Fractional Ownerships for Safe Memory Deallocation

Naoki Kobayashi and Kohei Suenaga

Tohoku University

Abstract. We propose a type system for a programming language with memory allocation/deallocation primitives, which prevents memory-related errors such as double-frees and memory leaks. The key idea is to augment pointer types with fractional ownerships, which express both capabilities and obligations to access or deallocate memory cells. Thanks to the use of fractions as ownerships, the type system admits a polynomial-time type inference algorithm, which serves as an algorithm for automatic verification of lack of memory-related errors. A prototype verifier has been implemented and tested for several programs manipulating lists, trees, and doubly-linked lists.

1 Introduction

In programming languages with manual memory management (like C and C++), a misuse of memory allocation/deallocation primitives often causes serious, hard-to-find bugs. We propose a new type system for statically preventing such memory-related errors. More precisely, the type system prevents memory leaks (forgetting to deallocate memory cells), double frees (deallocating memory cells more than once), and read/write accesses to deallocated memory. The type system is equipped with a polynomial-time type inference algorithm, so that programs can be automatically verified.

The key idea of our type system is to assign *fractional ownerships* to pointer types. An ownership ranges over the set of rational numbers in $[0, 1]$, and expresses both a capability (or permission) to access a pointer, and an obligation to deallocate the memory referred to by the pointer. As in Boyland's fractional permissions [1], a non-zero ownership expresses a permission to dereference the pointer, and an ownership of 1 expresses a permission to update the memory cell referenced by the pointer. In addition, a non-zero ownership expresses an obligation to eventually deallocate (the cell referenced by) the pointer, and an ownership of 1 also expresses a permission to deallocate the pointer. (Therefore, if one has an ownership of 0.5, one has to eventually combine it with other ownerships to obtain an ownership of 1, to fulfill the obligation to deallocate the pointer).

For example, $\mathbf{int}\ \mathbf{ref}_1\ \mathbf{ref}_1$ is the type of a pointer to an pointer to an integer, such that both the pointers can be read/written, and must be deallocated through the pointer. $\mathbf{int}\ \mathbf{ref}_0\ \mathbf{ref}_1$ is the type of a pointer to an pointer to

Fig. 1. List-like structure

an integer, such that only the first pointer can be read/written, and must be deallocated. To see how such types can be used, consider the following program, written in an ML-like language (but with memory deallocation primitive `free`).

`fun freeall(x) =`	$\texttt{freeall} : \mu\alpha.(\alpha\ \mathbf{ref}_1) \to \mu\alpha.(\alpha\ \mathbf{ref}_0)$
`  if null(x)`	$x : \mu\alpha.(\alpha\ \mathbf{ref}_1)$
`  then skip`	$x : \mu\alpha.(\alpha\ \mathbf{ref}_0)$
`  else let y = *x in`	$x : \mu\alpha.(\alpha\ \mathbf{ref}_1)$
`         (freeall(y);`	$x : (\mu\alpha.(\alpha\ \mathbf{ref}_0))\ \mathbf{ref}_1, y : \mu\alpha.(\alpha\ \mathbf{ref}_1)$
`          free(x)`	$x : (\mu\alpha.(\alpha\ \mathbf{ref}_0))\ \mathbf{ref}_1, y : \mu\alpha.(\alpha\ \mathbf{ref}_0)$
`         )`	$x : \mu\alpha.(\alpha\ \mathbf{ref}_0), y : \mu\alpha.(\alpha\ \mathbf{ref}_0)$

The function `freeall` takes as an argument a pointer `x` to a list structure shown in Figure 1, and deallocates all the pointers reachable from `x`. The right-hand side shows the type of function `freeall`, as well as the types assigned to `x` and `y` before execution of each line. (Our type system is flow-sensitive, so that different types are assigned at different program points.) Here, $\mu\alpha.\tau$ is a recursive type. In the type of `freeall` on the first line, $\mu\alpha.(\alpha\ \mathbf{ref}_1)$ and $\mu\alpha.(\alpha\ \mathbf{ref}_0)$ are the types of x before and after the call of the function. The type $\mu\alpha.(\alpha\ \mathbf{ref}_1)$ means that the argument `x` must hold the ownerships of all the pointers reachable from `x` when the function is called, while $\mu\alpha.(\alpha\ \mathbf{ref}_0)$ means that `x` holds no ownerships when the function returns (which implies that all the pointers reachable from `x` will be deallocated inside the function).

The type assignment at the beginning of the function indicates that all the memory cells reachable from `x` should be deallocated through variable `x`. In the then-branch, `x` is a null pointer, so that all the ownerships are cleared to 0. In the else-branch, $\mathbf{let}\ y = *x\ \mathbf{in}\ \cdots$ transfers a part of the ownerships held by `x` to `y`; after that, `x` has type $(\mu\alpha.(\alpha\ \mathbf{ref}_0))\ \mathbf{ref}_1$, indicating that x holds only the ownership of the pointer stored in `x`. The other ownerships (of the pointers that are reachable from `x`) are now held by `y`. After the recursive call to `freeall`, all the ownerships held by `y` become empty. Finally, after `free(x)`, the ownership of `x` also becomes empty.

The type system with fractional ownerships prevents: (i) memory leaks by maintaining the invariant that the total ownership for each memory cell is 1 until the cell is deallocated and by ensuring the ownerships held by a variable are empty at the end of the scope of the variable, (ii) double frees by ensuring that the ownership for a cell is consumed when the cell is deallocated, and (iii) illegal access to deallocated cells by requiring that a non-zero ownership is required for read/write operations.

Based on the type system, we have implemented a prototype verifier, and tested it for programs manipulating lists, trees, and doubly-linked lists.

Ownerships are also used in Heine and Lam's static analysis for detecting memory leaks [3]. Major differences are:

- In their system [3], a variable can hold only the ownership of the pointer stored in the variable, while in our type system, a variable can hold ownerships of any pointers reachable from the variable. In their system, therefore, an ownership cannot be passed to a pointer to a pointer or an array of pointers; for the example above, they cannot express a type like $\mu\alpha.\alpha\ \mathbf{ref}_1$, which holds all the ownerships of the reachable cells. Because of this restriction, we believe their system cannot verify list-manipulating C programs like the one given above.
- Their ownerships are integer-valued, while ours are fractions. Thus, the ownership (or type) inference is reduced to integer linear programming (which is NP-hard in general) in their type system, while in our type system, it is reduced to linear programming over rational numbers, which is solved in polynomial time.
- Their system does not prevent illegal access to deallocated pointers (in fact, no ownership is required for read/write operations in their system).

The rest of this paper is structured as follows. Section 2 introduces a simple imperative language that has only pointers as values. Section 3 presents our type system with fractional ownerships, proves its soundness, and discusses type inference issues. Section 4 discusses extensions to deal with data structures. Section 5 reports a prototype implementation of our type-based verification algorithm. Section 6 discusses related work and Section 7 concludes.

2 Language

This section introduces a simple imperative language with primitives for memory allocation/deallocation. For the sake of simplicity, the only values are (possibly null) pointers. See Section 4 for extensions of the language and the type system to deal with other language constructs.

The syntax of the language is given as follows.

Definition 1 (commands, programs)

$$\begin{array}{rl} s\ \textit{(commands)} ::= & \mathbf{skip} \mid *x \leftarrow y \mid s_1; s_2 \mid \mathbf{free}(x) \mid \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \\ & \mid \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s \mid \mathbf{let}\ x = y\ \mathbf{in}\ s \mid \mathbf{let}\ x = *y\ \mathbf{in}\ s \\ & \mid \mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid f(x_1, \ldots, x_n) \\ & \mid \mathbf{assert}(x = y) \mid \mathbf{assert}(x = *y) \\ d\ \textit{(definitions)} ::= & f(x_1, \ldots, x_n) = s \end{array}$$

A program is a pair (D, s), where D is a set of definitions.

The command **skip** does nothing. $*x \leftarrow y$ updates the target of x (i.e., the contents of the memory cell pointed to by x) with the value of y. The command $s_1; s_2$ is a sequential execution of s_1 and s_2. The command $\mathbf{free}(x)$ deallocates

(the cell referenced by) the pointer x. The command **let** $x = e$ **in** s evaluates e, binds x to the value of e, and executes s. The expression **malloc**() allocates a memory cell and returns a pointer to it. The expression **null** denotes a null pointer. $*y$ dereferences the pointer y. The command **ifnull**(x) **then** s_1 **else** s_2 executes s_1 if x is null, and executes s_2 otherwise. The command $f(x_1, \dots, x_n)$ calls function f. We require that $x_1, \dots, x_n$ are mutually distinct variables. (This does not lose generality, as we can replace $f(x, x)$ with **let** $y = x$ **in** $f(x, y)$.) There is no return value of a function call; values can be returned only by reference passing. The commands **assert**($x = y$) and **assert**($x = *y$) do nothing if the equality holds, and aborts the program otherwise. These are used as guides for our type-based analysis described in the next section. Usually, they can be automatically inserted during the transformation from a surface language (like C) into our language; for example, **assert**($x = y$) is automatically inserted at the end of a let-expression **let** $x = y$ **in** $\cdots$. Separate pointer analyses may also be used to insert assertions; in general, insertion of more assertions makes our analysis more precise.

Remark 1. Notice that unlike in C (and like in functional languages), variables are immutable; they are initialized in let-expressions, and are never re-assigned afterwards. The declaration `int x = 1; ...` in C is expressed as:

$$\textbf{let } \&x = \textbf{malloc}() \textbf{ in } (*\&x \leftarrow 1; \cdots; \textbf{free}(\&x))$$

in our language. Here, $\&x$ is treated as a variable name.

Operational Semantics We assume that there is a countable set $\mathcal{H}$ of *heap addresses.* A run-time state is represented by a triple $\langle H, R, s\rangle$, where H is a mapping from a finite subset of $\mathcal{H}$ to $\mathcal{H} \cup \{\textbf{null}\}$, R is a mapping from a finite set of variables to $\mathcal{H} \cup \{\textbf{null}\}$. Intuitively, H models the heap memory, and R models local variables stored in stacks or registers. The set of *evaluation contexts* is defined by $E ::= [\,] \mid E; s$. We write $E[s]$ for the command obtained by replacing $[\,]$ in E with s.

Figure 2 shows main rules for the transition of run-time states: See Appendix A for the other rules. In the figure, $f\{x \mapsto v\}$ denotes the function f' such that $dom(f) = dom(f') \cup \{x\}$, $f'(x) = v$, and $f'(y) = f(y)$ for every $y \in dom(f) \setminus \{x\}$. $[x'/x]s$ denotes the command obtained by replacing x in s with x'. $\tilde{x}$ abbreviates a sequence $x_1, \dots, x_n$. In the rules for let-expressions, we require that $x' \notin dom(R)$. In the rule for malloc, the contents v of the allocated cell can be any value in $\mathcal{H} \cup \{\textbf{null}\}$. There are three kinds of run-time errors: **NullEx** for accessing null pointers, **Error** for illegal read/write/free operations on deallocated pointers, and **AssertFail** for assertion failures. The type system in this paper will prevent only the errors expressed by **Error**.

Note that the function call $f(x_1, \dots, x_n)$ is just replaced by the function's body. Thus, preprocessing is required to handle functions in C: A function call `x = f(y)` in C is simulated by $f(y, \&x)$ in our language (where $\&x$ is a variable name), and a C function definition `f(y) {s; return v;}` is simulated by:

$$f(y, r) = \textbf{let } \&y = \textbf{malloc}() \textbf{ in } (*\&y \leftarrow y; s; *r \leftarrow v; \textbf{free}(\&y)).$$

$$\frac{R(x) \in dom(H)}{\langle H, R, E[*x \leftarrow y]\rangle \longrightarrow_D \langle H\{R(x) \mapsto R(y)\}, R, E[\mathbf{skip}]\rangle}$$

$$\frac{R(x) \in dom(H) \cup \{\mathbf{null}\}}{\langle H, R, E[\mathbf{free}(x)]\rangle \longrightarrow_D \langle H \setminus \{R(x)\}, R, E[\mathbf{skip}]\rangle}$$

$$\frac{}{\langle H, R, E[\mathbf{let}\ x = y\ \mathbf{in}\ s]\rangle \longrightarrow_D \langle H, R\{x' \mapsto R(y)\}, E[[x'/x]s]\rangle}$$

$$\frac{}{\langle H, R, E[\mathbf{let}\ x = *y\ \mathbf{in}\ s]\rangle \longrightarrow_D \langle H, R\{x' \mapsto H(R(y))\}, E[[x'/x]s]\rangle}$$

$$\frac{h \notin dom(H)}{\langle H, R, E[\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s]\rangle \longrightarrow_D \langle H\{h \mapsto v\}, R\{x' \mapsto h\}, E[[x'/x]s]\rangle}$$

$$\frac{}{\langle H, R\{x \mapsto \mathbf{null}\}, E[\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\rangle \longrightarrow_D \langle H, R\{x \mapsto \mathbf{null}\}, E[s_1]\rangle}$$

$$\frac{f(\tilde{y}) = s \in D}{\langle H, R, E[f(\tilde{x})]\rangle \longrightarrow_D \langle H, R, E[[\tilde{x}/\tilde{y}]s]\rangle}$$

$$\frac{R(x) = H(R(y))}{\langle H, R, E[\mathbf{assert}(x = *y)]\rangle \longrightarrow_D \langle H, R, E[\mathbf{skip}]\rangle}$$

$$\frac{R(x) = \mathbf{null}}{\langle H, R, E[*x \leftarrow y]\rangle \longrightarrow_D \mathbf{NullEx}} \qquad \frac{R(y) = \mathbf{null}}{\langle H, R, E[\mathbf{let}\ x = *y\ \mathbf{in}\ s]\rangle \longrightarrow_D \mathbf{NullEx}}$$

$$\frac{R(x) \notin dom(H) \cup \{\mathbf{null}\}}{\langle H, R, E[*x \leftarrow y]\rangle \longrightarrow_D \mathbf{Error}} \qquad \frac{R(y) \notin dom(H) \cup \{\mathbf{null}\}}{\langle H, R, E[\mathbf{let}\ x = *y\ \mathbf{in}\ s]\rangle \longrightarrow_D \mathbf{Error}}$$

$$\frac{R(x) \notin dom(H) \cup \{\mathbf{null}\}}{\langle H, R, E[\mathbf{free}(x)]\rangle \longrightarrow_D \mathbf{Error}} \qquad \frac{R(y) \notin dom(H) \vee R(x) \neq H(R(y))}{\langle H, R, E[\mathbf{assert}(x = *y)]\rangle \longrightarrow_D \mathbf{AssertFail}}$$

Fig. 2. Main Transition Rules

Here, the malloc and free commands above correspond to the allocation and deallocation of a stack frame.

3 Type System

This section introduces a type system that prevents memory leaks, double frees, and illegal read/write operations.

3.1 Types

The syntax of types is given by:

$$\begin{aligned} &\tau \text{ (value types)} ::= \alpha \mid \tau\ \mathbf{ref}_f \mid \mu\alpha.\tau \\ &\sigma \text{ (function types)} ::= (\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_n) \end{aligned}$$

We often write $\top$ for $\mu\alpha.\alpha$. The metavariable f ranges over rational numbers in $[0, 1]$. f, called an *ownership*, represents both a capability and an obligation to read/write/free a pointer.

α is a type variable, which gets bound by the recursive type constructor $\mu\alpha$. The type $\tau\ \mathbf{ref}_f$ describes a pointer whose ownership is f, and also expresses

the constraint that the value obtained by dereferencing the pointer should be used according to τ. For example, if x has type $\top\ \mathbf{ref}_1\ \mathbf{ref}_1$, not only the pointer x but also the pointer stored in the target of the pointer x must be eventually deallocated through x.

Type $(\tau_1, \ldots, \tau_n) \rightarrow (\tau'_1, \ldots, \tau'_n)$ describes a function that takes n arguments. The types $\tau_1, \ldots, \tau_n, \tau'_1, \ldots, \tau'_n$ describe how ownerships on arguments are changed by the function: the type of the i-th argument is τ_i at the beginning of the function, and it is τ'_i at the end of the function.

The semantics of (value) types is defined as a mapping from the set $\{\mathtt{0}\}^*$ (the set of finite sequences of the symbol $\mathtt{0}$) to the set of rational numbers.

Definition 2 *The mapping* $[\![\cdot]\!]$ *from the set of closed types to* $\{\mathtt{0}\}^* \rightarrow [0,1]$ *is the least function that satisfies the following conditions.*

$$[\![\tau\ \mathbf{ref}_f]\!](\epsilon) = f \qquad [\![\tau\ \mathbf{ref}_f]\!](\mathtt{0}w) = [\![\tau]\!](w) \qquad [\![\mu\alpha.\tau]\!] = [\![[\mu\alpha.\tau/\alpha]\tau]\!]$$

(Here, the order between functions from S *to* T *is defined by:* $f \leq_{S\rightarrow T} g$ *if and only if* $\forall x \in S. f(x) \leq_T g(x)$*.) We write* $\tau \approx \tau'$*, if* $[\![\tau]\!] = [\![\tau']\!]$*.*

Note that $\top (= \mu\alpha.\alpha) \approx \mu\alpha.(\alpha\ \mathbf{ref}_0)$, and $\mu\alpha.\tau \approx [\mu\alpha.\tau/\alpha]\tau$.

We write $\mathbf{empty}(\tau)$ if all the ownerships in τ are 0. We say that a type τ is *well-formed* if $[\![\tau]\!](w) \geq c \times [\![\tau]\!](w\mathtt{0})$ for every $w \in \{\mathtt{0}\}^*$. Here, we let c be the constant $1/2$, but the type system given below remains sound as long as c is a positive (rational) number. In the rest of this paper, we consider only types that satisfy the well-formedness condition. See Remark 2 for the reason why the well-formedness is required.

3.2 Typing

A type judgment is of the form $\Theta; \Gamma \vdash s \Rightarrow \Gamma'$, where Θ is a finite mapping from (function) variables to function types, Γ and Γ' are finite mappings from variables to value types. Γ describes the ownerships held by each variable before the execution of s, while Γ' describes the ownerships after the execution of s. For example, we have $\Theta; x\!:\!\top\ \mathbf{ref}_1 \vdash \mathbf{free}(x) \Rightarrow x\!:\!\top\ \mathbf{ref}_0$. Note that a variable's type describes how the variable should be used, and not necessarily the status of the value stored in the variable. For example, $x : \top\ \mathbf{ref}_0$ does not mean that the memory cell pointed to by x has been deallocated; it only means that deallocating the cell through x (i.e., executing $\mathbf{free}(x)$) is disallowed. There may be another variable y of type $\tau\ \mathbf{ref}_1$ that holds the same pointer as x.

Typing rules are shown in Figure 3. $\tau \approx \tau_1 + \tau_2$ and $\tau_1 + \tau_2 \approx \tau'_1 + \tau'_2$ mean $[\![\tau]\!] = [\![\tau_1]\!] + [\![\tau_2]\!]$ and $[\![\tau_1]\!] + [\![\tau_2]\!] = [\![\tau'_1]\!] + [\![\tau'_2]\!]$ respectively. In the rule for assignment $*x \leftarrow y$, we require that the ownership of x is 1 (see Remark 2). The ownerships of τ' must be empty, since the value stored in $*x$ is thrown away by the assignment. The ownerships of y (described by τ) is divided into τ_1, which will be transferred to x, and τ_2, which remains in y.

In the rule for **free**, the ownership of x is changed from 1 to 0. τ must be empty, since x can no longer be dereferenced. In the rule for **malloc**, the ownership of x is 1 at the beginning of s, indicating that x must be deallocated. At the end of s, we require that the ownership of x is 0, since x goes out of the scope. Note that this requirement does not prevent the allocated memory cell from escaping the scope of the let-expression: For example, $\textbf{let } x = \textbf{malloc}() \textbf{ in } *y \leftarrow x$ allows the new cell to escape through variable y. The ownership of x is empty at the end of the let-expression, since the ownership has been transferred to y.

In the rule for dereferencing ($\textbf{let } x = *y \textbf{ in } \cdots$), the ownership of y must be non-zero. The ownerships stored in the target of the pointer y, described by τ, are divided into τ_1 and τ_2. At the end of the let-expression, the ownerships held by x must be empty (which is ensured by $\textbf{empty}(\tau_1'))$, since x goes out of scope.

In the rule for **null**, there is no constraint on the type of x, since x is a null pointer. In the rule for conditionals, any type may be assigned to x in the then-branch. Thanks to this, $\textbf{ifnull}(x) \textbf{ then skip else free}(x)$ is typed as follows.

$$\frac{\Theta; x : \top\ \textbf{ref}_0 \vdash \textbf{skip} \Rightarrow x : \top\ \textbf{ref}_0 \qquad \Theta; x : \top\ \textbf{ref}_1 \vdash \textbf{free}(x) \Rightarrow x : \top\ \textbf{ref}_0}{\Theta; x : \top\ \textbf{ref}_1 \vdash \textbf{ifnull}(x) \textbf{ then skip else free}(x) \Rightarrow x : \top\ \textbf{ref}_0}$$

The rules for assertions allow us to shuffle the ownerships held by the same pointers.

Remark 2. The well-formedness condition approximates the condition: $\forall w \in \{\mathtt{0}\}^*.(\llbracket \tau \rrbracket(w) = 0 \Rightarrow \llbracket \tau \rrbracket(w\mathtt{0}) = 0)$. Types that violate the condition (like $(\top\ \textbf{ref}_1)\ \textbf{ref}_0$) make the type system unsound. For example, consider the following command s (here, some let-expressions are inlined):

$$\textbf{let } y = x \textbf{ in } (*y \leftarrow \textbf{null}; \textbf{assert}(x = y); \textbf{free}(*x); \textbf{free}(x)).$$

If we ignore the well-formedness condition, we can derive $\Theta; x : (\top\ \textbf{ref}_1)\ \textbf{ref}_1 \vdash s \Rightarrow x : (\top\ \textbf{ref}_0)\ \textbf{ref}_0$ from $\Theta; x : (\top\ \textbf{ref}_1)\ \textbf{ref}_0, y : (\top\ \textbf{ref}_0)\ \textbf{ref}_1 \vdash s' \Rightarrow x{:}(\top\ \textbf{ref}_0)\ \textbf{ref}_0, y{:}(\top\ \textbf{ref}_0)\ \textbf{ref}_0$ where s' is the body of s. However, the judgment is semantically wrong: the memory cell referenced by $*x$ is not deallocated by s (see Figure 4). The well-formedness condition ensures that if a variable (say, x) has an ownership of a pointer (say, p) reachable from x, then the variable must hold a fraction of ownerships for all the pointers between x and p, so that the pointers cannot be updated through aliases.

Example 1. Recall the example in Section 1. The part $\textbf{let } y = *x \textbf{ in } (\texttt{freeall}(y); \textbf{free}(x))$ is typed as follows.

$$\frac{\dfrac{\Theta; x : \tau, y : \mu\alpha.(\alpha\ \textbf{ref}_1) \vdash \texttt{freeall}(y) \Rightarrow x : \tau, y : \top \quad \Theta; x : \tau, y : \top \vdash \textbf{free}(x) \Rightarrow x : \top, y : \top}{\Theta; x : (\mu\alpha.(\alpha\ \textbf{ref}_0))\ \textbf{ref}_1, y : \mu\alpha.(\alpha\ \textbf{ref}_1) \vdash (\texttt{freeall}(y); \textbf{free}(x)) \Rightarrow x : \top, y : \top}}{\Theta; x : \mu\alpha.(\alpha\ \textbf{ref}_1) \vdash \textbf{let } y = *x \textbf{ in } (\texttt{freeall}(y); \textbf{free}(x)) \Rightarrow x : \top}$$

Here, $\tau = (\mu\alpha.(\alpha\ \textbf{ref}_0))\ \textbf{ref}_1$ and $\Theta = \texttt{freeall} : (\mu\alpha.(\alpha\ \textbf{ref}_1)) \to (\top)$.

$$\frac{}{\Theta; \Gamma \vdash \mathbf{skip} \Rightarrow \Gamma} \qquad \frac{\Theta; \Gamma \vdash s_1 \Rightarrow \Gamma'' \qquad \Gamma'' \vdash s_2 \Rightarrow \Gamma'}{\Theta; \Gamma \vdash s_1; s_2 \Rightarrow \Gamma'}$$

$$\frac{\tau \approx \tau_1 + \tau_2 \qquad \mathbf{empty}(\tau')}{\Theta; \Gamma, x : \tau'\ \mathbf{ref}_1, y : \tau \vdash *x \leftarrow y \Rightarrow \Gamma, x : \tau_1\ \mathbf{ref}_1, y : \tau_2}$$

$$\frac{\mathbf{empty}(\tau)}{\Theta; \Gamma, x : \tau\ \mathbf{ref}_1 \vdash \mathbf{free}(x) \Rightarrow \Gamma, x : \tau\ \mathbf{ref}_0} \qquad \frac{\begin{array}{c}\Theta; \Gamma, x : \tau\ \mathbf{ref}_1 \vdash s \Rightarrow \Gamma', x : \tau'\ \mathbf{ref}_0 \\ \mathbf{empty}(\tau) \qquad \mathbf{empty}(\tau')\end{array}}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Theta; \Gamma, x : \tau_1, y : \tau_2 \vdash s \Rightarrow \Gamma', x : \tau_1' \\ \tau \approx \tau_1 + \tau_2 \qquad \mathbf{empty}(\tau_1')\end{array}}{\Theta; \Gamma, y : \tau \vdash \mathbf{let}\ x = y\ \mathbf{in}\ s \Rightarrow \Gamma'} \qquad \frac{\begin{array}{c}\Theta; \Gamma, x : \tau_1, y : \tau_2\ \mathbf{ref}_f \vdash s \Rightarrow \Gamma', x : \tau_1' \\ f > 0 \qquad \tau \approx \tau_1 + \tau_2 \qquad \mathbf{empty}(\tau_1')\end{array}}{\Theta; \Gamma, y : \tau\ \mathbf{ref}_f \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s \Rightarrow \Gamma'}$$

$$\frac{\Theta; \Gamma, x : \tau \vdash s \Rightarrow \Gamma', x : \tau'}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s \Rightarrow \Gamma'} \qquad \frac{\Theta; \Gamma, x : \tau' \vdash s_1 \Rightarrow \Gamma' \qquad \Theta; \Gamma, x : \tau \vdash s_2 \Rightarrow \Gamma'}{\Theta; \Gamma, x : \tau \vdash \mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \Rightarrow \Gamma'}$$

$$\frac{\tau_1 + \tau_2 \approx \tau_1' + \tau_2'}{\Theta; \Gamma, x : \tau_1, y : \tau_2 \vdash \mathbf{assert}(x = y) \Rightarrow \Gamma, x : \tau_1', y : \tau_2'}$$

$$\frac{\tau_1 + \tau_2 \approx \tau_1' + \tau_2'}{\Theta; \Gamma, x : \tau_1, y : \tau_2\ \mathbf{ref}_f \vdash \mathbf{assert}(x = *y) \Rightarrow \Gamma, x : \tau_1', y : \tau_2'\ \mathbf{ref}_f}$$

$$\frac{\Theta(f) = (\tilde{\tau}) \rightarrow (\tilde{\tau}')}{\Theta; \Gamma, \tilde{x} : \tilde{\tau} \vdash f(\tilde{x}) \Rightarrow \Gamma, \tilde{x} : \tilde{\tau}'} \qquad \frac{\Gamma \approx \Gamma_1 \qquad \Gamma' \approx \Gamma_1' \qquad \Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_1'}{\Theta; \Gamma \vdash s \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Theta; \tilde{x} : \tilde{\tau} \vdash s : \tilde{x} : \tilde{\tau}' \qquad \Theta(f) = \tilde{\tau} \rightarrow \tilde{\tau}' \\ (\text{for each } f(\tilde{x}) = s \in D) \\ dom(\Theta) = dom(D)\end{array}}{\vdash D : \Theta} \qquad \frac{\vdash D : \Theta \qquad \Theta; \emptyset \vdash s \Rightarrow \emptyset}{\vdash (D, s)}$$

Fig. 3. Typing Rules

Example 2. The following function destructively appends two lists p and q, and stores the result in $*r$.

$$\begin{array}{l}\mathsf{app}(p, q, r) = \mathbf{ifnull}(p)\ \mathbf{then}\ *r \leftarrow q \\ \qquad\qquad\qquad \mathbf{else}\ (*r \leftarrow p; (\mathbf{let}\ x = *p\ \mathbf{in}\ \mathsf{app}(x, q, p)); \mathbf{assert}(p = *r))\end{array}$$

app has type $(\tau_1, \tau_1, \top\ \mathbf{ref}_1) \rightarrow (\top, \top, \tau_1)$, where $\tau_1 = \mu\alpha.(\alpha\ \mathbf{ref}_1)$. The else-part is typed as follows.

$$\frac{\Theta; \Gamma_1 \vdash *r \leftarrow p \Rightarrow \Gamma_1 \qquad \dfrac{\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2 \quad \Theta; \Gamma_2 \vdash \mathbf{assert}(p = *r) \Rightarrow p : \top, q : \top, r : \tau_1}{\Theta; \Gamma_1 \vdash s; \mathbf{assert}(p = *r) \Rightarrow p : \top, q : \top, r : \tau_1}}{\Theta; \Gamma_1 \vdash *r \leftarrow p; s; \mathbf{assert}(p = *r) \Rightarrow p : \top, q : \top, r : \tau_1}$$

Here, $s = \mathbf{let}\ x = *p\ \mathbf{in}\ \mathsf{app}(x, q, p)$, and $\Theta, \Gamma_1, \Gamma_2$ are given by:

$$\Theta = \mathsf{app} : (\tau_1, \tau_1, \top\ \mathbf{ref}_1) \rightarrow (\top, \top, \tau_1)$$
$$\Gamma_1 = p : \tau_1, q : \tau_1, r : \top\ \mathbf{ref}_1 \qquad \Gamma_2 = p : \tau_1, q : \top, r : \top\ \mathbf{ref}_1$$

Fig. 4. Snapshots of the heap during the execution of the program in Remark 2. The lefthand side and the righthand side show the states before and after executing $*y \leftarrow$ **null** respectively. The rightmost cell will be leaked.

$\Theta; \Gamma_1 \vdash s \Rightarrow \Gamma_2$ is derived from:

$$\Theta; p : \top\ \mathbf{ref}_1, q : \tau_1, r : \top\ \mathbf{ref}_1, x : \tau_1 \vdash \mathsf{app}(x, q, p) \Rightarrow \Gamma_2, x : \top$$

3.3 Type Soundness

The soundness of our type system is stated as follows.

Theorem 1. *If* $\vdash (D, s)$*, then the following conditions hold.*

1. $\langle \emptyset, \emptyset, s \rangle \not\longrightarrow_D^* \mathbf{Error}$.
2. *If* $\langle \emptyset, \emptyset, s \rangle \longrightarrow_D^* \langle H, R, \mathbf{skip} \rangle$*, then* $H = \emptyset$.

The first condition means that there is no illegal read/write/free access to deallocated memory. The second condition means that well-typed programs do not leak memory.

To prove the soundness, we need to show an invariant condition that the total ownership for each (live) heap address is always 1. The invariant is expressed by the relation $\mathbf{Con}(\Gamma, H, R)$ defined below. $[\![H, v]\!]$ is the mapping from $\{\mathtt{0}\}^*$ to the set **Addr** of addresses, defined by:

$$[\![H, v]\!](\epsilon) = v \qquad [\![H, v]\!](w0) = \begin{cases} H(h) \text{ if } [\![H, v]\!](w) = h \in \mathit{dom}(H) \\ \mathbf{null} \ \ \text{otherwise} \end{cases}$$

$\mathbf{own}(H, v, \tau)(w)$ is defined by:

$$\mathbf{own}(H, v, \tau)(w) = \begin{cases} \{h \mapsto [\![\tau]\!](w)\} \text{ if } [\![H, v]\!](w) = h \neq \mathbf{null} \\ \emptyset \qquad\qquad\qquad \text{if } v = \mathbf{null} \end{cases}$$

We write $\mathbf{ownall}(H, v, \tau)$ for $\sum_{w \in 0^*} \mathbf{own}(H, v, \tau)(w)$.
(Here, the sum of two functions $f_0 + f_1$ is defined by $\forall x \in \mathit{dom}(f_0) \cap \mathit{dom}(f_1).(f_0 + f_1)(w) = f_0(w) + f_1(w)$ and $\forall x \in \mathit{dom}(f_i) \setminus \mathit{dom}(f_{1-i}).(f_0 + f_1)(w) = f_i(w)$. $\mathbf{ownall}(H, v, \tau)(h) = \infty$ if $\sum_{w \in 0^*} \mathbf{own}(H, v, \tau)(w)(h)$ does not converge.) Intuitively, $\mathbf{ownall}(H, v, \tau)$ describes all the ownerships for the heap H, held by the value v of type τ.

A triple (Γ, H, R) is *consistent*, written $\mathbf{Con}(\Gamma, H, R)$, if the following conditions hold:

1. $\mathit{dom}(\Gamma) = \mathit{dom}(R)$
2. Let $F = \sum_{x \in \mathit{dom}(\Gamma)} .\mathbf{ownall}(H, R(x), \Gamma(x))$. $F(h) = 1$ for any $h \in \mathit{dom}(H)$ and $F(h) = 0$ for any $h \notin \mathit{dom}(F) \setminus \mathit{dom}(H)$.

We write $\mathbf{empty}_R(\Gamma)$ if $R(x) = \mathbf{null}$ or $\mathbf{empty}(\Gamma(x))$ holds for every $x \in dom(\Gamma)$. The followings are key lemmas: See the extended version of this paper [4] for the proofs.

Lemma 1 (lack of immediate error). *If $\Theta; \Gamma \vdash s \Rightarrow \Gamma''$ and $\vdash D : \Theta$ with $\mathbf{Con}(\Gamma, H, R)$, then $\langle H, R, s\rangle \not\longrightarrow_D \mathbf{Error}$.*

Lemma 2 (lack of memory leak). *If $\mathbf{empty}_R(\Gamma)$ and $\mathbf{Con}(\Gamma, H, R)$, then $H = \emptyset$.*

Lemma 3 (type preservation). *Suppose $\Theta; \Gamma \vdash s \Rightarrow \Gamma''$ and $\vdash D : \Theta$, with $\mathbf{Con}(\Gamma, H, R)$. If $\langle H, R, s\rangle \longrightarrow_D^* \langle H', R', s'\rangle$ and $(dom(R')\setminus dom(R))\cap dom(\Gamma) = \emptyset$, then there exist Γ' and Γ_0 such that $\Theta; \Gamma' \vdash s' \Rightarrow \Gamma'', \Gamma_0$ and $\mathbf{Con}(\Gamma', H', R')$ with $\mathbf{empty}_{R'}(\Gamma_0)$.*

We can now prove Theorem 1.

Proof of Theorem 1 Suppose $\vdash (D, s)$. Then, $\vdash D : \Theta$ and $\Theta; \emptyset \vdash s \Rightarrow \emptyset$. We have $\mathbf{Con}(\emptyset, \emptyset, \emptyset)$. To show the first property of the theorem, assume $\langle \emptyset, \emptyset, s\rangle \longrightarrow_D^* \langle H, R, s'\rangle$. By Lemma 3, we have $\Gamma_1 \vdash s' \Rightarrow \Gamma', \Gamma_0$ with $\mathbf{Con}(\Gamma_1, H, R)$. By Lemma 1, we have $\langle H, R, s'\rangle \not\longrightarrow_D \mathbf{Error}$.

To show the second property of the theorem, assume that $\langle \emptyset, \emptyset, s\rangle \longrightarrow_D^* \langle H, R, \mathbf{skip}\rangle$. By Lemma 3, we have $\Gamma_1 \vdash \mathbf{skip} \Rightarrow \Gamma', \Gamma_0$ and $\mathbf{Con}(\Gamma_1, H, R)$ with $\mathbf{empty}_R(\Gamma', \Gamma_0)$ for some Γ_0 and Γ_1. By the typing rules, it must be the case that $\mathbf{empty}_R(\Gamma_1)$. By the condition $\mathbf{Con}(\Gamma_1, H, R)$ and Lemma 2, we have $H = \emptyset$ as required. $\square$

3.4 Type Inference

By Theorem 1, for verifying lack of memory-related errors in a program, it sufficies to check that the program is well-typed. For the purpose of automated type inference, we restrict the syntax of types to those of the form $(\mu\alpha.\alpha\ \mathbf{ref}_{f_1})\ \mathbf{ref}_{f_2}$. This restriction does not seem so restrictive for realistic programs: in fact, all the correct programs we have checked so far (including those given in this paper) are typable in the restricted type system.

Given a program written in our language, type inference proceeds as follows.

1. For each n-ary function f, prepare a type template

   $$\begin{array}{l}((\mu\alpha.\alpha\ \mathbf{ref}_{\eta_{f,1,1}})\ \mathbf{ref}_{\eta_{f,1,2}}, \ldots, (\mu\alpha.\alpha\ \mathbf{ref}_{\eta_{f,n,1}})\ \mathbf{ref}_{\eta_{f,n,2}})\\ \to ((\mu\alpha.\alpha\ \mathbf{ref}_{\eta'_{f,1,1}})\ \mathbf{ref}_{\eta'_{f,1,2}}, \ldots, (\mu\alpha.\alpha\ \mathbf{ref}_{\eta'_{f,n,1}})\ \mathbf{ref}_{\eta'_{f,n,2}}),\end{array}$$

   where $\eta_{f,i,j}$ and $\eta'_{f,i,j}$ are variables to denote unknown ownerships. Also, for each program point p and for each variable x live at p, prepare a type template $(\mu\alpha.\alpha\ \mathbf{ref}_{\eta_{p,x,1}})\ \mathbf{ref}_{\eta_{p,x,2}}$.
2. Generate linear inequalities on ownership variables based on the typing rules and the well-formedness condition.

3. Solve the linear inequalities. If the inequalities have a solution, the program is well-typed.

The number of ownership variables and linear inequalities is quadratic in the size of the input program. Since linear inequalities (over rational numbers) can be solved in time polynomial in the size of the inequalities, the whole algorithm runs in time polynomial in the size of the input program.

Remark 3. The main advatange of using fractions in the type system given above is the exisitence of a polynomial-time algorithm for solving constraints. Fractions also increase the expressive power of the type system (compared with 0-1 ownerships). For example, consider the following program:

```
let f(x,y) = (let z=*x in let w=*y in skip) in f(p, p)
```

Our type system accepts the program above by assigning to `f` the type $(\top\ \mathbf{ref}_{0.5}, \top\ \mathbf{ref}_{0.5}) \rightarrow (\top\ \mathbf{ref}_{0.5}, \top\ \mathbf{ref}_{0.5})$. The program is not typable if we have only 0-1 ownerships (note that non-zero ownerships must be passed to x and y).

4 Extensions and Limitations

We have so far considered a very simple language which has only pointers as values. This section discusses extensions of the type system for other language features (mainly of the C language).

It is straightforward to extend the type system to handle primitive types such as integers and floating points. For structures with n elements (for the sake of simplicity, assume that each element has the same size as a pointer), we can introduce a type of the form $(\tau_0 \times \cdots \times \tau_{n-1})\ \mathbf{ref}_{w_0,\ldots,w_{n-1},f}$ as the type of a pointer to a structure. Here, τ_i is the type of the i-th element of the structure, f denotes the obligation to deallocate the structure, and w_i is a capability to read/write the i-th element; thus, an ownership has been split into a free obligation and read/write capabilities. Then the rules for pointer dereference and pointer arithmetics are given by:

$$\frac{\begin{array}{c}\Theta; \Gamma, x : \tau_{0,x}, y : (\tau_{0,y} \times \tau_1 \times \cdots \times \tau_{n-1})\ \mathbf{ref}_{w_0,\ldots,w_{n-1},f} \vdash s \Rightarrow \Gamma', x : \tau' \\ w_0 > 0 \qquad \tau_0 \approx \tau_{0,x} + \tau_{0,y} \qquad \mathbf{empty}(\tau')\end{array}}{\Theta; \Gamma, y : (\tau_0 \times \tau_1 \times \cdots \times \tau_{n-1})\ \mathbf{ref}_{w_0,\ldots,w_{n-1},f} \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s \Rightarrow \Gamma'}$$

$$\frac{\begin{array}{c}\Theta; \Gamma, x : (\tau_{i,x} \times \cdots \times \tau_{n-1,x}, \top, \ldots, \top)\ \mathbf{ref}_{w_{i,x},\ldots,w_{n-1,x},0,\ldots,0,0}, \\ y : (\tau_{0,y} \times \cdots \times \tau_{n-1,y})\ \mathbf{ref}_{w_{0,y},\ldots,w_{n-1,y}}, f \vdash s \Rightarrow \Gamma', x : \tau_x \\ \forall j \in \{0, \ldots, i-1\}.(\tau_{j,y} \approx \tau_j \wedge w_j = w_{j,y}) \\ \forall j \in \{i, \ldots, n-1\}.(\tau_j \approx \tau_{j,y} + \tau_{j,x} \wedge w_j = w_{j,x} + w_{j,y}) \qquad \mathbf{empty}(\tau_x)\end{array}}{\Theta; \Gamma, y : (\tau_0 \times \cdots \times \tau_{n-1})\ \mathbf{ref}_{w_0,\ldots,w_{n-1},f} \vdash \mathbf{let}\ x = y + i\ \mathbf{in}\ s \Rightarrow \Gamma'}$$

For example, consider the function `delnext` in Figure 5. It takes a doubly-linked list as shown in Figure 6, and deletes the next element of p. The function is given the type $(\tau_P \times \tau_N)\ \mathbf{ref}_{1,1,1} \rightarrow (\tau_P \times \tau_N)\ \mathbf{ref}_{1,1,1}$, where $\tau_P =$

```
fun delnext(p) =
  let nextp = p+1 in let next = *nextp in
  let nnp = next+1 in nn = *nnp in
     *nn <- p; *nextp <- nn; assert(nnp=next+1);
     free(next); assert(nextp=p+1);
```

Fig. 5. A function manipulating a doubly-linked list.

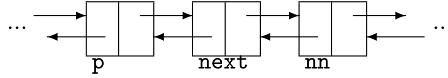

Fig. 6. A doubly-linked list given as an input of `delnext`. The cell `next` is removed and deallocated.

$\mu\alpha.((\alpha \times \top)\ \mathbf{ref}_{1,1,1})$ and $\tau_N = \mu\alpha.((\top \times \alpha)\ \mathbf{ref}_{1,1,1})$. The type $(\tau_P \times \tau_N)\ \mathbf{ref}_{1,1,1}$ means that the first element of `p` holds the capabilities and obligations on the cells reachable through the backward pointers, and the second element holds those on the cells reachable through the forward pointers. The types of variables at each program point are given in Appendix B.

An array of primitive values can be treated as one big reference cell, assuming that array boundary errors are prevented by other methods (such as dynamic checks or static analyses). At this moment, however, we do not know how to deal with arrays of pointers.

A dereference of a function pointer in C can be replaced with a non-deterministic choice of the functions it may point to, by using a standard flow analysis. It is not clear, however, how to deal with higher-order functions in functional languages, especially those stored in reference cells.

Cast operations can be handled in a conservative manner. For example, a pointer to a structure of type $(\tau_0 \times \cdots \times \tau_{n-1})\ \mathbf{ref}_{w_0,\ldots,w_{n-1},f}$ can be casted to a pointer of type $(\tau_0 \times \cdots \times \tau_{m-1})\ \mathbf{ref}_{w_0,\ldots,w_{m-1},f'}$ (if $m \leq n$). An integer can be casted to a pointer with 0 ownership (but it is useless).

Besides arrays of pointers and higher-order functions, one of the major limitations of our type system is that it cannot deal with cyclic structures well. For example, consider a cyclic list shown on the lefthand side of Figure 7. The only type that can be assigned to cycles of an arbitrary length is $\top$: Notice that if we assign $\mu\alpha.(\alpha\ \mathbf{ref}_f)$ to the cycle, then an ownership f can be extracted for *each path* (e.g., ϵ, `00`, `0000`,$\ldots$ for the cell on the lefthand side). We have to maintain the invariant that $f + f + f + \cdots \leq 1$, so that f must be 0. Thus, although a cyclic list can be constructed, it is useless as there is no ownership. Note, however, that this limitation does not apply to the case of doubly-linked lists discussed above, since cycles in doubly-linked lists are formed by two kinds of pointers; forward and backward pointers.

If we give up detecting illegal read operations (but still prevent illegal write/free operations), then cyclic lists can be constructed and used in a restricted manner, as shown on the righthand side of Figure 7. Here, y holds the ownerships of all the cells, but x can use the cylic list in a read-only manner.

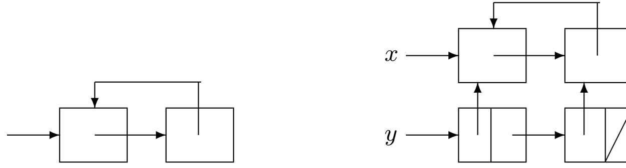

Fig. 7. Cyclic Lists

benchmark	IC	TOTAL	GEN_LP	SOLVE_LP	NASSERT
ll-app	36	269.4	144.1	124.9	8
ll-reverse	29	139.9	70.79	68.64	4
ll-search	27	115.7	56.66	58.73	5
ll-merge	40	303.6	171.0	59.18	7
dl-insert	53	715.8	387.4	327.9	14
dl-delete	47	505.9	271.5	234.0	10
bt-insert	41	256.0	138.0	117.3	6

Fig. 8. Benchmark result. Columns IC, TOTAL, GEN_LP, SOLVE_LP and NASSERT are the number of instructions, total execution time (msec), execution time for reducing constraints into linear inequalities (msec), execution time for solving linear inequalities (msec) and the number of manually-inserted assertions.

5 Experiments

We have implemented a prototype verifier for an extension of the language of Section 2 with cons cells (structures of size 2, as discussed in Section 4), and tested it for programs manipulating lists, doubly-linked lists and trees. The implementation, written in Objective Caml, is available at `http://www.kb.ecei.tohoku.ac.jp/~suenaga/mallocfree/`. As a linear programming solver, we used GLPK 4.15 wrapped by ocaml-glpk 0.1.5.

Figure 8 shows the result of the experiments. We used a machine with an Intel Core 2 1.06GHz CPU, 2MB cache and 2GB memory. The programs used for the experiments are:

- **ll-app**, **ll-reverse** and **ll-search**: create lists, perform specific operations on the lists (append for **ll-app**, reverse for **ll-reverse**, and list search for ll-search), and deallocate the lists.
- **dl-insert** and **dl-remove**: create doubly-linked lists, insert or delete a cell, and deallocating the doubly-linked lists.
- **bt-insert**: creates a binary tree, inserts a node, and then deallocates the tree.

All the programs have been verified correctly. It is worth noting that the programs manipulating doubly-linked lists could be verified. The benchmark results show that our analysis is reasonably fast, despite that our current implementation is rather naive.

Limitations of the current implementation are as follows.

– Requirement for insertion of appropriate assertions. Judging from the experiments, however, an extremely simple intra-procedural analysis is sufficient for inserting the assertions automatically, except the case for doubly-linked lists, for which assertions on the structure of doubly-linked lists (like `p->next->prev = p`) are required.
– Poor error messages. Given an ill-typed program, the current system does produce some diagnostic information to indicate a possible location of a bug, but more helpful messages are needed for end-users.

6 Related Work

There are a lot of studies and tools to detect or prevent memory-related errors. They are classified into static and dynamic analyses. We focus here on static analysis techniques. In general, dynamic analysis can only detect errors that occur in particular runs.

We have already discussed Heine and Lam's work [3] in Section 1. They use polymorphism on ownerships to make the analysis context-sensitive. The same technique would be applicable to our type system. Dor, Rodeh, and Sagiv [2] use shape analysis techniques to verify lack of memory-related errors in list-manipulating programs. Unlike ours, their analysis can also detect null-pointer dereferences. Advantages of our type system over their analysis are probably efficiency and simplicity. It is not clear whether their analysis can be easily extended to handle procedure calls and data structures (e.g., trees and doubly-linked lists) other than singly-linked lists in a efficient manner. Orlovich and Rugina [5] proposed a backward dataflow analysis to detect memory leaks. Their analysis does not detect double-frees and illegal accesses to deallocated memory. Xie and Aiken [9] use a SAT solver to detect memory leaks. Their analysis is unsound for loops and recursion.

Other potential advantages of our type-based approach are: By allowing programmers to declare ownership types, they may serve as good specifications of functions or modules, and also enhance modular verification. Our approach can be probably extended to deal with multi-threaded programs, along the line of previous work using fractional capabilities [1, 7, 8].

A main limitation of our approach is that our type system cannot handle cycles (recall the discussion in Section 4) and value-dependent (or, path-sensitive) behaviors. In practice, therefore, a combination of our technique with other techniques would be useful (e.g., shape analysis or separation logic for handling cycles, SAT-solver-based analysis [9] for handling value-dependent behaviors).

Boyland [1] is the pioneer who introduced fractions in the context of type-based program analysis. He used fractional permissions (for read/write operations) to prevent race conditions in multi-threaded programs. Terauchi [7, 8] later found another advantage of using fractions: inference of fractional permissions (or capabilities) can be reduced to a linear programming problem (rather than integer linear programming), which can be solved in polynomial time. The type system of this paper mainly exploits the latter advantage. In their work [1, 7, 8],

a fractional capability is assigned to an abstract location (often called a region), while our type system assigns a fractional ownership to each access path from a variable. The former approach is not suitable for the purpose of our analysis: for example, all the elements in a list are abstracted to the same location, so that a separate ownership cannot be assigned to each element of the list.

Swamy et al. [6] developed a language with safe manual memory management. Unlike C, their language requires programmers to provide various annotations (such as whether a pointer is aliased or not).

7 Conclusion

We have proposed a new type system that guarantees lack of memory-related errors. The type system is based on the notion of fractional ownerships, and is equipped with a polynomial-time type inference algorithm. The type system is quite simple (especially compared with previous techniques for analyzing similar properties), yet it can be used to verify tricky pointer-manipulating programs.

Acknowledgment

We would like to especially thank Toshihiro Wakatake and Kensuke Mano. Some of the ideas in this paper came from discussions with them. We would also like to thank members of our research group for comments and discussions.

References

1. J. Boyland. Checking interference with fractional permissions. In *Proceedings of SAS 2003*, volume 2694 of *LNCS*, pages 55–72. Springer-Verlag, 2003.
2. N. Dor, M. Rodeh, and S. Sagiv. Checking cleanness in linked lists. In *Proceedings of SAS 2000*, volume 1824 of *LNCS*, pages 115–134. Springer-Verlag, 2000.
3. D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *Proc. of PLDI*, pages 168–181, 2003.
4. N. Kobayashi and K. Suenaga. Fractional ownerships for safe memory deallocation. An extended version, available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/malloc.pdf`, 2008.
5. M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Proceedings of SAS 2006*, volume 4134 of *LNCS*, pages 405–424. Springer-Verlag, 2006.
6. N. Swamy, M. W. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.
7. T. Terauchi. Checking race freedom via linear programming. In *Proc. of PLDI*, pages 1–10, 2008.
8. T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *ACM Trans. Prog. Lang. Syst.*, 30(5), 2008.
9. Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 115–125, 2005.

Appendix

A The Transition Rules Omitted in Section 2

$$\frac{}{\langle H, R, E[\mathbf{skip}; s]\rangle \longrightarrow_D \langle H, R, E[s]\rangle}$$

$$\frac{x' \notin dom(R)}{\langle H, R, E[\mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s]\rangle \longrightarrow_D \langle H, R\{x' \mapsto \mathbf{null}\}, E[[x'/x]s]\rangle}$$

$$\frac{R(x) \neq \mathbf{null}}{\langle H, R, E[\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2]\rangle \longrightarrow_D \langle H, R, E[s_2]\rangle}$$

$$\frac{R(x) = R(y)}{\langle H, R, E[\mathbf{assert}(x = y)]\rangle \longrightarrow_D \langle H, R, E[\mathbf{skip}]\rangle}$$

$$\frac{R(x) \neq R(y)}{\langle H, R, E[\mathbf{assert}(x = y)]\rangle \longrightarrow_D \mathbf{AssertFail}}$$

B Typing for `delnext`

For the function `delnext` discussed in Section 4, the following types are given at each program point. ($\tau_1 \times \tau_2\ \mathbf{ref}_{w_1,w_2,f}$ should be read $(\tau_1 \times \tau_2)\ \mathbf{ref}_{w_1,w_2,f}$.)

```
fun delnext(p) =
```

$p : \tau_P \times \tau_N\ \mathbf{ref}_{1,1,1}$

```
  let nextp = p+1 in
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \tau_N \times \top\ \mathbf{ref}_{1,0,0}$

```
  let next = *nextp in
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \top \times \top\ \mathbf{ref}_{1,0,0}, \mathtt{next} : \tau_N$

```
  let nnp = next+1 in
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \top \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nnp} : \tau_N \times \top\ \mathbf{ref}_{1,0,0}$

```
  let nn = *nnp in
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \top \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{1,0,0}, \mathtt{nn} : \tau_N$

```
  *nn <- p;
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \top \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{1,0,0}, \mathtt{nn} : \tau_N$

```
  *nextp <- nn
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \tau_N \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{1,0,0}, \mathtt{nn} : \top$

```
  assert(nnp=next+1);
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \tau_N \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{1,1,1}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{0,0,0}, \mathtt{nn} : \top$

```
  free(next)
```

$p : \tau_P \times \top\ \mathbf{ref}_{1,0,1}, \mathtt{nextp} : \tau_N \times \top\ \mathbf{ref}_{1,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{0,0,0}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{0,0,0}, \mathtt{nn} : \top$

```
  assert(nextp=p+1);
```

$p : \tau_P \times \tau_N\ \mathbf{ref}_{1,1,1}, \mathtt{nextp} : \top \times \top\ \mathbf{ref}_{0,0,0},$
$\mathtt{next} : \top \times \top\ \mathbf{ref}_{0,0,0}, \mathtt{nnp} : \top \times \top\ \mathbf{ref}_{0,0,0}, \mathtt{nn} : \top$