Garbage Collection Based on a Linear Type System

Atsushi Igarashi
Dept. of Graphics and Computer Science
Graduate School of Arts and Sciences
University of Tokyo
igarashi@graco.c.u-tokyo.ac.jp

Naoki Kobayashi
Dept. of Information Science
Graduate School of Science
University of Tokyo
koba@is.s.u-tokyo.ac.jp

Abstract

We propose a type-directed garbage collection (GC) scheme for a programming language with static memory management based on a linear type system. Linear type systems, which can guarantee certain values (called *linear values*) to be used only once during program execution, are useful for memory management: memory space for linear values can be reclaimed immediately after they are used. However, conventional pointer-tracing GC does not work under such a memory management scheme: if the memory space for used linear values is still reachable through pointers, dangling pointers are created.

This problem is solved by exploiting static type information during garbage collection in a way similar to tag-free GC. Type information in our linear type system represents not only the shapes of heap objects but also how many times the heap objects are accessed in the rest of computation. Using such type information at GC-time, our GC can avoid tracing dangling pointers; in addition, our GC can reclaim even reachable garbage. We formalize such a GC algorithm and sketch a proof of its correctness.

1 Introduction

1.1 Memory Management and Linear Type Systems

Automatic memory management, one of the important features of modern high-level programming languages, releases programmers from burdens of correctly inserting explicit declarations of memory deallocation in programs. It is typically realized by garbage collection (GC), which is periodically invoked and finds unused memory spaces by traversing pointers. Although GC is indeed very useful run-time machinery, reclamation is delayed until the invocation of the garbage collector. Moreover, a traditional tracing garbage collector cannot reclaim memory space that is semantically garbage but reachable from the stack or registers.

To reuse memory space more eagerly, several techniques have been proposed based on region inference [TT94, AFL95, BTV96] or linear type systems[1] [GH90, Wad90, CGR92, TWM95, Iga97, Mog97, IK00b, Kob99, WJ99]. We follow the latter approach here. Linear (more precisely, affine linear) type systems can guarantee that certain values (called *linear values*) are used at most once, so we can reclaim memory space for such linear values immediately after they are used. The basic idea of linear type systems is to annotate type constructors with information about how often the memory space for values are accessed. For example, consider the expressions $x+1$ and $x+x$. In a conventional type system, both expressions are given type *Int* under the assumption that x is given type *Int*, written as follows:

$$x\colon Int \vdash x+1 : Int$$
$$x\colon Int \vdash x+x : Int$$

In a linear type system, each type *Int* is annotated with a *use*. It is either 0, 1, or ω and denotes how often a value is used for the primitive operation such as $+$ during the execution. For example, an expression of

[1]Strictly speaking, some of these type systems, including the one presented here, are classified as variants of affine linear type systems. Throughout this paper, we lump together type systems that can take into account how often values are used and refer to them as linear type systems.

the type Int^1 can be used at most once as an integer, that of the type Int^ω can be used an arbitrary number of times, and that of the type Int^0 cannot be used. Then, the type judgments below are both valid:

$$x\colon Int^1 \vdash x + 1 : Int^\omega$$
$$x\colon Int^\omega \vdash x + x : Int^\omega$$

while the type judgment

$$x\colon Int^1 \vdash x + x : Int^\omega$$

is not because x is used twice in $x + x$. By using such a type system, linear values can be statically found and the compiler can statically insert deallocation code at certain primitive operations. Such deallocation, however, is performed independently of reachability of the memory space from the run-time stack or registers. Thus, it may create *dangling pointers* in the memory space, making conventional tracing GC fail.

1.2 Our Approach

In this paper we propose a GC scheme that can coexist with static memory management based on a linear type system. The basic idea is to exploit static type information during GC in a way similar to tag-free GC [App89,Tol94,MFH95,Mor95,MH96]. It may help to review the idea of tag-free GC (for languages with a monomorphic type system) first. The intuition behind tag-free GC was that "types represent the shapes of values." When we begin to trace a variable in the environment, its static type tells how to trace it: if x is given type $(Int \times Int) \times Int$, then we know the memory space for x stores (a pointer to) an integer pair and an integer. Thus, we can perform GC without run-time tags by recording the static type information on the free variables of function closures and of the program point where GC may happen.

In our GC, the above intuition is extended as follows: "types represent not only the shapes of values but also how often they are used in a certain context." For example, if x is given type $Real^0$, then we know not only that x points to a real number but also it is no longer used. Since variables corresponding to dangling pointers are always given the use 0, the garbage collector can avoid tracing dangling pointers by ignoring variables with the use 0. In fact, it does not matter whether it is really a dangling pointer or not: even if there *is* a value at the address, the heap value need not be marked. As a result, our GC can collect semantic garbage reachable from the root set, as some GC schemes based on GC-time type inference can [GG92,Fra94,MFH95,HY98].

As in recent linear type systems [Mog97,Kob99], our type system can express non-uniform patterns of access to a single data structure in several contexts. For example, consider the expression $(\#1\,x) + (\#2\,x)$ (where $\#i$ extracts the i-th element of a pair). It is typed under the assumption that x is given type $Real^1 \times^\omega Real^1$ as follows:

$$\frac{\begin{array}{c} x\colon Real^1 \times^1 Real^0 \vdash \#1\,x : Real^1 \\ x\colon Real^0 \times^1 Real^1 \vdash \#2\,x : Real^1 \end{array}}{x\colon Real^{1+0} \times^{1+1} Real^{0+1} (= Real^1 \times^\omega Real^1) \vdash (\#1\,x) + (\#2\,x) : Real^\omega}$$

Since $\#1\,x$ does not use the second element of x, the use of the second element type is 0; similarly for $\#2\,x$. As a whole, x is given type $Real^1 \times^\omega Real^1$, which is obtained by adding each use. The obtained type means that a pair of the type can be accessed arbitrarily many times but each element can be used at most once *in total*. Thus, the memory space for the first element can be reclaimed immediately after the execution of $+$. Notice that element types of x are given different uses in each occurrence, expressing the context's own access pattern.

However, this non-uniformity introduces another subtlety not observed in conventional GC: our garbage collector may need to visit one heap value more than once. Consider the following Standard ML program:

```
let val p = (1.5, 2.2)
    fun f x = x + (#1 p)
    fun g x = x + (#2 p)
in f 3.1 + g 4.3 end
```

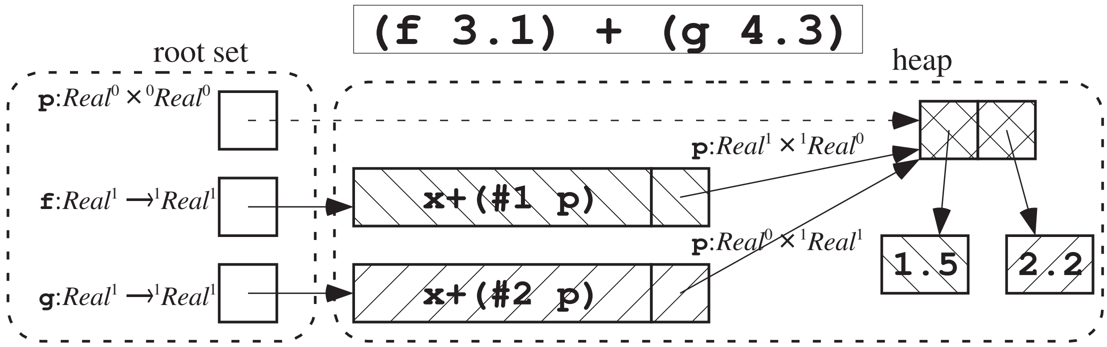

Figure 1: The real numbers `1.5` and `2.2` are marked by the traverses from `f` and `g`, respectively. The pair is marked twice by both traverses.

and suppose the garbage collector is invoked just before the execution of `f 3.1 + g 4.3`. As in the previous example, the variable `p` is given type $Real^1 \times^1 Real^0$ in `f` while the same variable is given type $Real^0 \times^1 Real^1$ in `g`. Thus, the traverse from the closure of `f` marks only the first element of `p`; the second element is marked when traverse from the closure of `g` happens, marking the pair for the second time. (See Figure 1.)

This multiple traverses on one value may cause a lot of verbose marking, or even nontermination in the presence of cycles in the heap space. To prevent it, we extend the mechanism of mark bits, used by conventional GC; our garbage collector keeps track of the types of the marked objects to remember which part of an object is already marked. If the garbage collector reaches a marked object again, it compares the type of heap object derived from the current scan set with the one derived from the marked objects. The garbage collector does not go further if the access pattern expressed by the latter type subsumes that by the former because it means that the heap objects that the current traverse tries to mark have been already marked (or scheduled to be marked) in the previous traverses. (Earlier linear type systems [TWM95,Iga97] cannot express the non-uniformity mentioned above; thus, the usual mark bits would work. However, static memory management would be less effective since such a type system cannot ensure that, for example, the elements in `p` above are linear. See Section 5 for discussion.)

1.3 Our Contribution

The contributions of the present work are formalization of our GC algorithm for a language with a monomorphic linear type system and a proof of correctness of the algorithm. Our formalization also includes operational semantics that takes account of immediate reclamation of memory space for linear values; as in the previous work [MFH95, Mor95, MH96] on formalization of memory management, the operational semantics of our language makes run-time mechanisms such as stacks or heap space explicit. We also prove soundness of our linear type system with respect to the operational semantics. Here, only one particular instance of monomorphic linear type systems is dealt with, but our technique would be applicable to a language based on another variant of linear type system, even to an extension with the ML-style polymorphism (and use polymorphism [Iga97,WJ99]). We could use techniques similar to the existing tag-free GC schemes [Tol94,MH96], which exploit run-time type passing.

1.4 Structure of the Paper

The rest of the paper is organized as follows: Section 2 introduces our target language and its operational semantics. Then, the type system is presented in Section 3; Section 4 presents the GC algorithm formally and states its correctness. After discussing related work in Section 5, we conclude this paper in Section 6 with discussion on future work in Section 7. For brevity, proofs are only sketched; interested readers are referred to the companion technical report [IK00a].

2 Language $\lambda^{\kappa}_{\mathrm{gc}}$

In this section, we define a language called $\lambda^{\kappa}_{\mathrm{gc}}$ and give its operational semantics. The language $\lambda^{\kappa}_{\mathrm{gc}}$ is based on a call-by-value lambda-calculus equipped with integers, pairs, and recursive functions. We use a variant of A-normal form [FSDF93] so that the evaluation order is made explicit and all temporal results are bound to variables. In addition, each heap-allocated value is associated with a *use*, which denotes how often a value is used. Expressions and functions are annotated with type information on their free variables; they are required by the garbage collector, as mentioned in the previous section. Our operational semantics is given in a way similar to preceding work on abstract models of GC [MFH95, Mor95, MH96]: the heap, stack, and register file are made explicit in the reduction relation. Moreover, deallocation of memory space for linear values is also explicit.

2.1 Syntax of Types

Before giving the syntax of expressions, we begin with the syntax of uses, types, and type environments.

2.1.1 Definition [uses, types]: The set of *uses*, ranged over by the metavariable κ, and the set of *types*, ranged over by the metavariable τ, are given by the following syntax.

$$\begin{array}{lcll} \kappa & ::= & 0 \mid 1 \mid \omega & (\textit{uses}) \\ \tau & ::= & \mathit{Int} \mid \tau_1 \rightarrow^{\kappa} \tau_2 \mid \tau_1 \times^{\kappa} \tau_2 & (\textit{types}) \end{array}$$

The type Int denotes integers, the type $\tau_1 \rightarrow^{\kappa} \tau_2$ functions from τ_1 to τ_2, and $\tau_1 \times^{\kappa} \tau_2$ pairs of values of the types τ_1 and τ_2. Uses in a type denote how a value of the type can be used: a use 0 means that a value is never used, 1 means that a value can be used at most once, and ω means that a value can be used an arbitrary number of times. For example, $\mathit{Int} \times^1 \mathit{Int}$ denotes a type of pairs, from which we can extract integers at most once. Uses are not attached to Int because integers are not "boxed" in the operational semantics defined below and we are not concerned with uses of unboxed values.

Type environments defined below are not only used in the type system as usual but also attached to expressions to represent type information on the program point at which GC may be invoked. Our garbage collector computes the types of the memory locations in the root set from such type environments; they are also stored in function closures, making it possible to compute the types of the free variables in a closure.

2.1.2 Definition [type environments]: The metavariables x, y, z, and w range over a countably infinite set $\mathcal{V}$ of *variables*. A *type environment* Γ is a mapping from a finite set of variables to the set of types.

2.1.3 Notation: We write $\mathit{dom}(\Gamma)$ for the domain of Γ and $x_1\colon\tau_1, \ldots, x_n\colon\tau_n$, abbreviated to $\tilde{x}\colon\tilde{\tau}$, for the type environment Γ such that $\mathit{dom}(\Gamma) = \{x_1, \ldots, x_n\}$ and $\Gamma(x_i) = \tau_i$ for each $i \in \{1, \ldots, n\}$. The empty type environment is written $\emptyset$. When $x \notin \mathit{dom}(\Gamma)$, we write $\Gamma, x\colon\tau$ for the type environment Γ' such that $\mathit{dom}(\Gamma') = \mathit{dom}(\Gamma) \cup \{x\}$, $\Gamma'(x) = \tau$ and $\Gamma'(y) = \Gamma(y)$ if $x \neq y$. The type environment $\Gamma \setminus \{x_1, \ldots, x_n\}$ denotes the restriction of Γ to the domain $\mathit{dom}(\Gamma) \setminus \{x_1, \ldots, x_n\}$.

2.2 Syntax of Expressions

The metavariable n ranges over the set of integers; the metavariable v ranges over the set of non-heap values; the metavariable e ranges over the set of expressions. The syntax of expressions is given by the following syntax:

$$\begin{array}{lcl} v & ::= & x \mid n \\ e & ::= & {}^{\langle\Gamma\rangle}x \mid {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = v\ \mathbf{in}\ e \mid {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = ({}^{\langle\Gamma_0\rangle}\mathbf{fun}\, y(z) = e_0)^{\kappa}\ \mathbf{in}\ e \\ & \mid & {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = (y_1, y_2)^{\kappa}\ \mathbf{in}\ e \mid {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y\ z\ \mathbf{in}\ e \mid {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y + z\ \mathbf{in}\ e \\ & \mid & {}^{\langle\Gamma\rangle}\,\mathbf{let}\ (x, y) = z\ \mathbf{in}\ e \mid {}^{\langle\Gamma\rangle}\,\mathbf{if0}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \end{array}$$

The form $\mathbf{fun}\, y(z) = e$ is a recursive function taking z as an argument; y refers to the function itself. An expression e is return from a function call, declaration of a value (i.e., a non-heap value, a function, or a pair), function application, addition of two integers, extraction from a pair[2], or a conditional branch. Type environments are attached to functions and each step of an expression, recording types of variables in them. We call them *type environment annotations* and often use $TE(e)$ to denote that of e. When they are not important, we often omit to write them explicitly.

The bound variables of expressions are defined in a customary fashion, i.e., (1) the variable x is bound in e of **let** $x = \cdots$ **in** e (2) the variables x and y are bound in e of **let** $(x, y) = z$ **in** e and $\mathbf{fun}\, x(y) = e$. A variable that is not bound will be called a free variable. We define substitutions of variables and α-conversion in a customary fashion and assume that the bound variables in an expression are pairwise distinct by α-conversion.

2.2.1 Remark: Note that programmers need not write explicit type and use annotations: $\lambda^{\kappa}_{\mathrm{gc}}$ is considered an intermediate form after type reconstruction. For type reconstruction, we can use techniques developed elsewhere [Iga97, IK00b, Kob99, Mog97]; some of them can handle pair types that can express non-uniform access patterns.

2.3 Operational Semantics

In our semantics, we make run-time mechanisms of program execution, such as stack frames, explicit. To represent such run-time citizens, we introduce *environments* representing register files, *heaps* representing memory space, and *stacks* representing run-time stacks. A state of program execution is represented as a quadruple consisting of a heap, a stack, an environment and an expression; execution is represented as rewriting of states.

2.3.1 Definition [environments]: An *environment* is a mapping from a finite set of variables to the set of non-heap values.

We use the metavariable V for environments. We write $dom(V)$ for the domain of V and $\{x_1 = v_1, \ldots, x_n = v_n\}$ for the environment V such that $dom(V) = \{x_1, \ldots, x_n\}$ and $V(x_i) = v_i$ for each $i \in \{1, \ldots, n\}$. When $dom(V_1) \cap dom(V_2) = \emptyset$, we write $V_1 \uplus V_2$ for the environment V such that $dom(V) = dom(V_1) \cup dom(V_2)$, $V(x) = V_i(x)$ for $x \in dom(V_i)$.

2.3.2 Definition [heap values, heaps]: The set of *heap values*, ranged over by the metavariable h, is given by the following syntax:

$$h \quad ::= \quad (v_1, v_2) \mid (V, {}^{\langle\Gamma\rangle}\mathbf{fun}\, x(y) = e)$$

A *heap* is a mapping from a finite set of variables to the set of pairs, written h^{κ}, of a heap value h and a non-zero use κ (i.e., either 1 or ω).

We use the metavariable H for a heap. The notations $dom(H)$, $\{x_1 = h_1^{\kappa_1}, \ldots, x_n = h_n^{\kappa_n}\}$, and $H_1 \uplus H_2$ are defined similarly to the corresponding notations for environments.

2.3.3 Definition [stacks]: A *stack* is a sequence whose elements have the form $[V, \Gamma, \lambda x.e]$, called a *stack frame.*

We use the metavariable S for a stack; we write $[\,]$ for the empty stack and $S[V, \Gamma, \lambda x.e]$ for a stack whose first element is $[V, \Gamma, \lambda x.e]$ and the remainder is S.

2.3.4 Definition [programs, answers]: A *program* P is defined as a quadruple (H, S, V, e) of a heap, a stack, an environment and an expression. In particular, a program $(H, [\,], V, {}^{\langle\Gamma\rangle}x)$ consisting of the empty stack and a return expression is called an *answer* program.

[2]We use this form of simultaneous extraction rather than ordinary projection operations: if we could not extract two components simultaneously, a pair would be considered non-linear as its two elements are used.

Then, we define rewriting rules for $\lambda^{\kappa}_{\mathrm{gc}}$ programs. We use the following auxiliary notations $H \oplus \{x = h^{\kappa}\}$ and H^{-x} in the definition:

$$
\begin{array}{lcl}
H \oplus \{x = h^{\kappa}\} & = & \left\{ \begin{array}{ll} H & (\text{if } \kappa = 0 \text{ and } x \notin \mathit{dom}(H)) \\ H \uplus \{x = h^{\kappa}\} & (\text{if } \kappa \neq 0 \text{ and } x \notin \mathit{dom}(H)) \\ \text{undefined} & (\text{otherwise}) \end{array} \right. \\
(H \uplus \{x = h^{1}\})^{-x} & = & H \\
(H \uplus \{x = h^{\omega}\})^{-x} & = & H \uplus \{x = h^{\omega}\}
\end{array}
$$

2.3.5 Definition: The relation $P \longmapsto P'$ is the least relation closed under the following rules:

$$(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y\ \mathbf{in}\ e) \longmapsto (H, S, V \uplus \{x = V(y)\}, e) \qquad \text{(R-VAR)}$$

$$(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = n\ \mathbf{in}\ e) \longmapsto (H, S, V \uplus \{x = n\}, e) \qquad \text{(R-INT)}$$

$$\frac{z \text{ fresh}}{(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = (y_1, y_2)^{\kappa}\ \mathbf{in}\ e') \longmapsto (H \oplus \{z = (V(y_1), V(y_2))^{\kappa}\}, S, V \uplus \{x = z\}, e')} \qquad \text{(R-PAIR)}$$

$$\frac{z \text{ fresh}}{\begin{array}{c}(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = ({}^{\langle\Gamma_0\rangle}\mathbf{fun}\, y(w) = e_0)^{\kappa}\ \mathbf{in}\ e') \\ \longmapsto (H \oplus \{z = (V, {}^{\langle\Gamma_0\rangle}\mathbf{fun}\, y(w) = e_0)^{\kappa}\}, S, V \uplus \{x = z\}, e')\end{array}} \qquad \text{(R-FUN)}$$

$$\frac{H(V(z)) = (v_1, v_2)^{\kappa}}{(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ (x, y) = z\ \mathbf{in}\ e) \longmapsto (H^{-V(z)}, S, V \uplus \{x = v_1, y = v_2\}, e)} \qquad \text{(R-EXT)}$$

$$\frac{n_1 = V(y_1) \qquad n_2 = V(y_2)}{(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y_1 + y_2\ \mathbf{in}\ e) \longmapsto (H, S, V \uplus \{x = \underline{n_1 + n_2}\}, e)} \qquad \text{(R-PLUS)}$$

$$\frac{H(V(y_1)) = (V_0, {}^{\langle\Gamma_0\rangle}\mathbf{fun}\, z(w) = e_0)^{\kappa}}{\begin{array}{c}(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y_1\ y_2\ \mathbf{in}\ e) \\ \longmapsto (H^{-V(y_1)}, S[V, \mathit{TE}(e) \setminus \{x\}, \lambda x.e], V_0 \uplus \{z = V(y_1), w = V(y_2)\}, e_0)\end{array}} \qquad \text{(R-APP)}$$

$$\frac{V(x) = 0}{(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{if0}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \longmapsto (H, S, V, e_1)} \qquad \text{(R-IFT)}$$

$$\frac{V(x) \neq 0}{(H, S, V, {}^{\langle\Gamma\rangle}\,\mathbf{if0}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) \longmapsto (H, S, V, e_2)} \qquad \text{(R-IFF)}$$

$$(H, S[V_0, \Gamma_0, \lambda x.e_0], {}^{\langle\Gamma\rangle}y) \longmapsto (H, S, V_0 \uplus \{x = V(y)\}, e_0) \qquad \text{(R-RET)}$$

We write $\longmapsto^*$ for the reflexive and transitive closure of $\longmapsto$.

The rules are fairly straightforward. In the rules R-PAIR and R-FUN, the heap value is allocated in the heap at a fresh address z; the execution continues after assigning z to x in the environment. When the program uses a heap value (R-EXT and R-APP), it is checked whether the value is linear; if so, the memory space will be reclaimed (using H^{-x}). In the rule R-PLUS, the notation $\underline{n_1 + n_2}$ denotes the summation of the two integers n_1 and n_2 (not the syntactic expression "$n_1 + n_2$"). The rules R-APP and R-RET are about function calls, involving manipulation of the stack. When a function is called (R-APP), the continuation $\lambda x.e$ after the call is pushed onto the stack together with its environment V and type environment $\mathit{TE}(e) \setminus \{x\}$; then, the execution continues with the function body e_0 under the closure's environment V_0 augmented with bindings of the actual argument $V(y_2)$ and the address $V(y_1)$ of the function itself. When execution reaches the end of a function (R-RET), the top of the stack is popped and the continuation is applied to the result $V(y)$.

2.3.6 Remark: Notice that type environment annotations do not play a significant role during execution. They are just stored in closures and stack frames, and will not be used unless GC occurs. Thus, it is not necessary, in practice, to attach them at every step; they are required only at (1) every function definition, (2) the program point after every function call, and (3) program points at which GC may happen.

2.3.7 Remark: As we see in the typing rules, a non-linear value can be passed where a linear value is expected. Thus, to reclaim the memory space for linear values, we have to perform a dynamic check, requiring uses in heaps. The cost of the check can be, however, fairly cheap (without requiring extra memory space for use tags), as discussed elsewhere [Kob99].

3 Type System

In this section, we give a type system for $\lambda^{\kappa}_{\mathrm{gc}}$. Our type system can ensure the lack of not only illegal operations (such as application of a non-function value) but also illegal access to already deallocated heap space. We begin with several operations on uses, types, and type environments, used in the typing rules.

3.1 Notational Preliminaries

The relation $\tau_1 \geq \tau_2$ below means that an expression of type τ_1 can be more frequently used than that of τ_2; we allow an expression of type τ_1 to be coerced to that of type τ_2. Similarly, $\Gamma_1 \geq \Gamma_2$ means that the type bound in Γ_2 is less than that in Γ_1 for each variable in $\mathit{dom}(\Gamma_2)$.

3.1.1 Definition: The binary relation $\geq$ between uses is the total order defined by $\omega \geq 1 \geq 0$. The binary relation $\tau_1 \geq \tau_2$ is defined by:

$$\begin{array}{rcll} \mathit{Int} & \geq & \mathit{Int} & \\ \tau_1 \rightarrow^{\kappa_1} \tau_2 & \geq & \tau_1 \rightarrow^{\kappa_2} \tau_2 & \text{if } \kappa_1 \geq \kappa_2 \\ \tau_{11} \times^{\kappa_1} \tau_{12} & \geq & \tau_{21} \times^{\kappa_2} \tau_{22} & \text{if } \kappa_1 \geq \kappa_2 \text{ and } \tau_{1i} \geq \tau_{2i} \text{ for } i = 1, 2 \end{array}$$

We also write $\Gamma_1 \geq \Gamma_2$ if $\mathit{dom}(\Gamma_1) \supseteq \mathit{dom}(\Gamma_2)$ and $\Gamma_1(x) \geq \Gamma_2(x)$ for all $x \in \mathit{dom}(\Gamma_2)$.

The relation $\tau_1 \geq \tau_2$ would correspond to a subtyping relation $\tau_1 \preceq \tau_2$, meaning τ_1 is subtype of τ_2. Note that $\geq$ is the inverse of the usual subtyping relation.

3.1.2 Remark: We could adopt usual structural subtyping for function types, without which the use analysis gets rather coarser. It is not adopted simply to make the presentation simpler (while pair types require a structural rule, which is crucial to express non-uniform access patterns, mentioned in Section 1). In fact, introduction of the structural subtyping for function types would not affect our GC algorithm itself (even though it would complicate a proof of correctness of the GC): our GC algorithm could work as long as attached uses are correct with respect to the operational semantics.

3.1.3 Example: $(x\colon (\mathit{Int} \times^{\omega} \mathit{Int}) \times^{1} \mathit{Int}, y\colon \mathit{Int} \rightarrow^{\omega} \mathit{Int}, z\colon \mathit{Int}) \geq (x\colon (\mathit{Int} \times^{0} \mathit{Int}) \times^{1} \mathit{Int}, y\colon \mathit{Int} \rightarrow^{1} \mathit{Int})$.

The summation of two types, defined below, is used to compute the total use of a variable. Suppose a variable is given type τ_1 in e_1 and τ_2 in e_2; if both expressions may be executed, then the total usage of the variable is represented by $\tau_1 + \tau_2$.

3.1.4 Definition: The *summation* of two uses, written $\kappa_1 + \kappa_2$, is the commutative and associative operation that satisfies $0 + 0 = 0$, $1 + 0 = 1$, and $1 + 1 = \omega + 0 = \omega + 1 = \omega + \omega = \omega$. The summation of two types, written $\tau_1 + \tau_2$, is defined as follows:

$$\begin{array}{lcl} \mathit{Int} + \mathit{Int} & = & \mathit{Int} \\ (\tau_1 \rightarrow^{\kappa_1} \tau_2) + (\tau_1 \rightarrow^{\kappa_2} \tau_2) & = & \tau_1 \rightarrow^{\kappa_1 + \kappa_2} \tau_2 \\ (\tau_{11} \times^{\kappa_1} \tau_{12}) + (\tau_{21} \times^{\kappa_2} \tau_{22}) & = & (\tau_{11} + \tau_{21}) \times^{(\kappa_1 + \kappa_2)} (\tau_{12} + \tau_{22}) \end{array}$$

The operation '+' on types are pointwise extended to type environments: the summation $\Gamma_1 + \Gamma_2$ of two type environments is defined by:

$$
\begin{array}{l}
dom(\Gamma_1 + \Gamma_2) = dom(\Gamma_1) \cup dom(\Gamma_2) \\
(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) + \Gamma_2(x) & \text{if } x \in dom(\Gamma_1) \cap dom(\Gamma_2) \\ \Gamma_1(x) & \text{if } x \in dom(\Gamma_1) \setminus dom(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \in dom(\Gamma_2) \setminus dom(\Gamma_1) \end{cases}
\end{array}
$$

As mentioned in Section 1, the summation of two pair types are obtained by adding uses pointwise because a use inside a pair type constructor denotes how may times an element in the pair is used *in total.* Thus, for example, the summation of two types $(Int \times^0 Int) \times^1 Int$ and $(Int \times^1 Int) \times^1 Int$ is defined to be $(Int \times^1 Int) \times^\omega Int$. The obtained type means that the inner pair can be used at most once in total even if the outer pair can be accessed arbitrarily many times.

3.1.5 Example: $(x\colon Int, y\colon (Int \times^1 Int) \times^1 Int) + (y\colon (Int \times^0 Int) \times^1 Int) = x\colon Int, y\colon (Int \times^1 Int) \times^\omega Int$.

The product of a use κ and a type is defined below as the summation of the type κ times.

3.1.6 Definition: The *product* of two uses, written $\kappa_1 \cdot \kappa_2$, is the commutative and associative operation that satisfies $0 \cdot 0 = 0 \cdot 1 = 0 \cdot \omega = 0$, $1 \cdot 1 = 1$, and $1 \cdot \omega = \omega \cdot \omega = \omega$. The product is extended to an operation on uses and types by:

$$
\begin{array}{lcl}
\kappa \cdot Int & = & Int \\
\kappa \cdot (\tau_1 \rightarrow^{\kappa'} \tau_2) & = & \tau_1 \rightarrow^{\kappa \cdot \kappa'} \tau_2 \\
\kappa \cdot (\tau_1 \times^{\kappa'} \tau_2) & = & (\kappa \cdot \tau_1) \times^{\kappa \cdot \kappa'} (\kappa \cdot \tau_2)
\end{array}
$$

It is further extended to an operation on uses and type environments by:

$$\kappa \cdot (x_1\colon \tau_1, \ldots, x_n\colon \tau_n) = x_1\colon \kappa \cdot \tau_1, \ldots, x_n\colon \kappa \cdot \tau_n$$

3.1.7 Example: $\omega \cdot (Int \times^1 (Int \times^0 Int)) = (Int \times^\omega (Int \times^0 Int))$.

3.1.8 Example: $0 \cdot (x\colon Int \times^\omega Int, y\colon Int) = x\colon Int \times^0 Int, y\colon Int$.

If a heap value refers to another heap value through a variable whose use is 0, then the referred heap value actually need not exist in the heap. To ignore such potential dangling pointers, we discard bindings of types with the use 0 from a type environment, by using the *truncation*, defined below.

3.1.9 Definition: The *truncation* $\lfloor \Gamma \rfloor$ of a type environment Γ is defined by:

$$\lfloor \Gamma \rfloor = \Gamma \setminus \{x \mid \Gamma(x) = \tau_1 \times^0 \tau_2 \text{ or } \tau_1 \rightarrow^0 \tau_2\}$$

3.1.10 Example: $\lfloor x\colon Int, y\colon Int \times^\omega Int, z\colon Int \rightarrow^0 Int \rfloor = x\colon Int, y\colon Int \times^\omega Int$.

3.2 Typing Rules

Typing Rules for Expressions

A type judgment for expressions is of the form $\Gamma \vdash e : \tau$. It means not only that e is well-typed in the ordinary sense, but also that each function and pair declared in e is used according to its use and each free variable is used according to the use of its type in Γ. For example, $\Gamma, x\colon Int \rightarrow^1 Int \vdash e : \tau$ means that e uses x *as a function on integers at most once.*

The formal rules are given in Figure 2. Since type environments contain information on how often variables are accessed, we need to take special care in merging type environments. For example, if $\Gamma_1, y\colon Int \rightarrow^1 Int \vdash y : Int \rightarrow^1 Int$ and $\Gamma_2, x\colon Int \rightarrow^1 Int, y\colon Int \rightarrow^1 Int \vdash e : \tau_2$, then y is used totally twice in **let** $x = y$ **in** e. Therefore, the total use of a variable in **let** $x = y$ **in** e should be obtained by adding the uses in two type environments as in the rule T-Dec. Similarly for the rules T-Pair, T-App, T-Sum, and T-Ext. On the other hand, in a conditional expression **if0** x **then** e_1 **else** e_2, either e_1 or e_2 is executed. Thus, the

Non-heap/heap values:

$$\frac{\Gamma(x) \geq \tau}{\Gamma \vdash x : \tau} \quad \text{(T-Var)} \qquad\qquad \Gamma \vdash n : \mathit{Int} \quad \text{(T-Int)}$$

Expressions:

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash {}^{\langle\Gamma\rangle}x : \tau} \quad \text{(T-Ret)}$$

$$\frac{\Gamma', x{:}\tau_2 \vdash e : \tau \qquad \Gamma = \Gamma' + (y_1{:}\tau_1 \rightarrow^1 \tau_2) + (y_2{:}\tau_2)}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y_1\, y_2\ \mathbf{in}\ e : \tau} \quad \text{(T-App)}$$

$$\frac{\Gamma_1 \vdash v : \tau' \qquad \Gamma_2, x{:}\tau' \vdash e : \tau}{\Gamma_1 + \Gamma_2 \vdash {}^{\langle\Gamma_1+\Gamma_2\rangle}\,\mathbf{let}\ x = v\ \mathbf{in}\ e : \tau} \quad \text{(T-Dec)}$$

$$\frac{\Gamma', x{:}\mathit{Int} \vdash e : \tau \qquad \Gamma = \Gamma' + y_1{:}\mathit{Int} + y_2{:}\mathit{Int}}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = y_1 + y_2\ \mathbf{in}\ e : \tau} \quad \text{(T-Sum)}$$

$$\frac{\Gamma_1, y{:}\tau_1 \rightarrow^{\kappa_1} \tau_2, z{:}\tau_1 \vdash e_0 : \tau_2 \qquad \Gamma_2, y{:}\tau_1 \rightarrow^{\kappa_2} \tau_2 \vdash e : \tau \qquad \kappa = \kappa_2 \cdot (\kappa_1 + 1) \qquad \Gamma = (\kappa \cdot \Gamma_1) + \Gamma_2}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = ({}^{\langle\Gamma_1\rangle}\mathbf{fun}\, y(z) = e_0)^{\kappa}\ \mathbf{in}\ e : \tau} \quad \text{(T-Fun)}$$

$$\frac{\Gamma', x{:}\tau_1, y{:}\tau_2 \vdash e : \tau \qquad \Gamma = \Gamma' + z{:}\tau_1 \times^1 \tau_2}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{let}\ (x, y) = z\ \mathbf{in}\ e : \tau} \quad \text{(T-Ext)}$$

$$\frac{\Gamma', x{:}\tau_1 \times^{\kappa} \tau_2 \vdash e : \tau \qquad \Gamma = \Gamma' + y_1{:}\tau_1 + y_2{:}\tau_2}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{let}\ x = (y_1, y_2)^{\kappa}\ \mathbf{in}\ e : \tau} \quad \text{(T-Pair)}$$

$$\frac{\Gamma' \vdash e_1 : \tau \qquad \Gamma' \vdash e_2 : \tau \qquad \Gamma = \Gamma' + x{:}\mathit{Int}}{\Gamma \vdash {}^{\langle\Gamma\rangle}\,\mathbf{if0}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau} \quad \text{(T-If)}$$

Figure 2: Typing rules for expressions

two branches should be typed under the same environment (T-If). In the rule T-App, the use of the type of the variable y_1 should be 1 since the function stored in y_1 is accessed there; similarly for the rule T-Ext. In the rule T-Fun, the two uses κ_1 and κ_2 represent the number of times of recursive calls and that of calls in e, respectively. Thus, the type environment of the function body Γ is multiplied by $\kappa_2 \cdot (\kappa_1 + 1)$, an upper bound of the total number of times that the function is applied [Kob99,IK00b]. (The type environment Γ for a single call is attached as the type environment annotation rather than $\kappa_2 \cdot (\kappa_1 + 1) \cdot \Gamma$; this is mainly for ease of our type soundness proof. Note that, as we will see later, it does not matter whether we attach Γ or $\kappa_2 \cdot (\kappa_1 + 1) \cdot \Gamma$ for the purpose of GC: the distinction between 1 and ω is not important in GC.) The rule T-Var allows coercion to a smaller type, making it possible to pass non-linear values to where linear values are expected. Annotated type environments must agree with ones from the type derivation to provide the garbage collector with correct type information.

Typing Rules for Programs

We also give a type system for environments, stacks, heaps, and programs. A type system for programs is important to show correctness of both static and dynamic memory management. As stated below, we can show that a well-typed program cannot go wrong under an execution without GC (Theorem 3.3.2), by the standard technique based on subject reduction. This means that, at least, static memory management based on uses is correct. In the next section, correctness of GC will also be stated in terms of well-typedness of a program: it is shown that, given a well-typed program, our garbage collection always succeeds and preserves well-typedness of the program after GC (with the garbage-collected heap). Then, the results of executions with and without GC agree. This means that GC, the dynamic memory management, is also correct.

We use the following type judgments for programs:

$\Gamma \vdash V : \Gamma'$	(V is a well-typed environment providing Γ' under Γ)
$\Gamma \vdash S : \tau_1 \rightarrow^1 \tau_2$	(S is a well-typed stack of type $\tau_1 \rightarrow^1 \tau_2$)
$\Gamma \vdash H : \Gamma_1; \Gamma_2$	(H is a well-typed heap described by Γ_1 and Γ_2)
$\vdash P : \tau$	(P is a well-typed program of type τ)

Environments, Heap Values:

$$\frac{\begin{array}{c} V = \{\tilde{x} = \tilde{z}\} \uplus \{\tilde{y} = \tilde{n}\} \\ x_1 : \tau_1, \ldots x_n : \tau_n, y_1 : \mathit{Int}, \ldots, y_m : \mathit{Int} \geq \Gamma \end{array}}{z_1 : \tau_1 + \cdots + z_n : \tau_n \vdash V : \Gamma} \quad \text{(T-Env)}$$

$$\frac{\begin{array}{c} \Gamma, y : \tau_1, x : \tau_1 \rightarrow^{\kappa_1} \tau_2 \vdash e : \tau_2 \\ \Gamma' \vdash V : \Gamma \qquad \kappa = \kappa_2 \cdot (\kappa_1 + 1) \end{array}}{\kappa \cdot \Gamma' \vdash (V, {}^{\langle \Gamma \rangle}\mathbf{fun}\, x(y) = e)^{\kappa} : \tau_1 \rightarrow^{\kappa_2} \tau_2} \quad \text{(T-HFun)}$$

$$\frac{\Gamma_1 \vdash v_1 : \tau_1 \qquad \Gamma_2 \vdash v_2 : \tau_2}{\Gamma_1 + \Gamma_2 \vdash (v_1, v_2)^{\kappa} : \tau_1 \times^{\kappa} \tau_2} \quad \text{(T-HPair)}$$

Stacks, Heaps, Programs:

$$\Gamma \vdash [\,] : \tau \rightarrow^1 \tau \quad \text{(T-Empty)}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash S : \tau_2 \rightarrow^1 \tau_3 \qquad \Gamma_2 \vdash V : \Gamma_3 \\ \Gamma_3, x : \tau_1 \vdash e : \tau_2 \end{array}}{\Gamma_1 + \Gamma_2 \vdash S[V, \Gamma_3, \lambda x.e] : \tau_1 \rightarrow^1 \tau_3} \quad \text{(T-Push)}$$

$$\frac{\begin{array}{c} \Gamma_1 \vdash h_1^{\kappa_1} : \tau_1 \qquad \cdots \qquad \Gamma_n \vdash h_n^{\kappa_n} : \tau_n \\ \Gamma' + (\tilde{x} : \tilde{\tau}) \geq \lfloor \Gamma + \Gamma_1 + \cdots + \Gamma_n \rfloor \end{array}}{\Gamma' \vdash \{x_1 = h_1^{\kappa_1}, \ldots, x_n = h_n^{\kappa_n}\} : \Gamma; \tilde{x} : \tilde{\tau}} \quad \text{(T-Heap)}$$

$$\frac{\begin{array}{c} \emptyset \vdash H : \Gamma_1 + \Gamma_2; \Gamma \qquad \Gamma_1 \vdash S : \tau' \rightarrow^1 \tau \\ \Gamma_2 \vdash V : \Gamma_3 \qquad \Gamma_3 \vdash e : \tau' \end{array}}{\vdash (H, S, V, e) : \tau} \quad \text{(T-Prog)}$$

Figure 3: Typing rules for stack, heap, and programs

The typing rules are given in Figure 3. In the rule T-Env, the type environment $x_1 : \tau_1, \ldots, x_n : \tau_n, y_1 : \mathit{Int}, \ldots, y_m : \mathit{Int}$ denotes how V can be used by an expression; the type environment $z_1 : \tau_1 + \cdots + z_n : \tau_n$ denotes the types of the heap addresses referred to by the environment V. To deal with possible aliasing (different variables x_i and x_j may maps to the same address z), the summation is required. The rules T-HPair and T-HFun are similar to the rules T-Pair and T-Fun, except that, in T-HFun, free variables of the function refer outside through the environment V. The empty stack is given type $\tau \rightarrow^1 \tau$ for any τ since it can be regarded as an identity function. If a top stack frame is a function of type $\tau_1 \rightarrow^1 \tau_2$ and the rest of the stack is given type $\tau_2 \rightarrow^1 \tau_3$, then the whole stack, regarded as composition of the functions, is given type $\tau_1 \rightarrow^1 \tau_3$. Since each stack frame will be applied at most once, the use of a stack is given 1. The type judgment form $\Gamma \vdash H : \Gamma_1; \Gamma_2$ for heaps and the rule T-Heap are explained as follows. The type environment Γ_2 describes how each heap value can be used; thus, provided that $\Gamma_{x_i} \vdash H(x_i) : \tau_i$ for each $x_i \in \mathit{dom}(H)$, the type environment Γ_2 is $x_1 : \tau_1, \ldots, x_n : \tau_n$. On the other hand, Γ_1 describes how the heap can be used by the stack and environment of a program. Since each Γ_{x_i} describes how the heap value $H(x_i)$ uses other heap values, their total use $\Gamma_1 + \Gamma_{x_1} + \cdots + \Gamma_{x_n}$ should be less than Γ_2. The truncation in the rule T-Heap represents the fact that the stack, environment, and heap values may include dangling pointers if they are given type with the use 0. The type environment Γ (on the left of $\vdash$) describes references to outside of the heap. In a well-typed program, the heap should be closed (except for dangling pointers) and Γ is empty. On the other hand, in a proof of correctness of our GC, it plays an important role to describe references from the collected heap values to the rest of heap values to be collected. Since the type environment Γ_2 is required mainly for conciseness of correctness of our GC and not important here, we often abbreviate $\Gamma \vdash H : \Gamma_1; \Gamma_2$ to $\Gamma \vdash H : \Gamma_1$. The rule T-Prog is straightforward: if a heap H is closed and well typed, a stack S is given type $\tau' \rightarrow^1 \tau$, an environment is well typed, and e is given type τ', then the program (H, S, V, e) is given type τ.

3.3 Soundness of Type System

The type system introduced in this section guarantees that a well-typed program can cause neither run-time type errors nor illegal memory access by dereferencing dangling pointers.

3.3.1 Theorem [Subject Reduction]: If $\emptyset \vdash P : \tau$ and $P \longmapsto P'$, then $\emptyset \vdash P' : \tau$.

Proof sketch: By a case analysis on the rule used to derive $P \longmapsto P'$. ∎

3.3.2 Theorem [Type Soundness]: If $\vdash P : \tau$ and $P \longmapsto^* P'$ with P' being a normal form, then P' is an answer and $\vdash P' : \tau$.

Proof sketch: By a case analysis on the form of the expressions in P, we show that, if $\vdash P : \tau$ with P being non-answer, then there exists P' such that $P \longmapsto P'$. Then, the conclusion is immediate from Theorem 3.3.1. ∎

4 Garbage Collection

In this section we describe our GC algorithm for $\lambda^{\kappa}_{\mathrm{gc}}$ formally and show its correctness.

4.1 GC Algorithm

First, we define three auxiliary functions TL_{env}, TL_{stack} and TL_{hval} to collect the type information on the addresses referred to by an environment, a stack and a heap value, respectively. The first two are used when the garbage collector is invoked: the initial scan set is computed by $TL_{\mathrm{stack}}[S] + TL_{\mathrm{env}}[V, TE(e)]$ where S, V, e are respectively the stack, environment, and expression from the program for which GC is invoked. The function TL_{env} is also used to compute type information on (continuation) closures stored in a heap or a stack. The last one is used when a heap value is marked and a new scan set is computed.

4.1.1 Definition: The functions TL_{env}, TL_{stack} and TL_{hval} are defined as follows:

$$TL_{\mathrm{env}}[V, \Gamma] = \omega \cdot \textstyle\sum_{x \in dom(\Gamma), \Gamma(x) \neq Int} V(x) : \Gamma(x)$$

$$\begin{aligned} TL_{\mathrm{stack}}[[\,]] &= \emptyset \\ TL_{\mathrm{stack}}[S[V, \Gamma, \lambda x.e]] &= TL_{\mathrm{stack}}[S] + TL_{\mathrm{env}}[V, \Gamma] \end{aligned}$$

$$\begin{aligned} TL_{\mathrm{hval}}[(V, {}^{\langle\Gamma\rangle}\mathbf{fun}\, x(y) = e)^{\kappa}, \tau_1 \to^{\kappa'} \tau_2] &= \omega \cdot TL_{\mathrm{env}}[V, \Gamma] \\ TL_{\mathrm{hval}}[(v_1, v_2)^{\kappa}, \tau_1 \times^{\kappa'} \tau_2] &= \omega \cdot \textstyle\sum_{\tau_i \neq Int} v_i : \tau_i \end{aligned}$$

($TL_{\mathrm{env}}[V, \Gamma]$ is undefined if $\Gamma(x) \neq Int$ and $V(x)$ is an integer for some $x \in dom(\Gamma)$. Similarly, $TL_{\mathrm{hval}}[(v_1, v_2)^{\kappa}, \tau_1 \times^{\kappa'} \tau_2]$ is undefined if $\tau_i \neq Int$ and v_i is an integer.)

Note that, the garbage collector need not distinguish between the uses 1 and ω in the scan set (and type information on the marked heap values). Hence, types are multiplied by ω to "normalize" type information.

Our GC algorithm is formally represented as rewriting of a quadruple $(H_f, \Gamma_s, H_t, \Gamma_t)$, consisting of two heaps and two type environments. Intuitively, H_f corresponds to a "from-space," H_t to a "to-space," and Γ_s to a "scan-set," which maintains locations to be traced together with their types. The type environment Γ_t, which maintains types of H_t, is used for avoiding verbose tracing, mentioned in Section 1. This algorithm could be implemented by extending either copying or mark-and-sweep GC, but, some implementation details such as forwarding pointers or the sweeping phase are abstracted out from the model.

4.1.2 Definition [GC algorithm]: The relation $(H_f, \Gamma_s, H_t, \Gamma_t) \Longrightarrow (H'_f, \Gamma'_s, H'_t, \Gamma'_t)$ is the least relation closed under the following rules:

$$(H_f \uplus \{x = h^{\kappa}\}, (\Gamma_s, x{:}\tau), H_t, \Gamma_t) \Longrightarrow (H_f, \Gamma_s + TL_{\mathrm{hval}}[h^{\kappa}, \tau], H_t \uplus \{x = h^{\kappa}\}, (\Gamma_t, x{:}\tau)) \quad \text{(GC-MARK)}$$

$$\frac{\Gamma_t(x) \not\geq \tau}{(H_f, (\Gamma_s, x{:}\tau), H_t, \Gamma_t) \Longrightarrow (H_f, \Gamma_s + TL_{\mathrm{hval}}[H_t(x), \tau], H_t, \Gamma_t + x{:}\tau)} \quad \text{(GC-REMARK)}$$

$$\frac{\tau = \tau_1 \times^0 \tau_2 \quad \text{or} \quad \tau = \tau_1 \to^0 \tau_2 \quad \text{or} \quad \Gamma_t(x) \geq \tau}{(H_f, (\Gamma_s, x{:}\tau), H_t, \Gamma_t) \Longrightarrow (H_f, \Gamma_s, H_t, \Gamma_t)} \quad \text{(GC-SKIP)}$$

The rule GC-MARK moves the heap value at x to the to-space, computes a new scan set $\Gamma_s + TL_{\text{hval}}[h^\kappa, \tau]$, and adds the type of the variable to Γ_t. As mentioned in Section 1, the garbage collector may mark one heap value more than once; the rule GC-REMARK is applied when the heap value itself has been already marked but something reachable from it is neither marked nor scheduled to be marked yet. The premise $\Gamma_t(x) \not\geq \tau$, which judges existence of such unmarked values, holds if and only if, for some use 0 in $\Gamma_t(x)$, there is a non-zero use in the corresponding position in the type τ. For example, suppose τ in the scan set is $(\mathit{Int} \times^\omega \mathit{Int}) \times^\omega \mathit{Int}$ and $\Gamma_t(x)$ is $(\mathit{Int} \times^0 \mathit{Int}) \times^\omega \mathit{Int}$. Then, the pair pointed by x has to be marked again since the previous traverses did not mark the inner pair of type $\mathit{Int} \times^0 \mathit{Int}$. After this traverse the type $\Gamma_t(x)$ becomes $(\mathit{Int} \times^\omega \mathit{Int}) \times^\omega \mathit{Int}$ and the rule GC-SKIP explained below will always be applied whenever x is selected from the scan set. The rule GC-SKIP is used for two cases. One is when the use of the type of the variable x in the scan set is 0, which means that the heap value at x does not have to be marked (sometimes, the memory space at the address has been deallocated); the other is when the values that the current traverse tries to copy are already marked or scheduled to be marked ($\Gamma_t(x) \geq \tau$). Note that, in the former case, the garbage collector may skip marking reachable garbage.

Finally, the whole GC process, which computes a garbage-collected heap called *collection* from a program, is defined below. Given a program, it computes an initial scan set by using TL_{env} and TL_{stack}, begins rewriting defined above with the given heap and the initial scan set until it reaches the empty scan set; then, the to-space is the collection:

4.1.3 Definition [collection]: A heap H' is a *collection* of a program (H, S, V, e) if and only if

$$(H,\ TL_{\text{stack}}[S] + TL_{\text{env}}[V,\ TE(e)], \emptyset, \emptyset) \Longrightarrow^* (H'', \emptyset, H', \Gamma_t)$$

where $\Longrightarrow^*$ is the reflexive and transitive closure of $\Longrightarrow$.

4.1.4 Example: A collection of a program P

$$P = (H, [\,], \{p = x_1, f = x_3, g = x_4\}, {}^{\langle p:\mathit{Int}\times^0(\mathit{Int}\times^0\mathit{Int}), f:\mathit{Int}\to^1\mathit{Int}, g:\mathit{Int}\to^1\mathit{Int}\rangle} f\ 1 + g\ 2)$$

where

$$H = \left\{ \begin{array}{l} x_1 = (2, x_2)^\omega, \\ x_2 = (3,4)^1, \\ x_3 = (\{p = x_1\}, {}^{\langle p:\mathit{Int}\times^1(\mathit{Int}\times^0\mathit{Int})\rangle}\mathbf{fun}\, f'(y) = y + (\#1\ p)), \\ x_4 = (\{p = x_1\}, {}^{\langle p:\mathit{Int}\times^1(\mathit{Int}\times^1\mathit{Int})\rangle}\mathbf{fun}\, g'(y) = y + (\#1\ (\#2\ p))) \end{array} \right\}$$

is H (for brevity, we use the direct-style and the projection operators #1 and #2 in Section 1); a possible sequence of GC rewriting steps is given as follows:

$$\begin{array}{ll}
 & (H, (x_3{:}\,\mathit{Int} \to^\omega \mathit{Int}, x_4{:}\,\mathit{Int} \to^\omega \mathit{Int}), \emptyset, \emptyset) \\
(\text{GC-MARK}) & \Longrightarrow (H \setminus \{x_3\}, (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^0 \mathit{Int}), x_4{:}\,\mathit{Int} \to^\omega \mathit{Int}), \{x_3 = H(x_3)\}, x_3{:}\,\mathit{Int} \to^\omega \mathit{Int}) \\
(\text{GC-MARK}) & \Longrightarrow (H \setminus \{x_1, x_3\}, (x_2{:}\,\mathit{Int} \times^0 \mathit{Int}, x_4{:}\,\mathit{Int} \to^\omega \mathit{Int}), \{x_1 = (2, x_2)^\omega, x_3 = H(x_3)\}, \\
 & \quad\ (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^0 \mathit{Int}), x_3{:}\,\mathit{Int} \to^\omega \mathit{Int})) \\
(\text{GC-SKIP}) & \Longrightarrow (H \setminus \{x_1, x_3\}, (x_4{:}\,\mathit{Int} \to^\omega \mathit{Int}), \{x_1 = (2, x_2)^\omega, x_3 = H(x_3)\}, \\
 & \quad\ (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^0 \mathit{Int}), x_3{:}\,\mathit{Int} \to^\omega \mathit{Int})) \\
(\text{GC-MARK}) & \Longrightarrow (\{x_2 = (3,4)^1\}, x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^\omega \mathit{Int}), H \setminus \{x_2\}, \\
 & \quad\ (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^0 \mathit{Int}), x_3{:}\,\mathit{Int} \to^\omega \mathit{Int}, x_4{:}\,\mathit{Int} \to^\omega \mathit{Int})) \\
(\text{GC-REMARK}) & \Longrightarrow (\{x_2 = (3,4)^1\}, x_2{:}\,\mathit{Int} \times^\omega \mathit{Int}, H \setminus \{x_2\}, \\
 & \quad\ (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^\omega \mathit{Int}), x_3{:}\,\mathit{Int} \to^\omega \mathit{Int}, x_4{:}\,\mathit{Int} \to^\omega \mathit{Int})) \\
(\text{GC-MARK}) & \Longrightarrow (\emptyset, \emptyset, H, (x_1{:}\,\mathit{Int} \times^\omega (\mathit{Int} \times^\omega \mathit{Int}), x_2{:}\,\mathit{Int} \times^\omega \mathit{Int}, x_3{:}\,\mathit{Int} \to^\omega \mathit{Int}, x_4{:}\,\mathit{Int} \to^\omega \mathit{Int}))
\end{array}$$

Notice how the rule GC-REMARK is applied and the type of x_1 is changed afterwards, and that the use of the rules GC-SKIP and GC-REMARK could be dispensed with if x_4 were traversed first.

4.2 Correctness of GC

Correctness of the GC algorithm is proved by showing that a collection H' of a well-typed program (H, S, V, e) is always obtained and both heaps are well typed with respect to the same type environment, i.e., $\emptyset \vdash H : \Gamma$ and $\emptyset \vdash H' : \Gamma$. If the above condition holds, we can safely replace the heap in a well-typed program with its collection and obtain the same result (if exists) because the collection is a subset of the original heap and the program after GC does not cause run-time type errors. A more formal argument would be similar to the discussions found in [MFH95, MH96].

We only state a correctness theorem with main lemmas and sketch their proofs here. The main lemmas (4.2.3, 4.2.4, and 4.2.5) ensure that a certain invariant (the well-formedness condition in Definition 4.2.2 below) holds during GC and that GC always terminates with the empty scan set without failure. Then, the main theorem (4.2.6) is an easy consequence from the three lemmas.

Before the definition of well-formedness, we introduce an operation to recover the actual types of the scan set Γ_s and the marked set Γ_t.

4.2.1 Definition: The *filtered* use of κ_1 by κ_2, written $\kappa_1 \sharp \kappa_2$, is defined by: $\kappa \sharp \omega = \kappa$ and $\kappa \sharp 0 = 0$. Moreover, it is pointwise extended to types and type environments:

$$\begin{array}{lcl} \mathit{Int} \sharp \mathit{Int} & = & \mathit{Int} \\ (\tau_1 \to^{\kappa_1} \tau_2) \sharp (\tau_1 \to^{\kappa_2} \tau_2) & = & \tau_1 \to^{\kappa_1 \sharp \kappa_2} \tau_2 \\ (\tau_{11} \times^{\kappa_1} \tau_{12}) \sharp (\tau_{21} \times^{\kappa_2} \tau_{22}) & = & (\tau_{11} \sharp \tau_{21}) \times^{\kappa_1 \sharp \kappa_2} (\tau_{21} \sharp \tau_{22}) \end{array}$$

$$\begin{array}{l} \mathit{dom}(\Gamma_1 \sharp \Gamma_2) = \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \\ (\Gamma_1 \sharp \Gamma_2)(x) = \Gamma_1(x) \sharp \Gamma_2(x) \quad \text{for each } x \in \mathit{dom}(\Gamma_1) \cap \mathit{dom}(\Gamma_2) \end{array}$$

4.2.2 Definition [Well-Formedness]: Suppose $\emptyset \vdash H : \Gamma; \Gamma_H$. The tuple $(H_f, \Gamma_s, H_t, \Gamma_t)$ is *well formed* with respect to $\emptyset \vdash H : \Gamma; \Gamma_H$ iff:

1. $H = H_f \uplus H_t$
2. $\lfloor \omega \cdot \Gamma + \sum_{x \in \mathit{dom}(\Gamma_t)} \mathit{TL}_{\mathrm{hval}}[H(x), \Gamma_H(x)] \rfloor \geq \lfloor \Gamma_s \rfloor + \Gamma_t$
3. $(\Gamma_H \sharp \lfloor \Gamma_s \rfloor) \vdash H_t : \Gamma; (\Gamma_H \sharp \Gamma_t)$

Intuitive meanings of the conditions are as follows: (1) a heap value is in either the from-space or the to-space; (2) the type of a variable in the scan set is compatible with the corresponding heap value's actual type (we can easily show that $\omega \cdot \Gamma_H \geq \lfloor \Gamma_s \rfloor + \Gamma_t$); and (3) the scan-set holds all free variables used by the heap values in the to-set. The filtered type environments $\Gamma_H \sharp \lfloor \Gamma_s \rfloor$ and $\Gamma_H \sharp \Gamma_t$ are used to recover the actual type information on the scan set and the types of marked heap values from the normalized ones Γ_s and Γ_t.

4.2.3 Lemma [GC Well-Formedness Preservation]: Suppose $\emptyset \vdash H : \Gamma; \Gamma_H$. If $(H_1, \Gamma_1, H_2, \Gamma_2)$ is well formed with respect to $\emptyset \vdash H : \Gamma; \Gamma_H$ and $(H_1, \Gamma_1, H_2, \Gamma_2) \Longrightarrow (H_1', \Gamma_1', H_2', \Gamma_2')$, then $(H_1', \Gamma_1', H_2', \Gamma_2')$ is well formed with respect to $\emptyset \vdash H : \Gamma; \Gamma_H$.

Proof sketch: By a case analysis on the rule used to derive $(H_1, \Gamma_1, H_2, \Gamma_2) \Longrightarrow (H_1', \Gamma_1', H_2', \Gamma_2')$. ∎

4.2.4 Lemma [GC Progress]: If $\emptyset \vdash H : \Gamma; \Gamma_H$ and $(H_1, \Gamma_1, H_2, \Gamma_2)$ is well-formed with respect to $\emptyset \vdash H : \Gamma; \Gamma_H$, then either Γ_1 is empty or $(H_1, \Gamma_1, H_2, \Gamma_2) \Longrightarrow (H_1', \Gamma_1', H_2', \Gamma_2')$ for some $(H_1', \Gamma_1', H_2', \Gamma_2')$.

Proof sketch: It is easy to show $\mathit{dom}(\lfloor \Gamma_1 \rfloor + \Gamma_2) \subseteq \mathit{dom}(\Gamma_H)$ from the second well-formedness condition and the assumption $\emptyset \vdash H : \Gamma; \Gamma_H$. Since $H = H_1 \uplus H_2$ and $\mathit{dom}(H) = \mathit{dom}(\Gamma_H)$, either $x \in \mathit{dom}(H_1)$ or $x \in \mathit{dom}(H_2)$ holds for each $x \in \mathit{dom}(\lfloor \Gamma_1 \rfloor)$. Then, it is easy to show that one of the three rules can be applied for a quadruple with a non-empty scan set Γ_1. (In any case, it is easy to show well-definedness of the right-hand side of $\Longrightarrow$.) ∎

4.2.5 Lemma [GC Termination]: If $(H_1, \Gamma_1, H_2, \Gamma_2)$ is well-formed with respect to $\emptyset \vdash H : \Gamma; \Gamma_H$, then there is no infinite sequence $(H_1, \Gamma_1, H_2, \Gamma_2) \Longrightarrow (H_1', \Gamma_1', H_2', \Gamma_2') \Longrightarrow \cdots$.

Proof sketch: We define a partial order $<$ by $(H_1, \Gamma_1, H_2, \Gamma_2) < (H_1', \Gamma_1', H_2', \Gamma_2')$ iff (1) $\Gamma_1 \subset \Gamma_1'$ with $\Gamma_2 = \Gamma_2'$ or (2) $\omega \cdot \Gamma_H \geq \Gamma_2 \geq \Gamma_2'$ with $\Gamma_2 \neq \Gamma_2'$. We can easily show that the order $<$ is well-founded and that $\Longrightarrow$ generates a monotonically decreasing sequence with respect to $<$. ∎

4.2.6 Theorem [Correctness of GC Algorithm]: If $\vdash (H, S, V, e) : \tau$, then there exists a collection H' of the program (H, S, V, e) and $\vdash (H', S, V, e) : \tau$.

Proof sketch: It is easy to show that, if a program is well typed, then the initial state of GC is well formed. Then, by Lemmas 4.2.5, 4.2.4 and 4.2.3, it is shown that the rewriting will terminate in a well-formed state with the empty scan set. By the third well-formedness condition, the collection is a well-typed heap described by the same type environment. ∎

5 Related Work

Linear type systems. In most of the existing linear type systems [Wad90, TWM95, CGR92, WJ99], the uses 0 and ω here are not distinguished. Thus, there would be no dangling pointers created in the heap space and conventional tracing GC could be applied. Mogensen [Mog97] and the authors [Iga97, IK00b] independently introduced 0 to the use information[3]. Under such a linear type system with the use 0, the garbage collector has to avoid tracing dangling pointers. In [Iga97, IK00b], the summation of two pair types is defined only when the element types are the same, i.e., elements of a data structure have to be uniformly accessed in every context where the data structure is accessed. Thus, unlike our system, the type information on marked objects would not be required and the usual mark-bit mechanism would be enough. Furthermore, Mogensen's type system and Kobayashi's quasi-linear type system [Kob99] removed the restriction on summation of types as in this paper. The new summation operator together with the use 0 play a significant role to refine the analysis to detect linear values and so improve effectiveness of the static memory management. On the other hand, as we have studied, GC has to know which part of the marked object has been marked.

Chirimar, Gunter, and Riecke [CGR92] formalized memory management based on reference counting for a language with a linear type system. There are no dangling pointers in the memory space and the memory management algorithm itself was fairly straightforward.

Type-directed GC. There have been two approaches towards tag-free GC for ML-style polymorphic languages. In order to recover the actual type arguments of a polymorphic function, in one approach, explicit type arguments are passed at run-time [App89, Tol94, MH96] and, in the other, type reconstruction is performed at GC-time [Gol91, GG92, Fra94, AFH94, MFH95, HY98]. Some type inference GC [GG92, Fra94, MFH95, HY98] for polymorphic languages can collect reachable garbage as our GC also can. In some cases, our GC can collect more garbage than the type inference GC schemes proposed so far. For example, consider an expression $f(x) + g(x) + g(y)$ and suppose the function f uses its argument but g does not. If both f and g are used in a monomorphic context (as they are function arguments, for example), then y must be given the same type as f's domain type, which is concrete (i.e., not a type variable); thus, type inference GC cannot collect an object at y. In our type system, on the other hand, the use of the type of y can be 0 even if the use of f's domain type is more than 0. Thus, our garbage collector can collect the object at y. We leave further comparison of our GC and type inference GC for future work.

Agesen, Detlefs, and Moss [ADM98] showed that, by a liveness analysis of local variables in a Java virtual machine, a garbage collector can avoid tracing useless references, thus reducing the required heap size. In fact, use of a liveness analysis seems to become fairly common in real compilers such as several Java JIT compilers. Since our garbage collector can avoid tracing useless variables (with the use 0) not only in the *stack* but also the *heap*, our technique can be more effective.

[3] The original idea of the use 0 is attributed to Bierman [Bie92] and the implicit idea of the use 0 is also found in the preceding paper on a linear type system for the π-calculus [KPT96].

Region-based static memory management. Tofte et al. have proposed another technique for static memory management, based on region inference [TT94, BTV96]; it analyzes the lifetime of *regions*, which are fragments of memory space with nested lifetime, by using an effect-based type system and inserts explicit allocation/deallocation primitives into programs. The region-based memory management may also deallocate elements of a data structure before the deallocation of itself, creating dangling pointers. Kariya and Kobayashi [KK99] have developed an algorithm of tracing GC under the region-based static memory management. The idea is similar to ours: since the static type information tells the garbage collector which regions are not used by a certain value, it can be avoided to trace dangling pointers.

6 Conclusions

We have studied a GC scheme for a programming language with static memory management based on a linear type system. Since it allows linear values to be reclaimed before the reclamation of values pointing to them, dangling pointers may occur in the heap space; in order to deal with them, we exploited static type information during GC; our GC scheme can not only avoid tracing dangling pointers but also collect some of reachable garbage. We have formalized our GC algorithm and stated its correctness.

7 Future Work

The work presented here is rather preliminary; much work is left to be done to adopt our technique to a real programming language like ML and evaluate a real impact.

First of all, integration with a polymorphic type system will be crucial. But, in fact, our GC scheme itself can be extended in a fairly straightforward manner: the technique of run-time type passing [Tol94, MH96] can be used to obtain actual type argument information on type variables, which the garbage collector will require. Moreover, this technique would also be extended to polymorphism on uses.

However, naive implementation of our memory management scheme described here will not be very effective. As in real implementation of region-based memory management [BTV96], auxiliary techniques will be needed to reduce the overheads. We briefly discuss main issues and possible solutions below:

Effectiveness of use analysis. The present type system, apart from the lack of polymorphism, would not be very useful for static memory management: as discussed elsewhere [Kob99], the condition of use-onceness is too restrictive. Kobayashi's quasi-linear type system [Kob99] (extended with polymorphism), a refinement of a linear type system for memory management, would be a good base of our system. We also should explore the design space about polymorphism, involving many engineering tradeoffs between the power of the type system and the overhead of type reconstruction [WJ99, WJ00].

Reduction of run-time cost. One of the main potential run-time overheads of our memory management would lie in deallocation of linear values involving dynamic check of uses, even though the check can be implemented without extra memory space for tags. To omit those checks, we might be able to modify the type system so that it also classifies primitive operations into two: those accepting only linear values with performing deallocation, and those accepting only non-linear values without deallocation. However, type reconstruction for such a type system would be impractical; we expect the complexity to be exponential. Instead, we will be able to benefit from flow analysis to determine at which deallocation points dynamic checks can be omitted. In the presence of polymorphism, the technique of type lifting [Tol94, Min96, SS98] can be used to reduce the cost of run-time type passing.

Reduction of GC-time cost. Our garbage collector is presented so that it keeps track of type information on the heap values already marked. However, in real implementation, the full type information is *not* needed: for example, for a function closure, only one bit information (0 or ω), corresponding to the outermost use of the function type, is sufficient. Concerning pair types, the required information can be represented by a bit vector of length $1 + n_1 + n_2$ where n_i is the length of such a bit vector for the i-th component type. Thus, without deeply nested pair types (or large tuple types), the cost could be comparable to conventional tracing GC, which uses a mark bit for each heap block. Moreover, such bit vectors may even be omitted by analyzing aliasing in the heap space.

As other theoretical issues, it is interesting to generalize our system so that the GC algorithm is parameterized by the underlying linear type system. Also, formal connection to type inference GC would be also worth investigating.

Acknowledgment

Comments from anonymous referees of TIC2000 helped us improve the presentation.

References

[ADM98] Ole Agesen, David Detlefs, and J. Eliot B. Moss. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 269–279, Montreal, Canada, June 1998.

[AFH94] Shail Aditya, Christine H. Flood, and James E. Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *Proceedings of the ACM Conference on Lisp and Functional Programming (LFP)*, pages 12–23, Orlando, FL, June 1994.

[AFL95] Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 174–185, San Diego, CA, June 1995.

[App89] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2(7):153–162, 1989.

[Bie92] Gavin M. Bierman. Type systems, linearity and functional languages. In *Proceedings of the CLICS Workshop*, pages 71–92, March 1992. Available as Aarhus University Technical Report DAIMI PB 397-I.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 171–183, St. Petersburg Beach, FL, January 1996.

[CGR92] Jawahar Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. In *Proceedings of the ACM Conference on Lisp and Functional Programming (LFP)*, pages 139–150, San Francisco, CA, July 1992.

[Fra94] Pascal Fradet. Collecting more garbage. In *Proceedings of the ACM Conference on Lisp and Functional Programming (LFP)*, pages 24–33, Orlando, FL, June 1994.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 237–247, Albuquerque, NM, June 1993.

[GG92] Benjamin Goldberg and Michael Gloger. Polymorphic type reconstruction for garbage collection without tags. In *Proceedings of the ACM Conference on Lisp and Functional Programming (LFP)*, pages 53–65, San Francisco, CA, June 1992.

[GH90] Juan C. Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science (LICS)*, pages 333–343, Philadelphia, PA, June 1990.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly typed programming languages. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 165–176, Toronto, Canada, June 1991.

[HY98] Haruo Hosoya and Akinori Yonezawa. Garbage collection via dynamic type inference — a formal treatment —. In *Proceedings of the 2nd International Workshop on Types in Compilation (TIC)*, volume 1473 of *Lecture Notes on Computer Science*, pages 215–239, Kyoto, Japan, March 1998. Springer-Verlag.

[Iga97] Atsushi Igarashi. Type-based analysis of usage of values for concurrent programming languages. Master's thesis, University of Tokyo, Tokyo, Japan, February 1997.

[IK00a] Atsushi Igarashi and Naoki Kobayashi. Garbage collection based on a linear type system. Technical report, Department of Information Science, University of Tokyo, Tokyo, Japan, 2000. In preparation. To be available through `http://www.graco.c.u-tokyo.ac.jp/~igarashi/papers.html`.

[IK00b] Atsushi Igarashi and Naoki Kobayashi. Type reconstruction for linear π-calculus with I/O subtyping. *Information and Computation*, 2000. To appear. A preliminary summary was presented in *Proceedings of SAS'97* under the title "Type-based Analysis of Communication for Concurrent Programming Languages," Springer LNCS 1302, pages 187–201.

[KK99] Hideki Kariya and Naoki Kobayashi. Combining region-based memory management with conventional garbage collection. *JSSST Computer Software*, 16(3):66–70, May 1999. (in Japanese).

[Kob99] Naoki Kobayashi. Quasi-linear types. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 29–42, San Antonio, TX, January 1999.

[KPT96] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 358–371, St. Petersburg Beach, FL, January 1996.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 66–77, San Diego, CA, June 1995.

[MH96] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. Technical Report CMU-CS-96-176, Carnegie Mellon University, Pittsburgh, PA, September 1996.

[Min96] Yasuhiko Minamide. Compilation based on a calculus for explicit type passing. In *Proceedings of the Second Fuji International Workshop on Functional and Logic Programming*, 1996.

[Mog97] Torben Æ. Mogensen. Types for 0, 1 or many uses. In *Proceedings of the 9th International Workshop on Implementation of Functional Languages*, number 1467 in Lecture Notes on Computer Science, pages 157–165, St. Andrews, Scotland, September 1997. Springer-Verlag.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, December 1995.

[SS98] Bratin Saha and Zhong Shao. Optimal type lifting. In *Proceedings of the 2nd International Workshop on Types in Compilation (TIC)*, volume 1473 of *Lecture Notes on Computer Science*, pages 156–177, Kyoto, Japan, March 1998. Springer-Verlag.

[Tol94] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming (LFP)*, pages 1–11, Orlando, FL, June 1994.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 188–201, Portland, OR, January 1994.

[TWM95] David N. Turner, Philip Wadler, and Christian Mossin. Once upon a type. In *Proceedings of the 7th International Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 1–11, San Diego, CA, June 1995.

[Wad90] Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*, Sea of Galilee, Israel, April 1990. North Holland. IFIP TC 2 Working Conference.

[WJ99] Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *Proceedings of the 26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 15–28, San Antonio, TX, January 1999.

[WJ00] Keith Wansbrough and Simon Peyton Jones. Simple usage polymorphism. In *Proceedings of the 3rd International Workshop on Types in Compilation (TIC)*, Montreal, Canada, September 2000.