' \overset{\mathbf{R}}{\longmapsto} P_1 \overset{\mathbf{ID}}{\longmapsto}^* P_1' \overset{\mathbf{R}}{\longmapsto}^* P_2 \overset{\mathbf{ID}}{\longmapsto}^* P_2' \overset{\mathbf{R}}{\longmapsto} \cdots.$$

By Corollary 6.7, we can construct an infinite sequence $P'', P_1'', P_2'', \cdots$ such that

$$P \overset{\mathbf{R}}{\longmapsto} P'' \overset{\mathbf{R}}{\longmapsto} P_1'' \overset{\mathbf{R}}{\longmapsto} P_2'' \overset{\mathbf{R}}{\longmapsto} \cdots$$

Thus, $P \Downarrow \bot$. Consequently, if $P \Downarrow_{\mathbf{IDR}} r$, then $P \Downarrow r$. □

Whereas the proof of the $\xrightarrow{\mathbf{ID}}$ postponement lemma relies crucially upon the ability to throw away frames in the middle of the stack, real implementations avoid pushing identity frames onto the stack. In effect, every transition that can push a frame on the stack is split into two transitions: If the frame to be pushed on the stack is an identity frame, then the frame is simply discarded; otherwise it is pushed on the stack.

In our abstract machine, only the **c-zero**, **c-succ**, **app**, and **tapp** rules push frames on the stack. Composing these transitions with the $\xrightarrow{\mathbf{ID}}$ transition (applied to the right-most stack frame) yields the following new transition rules:

$$(\textbf{tail-zero}) \quad \frac{H(E(x')) = \mathtt{0}}{(H, S, E, \mathtt{let}\ \ x{:}t = \mathtt{case}(x', e_0, \lambda x_1{:}t_1.e_1)\ \ \mathtt{in}\ \ x) \longmapsto (H, S, E, e_0)}$$

$$(\textbf{tail-succ}) \quad \frac{H(E(x')) = \mathtt{succ}\ \ l}{\begin{array}{l}(H, S, E, \mathtt{let}\ \ x{:}t = \mathtt{case}(x', e_0, \lambda x_1{:}t_1.e_1)\ \ \mathtt{in}\ \ x) \longmapsto \\ \qquad (H, S, E[x_1{:}t_1 \mapsto l], e_1)\end{array}}$$

$$(\textbf{tail-app}) \quad \frac{H(E(x_1)) = \langle\!\langle E', \mathtt{fix}\ \ x_1'{:}t_1'(x_2'{:}t_2').e' \rangle\!\rangle}{\begin{array}{l}(H, S, E, \mathtt{let}\ \ x{:}t = x_1\, x_2\ \ \mathtt{in}\ \ x) \longmapsto \\ \qquad (H, S, E'[x_1'{:}t_1' \mapsto E(x_1), x_2'{:}t_2' \mapsto E(x_2)], e')\end{array}}$$

$$(\textbf{tail-tapp}) \quad \frac{H(E(x_1)) = \langle\!\langle E', \Lambda t_1'.\, e' \rangle\!\rangle}{(H, S, E, \mathtt{let}\ \ x{:}t = x_1\, [t_1]\ \ \mathtt{in}\ \ x) \longmapsto (H, S, E'[t_1' \mapsto E(t_1)], e')}$$

It is easy to see that adding these new rules, and always choosing the appropriate **tail-** transition when possible, yields a computation without any identity stack frames. For certain classes of programs, this optimization is crucial in order to bound the amount of stack space needed to run programs. In particular, when programs are written in *continuation-passing style* (CPS) [22], then the **return** transition is never enabled until the end of the computation, assuming the computation terminates. Instead, each function is passed an extra argument function (the continuation), and the result of the function is passed to the continuation. In effect, all function applications turn into potential **tail-app** transitions. But if no tail-call elimination is performed, then this coding style delays all of the **return** transitions until the end of the computation (assuming the program terminates) and at worst results in unbounded stack-space requirements (assuming the program diverges). While this is reasonable behavior in an observational sense, it is unreasonable behavior in practice. Furthermore, a heap garbage collector must process and preserve all objects that are reachable from these unnecessary frames, so the total amount of garbage in the state of the abstract machine grows quickly.

Some languages, notably Scheme [17], require that all implementations faithfully implement tail-call elimination in order to address these practical concerns[2]. Yet, the standard models for Scheme make neither the control stack nor heap explicit [17], and thus the tail-call requirement is at best an informal contract between the language specification and its implementors. In contrast, the model we use here allows the language designer to specify the requirement precisely: asymptotically, an implementation should use no more space than our abstract machine requires with the **tail-** rules. Nevertheless, our model is sufficiently abstract that we can argue the correctness of such an implementation in the observational sense, without overly constraining implementations.

6.2 Environment Strengthening

Collecting garbage bindings in environments is much like collecting garbage bindings in the heap. In particular, a reasonable strategy for collecting bindings in an environment is to determine which bindings are inaccessible from the code associated with the environment, and drop those bindings.

We formulate an environment garbage collector by specifying two inference rules that allow us to *strengthen* the environment of a type or value closure:

$$(\mathbf{STE})\ \frac{FTV(\tau) \subseteq Dom(TE_1)}{\langle\!\langle TE_1 \uplus TE_2, \tau \rangle\!\rangle \overset{\mathbf{STE}}{\longmapsto} \langle\!\langle TE_1, \tau \rangle\!\rangle}$$

$$(\mathbf{SE})\ \frac{FV(e) \subseteq Dom(VE_1) \qquad FTV(VE_1) \cup FTV(e) \subseteq Dom(TE_1)}{\langle\!\langle (TE_1 \uplus TE_2,\ VE_1 \uplus VE_2), e \rangle\!\rangle \overset{\mathbf{SE}}{\longmapsto} \langle\!\langle (TE_1,\ VE_1), e \rangle\!\rangle}$$

The $\overset{\mathbf{STE}}{\longmapsto}$ rule allows us to strengthen the type environment of a type closure by discarding those bindings not referenced by the type. Similarly, the $\overset{\mathbf{SE}}{\longmapsto}$ rule allows us to strengthen the type and value environment of a value closure, as long as all of the free variables of the code are in the domain of the resulting value environment, and all of the free type variables of both the code and the value environment are in the domain of the resulting type environment.

We use $\overset{\mathbf{STE}}{\longmapsto}$ and $\overset{\mathbf{SE}}{\longmapsto}$ to formulate rewriting rules that allow us to strengthen the various environments that may arise in an abstract machine state:

$$(\mathbf{SE\text{-}th})\ \frac{\langle\!\langle TE_1, \tau \rangle\!\rangle \overset{\mathbf{STE}}{\longmapsto} \langle\!\langle TE_2, \tau \rangle\!\rangle}{(H[p \mapsto \langle\!\langle TE_1, \tau \rangle\!\rangle], S, E, e) \overset{\mathbf{SE\text{-}th}}{\longmapsto} (H[p \mapsto \langle\!\langle TE_2, \tau \rangle\!\rangle], S, E, e)}$$

[2]See Chase [14] for a further discussion of practical space safety issues.

$$(\textbf{SE-vh})\ \frac{\langle\!\langle E_1, e'\rangle\!\rangle \overset{\textbf{SE}}{\longmapsto} \langle\!\langle E_2, e'\rangle\!\rangle}{(H[l \mapsto \langle\!\langle E_1, e'\rangle\!\rangle], S, E, e) \overset{\textbf{SE-vh}}{\longmapsto} (H[l \mapsto \langle\!\langle E_2, e'\rangle\!\rangle], S, E, e)}$$

$$(\textbf{SE-stack})\ \frac{\langle\!\langle E_1, \lambda x'{:}t'.e'\rangle\!\rangle \overset{\textbf{SE}}{\longmapsto} \langle\!\langle E_2, \lambda x'{:}t'.e'\rangle\!\rangle}{\begin{array}{l}(H, S_1 \circ \langle\!\langle E_1, \lambda x'{:}t'.e'\rangle\!\rangle \circ S_2, E, e) \overset{\textbf{SE-stack}}{\longmapsto} \\ \quad (H, S_1 \circ \langle\!\langle E_2, \lambda x'{:}t'.e'\rangle\!\rangle \circ S_2, E, e)\end{array}}$$

$$(\textbf{SE-env})\ \frac{\langle\!\langle E_1, e\rangle\!\rangle \overset{\textbf{SE}}{\longmapsto} \langle\!\langle E_2, e\rangle\!\rangle}{(H, S, E_1, e) \overset{\textbf{SE-env}}{\longmapsto} (H, S, E_2, e)}$$

Taken together, these rules allow us to trim the size of all the kinds of environments that may occur in a machine state. As with tail-call collection, trimming environments may also allow more heap garbage to be collected, since there are fewer references to heap allocated objects. Finally, proving that these rules preserve types and do not affect the observable behavior of programs can be accomplished in the same way we argued these results for the $\overset{\textbf{GC}}{\longmapsto}$ and $\overset{\textbf{ID}}{\longmapsto}$ rules[3].

In practice, implementations only perform environment strengthening at certain points during evaluation, just as most implementations only perform tail-call elimination in conjunction with function application. In particular, Appel suggests that a *space-safe* implementation strategy for closures is to trim their environment only to those variables that occur free in the associated code [4]. However, it is often impractical to trim the current environment (at every instruction), or to even trim the environments of closures as they are pushed on the stack. Instead, some implementations, including the TIL/ML compiler, delay trimming the current environment and the environments of stack frames until garbage collection is invoked.

Many implementations of functional languages perform a program transformation known as *closure conversion* [6, 4, 29] to eliminate nested, higher-order functions. As a result of the transformation, nested functions are replaced with a record, where the first component is a pointer to some code, and the second component is a pointer to a data structure containing bindings for the free variables of the original function. The code abstracts both the environment and the argument to the function. Application is replaced with operations to extract the code and environment, and to apply the code simultaneously to the environment and the argument. In this respect, closure conversion reifies the heap closures of our abstract machine as language constructs in the same fashion that CPS conversion reifies the stack closures of our abstract machine as functions.

[3]Some care must be taken, as with the $\overset{\textbf{GC}}{\longmapsto}$ transition, to rule out infinite sequences of environment strengthening.

During closure conversion, when emitting the operations to construct the environment for a particular closure, it is possible to strengthen the environment. It is also possible for different closures to share portions of the same environment. In particular, many implementations, such as the CAM [18], allow closures in the same lexical scope to share an environment for that scope. However, care must be taken when two closures share an environment in order to avoid a class of space leaks [37]. In particular, the shared portion of the environment should only contain bindings for those variables that occur free in the code of both closures. Though the abstract machine presented here does not support shared environments, it is fairly straightforward to add environments to the set of heap-allocated values so that they may be shared.

7 Related Work

The ideas in this paper are derived from our previous work with Matthias Felleisen [31] and Morrisett's dissertation [30], where we presented a much simpler abstract machine that relied upon meta-level substitution and meta-level evaluation contexts to implicitly represent the control state and environment of a computation. In this paper, we chose to make these details explicit so that issues like tail-call elimination and environment strengthening could be addressed. We further extended the previous work by addressing the issues of allocating and collecting type information, and by giving a full treatment of tag-free collection in the context of a polymorphic language.

The literature on garbage collection in sequential programming languages contains few papers that attempt to provide a compact characterization of algorithms or correctness proofs. Demers *et al.* [19] 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, tag-free collector of Section 5. Nettles [32] provides a concrete specification of a copying garbage collection algorithm using the Larch specification language. Our $\overset{\mathbf{GC}}{\longmapsto}$ transition rule is essentially a high-level, one-line description of his specification, and the $\overset{\mathbf{tf\text{-}gc}}{\longmapsto}$ rule is a particular implementation.

Hudak gives a denotational model that tracks reference counts for a first-order language [27]. 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 [15]. 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 accessible" [9]. This provides a compiler enough information to determine when certain objects may be garbage collected or over-written.

Tolmach [40] built a type-recovery collector for a variant of SML that passes type information to polymorphic routines during execution, effectively implementing our $\lambda_{\mathrm{gc}}^{\rightarrow\forall}$ language and the type-based, tag-free collector of Section 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]; 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, tag-free collection in monomorphic [11, 43, 12] and polymorphic [3, 25, 26, 23] 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 [8] 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 [26] 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 [23] gave another argument based on Reynolds's abstraction/parametricity theorem [35]. 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.*

Blelloch and Greiner give an abstract machine for evaluation of the parallel programming language NESL [10]. The goal of their work was to provide provable space and time bounds for an implementation of NESL. Their machine is based directly on the CESK machine [20]. However, some details in their formulation, such as the representation of control information, are left implicit.

8 Summary and Conclusions

We have presented an abstract machine for describing the evaluation of polymorphically-typed, functional programs. Unlike traditional models of functional languages, our abstract machine exposes many important details of memory management such as the heap, control stack, and environment. Nevertheless, our machine is sufficiently abstract that we are able to use conventional techniques to specify its static semantics, and to prove soundness of the static semantics with respect to the transitions of the abstract machine.

Since the abstract machine exposes memory management issues, we are able to precisely specify important classes of garbage that arise during the evaluation of programs, including unreachable heap values, tail-call stack frames, and unreferenced environment bindings. For each class of garbage, we presented an abstract specification of a collector which reclaims the garbage objects, and proved that these collectors do not affect the observable behavior of well-formed programs. In addition, we gave a detailed specification of Tolmach's type-based, tag-free, copying garbage collector [40] and proved its correctness. The techniques used to specify and prove correctness for all of the collectors were based on those used to establish type soundness for the abstract machine.

Admittedly, our machine abstracts many important low-level implementation details for memory management. In particular, we *a priori* considering programs equivalent up to α-conversion of bound variables and any re-ordering of heap or environment bindings; we also ignore representation and sharing issues for environments. However, abstracting these details greatly simplifies our reasoning and keeps us from over-constraining implementations. Hence, we claim that such a model provides an important intermediate step in establishing the correctness of a wide class of existing and future implementations.

9 Acknowledgments

This paper is based on a preliminary version by Matthias Felleisen and the authors [31, 30]. We very gratefully acknowledge the important contributions of Matthias Felleisen to this work.

We are grateful to Chris Stone, Todd Wilson, and Tanya Morrisett for their careful reading of this paper and for their many useful comments and suggestions.

References

[1] Shail Aditya and Alejandro Caro. Compiler-directed type reconstruction for

polymorphic languages. In *ACM Conference on Functional Programming and Computer Architecture*, pages 74–82, Copenhagen, June 1993.

[2] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *ACM Conference on Lisp and Functional Programming*, pages 12–23, Orlando, June 1994.

[3] Andrew W. Appel. Run-time tags aren't necessary. *LISP and Symbolic Computation*, 2:153–162, 1989.

[4] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[5] Andrew W. Appel. A critique of Standard ML. Technical Report CS–TR–364–92, Princeton University, Princeton, NJ, February 1992.

[6] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 293–302, Austin, TX, January 1989.

[7] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In J. Maluszynski and M. Wirsing, editors, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[8] Henry Baker. Unify and conquer (garbage, updating, aliasing ...) in functional languages. In *ACM Conference on Lisp and Functional Programming*, pages 218–226, Nice, 1990.

[9] Erik Barendsen and Sjaak 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, 1993. Springer-Verlag. Extended abstract.

[10] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of NESL. In *ACM Conference on Functional Programming and Computer Architecture*, pages 213–225, Philadelphia, PA, May 1996.

[11] 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.

[12] Dianne Ellen Britton. Heap storage management for the programming language Pascal. Master's thesis, University of Arizona, 1975.

[13] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[14] David Chase. Safety considerations for storage allocation optimizations. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[15] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *ACM Conference on Lisp and Functional Programming*, pages 139–150, San Francisco, June 1992.

[16] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, Laurent Hascoet, and Gilles Kahn. Natural semantics on the computer. Technical Report RR 416, INRIA, Sophia–Antipolis, France, June 1985.

[17] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-Sep. 1991.

[18] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The categorical abstract machine. In J. P. Jouannaud, editor, *1985 Conference on Functional Programming and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[19] A. Demers, M. Weiser, B. Hayes, H. Boehm, D. Bobrow, and S. Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Seventeenth ACM Symposium on Principles of Programming Languages*, pages 261–269, San Francisco, January 1990.

[20] Matthias Felleisen and Daniel P. Friedman. A calculus for assignments in higher-order languages. In *Thirteenth ACM Symposium on Principles of Programming Languages*, January 1987.

[21] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[22] Michael J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6(3/4):259–288, November 1993.

[23] Pascal Fradet. Collecting more garbage. In *ACM Conference on Functional Programming and Computer Architecture*, pages 24–33, Orlando, June 1994.

[24] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure.* PhD thesis, Université Paris VII, 1972.

[25] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 165–176, Toronto, June 1991.

[26] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *ACM Conference on Lisp and Functional Programming*, pages 53–65, San Francisco, June 1992.

[27] Paul Hudak. A semantic model of reference counting and its abstraction. In *ACM Conference on Lisp and Functional Programming*, pages 351–363, August 1986.

[28] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[29] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, FL, January 1996.

[30] Greg Morrisett. *Compiling with Types*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1995. (Available as Carnegie Mellon University School of Computer Science technical report CMU–CS–95–226.).

[31] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Functional Programming and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995. ACM.

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

[33] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI–FN–19, Computer Science Department, Aarhus University, 1981.

[34] John C. Reynolds. Towards a theory of type structure. In *Colloq. sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer-Verlag, 1974.

[35] John C. Reynolds. Types, abstraction, and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83*, pages 513–523. Elsevier Science Publishers B. V., 1983.

[36] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *1992 ACM Conference on LISP and Functional Programming*, pages 288–298, San Francisco, CA, June 1992. ACM.

[37] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, pages 150–161, Orlando, June 1994.

[38] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. Til: A type-directed optimizing compiler for ml. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages ?–?, Philadelphia, PA, May 1996.

[39] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89:1–34, November 1990.

[40] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *Proc. 1994 ACM Conference on LISP and Functional Programming*, pages 1–11, Orlando, FL, June 1994. ACM.

[41] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, St. Malo, September 1992. Springer-Verlag.

[42] Paul R. Wilson. Garbage collection. *Computing Surveys*, 1995. Expanded version of [41]. Draft available via anonymous internet FTP from `cs.utexas.edu` as `pub/garbage/bigsurv.ps`. In revision, to appear.

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

[44] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.

A Typing Rules for the Abstract Machine

$\boxed{5.\quad \Phi \vdash TE \Downarrow \Delta}$

$$\textbf{(empty-te)}\ \ \Phi \vdash \emptyset \Downarrow \emptyset$$

$$\textbf{(cons-te)}\ \frac{\Phi \vdash TE \Downarrow \Delta \qquad \Phi(p) = \sigma}{\Phi \vdash TE[t \mapsto p] \Downarrow \Delta[t = \sigma]}$$

$\boxed{6.\quad \Phi \vdash \mu \Downarrow \sigma}$

$$\textbf{(th-nat)}\ \Phi \vdash \mathsf{nat} \Downarrow \mathsf{nat} \qquad\qquad \textbf{(th-unit)}\ \Phi \vdash \mathsf{unit} \Downarrow \mathsf{unit}$$

$$\textbf{(th-prod)}\ \frac{\Phi(p_i) = \sigma_i}{\Phi \vdash p_1 \times p_2 \Downarrow \sigma_1 \times \sigma_2} \quad (i = 1, 2)$$

$$\textbf{(th-arrow)}\ \frac{\Phi(p_i) = \sigma_i}{\Phi \vdash t_1 \to t_2 \Downarrow \sigma_1 \to \sigma_2} \quad (i = 1, 2)$$

$$\textbf{(th-all)}\ \frac{\Phi \vdash TE \Downarrow \Delta \qquad \Delta \vdash \forall t.\tau \Downarrow \forall t.\sigma}{\Phi \vdash \langle\!\langle TE, \forall t.\tau \rangle\!\rangle \Downarrow \forall t.\sigma}$$

$\boxed{7.\quad \vdash TH \Downarrow \Phi}$

$$\textbf{(empty-th)}\ \vdash \emptyset \Downarrow \emptyset$$

$$\textbf{(cons-th)}\ \frac{\vdash TH \Downarrow \Phi \qquad \Phi \vdash \mu \Downarrow \sigma}{\vdash TH[p \mapsto \mu] \Downarrow \Phi[p = \sigma]}$$

Figure 10: Type Environment and Heap Reduction

$\boxed{8.\quad \Psi;\Delta \vdash VE : \Gamma}$

$$\textbf{(empty-ve)}\ \ \Psi;\Delta \vdash \emptyset : \emptyset \quad (Abstr(\Delta) = \emptyset)$$

$$\textbf{(cons-ve)}\ \ \frac{\Psi;\Delta \vdash VE : \Gamma \qquad \Psi(l) = \sigma \qquad \Delta \vdash t \Downarrow \sigma}{\Psi;\Delta \vdash VE[x{:}t \mapsto l] : \Gamma[x{:}\sigma]}$$

$\boxed{9.\quad \Phi;\Psi \vdash h : \sigma}$

$$\textbf{(vh-zero)}\ \ \Phi;\Psi \vdash \mathsf{0} : \mathsf{nat} \qquad\qquad \textbf{(vh-succ)}\ \ \frac{\Psi(l) = \mathsf{nat}}{\Phi;\Psi \vdash \mathsf{succ}\ \ l : \mathsf{nat}}$$

$$\textbf{(vh-unit)}\ \ \Phi;\Psi \vdash \langle\rangle : \mathsf{unit}$$

$$\textbf{(vh-prod)}\ \ \frac{\Psi(l_i) = \sigma_i}{\Phi;\Psi \vdash \langle l_1, l_2\rangle : \sigma_1 \times \sigma_2} \quad (i = 1, 2)$$

$$\textbf{(vh-arrow)}\ \ \frac{\Phi \vdash TE \Downarrow \Delta \qquad \Psi;\Delta \vdash VE : \Gamma \qquad \Delta;\Gamma \vdash \texttt{fix}\ \ x{:}t(x_1{:}t_1).e : \sigma_1 \to \sigma_2}{\Phi;\Psi \vdash \langle\!\langle (TE, VE), \texttt{fix}\ \ x{:}t(x_1{:}t_1).e \rangle\!\rangle : \sigma_1 \to \sigma_2}$$

$$\textbf{(vh-all)}\ \ \frac{\Phi \vdash TE \Downarrow \Delta \qquad \Psi;\Delta \vdash VE : \Gamma \qquad \Delta;\Gamma \vdash \Lambda t.\, e : \forall t.\sigma}{\Phi;\Psi \vdash \langle\!\langle (TE, VE), \Lambda t.\, e \rangle\!\rangle : \forall t.\sigma}$$

$\boxed{10.\quad \Phi \vdash VH : \Psi}$

$$\textbf{(empty-vh)}\ \ \Phi \vdash \emptyset : \emptyset$$

$$\textbf{(cons-vh)}\ \ \frac{\Phi \vdash VH : \Psi \qquad \Phi;\Psi \vdash h : \sigma}{\Phi \vdash VH[l \mapsto h] : \Psi[l{:}\sigma]}$$

Figure 11: Value Environment and Heap Typing

$\boxed{11. \quad \Phi;\Psi \vdash S : \sigma_1 \to \sigma_2}$

$$\textbf{(empty-s)} \quad \Phi;\Psi \vdash []_\sigma : \sigma \to \sigma \quad (FTV(\sigma) = \emptyset)$$

$$\textbf{(cons-s)} \quad \frac{\begin{array}{c}\Phi;\Psi \vdash S : \sigma_3 \to \sigma_2 \qquad \Phi \vdash TE \Downarrow \Delta \qquad \Psi;\Delta \vdash VE : \Gamma \\ \Delta \vdash t \Downarrow \sigma_1 \qquad \Delta;\Gamma[x{:}\sigma_1] \vdash e : \sigma_3\end{array}}{\Phi;\Psi \vdash S \circ \langle\!\langle (TE, VE), \lambda x{:}t.e \rangle\!\rangle : \sigma_1 \to \sigma_2}$$

$\boxed{12. \quad \vdash P : \sigma}$

$$\textbf{(prog)} \quad \frac{\begin{array}{c}\vdash TH \Downarrow \Phi \qquad \Phi \vdash VH : \Psi \\ \Phi;\Psi \vdash S : \sigma' \to \sigma \qquad \Phi \vdash TE \Downarrow \Delta \\ \Psi;\Delta \vdash VE : \Gamma \qquad \Delta;\Gamma \vdash e : \sigma'\end{array}}{\vdash ((TH, VH), S, (TE, VE), e) : \sigma}$$

Figure 12: Stack and Program Typing

B Proofs of Main Lemmas

Lemma 3.10 (Type Preservation) *If* $\vdash P : \sigma$ *and* $P \xmapsto{\mathbf{R}} P'$, *then* $\vdash P' : \sigma$.

Proof: Let $P = ((TH_0, VH_0), S_0, (TE_0, VE_0), e_0)$. The judgment $\vdash P : \sigma$ can only hold via the **prog** rule, so there must exist Φ_0, Ψ_0, Δ_0, Γ_0, and σ_0 such that:

(a) $\vdash TH_0 \Downarrow \Phi_0$,
(b) $\Phi_0 \vdash VH_0 : \Psi_0$,
(c) $\Phi_0;\Psi_0 \vdash S_0 : \sigma_0 \to \sigma$,
(d) $\Phi_0 \vdash TE_0 : \Delta_0$,
(e) $\Psi_0;\Delta_0 \vdash VE_0 : \Gamma_0$, and
(f) $\Delta_0;\Gamma_0 \vdash e_0 : \sigma_0$.

The proof proceeds by cases on the rewriting rule that takes P to P' (see Figure 8), using the syntax-directed nature of the typing rules.

case return: e_0 is x' and S_0 is $S \circ \langle\!\langle (TE, VE), \lambda x{:}t.e \rangle\!\rangle$ for some x', S, TE, VE, x, t, and e, and

$$P' = ((TH_0, VH_0), S, (TE, VE[x{:}t \mapsto VE_0(x')]), e).$$

Hence, (c) must be derived from the **cons-s** rule and thus there exists a σ_1, Δ, and Γ such that $\Phi_0;\Psi_0 \vdash S : \sigma_1 \to \sigma$, $\Phi_0 \vdash TE \Downarrow \Delta$, $\Psi_0;\Delta \vdash VE : \Gamma$, $\Delta \vdash t \Downarrow \sigma_0$, and $\Delta; \Gamma[x{:}\sigma_0] \vdash e : \sigma_1$. Now, (f) must hold via the **var** rule and thus $\Gamma_0(x') = \sigma_0$. Therefore, (e) must hold via the **cons-ve** rule and thus $\Psi_0 \vdash VE_0(x') : \sigma_0$. Hence, by the **cons-ve** rule, $\Psi_0; \Delta \vdash VE[x{:}t \mapsto l] : \Gamma[x{:}\sigma_0]$.

case talloc: e_0 is $\texttt{let type } t = \nu \texttt{ in } e$ for some t, ν, and e, and

$$P' = ((TH_0[p \mapsto \hat{TE}_0(\nu)], VH_0), S, (TE_0[t \mapsto p], VE_0), e).$$

Hence, (f) holds via the **let-type-exp** rule, so there exists a σ' such that $\Delta_0 \vdash \nu \Downarrow \sigma'$, and $\Delta_0[t = \sigma']; \Gamma_0 \vdash e : \sigma$. Since (a) and (d) hold, Lemma 3.6 implies that $\Phi_0 \vdash \hat{TE}_0(\nu) \Downarrow \sigma'$. Therefore, by **cons-th**, $\vdash TH_0[p \mapsto \hat{TE}_0(\nu)] : \Phi_0[p = \sigma']$. Hence, by **cons-te**, $\Phi_0[p = \sigma'] \vdash TE_0[t \mapsto p] : \Delta_0[t = \sigma']$.

For the remainder of the cases, e_0 must be of the form $\texttt{let } x{:}t = b \texttt{ in } e$ for some x, t, b, and e. Therefore, (f) must hold via the **let-exp** rule, and hence there exists some σ' such that:

(g) $\Delta_0 \vdash t \Downarrow \sigma'$,

(h) $\Delta_0; \Gamma_0 \vdash b : \sigma'$, and

(i) $\Delta_0; \Gamma_0[x{:}\sigma'] \vdash e : \sigma_0$.

The proof proceeds by cases on b.

case valloc: b is some a and

$$P' = ((TH_0, VH_0[l \mapsto \hat{E}_0(a)]), S_0, (TE_0, VE_0[x{:}t \mapsto l]), e),$$

where $E_0 = (TE_0, VE_0)$. Since (a), (b), (d) and (e) hold, Lemma 3.7 implies that $\Phi_0; \Psi_0 \vdash \hat{E}_0(a) : \sigma'$. Therefore by **cons-vh**, $\Phi_0 \vdash VH_0[l \mapsto \hat{E}_0(a)] : \Psi_0[l{:}\sigma']$. Thus, by **cons-ve**, $\Psi_0[l{:}\sigma'] \vdash VE_0[x{:}t \mapsto l] : \Gamma_0[x{:}\sigma']$.

case c-zero: b is $\mathsf{case}(x', e_0', \lambda x_1{:}t_1.e_1)$ for some x', e_0', x_1', t_1, and e_1. Furthermore, $VH_0(VE_0(x')) = \mathsf{0}$ and thus

$$P' = ((TH_0, VH_0), S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e \rangle\!\rangle, (TE_0, VE_0), e_0').$$

By (c), (d), (e), (g), (i) and the **cons-s** rule, $\Delta_0; \Gamma_0 \vdash S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e \rangle\!\rangle : \sigma' \to \sigma$. The only way (h) can be derived is via the **nat-E** rule, and thus $\Delta_0; \Gamma_0 \vdash e_0' : \sigma'$.

case c-succ: e_0 is $\mathsf{case}(x', e_0', \lambda x_1{:}t_1.e_1)$ for some x', e_0', x_1', t_1, and e_1. Furthermore, $VH_0(VE_0(x')) = \mathsf{succ}\ l$ for some l and thus

$$P' = ((TH_0, VH_0), S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e \rangle\!\rangle, (TE_0, VE_0[x_1{:}t_1 \mapsto l]), e_1).$$

By (c), (d), (e), (g), (i) and the **cons-s** rule, $\Delta_0; \Gamma_0 \vdash S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e \rangle\!\rangle : \sigma' \to \sigma$. Now (i) must hold via **nat-E**, so we can conclude that $\Delta_0; \Gamma_0 \vdash x' : \mathsf{nat}$, $\Delta_0 \vdash t_1 \Downarrow \mathsf{nat}$, and $\Delta_0; \Gamma_0[x_1{:}\mathsf{nat}] \vdash e_1 : \sigma'$. Therefore, through (e) and the **cons-ve** rule, we can conclude that $\Psi_0(VE_0(x')) = \mathsf{nat}$. Thus, through (b) and the **cons-vh** rule, $\Phi_0;\Psi_0 \vdash VH_0(VE_0(x')) : \mathsf{nat}$ and consequently, $\Phi_0;\Psi_0 \vdash \mathsf{succ}\ l : \mathsf{nat}$. Hence, working backwards from the **vh-succ** rule, $\Psi_0(l) = \mathsf{nat}$. Therefore, $\Psi_0;\Delta_0 \vdash TE_0[x_1{:}t \mapsto l] : \Gamma_0[x_1{:}\mathsf{nat}]$.

case proj: b is $\pi_i\, x'$ for some x', i is either 1 or 2, and $VH_0(VE_0(x')) = \langle l_1, l_2\rangle$ for some l_1 and l_2. Thus,

$$P' = ((TH_0, VH_0), S_0, (TE_0, VE_0[x{:}t \mapsto l_i]), e).$$

Now (h) must hold via the **prod-E** rule, and thus $\Delta_0;\Gamma_0 \vdash x' : \sigma_1 \times \sigma_2$ for some σ_1 and σ_2, where $\sigma' = \sigma_i$. Thus, through (b) and (e), we can conclude that $\Psi_0(VE_0(x')) = \sigma_1 \times \sigma_2$. Therefore, working backwards from the **vh-prod** rule, $\Psi_0(l_i) = \sigma_i$. Thus, $\Psi_0;\Delta_0 \vdash VE_0[x{:}t \mapsto l_i] : \Gamma_0[x{:}\sigma_i]$.

case app: b is of the form $x_1\, x_2$ for some x_1 and x_2, and $VH_0(VE_0(x_1))$ is $\langle\!\langle (TE, VE), \mathtt{fix}\ x_1'{:}t_1'(x_2'{:}t_2').e'\rangle\!\rangle$. Thus, P' is:

$$((TH_0, VH_0), S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e\rangle\!\rangle, (TE, VE[x_1'{:}t_1' \mapsto VE_0(x_1), x_2'{:}t_2' \mapsto VE_0(x_2)]), e').$$

By (c), (d), (e), (g), (i), and the **cons-s** rule, $\Phi_0;\Psi_0 \vdash S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e\rangle\!\rangle : \sigma' \to \sigma$. Now (h) can only hold via the **arrow-E** rule and thus there exists a σ_1 such that $\Delta_0;\Gamma_0 \vdash x_1 : \sigma_1 \to \sigma'$ and $\Delta_0;\Gamma_0 \vdash x_2 : \sigma_1$. Therefore, $\Psi_0;\Delta_0 \vdash VE_0(x_1) : \sigma_1 \to \sigma'$ and $\Psi_0;\Delta_0 \vdash VE_0(x_2) : \sigma_1$. Consequently, $\Phi_0;\Psi_0 \vdash \langle\!\langle (TE, VE), \mathtt{fix}\ x_1'{:}t_1'(x_2'{:}t_2').e'\rangle\!\rangle : \sigma_1 \to \sigma_2$. Now this can only hold via the **vh-arrow** rule. Thus, there exists a Δ and Γ such that $\Phi_0 \vdash TE \Downarrow \Delta$, $\Delta \vdash t_1' \Downarrow \sigma_1 \to \sigma'$, $\Delta \vdash t_2' \Downarrow \sigma_1$, $\Psi_0;\Delta \vdash VE : \Gamma$, and $\Delta;\Gamma[x_1'{:}\sigma_1 \to \sigma', x_2'{:}\sigma_1] \vdash e' : \sigma'$. By **cons-ve**, $\Psi_0;\Delta_0 \vdash VE[x_1'{:}t_1' \mapsto VE_0(x_1), x_2'{:}t_2' \mapsto VE_0(x_2)] : \Gamma[x_1'{:}\sigma_1 \to \sigma', x_2'{:}\sigma_1]$.

case tapp: b is $x_1\,[t_2]$ for some x_1 and t_2, and $VH_0(VE_0(x_1)) = \langle\!\langle (TE, VE), \Lambda t'.\, e'\rangle\!\rangle$ for some (TE, VE), t', and e'. Thus,

$$P' = ((TH_0, VH_0), S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e\rangle\!\rangle, (TE[t' \mapsto TE_0(t_2)], VE), e').$$

From (c), (d), (e), (g), (i), and the **cons-s** rule, we can conclude that $\Phi_0;\Psi_0 \vdash S_0 \circ \langle\!\langle (TE_0, VE_0), \lambda x{:}t.e\rangle\!\rangle : \sigma' \to \sigma$. Now (h) must hold via the **all-E** rule, and thus there exists σ_1 and σ_2 such that $\Delta_0;\Gamma_0 \vdash x_1 : \forall t'.\sigma_1$,$\Delta_0 \vdash t_2 \Downarrow \sigma_2$, and $\sigma' = \{\sigma_2/t'\}\sigma_1$. From this, we can conclude that $\Phi_0;\Psi_0 \vdash \langle\!\langle (TE, VE), \Lambda t'.\, e'\rangle\!\rangle : \forall t'.\sigma_1$. Now this can only hold through the **vh-all** rule, and thus there exists Δ and Γ such that $\Phi_0 \vdash TE \Downarrow \Delta$, $\Psi_0;\Delta \vdash VE : \Gamma$, and $\Delta[t'];\Gamma \vdash e' : \sigma_1$. Since $\Delta_0 \vdash t_2 \Downarrow \sigma_2$, we can conclude from the **cons-te** rule that $TE[t' \mapsto TE_0(t_2)] \Downarrow \Delta[t' = \sigma_2]$. Now, by Lemma 3.9, $\Delta[t' = \sigma_2];\Gamma \vdash e' : \{\sigma_2/t'\}\sigma_1 = \sigma'$. □

Lemma 3.12 (Progress) *If $\vdash P : \sigma$, then either P is an answer or else there exists a P' such that $P \overset{\mathbf{R}}{\longmapsto} P'$.*

Proof: Let $P = (H_0, S_0, E_0, e_0)$, where $H_0 = (TH_0, VH_0)$ and $E_0 = (TE_0, VE_0)$. Since $\vdash P : \sigma$ must be derived from the **prog** rule, there exists Φ_0, Ψ_0, Δ_0, Γ_0, and σ_0 such that:

(a) $\vdash TH_0 \Downarrow \Phi_0$,
(b) $\Phi_0 \vdash VH_0 : \Psi_0$,
(c) $\Phi_0; \Psi_0 \vdash S_0 : \sigma_0 \rightarrow \sigma$,
(d) $\Phi_0 \vdash TE_0 : \Delta_0$,
(e) $\Psi_0; \Delta_0 \vdash VE_0 : \Gamma_0$, and
(f) $\Delta_0; \Gamma_0 \vdash e_0 : \sigma_0$.

The proof proceeds by cases on e.

case: $e_0 = x'$. If $S_0 = []_{\sigma'}$, then P is an answer. Suppose $S_0 = S \circ \langle\langle E, \lambda x{:}t.e \rangle\rangle$. Since (f) holds, it must be the case that $x' \in Dom(\Gamma_0)$. By (e), this implies that $x' \in Dom(VE_0)$. Consequently, $P \longmapsto (H_0, S, E[x{:}t \mapsto VE_0(x')], e)$ by the **return** rule.

case: $e_0 =$ `let type` $t = \nu$ `in` e. Thus (f) must hold via the **let-type** rule and we can conclude that, since $\Delta_0 \vdash \nu \Downarrow \sigma'$ for some σ', $FTV(\nu) \subseteq Dom(\Delta_0)$. Since $\Phi_0 \vdash TE_0 : \Delta_0$, we know that $Abstr(\Delta_0) = \emptyset$. Thus, $FTV(\nu) \subseteq Dom(TE_0)$. Consequently, $\hat{TE}_0(\nu)$ is defined. Thus $P \longmapsto (H_0[p \mapsto \hat{TE}_0(\nu)], S_0, E_0[t \mapsto p], e)$ by the **talloc** rule.

For the remainder of the cases, e_0 must be of the form `let` $x{:}t = b$ `in` e for some x, t, b, and e. Therefore, (f) must hold via the **let-exp** rule, and hence there exists some σ' such that:

(g) $\Delta_0 \vdash t \Downarrow \sigma'$,
(h) $\Delta_0; \Gamma_0 \vdash b : \sigma'$, and
(i) $\Delta_0; \Gamma_0[x{:}\sigma'] \vdash e : \sigma_0$.

The proof proceeds by cases on b.

case: $b = a$. Since (h) holds, $FV(a) \subseteq Dom(\Gamma_0)$. Since $\Psi_0; \Delta_0 \vdash VE_0 : \Gamma_0$, we know that $FV(a) \subseteq Dom(VE_0)$. Consequently, $\hat{E}_0(a)$ is defined. Thus $P \longmapsto (H_0[l \mapsto \hat{E}_0(a)], S_0, E_0[x{:}t \mapsto l], e)$ by the **valloc** rule.

case: $b =$ `case`$(x', e_0', \lambda x_1'{:}t_1'.e_1')$. Now (h) must hold through the **nat-E** rule. Thus, $\Delta_0; \Gamma_0 \vdash x' :$ `nat`. Since (h) holds, $FV(b) \subseteq Dom(\Gamma_0)$ and thus $x' \in Dom(\Gamma_0)$. But then (e) implies $x' \in Dom(VE_0)$. In turn, this implies that $\Psi_0(VE_0(x')) =$ `nat`. Since (b) holds, this implies that $\Phi_0; \Psi_0 \vdash VH_0(VE_0(x')) :$ `nat`. By the Canonical Forms lemma, $VH_0(VE_0(x'))$ is either `0` or `succ` l for some l. Thus, $P \longmapsto (H_0, S_0 \circ \langle\langle E_0, \lambda x{:}t.e \rangle\rangle, E', e')$ where either $E' = E_0$ and $e' = e_0'$, or else $E' = E_0[x_1'{:}t_1' \mapsto l]$ and $e' = e_1'$, by either the **c-zero** or **c-succ** rule respectively.

case: $b = \pi_i\, x'$. Now (h) must hold via the **prod-E** rule. Thus, $\Delta_0; \Gamma_0 \vdash x' : \sigma_1 \times \sigma_2$ for some σ_1 and σ_2 such that $\sigma_i = \sigma'$. Since (e) holds, $x' \in Dom(VE_0)$. Therefore, $VE_0(x') \in Dom(\Psi_0)$ and thus $VE_0(x') \in Dom(VH_0)$. Since (b) holds, $\Phi_0; \Psi_0 \vdash VH_0(VE_0(x')) : \sigma_1 \times \sigma_2$. By the Canonical Forms lemma, $VH_0(VE_0(x'))$

must be of the form $\langle l_1, l_2 \rangle$ for some l_1 and l_2. Thus, $P \longmapsto (H_0, S_0, E_0[x{:}t \mapsto l_i], e)$ by the **proj** rule.

case: $b = x_1\, x_2$. Now (h) must hold via the **arrow-E** rule. Thus, $\Delta_0; \Gamma_0 \vdash x_1 : \sigma_1 \to \sigma'$ and $\Delta_0; \Gamma_0 \vdash x_2 : \sigma_1$ for some σ_1. Thus, x_1 and x_2 are in $Dom(\Gamma_0)$ and since (e) holds, x_1 and x_2 are in $Dom(VE_0)$. Furthermore, $VE_0(x_1) \in Dom(\Psi_0)$ and since (b) holds, $VE_0(x_1) \in Dom(VH_0)$ and $\Phi_0; \Psi_0 \vdash VH_0(VE_0(x_1)) : \sigma_1 \to \sigma'$. By the Canonical Forms lemma, $VH_0(VE_0(x_1))$ must be of the form $\langle\!\langle E, \mathtt{fix}\ x_1'{:}t_1'(x_2'{:}t_2').e' \rangle\!\rangle$ for some E, x_1', t_1', x_2', t_2', and e'. Thus, $P \longmapsto (H_0, S_0 \circ \langle\!\langle E_0, \lambda x{:}t.e \rangle\!\rangle, E[x_1'{:}t_1' \mapsto VE_0(x_1), x_2'{:}t_2' \mapsto VE_0(x_2)], e')$ by the **app** rule.

case: $b = x_1\,[t_1]$. Now (h) must hold via the **all-E** rule. Thus $\Delta_0; \Gamma_0 \vdash x_1 : \forall t'.\sigma_1$ and $\Delta_0 \vdash t_1 \Downarrow \sigma_2$ such that $\{\sigma_2/t'\}\sigma_1 = \sigma'$. Since (d) holds, $Abstr(\Delta_0) = \emptyset$. Thus, $TE_0(t_1)$ is defined. Furthermore, since (e) holds, $x_1 \in Dom(\Gamma_0)$ and thus $x_1 \in Dom(VE_0)$. Furthermore, $VE_0(x_1) \in Dom(\Psi_0)$ and by (b), $VE_0(x_1) \in Dom(VH_0)$. Consequently, $\Phi_0; \Psi_0 \vdash VH_0(VE_0(x_1)) : \forall t'.\sigma_1$. By the Canonical Forms lemma, $VH_0(VE_0(x_1))$ must be of the form $\langle\!\langle E, \Lambda t'.\,e' \rangle\!\rangle$ for some E, t', and e'. Thus $P \longmapsto (H_0, S_0 \circ \lambda x{:}t.e, E[t' \mapsto TE_0(t_1)], e')$ by the **tapp** rule.

□

Lemma 5.6 (GC State Preservation) *If X is well-formed with respect to P and $X \Longrightarrow X'$, then X' is well-formed with respect to P.*

Proof: Let $P = (H, S, E, e)$, $H = (TH, VH)$, $\vdash TH \Downarrow \Phi$, $\Phi \vdash VH : \Psi$, and let $X = (H_f, Q, L, H_t)$ be well-formed with respect to P and $X \Longrightarrow X'$. Since X is well-formed, we know that:

(a) $H_f \uplus H_t = H$,
(b) $FP(H_t, S, E, e) \subseteq Q$,
(c) $FL(H_t, S, E, e) \subseteq Dom(L)$,
(d) for all $l{:}p \in L$, $\Phi(p) = \Psi(l)$,
(e) $P \subseteq Dom(TH)$, and
(f) for all $l{:}p$ in L, $l \in Dom(VH)$ and $p \in Dom(TH)$.

The argument continues by cases on the rule taking X to X'. Throughout, it is clear that any free pointers or locations entered into the scan-set must be bound in the original heap.

case gc-1: $H_f = H_f'[p \mapsto \mu]$, $Q = Q' \uplus \{p\}$, and $X' = (H_f', Q' \cup FP(\mu), L, H_t[p \mapsto \mu])$. By (a), we know that $H_f' \uplus H_t[p \mapsto \mu] = H$. Suppose $H_t = (TH_t, VH_t)$. Then from (b), we know that $FP(H_t) \subseteq Q' \uplus \{p\}$. Therefore,

$$\begin{aligned}
FP(H_t[p \mapsto \mu]) &= (FP(H_t) \setminus \{p\}) \cup (FP(\mu) \setminus (Dom(TH_t) \cup \{p\})) \\
&\subseteq ((Q' \uplus \{p\}) \setminus \{p\}) \cup (FP(\mu) \setminus (Dom(TH_t) \cup \{p\})) \\
&= Q' \cup (FP(\mu) \setminus (Dom(TH_t) \cup \{p\})) \\
&\subseteq Q' \cup FP(\mu).
\end{aligned}$$

Thus, $FP(H_t[p \mapsto \mu], S, E, e) \subseteq Q'$. Hence, from (c) and (d) we can conclude that X' is well-formed with respect to P.

case gc-2: $Q = Q' \uplus \{p\}$, $p \notin Dom(H_f)$, and $X' = (H_f, Q', L, H_t)$. By (a) and the fact that H is well-formed, we can conclude that p must be bound in either H_f or H_t. Since p is not bound in H_f, it must be bound in H_t. Thus, $p \notin FP(H_t, S, E, e)$ and from (b) we can conclude that $FP(H_t, S, E, e) \subseteq Q'$. Hence from (d) and (e), we can conclude that X' is well-formed with respect to P.

case gc-3: $H_f = H'_f[l \mapsto h]$, $L = L' \uplus \{l{:}p\}$, and $X' = (H_f, Q \cup Q_1, L \cup L_2, H_t[l \mapsto h])$ where $\mathcal{F}[(H_f \cup H_t)(p)](p, h) = (Q_1, L_1)$. Since X is well-formed, we know that $\Phi(p) = \Psi(l)$ and thus $FP(h) \subseteq Q_1$ and $FL(h) \subseteq Dom(L_1)$. Thus, from (b), (c), (d) and Lemma 5.3, we know that $FP(H_t[l \mapsto h], S, E, e) \subseteq Q \cup Q_1$ and $FL(H_t[l \mapsto h], S, E, e) \subseteq Dom(L \cup L_1)$, and for all $l'{:}p' \in L \cup L_1$, $\Phi(p') = \Psi(l')$. Therefore, X' is well-formed with respect to P.

case gc-4: $L = L' \uplus \{l{:}p\}$, $l \notin Dom(H_f)$, and $X' = (H_f, Q, L', H_t)$. By (a) and the fact that H is well-formed, we can conclude that l must be bound in either H_f or H_t. Since l is not bound in H_f, it must be bound in H_t. Thus, $l \notin FL(H_t, S, E, e)$ and from (c) we can conclude that $FL(H_t, S, E, e) \subseteq L'$. Hence from (b) and (e), we can conclude that X' is well-formed with respect to P. □

Lemma 5.7 (GC State Progress) *If $X = (H_f, Q, L, H_t)$ is well-formed with respect to P, then either Q and L are empty or else there exists a GC state X' such that $X \Longrightarrow X'$.*

Proof: Let $X = (H_f, Q, L, H_t)$ be a GC state such that Q or L is non-empty and assume X is well-formed with respect to the program $P = (H, S, E, e)$. Thus,

(a) $H_f \uplus H_t = H$,
(b) $FP(H_t, S, E, e) \subseteq Q$,
(c) $FL(H_t, S, E, e) \subseteq Dom(L)$,
(d) for all $l{:}p \in L$, $\Phi(p) = \Psi(l)$,
(e) $P \subseteq Dom(TH)$, and
(f) for all $l{:}p$ in L, $l \in Dom(VH)$ and $p \in Dom(TH)$.

If Q is non-empty, then $Q = Q' \uplus \{p\}$ for some Q' and p. By conditions (a) and (e), we know that p is either bound in H_f or H_t. If p is bound in H_t then **gc-2** applies. If p is bound in H_f, then **gc-1** applies.

If L is non-empty, then $L = L' \uplus \{l{:}p\}$ for some L', l, and p. By condition (a) and (f), l must be bound in either H_f or H_t. If l is bound in H_t, then the **gc-4** rule applies. If l is bound in H_f, then there exists an H'_f and h such that $H_f(l) = h$. By condition (a) and (f), p is bound in $H_f \uplus H_t$. By conditions (e) and (d), the well-formedness of P, and Lemma 5.3, we know that $\mathcal{F}[(H_f \uplus H_t)(p)](p, h) = (Q', L')$ for some Q' and L'. Consequently, **gc-3** applies. □