Safe Memory Deallocation for Non-terminating Programs

Author Name

1

sample@example.ac.jp

2

example@sample.net

Abstract We propose a type system to prevent programs infinitely consuming memory cells. The main idea is based on the previous type system that augments pointer types with fractional ownerships and behavioral type system that abstracts the behavior of programs. We design a behavioral type system for verification of behavioral correctness.

1 Introduction

Manual memory management primitives (e.g., `malloc` and `free` in C) often cause serious problems such as double frees, memory leaks, and illegal read/write to a deallocated memory cell. Verifying *safe memory deallocation* – a program not leading to such an unsafe state – is an important problem.

Most of safe memory deallocation verification techniques proposed so far [**?**, **?**, **?**, **?**] focus on the memory leak for *terminating* programs: If a program terminates, the program satisfies safe memory dealloction. For example, the type system by Suenaga and Kobayashi [**?**] guarrantees that (1) a well-typed program does not perform read/write/free operations to any deallocated memory cell and that (2) after execution of a well-typed program, all the memory cells are deallocated.

Currently, we are investigating safe memory deallocation for non-terminating programs, because the safe memory deallocation is very important in real world programs such as Web servers and operating systems.

The main idea of our approach is to decompose this problem into two subproblems: (1) partial correctness and (2) *behavioral correctness*. The former is verified based on the previous type system [**?**], whereas the latter based on the behavioral type system that is mainly used to abstact the behavior of a program. I will describe the concept "memory leak" and our idea by several examples written in an ML-like language as shown in Figure 1 and Figure 2:

```
 f(x)=                        g(x)=
let x = malloc() in          let x = malloc() in
 free(x); f(x)               g(x); free(x)
```

Figure 1. Examples of memory leak.

A program *leaks* memory if the program consumes unbounded number of memory cells. For example, the left-hand side program in Example 1 does not leak memory, whereas the right-hand side does; the former consumes at most one memory cell at once but the latter consumes unbounded number of memory cells. Notice that these two programs are all partially corrected by the previous type system, because they do not terminate. Let us consider another examples as follows: The same to examples in Figure 1, the left-hand side program does not leak memory but the right-hand side does; these two are all partially corrected by previous type system. Another

```
 h(x)=                                    h'(x)=
let x = malloc() in               let x = malloc() in
 let y = malloc() in               let y = malloc() in
 free(x); free(y) ; h(x);                    free(x);h'(x);free(y)
```

Figure 2. Example for demonstrating the main observation.

thing about programs in Figure 2 we should notice is that once partial correctness is guaranteed, we can guarantee memory-leak freedom by estimating upper bound of memory consumption *ignoring the relation between variables and pointers to memory cells.* Specially speaking, partial correctness has proved that there is no double free in programs, which means if allocating a memory cell in the program, definitely there is a deallocation to that pointer. Say, in order to verify h consumes at most two cells at once, we can ignore x and y in h; We can focus on the fact that h executes **malloc** twice, **free** twice, and then calls h. This abstraction is sound because the correspondence between allocations and deallocations is guaranteed by the partial correctness verification. And from the abstraction of h, we know the number of **malloc** and **free** is balanced, so we can say that function h is memory-leak freedom. About the function h', its behavior is abstracted as **malloc** twice,**free** once, calls h' itself ,and then **free** once. So the number of **malloc** exceeds the number of **free** once, which makes the recursive function h' consume infinitely memory cells.

Thanks to the previous type system, which guarantes partial correctness, We can focus on the abstraction of behaviour by *behavioral type system.* In our paper, the behaviour of a program is abstracted by CCS-like processes. For example, the behaviour of f is abstracted as $\mu\alpha.\mathbf{malloc};\mathbf{free};\alpha$; the behaviour of g is abstracted as $\mu\alpha.\mathbf{malloc};\alpha;\mathbf{free}$; the behaviour of h is abstracted as $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{free};\mathbf{free};\alpha$; the behaviour of h' is abstracted as $\mu\alpha.\mathbf{malloc}$; $\mathbf{malloc}$;free;α; $\mathbf{free}$.

The rest of this paper is structed as follows. Section 2 introduces a simple imperative language, as well as its syntax and operational semantics. Section 3 introduces the behavioral type system and its semantics.

2 Language

This section introduces a sublanguage of Suenaga and Kobayashi [?] with primitives for memory allocation/deallocation. And the values in our paper are only pointers.
The syntax of language is as follows.

2.1 Syntax

$$\begin{aligned}
s\ (\textit{statements}) \quad &::= \mathbf{skip} \mid s_1;s_2 \mid\ *x \leftarrow y \mid \mathbf{free}(x)\\
&\mid \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \mid \mathbf{let}\ x = \mathbf{null\ in}\ s\\
&\mid \mathbf{let}\ x = y\ \mathbf{in}\ s \mid \mathbf{let}\ x = *y\ \mathbf{in}\ s\\
&\mid \mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid f(\vec{x})d\ (\textit{definition})\\
::= f(x_1,\ldots,x_n) = s &
\end{aligned}$$

2.2 Semantics

H models the heap address.
R models the registers.

$$\frac{n \in N}{\langle H,\ R,\ \mathbf{skip};s,n\rangle \longrightarrow_D \langle H,\ R,\ s,n\rangle} \quad \text{(E-Skip)}$$

$$\frac{R(x) \in dom(H), n \in N}{\langle H,\ R,\ *x \leftarrow y, n\rangle \longrightarrow_D \langle H\left\{R(x) \to R(y)\right\},\ R,\ \mathbf{skip}, n\rangle} \quad \text{(E-Assign)}$$

$$\frac{R(x) \in dom(H), n \in N}{\langle H,\ R,\ \mathbf{free(x)}\ , n\rangle \mathbf{free}_D \langle H \backslash \left\{R(x)\right\},\ R,\ \mathbf{skip}, n+1\rangle} \quad \text{(E-Free)}$$

$$\frac{x' \notin dom(R)}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\left\{x' \to \mathbf{null}\ \right\},\ [x'/x]\, s, n\rangle} \quad \text{(E-LetNull)}$$

$$\frac{x' \notin dom(R)}{\langle H,\ R,\ \mathbf{let}\ x = y\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\left\{x' \to R(y)\right\},\ [x'/x]\, s, n\rangle} \quad \text{(E-LetEq)}$$

$$\frac{x' \notin dom(R)}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\left\{x' \to H(R(y))\right\},\ [x'/x]\, s, n\rangle} \quad \text{(E-LetDref)}$$

$$\frac{h \notin dom(H)}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, n\rangle \mathbf{malloc}_D \langle H\left\{h \to v\right\},\ R\left\{x' \to h\right\},\ [x'/x]\, s, n-1\rangle} \quad \text{(E-Malloc)}$$

$$\frac{R(x) = \mathbf{null}}{\langle H,\ R,\ \mathbf{ifnull}\ (x)\mathbf{then}\ s_1\mathbf{else}\ \ s_2,\ n\rangle \longrightarrow_D \langle H,\ R,\ s_1,\ n\rangle} \quad \text{(E-IfNullT)}$$

$$\frac{R(x) \neq \mathbf{null}}{\langle H,\ R,\ \mathbf{ifnull}\ (x)\mathbf{then}\ s_1\mathbf{else}\ s_2,\ n\rangle \longrightarrow_D \langle H,\ R,\ s_2,\ n,\rangle} \quad \text{(E-IfNullF)}$$

$$\frac{f(\vec{y}) = s \in D}{\langle H,\ R,\ f(\vec{x}), n\rangle \longrightarrow_D \langle H,\ R,\ [\vec{x}/\vec{y}]\, s, n\rangle} \quad \text{(E-Call)}$$

$$\frac{R(x) = null}{\langle H,\ R,\ *x \leftarrow y, n\rangle \longrightarrow_D NullEx} \quad \text{(E-AssignNullError)}$$

$$\frac{R(y) = null}{\langle H,\ R,\ x = *y, n\rangle \longrightarrow_D NullEx} \quad \text{(E-DrefNullError)}$$

$$\frac{R(x) \notin dom(H) \cup \{null\}}{\langle H,\ R,\ *x \leftarrow y, n\rangle \longrightarrow_D Error} \quad \text{(E-AssignError)}$$

$$\frac{R(y) \notin dom(H) \cup \{null\}}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n\rangle \longrightarrow_D Error} \quad \text{(E-DrefError)}$$

$$\frac{R(x) \notin dom(H) \cup \{null\}}{\langle H,\ R,\ \mathbf{free(x)}\ , n\rangle \mathbf{free}_D Error} \quad \text{(E-FreeError)}$$

$$\frac{R(x) = null}{\langle H,\ R,\ \mathbf{free(x)}\ , n\rangle \mathbf{free}_D NullEx} \quad \text{(E-FreeNullError)}$$

$$\langle H,\ R,\ \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, 0\rangle \mathbf{malloc}_D Error \quad \text{(E-MallocError)}$$

3 Type System

3.1 Syntax of Type

$$
\begin{aligned}
P(\textit{behaviral types}) ::= \quad & \mathbf{0} \mid P_1; P_2 \mid P_1 + P_2 \mid \mathbf{malloc} \\
& \mid \mathbf{free} \mid \alpha \mid \mu\alpha.P \\
\tau(\textit{value types}) ::= \quad & \mathbf{Ref} \\
\sigma(\textit{function types}) ::= \quad & (\tau_1, \ldots, \tau_n)P
\end{aligned}
$$

3.2 Semantics of Behavioral Types

$$0; P \to P$$

$$\mu\alpha.P \to [\mu\alpha.P/\alpha]P$$

$$\mathbf{mallocmalloc}0$$

$$\mathbf{freefree}0$$

$$\frac{P_1 \alpha P_1'}{P_1; P_2 \alpha P_1'; P_2}$$

$$P_1 + P_2 \longrightarrow P_1$$

$$P_1 + P_2 \longrightarrow P_2$$

where $\alpha ::= \mathbf{malloc} \mid \mathbf{free}$

3.3 Type Judgments

$$\Theta; \Gamma \vdash s : P$$

Θ : a finite mapping from function variables to function types.
Γ : a finite mapping from variables to value types.

3.4 Typing Rules

$$\Theta; \Gamma \vdash \mathbf{skip} : \mathbf{0} \quad \text{(T-Skip)}$$

$$\frac{\Theta; \Gamma \vdash s_1 : P_1 \quad \Theta; \Gamma \vdash s_2 : P_2}{\Theta; \Gamma \vdash s_1; s_2 : P_1; P_2} \quad \text{(T-Seq)}$$

$$\frac{\Theta; \Gamma \vdash y : \mathbf{Ref} \quad \Theta; \Gamma \vdash x : \mathbf{Ref}}{\Theta; \Gamma \vdash *x \leftarrow y : \mathbf{0}} \quad \text{(T-Assign)}$$

$$\frac{\Theta; \Gamma \vdash x : \mathbf{Ref}}{\Theta; \Gamma \vdash \mathbf{free}(x) : \mathbf{free}; 0} \quad \text{(T-Free)}$$

$$\frac{\Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \ \mathbf{in}\ s : \mathbf{malloc}; P} \quad \text{(T-Malloc)}$$

$$\frac{\Theta; \Gamma \vdash y : \mathbf{Ref} \quad \Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = y\ \mathbf{in}\ s : P} \quad \text{(T-LetEq)}$$

$$\frac{\Theta; \Gamma \vdash y : \mathbf{Ref} \quad \Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s : P} \quad \text{(T-LetDref)}$$

$$\frac{\Theta; \Gamma, x \vdash s : P}{\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s : P} \quad \text{(T-LetNull)}$$

$$\frac{\Theta;\Gamma \vdash x : \mathbf{Ref} \quad \Theta;\Gamma \vdash s_1 : P \quad \Theta;\Gamma \vdash s_1 : P}{\Theta;\Gamma \vdash \mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 : P} \quad \text{(T-IfNull)}$$

$$\frac{\Theta(f) = P}{\Theta;\Gamma, \vec{x} : \vec{\tau} \vdash f(\vec{x}) : P} \quad \text{(T-Call)}$$

$$\frac{\vdash D : \Theta \quad \Theta;\emptyset \vdash s : P \quad OK_n(P)}{\vdash (D, s)} \quad \text{(T-Program)}$$

$$\frac{\Theta;\Gamma \vdash s : P_1 \quad P_1 \leq P_2}{\Theta;\Gamma \vdash s : P_2} \quad \text{(T-Sub)}$$

4 Type Soundness

Theorem 4.1 *If $\vdash (D, s)$ then (D, s) does not lead to memory leak.*
Memory leak freedom: $\exists n \in N$ s.t. $\langle \emptyset, \emptyset, s, n\rangle^* Error$

Lemma 4.2 (Preservation I) *If $OK_n(P)$, $\Theta;\Gamma \vdash s : P$ and $\langle H,\ R,\ s,\ n\rangle \alpha \langle H',\ R',\ s',\ n'\rangle$, then $\exists P'$ s.t.*
(1) $\Theta;\Gamma \vdash s' : P'$
(2) $P\alpha \Longrightarrow P'$
(3) $OK_{n'}(P')$

Lemma 4.3 (Preservation II) *If $OK_n(P)$, $\Theta;\Gamma \vdash s : P$ and $\langle H,\ R,\ s,\ n\rangle \rightarrow \langle H',\ R',\ s',\ n'\rangle$, then $\exists P'$ s.t.*
(1) $\Theta;\Gamma \vdash s' : P'$
(2) $P\tau^* P'$
(3) $OK_{n'}(P')$

Lemma 4.4 *when partial correctness is guaranteed $\vdash \langle H, R, s\rangle$and if $\vdash \langle H, R, s, n\rangle$, then $\vdash \langle H', R', s', n'\rangle Error$*

5 Syntax Directed Typing Rules

C is contraint for subtype.

$$\frac{C = \emptyset}{\Theta;\Gamma;C \vdash \mathbf{skip} : \mathbf{0}} \quad \text{(ST-Skip)}$$

$$\frac{\Theta;\Gamma;C_1 \vdash s_1 : P_1 \quad \Theta;\Gamma;C_2 \vdash s_2 : P_2 \quad C = C_1 \cup C_2 \cup \{P_1; P_2 \leq P\}}{\Theta;\Gamma;C \vdash s_1; s_2 : P} \quad \text{(ST-Seq)}$$

$$\frac{\Theta;\Gamma;C_1 \vdash y \quad \Theta;\Gamma;C_2 \vdash x : \mathbf{Ref} \quad C = C_1 \cup C_2}{\Theta;\Gamma;C \vdash *x \leftarrow y : \mathbf{0}} \quad \text{(ST-Assign)}$$

$$\frac{C = \emptyset}{\Gamma;C \vdash \mathbf{free}() : \mathbf{free}; \mathbf{0}} \quad \text{(ST-Free)}$$

$$\frac{\Theta;\Gamma, x;C_1 \vdash s : P_1 \quad C = C_1 \cup \{P_1 \leq P\}}{\Theta;\Gamma;C \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s : \mathbf{malloc}; P} \quad \text{(ST-Malloc)}$$

$$\frac{\Theta;\Gamma;C_1 \vdash y \quad \Theta;\Gamma, x;C_2 \vdash s : P_1 \quad C = C_1 \cup C_2 \cup \{P_1 \leq P\}}{\Theta;\Gamma;C \vdash \mathbf{let}\ x = y\ \mathbf{in}\ s : P} \quad \text{(ST-LetEq)}$$

$$\frac{\Theta;\Gamma;C_1 \vdash y : \mathbf{Ref} \quad \Theta;\Gamma,x;C_2 \vdash s : P_1 \quad C = C_1 \cup C_2 \cup \{P_1 \leq P\}}{\Theta;\Gamma;C \vdash \mathbf{let}\ x = *y\ \mathbf{in}\ s : P} \quad \text{(ST-LetDref)}$$

$$\frac{\Theta;\Gamma;C_1 \vdash x \quad \Theta;\Gamma;C_2 \vdash s_1 : P_1 \quad \Theta;\Gamma;C_3 \vdash s_2 : P_2 \quad C = C_1 \cup C_2 \cup C_3\{P_1 \leq P, P_2 \leq P\}}{\Theta;\Gamma;C \vdash \mathbf{ifnull}\ (x)\mathbf{then}\ s_1\mathbf{else}\ s_2 : P} \quad \text{(ST-IfNull)}$$

$$\frac{\Theta(f) = P_1 \quad C = P_1 \leq P}{\Gamma, \vec{x} : \vec{\tau} \vdash f(\vec{x}) : P} \quad \text{(ST-Call)}$$

$$\frac{\Theta \vdash D : \Theta \quad \Theta;\emptyset;C_1 \vdash s : P \quad C = C_1 \cup \{OK_n(P)\}}{C \vdash (D, s)} \quad \text{(ST-Prog)}$$

6 Type Inference

$PT_v(x) = (\emptyset, \emptyset)$, where x maybe a value or reference.
PT_Θ is a mapping from statements to a pair of constraints and types with function Θ from function names to function types, like $\Theta(f) = P$. $\mathrm{PT}_\Theta(f) =$

```
let $\alpha = \Theta(f)$
in $(C = \{\alpha \leq \beta\}, \beta)$
    $\mathrm{PT}_\Theta(\mathbf{skip}) = (\emptyset, 0)$
    $\mathrm{PT}_\Theta(s_1; s_2) =$
let $(C_1, P_1) = PT_\Theta(s_1)$
$(C_2, P_2) = PT_\Theta(s_2)$
in $(C_1 \cup C_2 \cup \{P_1; P_2 \leq \beta\}, \beta)$
    $\mathrm{PT}_\Theta(*x \leftarrow y) =$
let $(C_1, \emptyset) = PT_v(*x)$
$(C_2, \emptyset) = PT_v(y)$
in $(C_1 \cup C_2, 0)$
    $\mathrm{PT}_\Theta(\mathbf{free}(x)) = (\emptyset, \mathbf{free}; 0)$
    $\mathrm{PT}_\Theta(\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s) =$
let $(C_1, P_1) = PT_v(s)$
in $(C_1 \cup \{P_1 \leq \beta\}, \mathbf{malloc}; \beta)$
    $\mathrm{PT}_\Theta(\mathbf{let}\ x = y\ \mathbf{in}\ s) =$
let $(C_1, \emptyset) = PT_v(y)$
$(C_2, P_1) = PT_\Theta(s)$
in $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)$
    $\mathrm{PT}_\Theta(\mathbf{let}\ x = *y\ \mathbf{in}\ s) =$
let $(C_1, \emptyset) = PT_v(y)$
$(C_2, P_1) = PT_\Theta(s)$
in $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)$
    $\mathrm{PT}_\Theta(\mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2) =$
let $(C_1, P_1) = PT_\Theta(s_1)$
$(C_2, P_2) = PT_\Theta(s_2)$
$(C_3, \emptyset) = PT_v(x)$
in $(C_1 \cup C_2 \cup C_3 \cup \{P_1 \leq \beta, P_2 \leq \beta\}, \beta)$
    PT(¡D, s¿)  =
let $\Theta = \{f_1 : \alpha_1, \ldots, f_n : \alpha_n\}$
$where\ \{f_1, \ldots, f_n\} = dom(D)\ and\ \alpha_1, \ldots, \alpha_n\ are\ fresh$
in let $(C_i, P_i) = PT_\Theta(D(f_i))\ for\ each\ i$
in let $C_i' = \{\alpha_i \leq P_i\}\ for\ each\ i$
in let $(C, P) = PT_{\Theta}(s)$
in $(C_i \cup C_i') \cup C \cup \{OK(P)\}, P)$
```

Experiments

7 Conclusion

this is the end

References