Safe Memory Deallocation for Non-Terminating Programs

Author Name

1

sample@example.ac.jp

2

example@sample.net

Abstract We propose an approach to guarantee safe memory deallocation for non-terminating programs. The main idea is based on two type systems: previous type system [1] and behavioral type system. The former guarantees partial correctness, and the latter abstracts the behavior of a program. Thanks to the previous type system, we can focus on the behavioral type system to count the upper bound of the consumed memory cells soundly.

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 [1, 2, 3, 4] focus on the *partial correctness*: if a program terminates, the program satisfies safe memory dealloction. For example, the type system by Suenaga and Kobayashi [1] 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. The function g shown in Figure 1 describe this situation.

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

Figure 1.

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.* Partial correctness means that if a program termintates, the program satisfies safe memory deallocation. Behavioral correctness means a program does not *leak* memory . The former is verified based on the previous type system [1], whereas the latter based on the behavioral type system that is mainly used to abstact the behavior of a program. Behavioral types are heavily used in the context of concurrent program verification [5, 6, 7]. A program *leaks* memory if

$f(x)=$
$\textbf{let}\ \ x = \textbf{malloc}()\ \ \textbf{in}$
$\textbf{free}(x);\quad f(x)$

$g(x)=$
$\textbf{let}\ \ x = \textbf{malloc}()\ \ \textbf{in}$
$g(x);\quad \textbf{free}(x)$

Figure 2. Explanation for memory leak.

the program consumes unbounded number of memory cells. For example, the left-hand side program in Figure 2 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. We should notice that once partial

$h(x)=$
$\textbf{let}\ \ x = \textbf{malloc}()\ \ \textbf{in}$
$\textbf{let}\ \ y = \textbf{malloc}()\ \ \textbf{in}$
$\textbf{free}(x);\quad \textbf{free}(y);\ h(x)$

Figure 3. Example for demonstrating the main observation.

correctness is guaranteed, which especially about no double frees, we can guarantee memory-leak freedom by estimating the upper bound of memory consumption ignoring the relationship between variables and pointers to memory cells. For demonstrating this observation, we use example in Figure 3. The function h is partially corrected. The behavior of h is that it consumes two memory cells at once. In order to verify this behavior, we ignore the variables x and y in h to 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

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 as CCS-like processes. For example, the behaviour of f is abstracted as $\mu\alpha.\textbf{malloc};\textbf{free};\alpha$; the behaviour of g is abstracted as $\mu\alpha.\textbf{malloc};\alpha;\textbf{free}$; the behaviour of h is abstracted as $\mu\alpha.\textbf{malloc};\textbf{malloc};\textbf{free};\textbf{free};\alpha$. And then passing these behavior(types) to $OK_n(P)$, where P denotes behavior(types). The $OK_n(P)$ elaborated in subsection 3.3 traces every steps about allocation and deallocation, and it will reject a memory-leak program.

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. Section 4 proposes an inference algorithm, and talks about syntax directed typing rules. Section 5 descibe current status and future work.

2 Language

This section introduces a sublanguage of Suenaga and Kobayashi [1] 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

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

A program is a pair (D, s), where D is the set of definition.
The command **skip** does nothing. The command $s_1; s_2$ is executed as a sequence, first executing s_1 and then s_2. The command $*x \leftarrow y$ updates the content of the memory cell which is pointed by pointer x with value y. The command $\mathbf{free}(x)$ deallocates the memory cell which is pointed by a pointer x. Then command **let** $x = e$ **in** s first evaluates the expresstion e and binds the return value of e to x and then executes statement s. The command **let** $x =$ **malloc in** s first allocates a memory cell to a pointer x and then executes the statement s. The command **let** $x =$ **null in** s first allocates a null pointer to x and then executes s. The command **let** $x = y$ **in** s assign the pointer y to x, so the pointer x and y are said aliases for the same memory cell, and then executes statement s. The command **let** $x = *y$ **in** s transfers a part of memory cells pointed by y and then executes statement s. The command **ifnull** (x) **then** s_1 **else** s_2 denotes that executing statement s_1 if pointer x is a null pointer, otherwise executing statement s_2. The command $f(\vec{x})$ is a function call in which $\vec{x}$ denotes mutually distinct variables like $\{x_1, \ldots, x_n\}$. The notation d denotes the definition of function $f(\vec{x})$ which has a body of statement s. And examples are described by this syntax you can see in Figure 1 and Figure 2.

2.2 Operational Semantics

Because we want to estimate the number of available memory cells at every operation step, we extend the triple $\langle H,\ R,\ s\rangle$ that is represented as run-time state in previous type system to a quadruple $\langle H,\ R,\ s,\ n\rangle$ in our paper. The introduced notation n denotes the number of available memory cells, a nature number. When executing the operation **malloc**, the number of available memory cells will decrease 1, which is denoted as $(n-1)$; when executing the opreation **free**, the number of available memory cells will increase 1, which is denoted as $(n+1)$. The notation H, which models heap memory, is a mapping from finite subset of $\mathcal{H}$ to $\mathcal{H} \cup \{null\}$, where $\mathcal{H}$ represents the set of *heap addresses*. R, which models registers, is a mapping from finite set of variables to $\mathcal{H} \cup \{null\}$.

Transition rules are listed in Figure 3. In these rules, $f\{x \to v\}$ is defined as a function f' such that $f'(y) = v$ if $x = y$, otherwise $f'(y) = f(y)$ and $y \in dom(f)$. There are three rules about **NullEx** which denotes accessing a null pointer, three rules aoubt **Error** for accessing a deallocated memory cell, and one rule about **Error** which denotes allocating a memory cell when there is no memory space.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Operational Semantics

3 Type System

This section elaborate the behavioral type system to prevent leaking memory in non-terminating programs. We define behavioral types, CCS-like processes that abstract the behavior of programs, as follows.

3.1 Syntax of Types

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

The type $\mathbf{0}$ abstracts the behavior of **skip** and means "does nothing". $P_1; P_2$ is for sequential exetution. $P_1 + P_2$ is abstracted as contional. **malloc** is the behavior of a statement that allocates a memory cell exactly once. **free** is for deallocating memory cell exactly once. $\mu\alpha.P$ is a recursive type. For example, the behavior of the body of function h in Figure 2 is abstracted as $\mu\alpha.\mathbf{malloc}; \mathbf{malloc}; \mathbf{free}; \mathbf{free}; \alpha$. α is a type variable and bounded to the recursive constructor $\mu\alpha$.

The only value in our paper is reference, and its type is **Ref**.

The function type is described as $(\tau_1, \ldots, \tau_n)P$, which means a function receives some pointers as arguments and its body is abstracted as a behavioral type P.

3.2 Semantics of Behavioral Types

The semantics of behavioral type are given by labeled transition system, and listed as follows:

$$0; P \to P$$

$$\mathbf{malloc} \xrightarrow{\mathbf{malloc}} 0$$

$$\mathbf{free} \xrightarrow{\mathbf{free}} 0$$

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

$$P_1 + P_2 \longrightarrow P_1$$

$$P_1 + P_2 \longrightarrow P_2$$

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

The notation $\rightarrow$ denotes that a behavioral type can be reduced by the internal action. Notation $\xrightarrow{\alpha}$ means that a behavioral type can be reduced by executing α actions, and the α here is $\{\mathbf{malloc}, \mathbf{free}\}$.

3.3 Typing Rules

The type judgement of our type system is given by the form $\Theta; \Gamma \vdash s : P$, where Θ is a mapping from function varibles to function types, Γ is a type environment that denotes a mapping from variables to value types. It reads "the behavior of s is abstracted as P under Θ and Γ environments". We design the type system so that this type judgement implies the property: when s executes **malloc**(resp.**free**), then P is equivalent to $\mathbf{malloc}; P'$(resp.$\mathbf{free}; P'$) for a type P' such that $\Theta; \Gamma \vdash s' : P'$, where s' is the continuation of s. This property guarantees the behavioral type soundly abstracts the upper bound of the consumed memory cells.

Typing rules are presented in Figure 4. In the rule for assignment, the behavior of $*x \leftarrow y$ is $\mathbf{0}$. The rule for **free** represents that the behavior of $\mathbf{free}(x)$ is **free**. The rule T-Malloc represents that $\mathbf{let}\ x = \mathbf{malloc}()\ \ \mathbf{in}\ s$ has the behavior $\mathbf{malloc}; P$, where P is the behavior of statement s. The rule for function call represents that function f has the behavior P which is the behavior of the body of this function .

In the rule for subtyping, $P_1 \leq P_2$ represents that P_1 is the subtype of P_2 and means that:
(1) if $P_1 \xrightarrow{\alpha} P_1'$ then $\exists P_2'$ s.t. $P_2 \overset{\alpha}{\Longrightarrow} P_2'$ and $P_1' \leq P_2'$
(2) if $P_1 \rightarrow P_1'$ then $\exists P_2'$ s.t. $P_2 \rightarrow^* P_2'$ and $P_1' \leq P_2'$
where $\overset{\alpha}{\Longrightarrow}$ means that: $\rightarrow^* \xrightarrow{\alpha} \rightarrow^*$.

In the rule for program, the main statement s is executed under Θ and Γ environments without free variables. At the end of s, memory leak freedom is guaranteed by $OK_n(P)$,where P is the behavior of s. $OK_n(P)$ is defined as **Definition 1** in which $\sharp_{malloc}(\alpha)$ and $\sharp_{free}(\alpha)$ are funcitons to count the number of **malloc** and **free** actions in α respectively. This definition, intuitively, means at every running step the number of allocated memory cells will never go out of memory scope.

Definition 1. $OK_n(P) \iff \forall P',\ P \xrightarrow{\alpha}{}^{*} P'$ *then* $\sharp_{malloc}(\alpha) - \sharp_{free}(\alpha) \leq n$.

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4. Typing Rules

3.4 Type Soundness

This subsection describes some theorems and lemmas for type safety.

Theorem 3.1. *If* $\vdash (D,s)$ *then* (D,s) *does not lead to memory leak.*
Memory leak freedom: $\exists n\in\mathbb{N}$ *s.t.* $\langle\emptyset,\emptyset,s,n\rangle \not\rightarrow^* Error$

This theorem says that a well typed program guarantees memory leak freedom.

Lemma 3.2 (Preservation **I**)**.** *If* $OK_n(P)$*,* $\Theta;\Gamma\vdash s:P$ *and* $\langle H,R,s,n\rangle \xrightarrow{\alpha} \langle H',R',s',n'\rangle$*,*
then $\exists P'$ *s.t.*
(1) $\Theta;\Gamma\vdash s':P'$
(2) $P \stackrel{\alpha}{\Longrightarrow} P'$
(3) $OK_{n'}(P')$

Lemma 3.3 (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) $P\rightarrow^* P'$
(3) $OK_{n'}(P')$

Lemma 3.4. *The partial correctness is guaranteed* $\vdash \langle H,R,s\rangle$ *, so that if* $\vdash \langle H,R,s,n\rangle$*,*
then $\vdash \langle H',R',s',n'\rangle \not\rightarrow Error$

4 Type Inference Algorithm

This section describes how to construct syntax directed typing rules according to the typing rules of above section ,and it provides an algorithm which inputs statements and returns a pair containing constraints and behavior types.

4.1 Syntax Directed Typing Rules

Typing rules showed in Figure 4 are not immediately suitable for type inference. The reason is that the subtyping rule can be applied to any kind of term. This means that, any kind of term s can be applied by either subtyping rule or the other rule whose conclusion mathes the shape of the s [8].

In order to yield a type inference algorithm, we should do something with the subtyping rule. The method is to merge the subtyping rule with the other rules by introducing a set C of constraints, where C consists of subtype constraints on behavioral types of the form $P_1 \leq P_2$ and $OK_n(P)$.

Syntax directed typing rules are listed in Figure 5.

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

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

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

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

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

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

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

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

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

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

Figure 5. Syntax Directed Typing Rules

4.2 Algorithm

By syntax directed typing rules, the type inference algorithm has been designed as in Figure 6.

Function $PT_v(x) = (C, \emptyset)$ denotes that it receives a pointer variable x and outputs a pair consisting of constraints set C and an empty set. $PT_\Theta(s) = (C, P)$ is a mapping from statements to a pair – constraints set C and behavioral types P, where Θ is mapping from function names to function types. $PT(< D, s >) = (C, P)$ denotes that it receives a program and produces a pair (C, P). α_i and β are fresh type variables.

$$
\begin{aligned}
&PT_\Theta(f) = \\
&\quad \textbf{let } \alpha = \Theta(f) \\
&\quad \textbf{in } (C = \{\alpha \leq \beta\}, \beta)
\end{aligned}
$$

$$PT_\Theta(\textbf{skip}) = (\emptyset, 0)$$

$$
\begin{aligned}
&PT_\Theta(s_1; s_2) = \\
&\quad \textbf{let } (C_1, P_1) = PT_\Theta(s_1) \\
&\quad\quad (C_2, P_2) = PT_\Theta(s_2) \\
&\quad \textbf{in } (C_1 \cup C_2 \cup \{P_1; P_2 \leq \beta\}, \beta)
\end{aligned}
$$

$$
\begin{aligned}
&PT_\Theta(*x \leftarrow y) = \\
&\quad \textbf{let } (C_1, \emptyset) = PT_v(*x) \\
&\quad\quad (C_2, \emptyset) = PT_v(y) \\
&\quad \textbf{in } (C_1 \cup C_2, 0)
\end{aligned}
$$

$$PT_\Theta(\textbf{free}(x)) = (\emptyset, \textbf{free})$$

$$
\begin{aligned}
&PT_\Theta(\textbf{let } x = \textbf{malloc}() \textbf{ in } s) = \\
&\quad \textbf{let } (C_1, P_1) = PT_v(s) \\
&\quad \textbf{in } (C_1 \cup \{P_1 \leq \beta\}, \textbf{malloc}; \beta)
\end{aligned}
$$

$$
\begin{aligned}
&PT_\Theta(\textbf{let } x = y \textbf{ in } s) = \\
&\quad \textbf{let } (C_1, \emptyset) = PT_v(y) \\
&\quad\quad (C_2, P_1) = PT_\Theta(s) \\
&\quad \textbf{in } (C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)
\end{aligned}
$$

$$
\begin{aligned}
&PT_\Theta(\textbf{let } x = *y \textbf{ in } s) = \\
&\qquad \textbf{let } (C_1, \emptyset) = PT_v(y) \\
&\qquad\quad (C_2, P_1) = PT_\Theta(s) \\
&\qquad \textbf{in } (C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)
\end{aligned}
$$

$$
\begin{aligned}
&PT_\Theta(\textbf{ifnull } (x) \textbf{ then } s_1 \textbf{ else } \ s_2) = \\
&\qquad\qquad \textbf{let } (C_1, P_1) = PT_\Theta(s_1) \\
&\qquad\qquad\quad (C_2, P_2) = PT_\Theta(s_2) \\
&\qquad\qquad\quad (C_3, \emptyset) = PT_v(x) \\
&\qquad\qquad \textbf{in } (C_1 \cup C_2 \cup C_3 \cup \{P_1 \leq \beta, P_2 \leq \beta\}, \beta)
\end{aligned}
$$

$$
\begin{aligned}
&PT(< D, s >) = \\
&\qquad\qquad \textbf{let } \Theta = \{f_1 : \alpha_1, \ldots, f_n : \alpha_n\} \\
&\qquad\qquad\quad \textit{where } \{f_1, \ldots, f_n\} = dom(D) \textit{ and } \alpha_1, \ldots, \alpha_n \textit{ are fresh} \\
&\qquad\qquad \textbf{in let } (C_i, P_i) = PT_\Theta(D(f_i)) \textit{ for each } i \\
&\qquad\qquad \textbf{in let } C_i^{'} = \{\alpha_i \leq P_i\} \textit{ for each } i \\
&\qquad\qquad \textbf{in let } (C, P) = PT_\Theta(s) \\
&\qquad\qquad \textbf{in } (C_i \cup C_i^{'}) \cup C \cup \{OK(P)\}, P)
\end{aligned}
$$

Figure 6. Type Inference Algorithm

5 Future Work

We have described a type-based approach to safe memory deallocation for non-terminating programs. The approach is based on the idea of decomposing safe memory memory deallocation into partial correctness, which is verified by previous type system, and behavioral correctness. We designed a behavioral type system in our paper for verification of behavioral correctness. Currently, we are looking for a model checker to estimate an upper bound of cousumption given a behavioral type and planning to implement a verifer and conduct experiment to see whether our approach is feasible.

References

[1] K. Suenaga and N. Kobayashi, "Fractional ownerships for safe memory deallocation," in *APLAS*, ser. Lecture Notes in Computer Science, Z. Hu, Ed., vol. 5904. Springer, 2009, pp. 128–143.

[2] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive c and c++ memory leak detector," in *PLDI*, R. Cytron and R. Gupta, Eds. ACM, 2003, pp. 168–181.

[3] Y. Xie and A. Aiken, "Context- and path-sensitive memory leak detection," in *ESEC/SIGSOFT FSE*, M. Wermelinger and H. Gall, Eds. ACM, 2005, pp. 115–125.

[4] N. Swamy, M. W. Hicks, G. Morrisett, D. Grossman, and T. Jim, "Safe manual memory management in cyclone," *Sci. Comput. Program.*, vol. 62, no. 2, pp. 122–144, 2006.

[5] N. Kobayashi, K. Suenaga, and L. Wischik, "Resource usage analysis for the p-calculus," *Logical Methods in Computer Science*, vol. 2, no. 3, 2006.

[6] A. Igarashi and N. Kobayashi, "A generic type system for the pi-calculus," *Theor. Comput. Sci.*, vol. 311, no. 1-3, pp. 121–163, 2004.

[7] K. Honda, V. T. Vasconcelos, and M. Kubo, "Language primitives and type discipline for structured communication-based programming," in *ESOP*, ser. Lecture Notes in Computer Science, C. Hankin, Ed., vol. 1381. Springer, 1998, pp. 122–138.

[8] B. C. Pierce, *Types and Programming Languages.* The MIT Press, 2002.

A Appendix