Fractional Ownerships for Safe Memory Deallocation

Kohei Suenaga and Naoki Kobayashi

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 main idea is to augment pointer types with fractional ownerships, which express both capabilities and obligations to access or deallocate memory cells. By assigning an ownership to each pointer type constructor (rather than to a variable), our type system can properly reason about list/tree-manipulating programs. Furthermore, 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 C programs.

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-based method for static verification of lack of such memory-related errors. More precisely, we construct a type system that guarantees that well-typed programs do not suffer from memory leaks (forgetting to deallocate memory cells), double frees (deallocating memory cells more than once), and illegal read/write accesses to deallocated memory. We then construct a polynomial-time type inference algorithm, so that programs can be verified without any type annotations.

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 a non-zero ownership less than 1, one has to eventually combine it

** Suenaga's Current Affiliation: IBM Research with other ownerships to obtain an ownership of 1, to fulfill the obligation to deallocate the pointer).

Ownerships are also used in Heine and Lam's static analysis for detecting memory leaks [2], although their ownerships range over integer values $\{0, 1\}$. The most important deviation from their system is that our type system assigns an ownership to each *pointer type constructor*, rather than to a variable. For example, $\mathbf{int}\ \mathbf{ref}_1\ \mathbf{ref}_1$ is the type of a pointer to a 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 an integer, such that only the first pointer can be read/written, and must be deallocated. The type $\mu\alpha.(\alpha\ \mathbf{ref}_1)$ (where $\mu\alpha.\tau$ is a recursive type) describes a pointer to a list structure shown in Figure 1, where the pointer holds the ownerships of all the pointers reachable from it. This allows us to properly reason about list- and tree-manipulating programs, unlike Heine and Lam's analysis.

For example, consider the following program, written in an ML-like language (but with memory deallocation primitive `free`).

`fun freeall(x) =`	`freeall` $: \mu\alpha.(\alpha\ \mathbf{ref}_1) \rightarrow \mu\alpha.(\alpha\ \mathbf{ref}_0)$
`  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, and deallocates all the pointers reachable from `x`. The righthand 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.) 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 $\mu\alpha.(\alpha\ \mathbf{ref}_0)$ 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

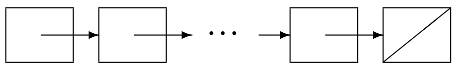

Fig. 1. List-like structure

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.

Thanks to the use of fractional ownerships, the type inference problem can be reduced to a linear programming problem over rational numbers, which can be solved in polynomial time. If ownerships are integer-valued, the type inference problem is reduced to an integer linear programming problem, which is NP-hard.[1] Furthermore, fractional ownerships make the type system more expressive: see Example 3 in Section 3.

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

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, including Ueda's work [3] on fractional capabilities for GHC, to which our type system seems closely related, despite the differences of the target languages. Section 7 concludes the paper.

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.

[1] With the recent advance of SAT solvers, it may still be the case that the integer linear programming problem generated by the type inference can be solved efficiently in practice; that may be left as a subject for further investigation.

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 $\mathbf{let}\ x = e\ \mathbf{in}\ s$ evaluates e, binds x to the value of e, and executes s. The expression $\mathbf{malloc}()$ allocates a memory cell and returns a pointer to it. The expression **null** denotes a null pointer. $*y$ dereferences the pointer y. The command $\mathbf{ifnull}(x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$ 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 $\mathbf{let}\ y = x\ \mathbf{in}\ f(x, y)$.) There is no return value of a function call; values can be returned only by reference passing. The commands $\mathbf{assert}(x = y)$ and $\mathbf{assert}(x = *y)$ do nothing if the equality holds, and aborts the program otherwise. These are introduced to simplify the type system and the proof of its soundness in Section 3. Usually, assert commands can be automatically inserted during the transformation from a surface language (like C) into our language; for example, $\mathbf{assert}(x = y)$ is automatically inserted at the end of a let-expression $\mathbf{let}\ x = y\ \mathbf{in}\ \cdots$. 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:

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

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 $\mathcal{H} \cup \{\mathbf{null}\}$, R is a mapping from a finite set of variables to $\mathcal{H} \cup \{\mathbf{null}\}$. 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 the transition rules for run-time states. In the figure, $f\{x \mapsto v\}$ denotes the function f' such that $\mathit{dom}(f) = \mathit{dom}(f') \cup \{x\}$, $f'(x) = v$, and $f'(y) = f(y)$ for every $y \in \mathit{dom}(f) \setminus \{x\}$. $[x'/x]s$ 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 $x' \notin \mathit{dom}(R)$. In the rule for malloc, the contents v of the allocated cell can be any value in $\mathcal{H} \cup \{\mathbf{null}\}$. 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**. In the rules for assertions on the last line, the relation $H, R \models P$ is defined by: $H, R \models x = y$ iff $R(x) = R(y)$, and $H, R \models x = *y$ iff $R(x) = H(R(y))$.

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

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

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

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

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

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

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

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

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

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

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

$$\frac{R(x) \notin \mathit{dom}(H) \cup \{\mathbf{null}\}}{\langle H, R, E[\mathbf{free}(x)]\rangle \longrightarrow_D \mathbf{Error}} \qquad \frac{f(\tilde{y}) = s \in D}{\langle H, R, E[f(\tilde{x})]\rangle \longrightarrow_D \langle H, R, E[[\tilde{x}/\tilde{y}]s]\rangle}$$

$$\frac{H, R \models P}{\langle H, R, E[\mathbf{assert}(P)]\rangle \longrightarrow_D \langle H, R, E[\mathbf{skip}]\rangle} \qquad \frac{H, R \not\models P}{\langle H, R, E[\mathbf{assert}(P)]\rangle \longrightarrow_D \mathbf{AssertFail}}$$

Fig. 2. Transition Rules

Note that the function call $f(x_1, \ldots, x_n)$ 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:

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

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:

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

We often write $\top$ for $\mu\alpha.\alpha$, which describes pointers carrying no ownerships. The metavariable f ranges over rational numbers in $[0, 1]$. It is called an *ownership*, and 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 $(\tau_1, \ldots, \tau_n) \to (\tau'_1, \ldots, \tau'_n)$ 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. Intuitively, the type $[\![\,\tau\,]\!](\epsilon)$ of a pointer represents the ownership for the memory cell directly pointed to by the pointer, and $[\![\,\tau\,]\!](\mathtt{0}^k)$ represents the ownership for the memory cell reached by k hops of pointer traversals. (If the language is extended with structures with n elements as discussed in Section 4, $[\![\,\tau\,]\!]$ should be extended to a mapping from $\{0, \ldots, , n-1\}^*$ to the set of rational numbers.)

Definition 2 *The mapping $[\![\,\cdot\,]\!]$ from the set of closed types to $\{\mathtt{0}\}^* \to [0, 1]$ 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: $f \leq_{S\to T} g$ 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, $\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ *y \leftarrow x$ 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 ($\mathbf{let}\ x = *y\ \mathbf{in}\ \cdots$), 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 $\mathbf{empty}(\tau_1')$), 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, $\mathbf{ifnull}(x)\ \mathbf{then\ skip\ else\ free}(x)$ is typed as follows.

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

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

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

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

If we ignore the well-formedness condition, we can derive $\Theta; x:(\top\ \mathbf{ref}_1)\ \mathbf{ref}_1 \vdash s \Rightarrow x:(\top\ \mathbf{ref}_0)\ \mathbf{ref}_0$ from $\Theta; x:(\top\ \mathbf{ref}_1)\ \mathbf{ref}_0, y:(\top\ \mathbf{ref}_0)\ \mathbf{ref}_1 \vdash s' \Rightarrow x{:}(\top\ \mathbf{ref}_0)\ \mathbf{ref}_0, y{:}(\top\ \mathbf{ref}_0)\ \mathbf{ref}_0$ where s' is the body of s. However, the judgment is semantically wrong: the memory cell referenced by $*x$ is not deallocated by s

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

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

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

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

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

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

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

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

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

Fig. 3. Typing Rules

(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 **let** $y = *x$ **in** ($\texttt{freeall}(y)$; **free**(x)) is typed as follows.

$$\frac{\dfrac{\Theta; x:\tau, y:\mu\alpha.(\alpha\ \mathbf{ref}_1) \vdash \texttt{freeall}(y) \Rightarrow x:\tau, y:\top \qquad \Theta; x:\tau, y:\top \vdash \mathbf{free}(x) \Rightarrow \Theta_0}{\Theta; x:(\mu\alpha.(\alpha\ \mathbf{ref}_0))\ \mathbf{ref}_1, y:\mu\alpha.(\alpha\ \mathbf{ref}_1) \vdash (\texttt{freeall}(y); \mathbf{free}(x)) \Rightarrow \Theta_0}}{\Theta; x:\mu\alpha.(\alpha\ \mathbf{ref}_1) \vdash \mathbf{let}\ y = *x\ \mathbf{in}\ (\texttt{freeall}(y); \mathbf{free}(x)) \Rightarrow x:\top}$$

Here, $\tau = (\mu\alpha.(\alpha\ \mathbf{ref}_0))\ \mathbf{ref}_1$, $\Theta = \texttt{freeall} : (\mu\alpha.(\alpha\ \mathbf{ref}_1)) \rightarrow (\top)$, and $\Theta_0 = x:\top, y:\top$.

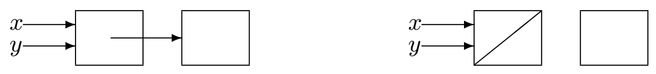

Fig. 4. Snapshots of the heap during the execution of the program in Remark 2. The lefthand side and the righthand side show the states before and after executing $*y \leftarrow$ **null** respectively. The rightmost cell will be leaked.

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

$$\begin{aligned}\mathsf{app}(p,q,r) = {} & \textbf{ifnull}(p)\ \textbf{then}\ *r \leftarrow q \\ & \textbf{else}\ (*r \leftarrow p; (\textbf{let}\ x = *p\ \textbf{in}\ \mathsf{app}(x,q,p)); \textbf{assert}(p = *r))\end{aligned}$$

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

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

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

$$\begin{aligned}&\Theta = \mathsf{app} : (\tau_1, \tau_1, \top\ \textbf{ref}_1) \rightarrow (\top, \top, \tau_1) \\ &\Gamma_1 = p : \tau_1, q : \tau_1, r : \top\ \textbf{ref}_1 \qquad \Gamma_2 = p : \tau_1, q : \top, r : \top\ \textbf{ref}_1\end{aligned}$$

Example 3. Consider the following functions f and g:

$$\begin{aligned}f(x) &= \textbf{let}\ y = x\ \textbf{in}\ g(x,y); \textbf{assert}(x = y) \\ g(x,y) &= \textbf{let}\ z = *x\ \textbf{in}\ \textbf{let}\ w = *y\ \textbf{in}\ \textbf{skip}\end{aligned}$$

Then, f and g can be given types $\top\ \textbf{ref}_1 \rightarrow \top\ \textbf{ref}_1$ and $(\top\ \textbf{ref}_{0.5}, \top\ \textbf{ref}_{0.5}) \rightarrow (\top\ \textbf{ref}_{0.5}, \top\ \textbf{ref}_{0.5})$. Without fractional types, f is not typable because the ownership of x cannot be split into the first and second arguments of g. Although the situation above is not likely to occur so often in actual sequential programs, we expect that fractional ownerships will play a more fundamental role in a multi-threaded setting, where ownerships for shared variables need to be split for multi-threads.

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. $\langle \emptyset, \emptyset, s \rangle \not\longrightarrow^*_D$ **Error**.
2. *If* $\langle \emptyset, \emptyset, s \rangle \longrightarrow^*_D \langle H, R, \textbf{skip} \rangle$*, 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. See the longer version [4] for the proof.

3.4 Type Inference

By Theorem 1, verification of lack of memory-related errors is reduced to type inference. 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 makes the type system slightly less expressive, by precluding types like $\mu\alpha.(\alpha\ \mathbf{ref}_{0.5}\ \mathbf{ref}_{0.7})$. The restriction, however, 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

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

   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.

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:

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

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

Fig. 5. A function manipulating a doubly-linked list and its typing

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

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 $\tau_P=\mu\alpha.((\alpha\times\top)\ \mathbf{ref}_{1,1,1})$ 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.

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.

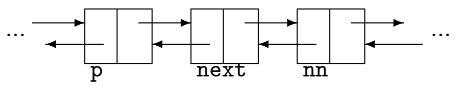

Fig. 6. A doubly-linked list given as an input of `delnext`. The cell `next` is removed and deallocated.

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. The only type that can be assigned to cyclic lists of 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`,... 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, since cycles in doubly-linked lists are formed by two kinds of pointers; forward and backward pointers (recall the example in Figure 5). In order to handle cyclic lists, we need to extend pointer types to $\tau\ \mathbf{ref}_f^P$, which means that the pointer is an element of the set P or has an ownership f. The pointer type $\tau\ \mathbf{ref}_f$ is then just a special case of $\tau\ \mathbf{ref}_f^{\{\mathtt{NULL}\}}$.

5 Preliminary Experiments

We have implemented a prototype verifier for C programs, and tested it for several programs. 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. The implementation is based on the type system described in Section 3, with the extension for structures discussed in Section 4.

The limitations of the current implementation are: (i) Unsound treatment of arrays of pointers (recall the discussion in Section 4): An array of pointers is handled as an array of size 1; (ii) Poor error reporting: when a program is ill-typed, the current system does produce some diagnostic information to indicate a possible location of a bug, but it is probably incomprehensible for end-users; (iii) Lack of support of several C statements: for example, a function call of the form `f(&x->f)` has to be manually rewritten to a sequence of statements `p = &x->f; f(p); assert(&x->f, p)`; and (iv) Need for manual insertion of assertions ($\mathbf{assert}(x = y)$ and $\mathbf{assert}(x = *y)$ in Section 2).

benchmark	LOC	Time (total)	Time (LP)	NASSERT	SIZE_LP	NVAR
ll-app	62	0.09	0.002	2	196	403
ll-reverse	67	0.10	0.002	2	217	430
ll-search	70	0.09	0.002	1	192	398
ll-merge	69	0.10	0.003	3	227	460
dl-insert	80	0.14	0.011	9	806	954
dl-delete	87	0.15	0.014	8	919	1134
bt-insert	64	0.09	0.003	0	188	479
authfd.c	6463	0.44	0.07	16	739	5408
cdrom.c	13429	26.49	19.85	14	35197	47185

Fig. 7. Benchmark result. The meaning of each column is as follows. LOC: the number of lines of code. Time (total): total execution time (sec). Time (LP): execution time for solving linear inequalities (sec). NASSERT: the number of manually-inserted assertions. SIZE_LP: the number of linear inequality constraints (after preprocessing of trivial constraints). NVAR: the number of variables contained in generated linear inequalities.

Figure 7 shows the result of the experiments. We used a machine with an Intel(R) Xeon(R) 3.00Hz CPU, 4MB cache and 8GB memory. The programs used for the experiments are described as follows:

– **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 deallocate the doubly-linked lists.

– **bt-insert** constructs a binary tree, performs an insertion, and then deallocates the tree.

– **authfd.c** is a preprocessed file taken from `openssh-5.2p1`. (A large part of the preprocessed file consists of type declarations; the rest of the code consists of about 600 lines.)

– **cdrom.c** is a *fragment*[2] of Linux device driver `/drivers/cdrom/cdrom.c`.

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, even for **cdrom.c**, which consists of 13K LOC.

Note that only 14 assertions were required for **cdrom.c**. (Thus, although the microbenchmarks used in this experiment are quite small, they are actually tricky programs.) All of those assertions were of the form `assert(p=NULL)`, except the following assertion, which asserts that `prev` points to the previous element of `cdi` in a singly-linked list.

```
while (cid && cdi != unreg){
  assert(cdi, prev->next); prev = cdi; cdi = prev->next;}
```

[2] For the rest of the driver code, we have not yet checked whether it is typable by appropriate insertion of assertion commands.

This suggests that most of the assertions manually inserted in the experiments above can be automatically inferred by a rather straightforward intra procedural analysis like the one mentioned in Section 2 (except those for doubly-linked lists, which require knowledge of the data structure invariant).

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. Here we focus on static analysis techniques.

We have already discussed Heine and Lam's work [2] in Section 1. They use polymorphism on ownerships to make the analysis context-sensitive, which would be applicable to our type system. Dor, Rodeh, and Sagiv [5] 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 the simplicity and efficiency. 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 an efficient manner. Orlovich and Rugina [6] 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 [7] use a SAT solver to detect memory leaks. Their analysis is unsound for loops and recursion. Boyapati et al. [8] uses ownership types for safe memory management for real-time Java, but their target is region-based memory management, and assume explicit type annotations. Swamy et al. [9] also 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).

Yang et al. [10, 11] applied separation logic to automated verification of pointer safety in systems code. The efficiency of their verification method [10] seems comparable to ours. However, they do not deal with doubly linked lists ([10], Section 2).[3] Like our technique, their tool cannot handle arrays of pointers.

Other potential advantages of our type-based approach are: (i) By allowing programmers to declare ownership types, they may serve as good specifications of functions or modules, and also enhance modular verification, (ii) Our approach can probably be extended to deal with multi-threaded programs, along the line of previous work using fractional capabilities [1, 12, 13], and (iii) There is a clear proof of soundness of the analysis, based on a standard technique for proving type soundness (see the longer version [4]). A main limitation of our approach is that our type system cannot properly 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.

Technically, our type system is based on the notion of ownerships and fractional permissions/capabilities. Although there are many pieces of previous work

[3] Berdine et al. [11] can handle doubly linked lists, but the verification tool is much slower according to their experimental results.

that use ownerships and fractional capabilities, our work is original in the way they are integrated into a type system (in particular, pointer types that can represent an ownership of each memory cell reachable from a pointer, and typing rules that allow automated inference of such pointer types). The idea of fractional capabilities can be traced back to Ueda's work [3] on GHC (a concurrent logic programming language). He extended input/output modes to capabilities ranging over $[-1, 1]$, and used them to guarantee that there is no leakage of memory cells for storing constructors. Our type system actually seems closer to his system than to other later fractional capability systems [1, 12, 13]. In particular, his system assigns a capability (or, an ownership in our terminology) to each node reachable from a variable (just as our type system assigns an ownership to each pointer reachable from a variable), and the unification constraint $X = Y$ between variables plays a role similar to our assert commands. Nevertheless, the details are different: our ownerships range over $[0, 1]$ while theirs range over $[-1, 1]$, and both the well-formedness conditions on types, and the constraints imposed by the type systems are different. This seems to come from the differences in the language primitives: sequential vs concurrent compositions, and pointers vs unification variables. Note that, for example, updating a pointer does not consume any capability, while writing to a unification variable consumes a write capability.

Boyland [1] used fractional permissions (for read/write operations) to prevent race conditions in multi-threaded programs. Terauchi [12, 13] 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, 12, 13], 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. More specifically, in their work [1, 12, 13], a pointer type is represented as $\tau\ \mathbf{ref}_{\rho_i}$ with a separate map $\{\rho_1 \mapsto f_1, \ldots, \rho_n \mapsto f_n\}$ from abstract locations to fractions, whereas our pointer type $\tau\ \mathbf{ref}_f$ may be regarded as a kind of existential type $\exists\rho :: \{\rho \mapsto f\}.\tau\ \mathbf{ref}_\rho$. The former approach is not suitable for the purpose of our analysis: for example, without existential types, 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. Our pointer types (e.g. of the form $\mu\alpha.\alpha\ \mathbf{ref}_{0.5}\ \mathbf{ref}_1$) seem to have some similarity with the notion of fractional permissions with nesting [14], as both can express ownerships for nested data structures. Boyland [14] gives the semantics of fractional permissions with nesting, but does not discuss their application to program analysis.

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. It is left for future work to carry out more experiments to evaluate the effectiveness of the type system, and to construct a practical memory-leak verification tool for C 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 anonymous referees for valuable comments, especially for pointing out a close connection to Ueda's work, and members of our research group for comments and discussions.

References

1. Boyland, J.: Checking interference with fractional permissions. In: Proceedings of SAS 2003. Volume 2694 of LNCS., Springer-Verlag (2003) 55–72
2. Heine, D.L., Lam, M.S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In: Proc. of PLDI. (2003) 168–181
3. Ueda, K.: Resource-passing concurrent programming. In: Proceedings of 4th International Symposium on Theoretical Aspects of Computer Science (TACS2001). Volume 2215 of LNCS., Springer-Verlag (2001) 95–126
4. Suenaga, K., Kobayashi, N.: Fractional ownerships for safe memory deallocation. A longer version, available from `http://www.kb.ecei.tohoku.ac.jp/~koba/papers/malloc.pdf` (2009)
5. Dor, N., Rodeh, M., Sagiv, S.: Checking cleanness in linked lists. In: Proceedings of SAS 2000. Volume 1824 of LNCS., Springer-Verlag (2000) 115–134
6. Orlovich, M., Rugina, R.: Memory leak analysis by contradiction. In: Proceedings of SAS 2006. Volume 4134 of LNCS., Springer-Verlag (2006) 405–424
7. Xie, Y., Aiken, A.: Context- and path-sensitive memory leak detection. In: ACM SIGSOFT International Symposium on Foundations of Software Engineering. (2005) 115–125
8. Boyapati, C., Salcianu, A., Beebee, W.S., Rinard, M.C.: Ownership types for safe region-based memory management in real-time Java. In: Proc. of PLDI. (2003) 324–337
9. Swamy, N., Hicks, M.W., Morrisett, G., Grossman, D., Jim, T.: Safe manual memory management in Cyclone. Sci. Comput. Program. **62**(2) (2006) 122–144
10. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Proceedings of CAV 2008. Volume 5123 of LNCS., Springer-Verlag (2008) 385–398
11. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Proceedings of CAV 2007. Volume 4590 of LNCS., Springer-Verlag (2007) 178–192
12. Terauchi, T.: Checking race freedom via linear programming. In: Proc. of PLDI. (2008) 1–10
13. Terauchi, T., Aiken, A.: A capability calculus for concurrency and determinism. ACM Trans. Prog. Lang. Syst. **30**(5) (2008)
14. Boyland, J.: Semantics of fractional permissions with nesting. UWM EECS Technical Report CS-07-01 (2007)