# A Behavioral Type System for Memory-Leak Freedom

Qi Tan, Kohei Suenaga, and Atsushi Igarashi

Department of Communications and Computer Engineering
Graduate School of Informatics
Kyoto University
{tanki,ksuenaga,igarashi}@fos.kuis.kyoto-u.ac.jp

**Abstract** We propose a type system to abstract the behavior of a program under manual memory management. Our type system uses sequential processes as types where each action corresponds to an allocation and a deallocation of a fixed-size memory block. The abstraction obtained by our type system makes it possible to estimate an upper bound of memory consumption of a program. Hence, by using our type system with another safe-memory-deallocation analysis proposed by Suenaga and Kobayashi, we can verify memory-leak freedom even for nonterminating programs. We define the type system, prove type soundness, and show a type reconstruction procedure that estimates an upper bound of memory consumption using an off-the-shelf model checker.

## 1 Introduction

Dynamic memory management is a crucial function of programming languages. Correct allocation and disposal of memory cells are fundamental for software to be reliable.

Correct dynamic memory management is challenging if a programming language is equipped with manual memory management primitives (e.g., `malloc` and `free` in the C language). With such primitives, one can write a program that accesses to deallocated memory cells (i.e., accesses to dangling pointers) and that does not dispose memory cells even when they become unnecessary (i.e., memory leaks). In order to detect bugs related to such primitives at the early stage of software development, many static verification methods have been proposed [4, 9–12, 15].

The analyses proposed so far put less emphasis to nonterminating programs although memory leaks in such programs are more serious (e.g., operating systems and Web servers) than in terminating ones. They rather verify *partial* memory-leak freedom: All the allocated memory cells are eventually deallocated *if a program terminates*. We say a program is *totally* memory-leak free if it does not consume unbounded amount of memory during execution.

Total memory-leak freedom is indeed an important property in real-world programs. For example, memory-leak freedom of a controller in an embedded system is crucial.

However, partial memory-leak freedom is not enough for such software that does not terminate.

```
1   h()=                              h'()=
2       let x = malloc() in               let x = malloc() in
3       let y = malloc() in               let y = malloc() in
4       free(x); free(y); h()             h'(); free(x); free(y)
```

**Figure 1.** Memory leaks in nonterminating programs.

**Example 1.1.** Figure 1 describes partial and total memory-leak freedom. Both $h$ and $h'$ are partially memory-leak free because they do not terminate. The function $h$ is totally memory-leak free since it consumes at most two cells[1]. However, the function $h'$, when it is invoked, consumes unbounded number of memory cells; hence $h'$ is not totally memory-leak free.

As a first step to the verification of total memory-leak freedom, this paper proposes a *behavioral type system* [6–8] for a programming language with manual memory-management primitives. Our type system approximates the behavior of a program by a sequential process whose actions represent memory allocation and deallocation. For example, our type system can assign a type $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{free};\mathbf{free};\alpha$ to the function $h$ above. This type expresses that $h$ can allocate a memory cell twice, deallocate a memory cell twice, and then iterate this behavior. The type assigned to $h'$ is $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\alpha;\mathbf{free};\mathbf{free}$, which expresses that $h'$ can allocate a memory cell twice, call itself recursively, and then deallocate a memory cell twice. Hence, by inspecting the inferred types (by using off-the-shelf model checkers, for example), one can estimate the upper bound required to execute $h$ and $h'$.

One may not observe, in the example above, much difference between applying a model checker to the original programs and to the assigned behavioral types. However, we expect the latter is faster than the former in many programs because a behavioral type focuses on the actions related to allocations and deallocations, abstracting away the other actions.

Notice that our type system alone does not prevent incorrect usage of **malloc** and **free**. Indeed, as observed from the type assigned to $h$ and $h'$ above, our types include information only about the number and the order of allocations, deallocations, and recursive function calls; hence, the type system does not guarantee, for example, that there is no access to a deallocated cell. For such properties, we can use other no-illegal-access verifiers [10].

The rest of this paper is structured as follows. Section 2 introduces a simple imperative language and the operational semantics of the language. Section 3 introduces the behavioral type system and states its type soundness. Section 4 describes a type reconstruction procedure. Section 5 presents the preliminary experiments. Section 6 discusses

[1] We assume that every memory cell allocated by **malloc** is fixed size to simplify our type system introduced in Section 3. Extension with variable-length cells is one of our future work.

the related work. Section 7 concludes the paper. The proof of type soundness and the detailed definition of the type reconstruction procedure are in the full version [13].

**Notation** We write $\vec{X}$ for a finite sequence of $X$. We write $[\vec{x'}/\vec{x}]s$ for the term obtained by replacing every free occurrence of $\vec{x}$ in $s$ with $\vec{x'}$. We write $\mathbf{Dom}(f)$ for the domain of the map $f$. For a map $f$, we write $f\{x \mapsto v\}$ and $f\backslash x$ for the maps defined as follows:

$$f\{x \mapsto v\}(w) = \begin{cases} v & \text{if } x = w \\ f(w) & \text{otherwise.} \end{cases}$$

$$(f\backslash x)(w) = \begin{cases} \text{undefined} & \text{if } x = w \\ f(w) & \text{otherwise.} \end{cases}$$

## 2 Language $\mathcal{L}$

This section gives the definition of language $\mathcal{L}$, an imperative language with memory allocation/deallocation primitives. The language is essentially the same as one used by Sueanga et al. [10], where they propose a type-based analysis for partial memory-leak freedom analysis.

The syntax of language $\mathcal{L}$ is as follows.

$$\begin{array}{rcl} x, y, z, \ldots \text{ (variables)} & \in & \mathbf{Var} \\ s \text{ (statements)} & ::= & \mathbf{skip} \mid s_1; s_2 \mid *x \leftarrow y \mid \mathbf{free}(x) \\ & \mid & \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \mid \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s \\ & \mid & \mathbf{let}\ x = y\ \mathbf{in}\ s \mid \mathbf{let}\ x = *y\ \mathbf{in}\ s \\ & \mid & \mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \mid f(\vec{x}) \\ d \text{ (proc. defs.)} & ::= & \{f \mapsto (x_1, \ldots, x_n)s\} \\ D\text{(definitions)} & ::= & \langle d_1 \cup \cdots \cup d_n \rangle \\ P \text{ (programs)} & ::= & \langle D, s \rangle \end{array}$$

The language is equipped with procedure calls, dynamic memory allocation and deallocation, and memory accesses with pointers. **Var** is a countably infinite set of *variables*. The statement **skip** does nothing. The statement $s_1; s_2$ executes $s_1$ and $s_2$ sequentially. The statement $*x \leftarrow y$ writes $y$ to the memory cell that $x$ points to. The statement $\mathbf{let}\ x = e\ \mathbf{in}\ s$ evaluates the expression $e$, binds $x$ to the result, and executes $s$. The expression $\mathbf{malloc}()$ allocates a new memory cell and evaluates to the pointer to the cell. The expression **null** evaluates to the null pointer. The expression $y$ evaluates to its value. The expression $*y$ evaluates to the value in the memory cell that $y$ points to. The statement $\mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$ executes $s_1$ if $x$ is **null** and executes $s_2$ otherwise. The statement $f(\vec{x})$ calls procedure $f$ with arguments $\vec{x}$.

A procedure definition ranged over by $d$ is a map from a procedure name to an abstraction of the form $(\vec{x})s$. We use a metavariable $D$ for a set of function definitions $d_1 \cup \cdots \cup d_n$. We assume that there are no duplicated definitions of each function. We

also assume that there is no arity mismatch between function definitions and function calls. A program is a pair of function definitions $D$ and a main statement $s$.

## 2.1 Operational semantics

This section introduces the operational semantics of $\mathcal{L}$. We assume a countably infinite set $\mathcal{H}$ of *locations* ranged over by $l$.

We express a state of computation by *configuration* $\langle H, R, s, n\rangle$. A configuration consists of the following four components:

- $H$, a *heap*, is a finite mapping from $\mathcal{H}$ to $\mathcal{H} \cup \{\mathbf{null}\}$;
- $R$, an *environment*, is a finite mapping from **Var** to $\mathcal{H} \cup \{\mathbf{null}\}$;
- $s$ is the statement that is being executed; and
- $n$ is a natural number that represents the number of memory cells available for allocation.

We later use $n$ to formalize memory leaks caused by nonterminating program.

The operational semantics is given by relation $\langle H, R, s, n\rangle \xrightarrow{\rho}_D \langle H', R', s', n'\rangle$ where $\rho$, an *action*, is **malloc**, **free**, or $\tau$. The action **malloc** expresses an allocation of a memory cell; **free** expresses a deallocation; $\tau$ expresses the other actions. We often omit $\tau$ in $\xrightarrow{\tau}_D$. We use a metavariable $\sigma$ for a finite sequence of actions $\rho_1 \dots \rho_n$. We write $\xrightarrow{\rho_1 \dots \rho_n}_D$ for $\xrightarrow{\rho_1}_D \xrightarrow{\rho_2}_D \dots \xrightarrow{\rho_n}_D$. We write $\overset{\rho}{\Longrightarrow}_D$ for $\longrightarrow_D^* \xrightarrow{\rho}_D \longrightarrow_D^*$. We write $\overset{\rho_1 \dots \rho_n}{\Longrightarrow}_D$ for $\overset{\rho_1}{\Longrightarrow}_D \dots \overset{\rho_n}{\Longrightarrow}_D$.

Figure 2 defines the relation $\xrightarrow{\rho}_D$. We add explanation to several important rules.

- Sem-Free: Deallocation of a memory cell pointed to by $x$ is expressed by deleting the entry for $R(x)$ from the heap. This action increments the number of available cells (i.e., $n$) by one (i.e., $n + 1$).
- Sem-Malloc and Sem-OutOfMem: Allocation of a memory cell is expressed by adding a fresh entry to the heap. This action is allowed only if the number of available cells is positive; if the number is zero, then the configuration leads to an error state **OutOfMemory**.
- Sem-*Exn: These rules express an illegal access to memory. If such action is performed, then the configuration leads to exceptional state **MemEx**. This state **MemEx** is not seen as an erroneous state in the current paper, hence is not excluded by the type system in Section 3. The command **free**(**x**) ($x$), if $x$ is a null pointer, leads to **MemEx** in the current semantics, although it is equivalent to **skip** in the C language.

As mentioned in Section 1, a program is said to leak memory if it consumes unbounded number of memory cells. Formally, this is defined as follows.

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

$$\frac{\langle H, R, s_1, n\rangle \xrightarrow{\rho}_D \langle H', R', s_1', n'\rangle}{\langle H, R, s_1; s_2, n\rangle \xrightarrow{\rho}_D \langle H', R', s_1'; s_2, n'\rangle} \quad \text{(Sem-Seq)}$$

$$\frac{x' \notin \mathbf{Dom}(R)}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\{x' \mapsto \mathbf{null}\},\ [x'/x]\, s, n\rangle} \quad \text{(Sem-LetNull)}$$

$$\frac{x' \notin \mathbf{Dom}(R)}{\langle H,\ R,\ \mathbf{let}\ x = y\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\{x' \mapsto R(y)\},\ [x'/x]\, s, n\rangle} \quad \text{(Sem-LetEq)}$$

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

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

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

$$\langle H\{R(x) \mapsto v\}, R, *x \leftarrow y, n\rangle \longrightarrow_D \langle H\{R(x) \mapsto R(y)\}, R, \mathbf{skip}, n\rangle \quad \text{(Sem-Assign)}$$

$$\frac{x' \notin \mathbf{Dom}(R) \qquad R(y) \in \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n\rangle \longrightarrow_D \langle H,\ R\{x' \mapsto H(R(y))\},\ [x'/x]\, s, n\rangle} \quad \text{(Sem-LetDeref)}$$

$$\frac{R(x) \neq \mathbf{null} \text{ and } R(x) \in \mathbf{Dom}(H)}{\langle H\{R(x) \mapsto v\},\ R,\ \mathbf{free}(x), n\rangle \xrightarrow{\mathbf{free}}_D \langle H \backslash R(x),\ R,\ \mathbf{skip}, n+1\rangle} \quad \text{(Sem-Free)}$$

$$\frac{l \notin \mathbf{Dom}(H) \qquad n > 0}{\langle H,\ R,\ \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, n\rangle \xrightarrow{\mathbf{malloc}}_D \langle H\{l \mapsto v\},\ R\{x' \mapsto l\},\ [x'/x]\, s, n-1\rangle} \quad \text{(Sem-Malloc)}$$

$$\frac{R(x) = \mathbf{null} \text{ or } R(x) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ *x \leftarrow y, n\rangle \longrightarrow_D \mathbf{MemEx}} \quad \text{(Sem-AssignExn)}$$

$$\frac{R(y) = \mathbf{null} \text{ or } R(y) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{let}\ x = *y\ \mathbf{in}\ s, n\rangle \longrightarrow_D \mathbf{MemEx}} \quad \text{(Sem-DerefExn)}$$

$$\frac{R(x) = \mathbf{null} \text{ or } R(x) \notin \mathbf{Dom}(H)}{\langle H,\ R,\ \mathbf{free}(x), n\rangle \xrightarrow{\mathbf{free}}_D \mathbf{MemEx}} \quad \text{(Sem-FreeExn)}$$

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

**Figure 2.** Operational semantics of $\mathcal{L}$.

$$\mathbf{0}; P \xrightarrow{\tau} P \quad \text{(Tr-Skip)} \qquad \mathbf{malloc} \xrightarrow{\mathbf{malloc}} \mathbf{0} \quad \text{(Tr-Malloc)} \qquad \mathbf{free} \xrightarrow{\mathbf{free}} \mathbf{0} \quad \text{(Tr-Free)}$$

$$\mu\alpha.P \xrightarrow{\tau} [\mu\alpha.P/\alpha]P \quad \text{(Tr-Rec)} \qquad P_1 + P_2 \xrightarrow{\tau} P_1 \quad \text{(Tr-ChoiceL)} \qquad P_1 + P_2 \xrightarrow{\tau} P_2 \quad \text{(Tr-ChoiceR)}$$

$$\frac{P_1 \xrightarrow{\rho} P_1'}{P_1; P_2 \xrightarrow{\rho} P_1'; P_2} \quad \text{(Tr-Seq)}$$

**Figure 3.** Semantics of behavioral types.

**Definition 1** (Memory leaks)**.** *A configuration* $\langle H, R, s, n\rangle$ goes overflow *if there is* $\sigma$ *such that* $\langle H, R, s, n\rangle \stackrel{\sigma}{\Longrightarrow} \mathbf{OutOfMemory}$*. A program* $\langle D, s\rangle$ requires at least $n$ cells *if* $\langle \emptyset, \emptyset, s, n\rangle$ *goes overflow. A program* $\langle D, s\rangle$ *is* totally memory-leak free *if there is a natural number* $n$ *such that it does not require more than* $n$ *cells.*

# 3 Type system

## 3.1 Types

The syntax of the types is as follows.

| | | | |
|---|---|---|---|
| $P$ | (behavioral types) | ::= | $\mathbf{0} \mid P_1; P_2 \mid P_1 + P_2 \mid \mathbf{malloc} \mid \mathbf{free} \mid \alpha \mid \mu\alpha.P$ |
| $\Gamma$ | (variable type environments) | ::= | $\{x_1, x_2, \ldots, x_n\}$ |
| $\Theta$ | (function type environments) | ::= | $\{f_1 : P_1, \ldots, f_n : P_n\}$ |

We use metavariable $P$ for *behavioral types*, which are abstractions of the behavior of programs. The type $\mathbf{0}$ represents do-nothing behavior. The type $P_1; P_2$ represents the sequential execution of $P_1$ and $P_2$. The type $P_1 + P_2$ represents the choice between $P_1$ and $P_2$. The type **malloc** represents an allocation of a memory cell exactly once. The type **free** represents a deallocation. $\alpha$ is a type variable. The type $\mu\alpha.P$ represents the behavior of $\alpha$ defined by the recursive equation $\alpha = P$.

Type environments for variables, ranged over by $\Gamma$, are set of variables. Since our interest is the behavior of a program, not the types of values, a variable type environment does not carry information on the types of variables. We also designate type environments for function names ranged over by $\Theta$. A function type environment carries information on the behavior of functions represented by behavioral types. We often omit the curly braces around variable and function environments.

The semantics of behavioral type are given by the labeled transition system in Figure 3. $P \xrightarrow{\rho} P'$ means that $P$ can make an action $\rho$ and, after $P$ make the action $\rho$, $P$ turns into $P'$.

## 3.2 Typing rules

We need several auxiliary definitions to present typing rules. We first define predicate $OK_n(P)$ that means that $P$ describes the behavior of a program that requires at most

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

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

$$\Theta;\Gamma, x, y \vdash *x \leftarrow y : \mathbf{0}\ \text{(T-Assign)} \qquad \Theta;\Gamma, x \vdash \mathbf{free}(x) : \mathbf{free}\ \text{(T-Free)}$$

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

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

$$\frac{\Theta;\Gamma, x \vdash s_1 : P \qquad \Theta;\Gamma, x \vdash s_2 : P}{\Theta;\Gamma, x \vdash \mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 : P}\ \text{(T-IfNull)} \qquad \Theta, f{:}P;\Gamma, \vec{x} \vdash f(\vec{x}) : P\ \text{(T-Call)}$$

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

$$\frac{\mathbf{Dom}(D) = \mathbf{Dom}(\Theta) \qquad \Theta; x_1, \ldots, x_n \vdash s{:}\Theta(f) \text{ for each } f \mapsto (x_1, \ldots, x_n)s \in D.}{\vdash D{:}\Theta}\ \text{(T-Def)}$$

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

**Figure 4.** Typing rules

$n$ memory cells.

**Definition 2** ($\sharp_\rho(\sigma)$)**.** *$\sharp_\rho(\sigma)$ is the number of $\rho$ in the sequence $\sigma$.*

**Definition 3.** *$OK_n(P)$ holds if, for any $P'$, if $P \xrightarrow{\sigma} P'$ then $\sharp_{\mathbf{malloc}}(\sigma) - \sharp_{\mathbf{free}}(\sigma) \leq n$.*

We also define subtyping. Following the idea of Kobayashi et al. [7], we define subtyping by using simulation between behavioral types.

**Definition 4** (Subtyping)**.** *$P_1 \leq P_2$ is the largest relation such that, for any $P_1'$ and $\rho$, if $P_1 \xrightarrow{\rho} P_1'$, then there exists $P_2'$ such that $P_2 \stackrel{\rho}{\Longrightarrow} P_2'$ and $P_2 \leq P_2'$.*

The type judgment for statements is of the form $\Theta;\Gamma \vdash s : P$. It expresses that, under the variable type environment $\Gamma$ and the function type environment $\Theta$, the behavior of $s$ is abstracted by behavior $P$.

Figure 4 shows the typing rules. It should be easy to observe correspondence between a construct in a statement and that in the behavioral type assigned to the statement.

For example, rule T-Malloc expresses that the behavior of **let** $x = \mathbf{malloc}()$ **in** $s$ is abstracted by $\mathbf{malloc}; P$ where $P$ is the behavior of $s$; this statement first allocates a memory cell once, and then executes $s$.

$\vdash D : \Theta$ is the judgment for function definitions. This judgment represents that the behavior assigned to each function symbol $f$ in $\Theta$ indeed abstracts the behavior of its function body. This property is guaranteed by T-Def.

The type judgment for programs, $\vdash \langle D, s\rangle : n$, expresses that the program does not require more than $n$ memory cells when it is run. In order to guarantee this, rule T-Prog forces a side condition $OK_n(P)$ where $P$ is a behavioral type assigned to $s$ under $\Theta$ assigned to $D$.

**Remark 3.1.** *The constructs $P_1 + P_2$ and $\mu\alpha.P$ may seem redundant since they do not appear in the typing rules. We designate these constructs so that there is (1) the join for arbitrary two behavioral types $P_1$ and $P_2$ and (2) the least fixed point for a map $\{\alpha \mapsto P\}$. Without these constructs, for example, a program* **ifnull** $(x)$ **then free**$(\mathbf{x})$ **else skip** *is not well-typed under a type environment $x$.*

### 3.3 Type soundness

The following theorem states soundness of the type system. The proof is in the full version [13].

**Theorem 3.1.** *If $\vdash \langle D, s\rangle : n$ for some $n$, then $\langle D, s\rangle$ is totally memory-leak free.*

The proof is based on the following lemmas: preservation and lack of immediate overflow.

**Lemma 3.2** (Preservation)**.** *If $OK_n(P)$, $\Theta; \Gamma \vdash s : P$, $\vdash D : \Theta$, and $\langle H, R, s, n\rangle \xrightarrow{\rho} \langle H', R', s', n'\rangle$, then there exists $P'$ such that (1) $\Theta; \Gamma' \vdash s' : P'$, (2) $P \stackrel{\rho}{\Longrightarrow} P'$, and (3) $OK_{n'}(P')$.*

**Lemma 3.3** (Lack of immediate overflow)**.** *If $\Theta; \Gamma \vdash s : P$, $\vdash D : \Theta$, and $OK_n(P)$, then $\langle H, R, s, n\rangle \not\xrightarrow{\mathbf{malloc}}$* **OutOfMemory**.

## 4 Type reconstruction

This section describes a type reconstruction procedure for the type system in Section 3[2]. Since the procedure is essentially the same as one in Kobayashi et al. [8], we do not give a concrete definition here.

The reconstruction procedure is a constraint-based one. It generates, given a program, constraints for the program to be well-typed by constructing a derivation tree based on the rules in Figure 4. A constraint is either a subtyping constraint $\alpha \geq P$, or $OK_\nu(\alpha)$, where $\nu$ is a symbol for an unknown natural number. Since T-Prog is the only place where the condition $OK_n(P)$ is involved, the constraint set includes exactly

[2]The procedure described here is not complete; see Section 7. We could reject a program that is well-typed in the type system.

one constraint of the form $OK_\nu(\alpha)$. The concrete definition of the constraint generation is in the full version [13].

By using the result obtained by Kobayashi et al. [8, Lemma 3.8], a subtyping constraint $\alpha \geq P$ can be resolved by setting $\alpha = \mu\alpha.P$, which is the least solution of the constraint. Hence, the generated constraint set is reduced to a single constraint $OK_\nu(P')$ for some behavioral type $P'$.

By definition, $OK_\nu(P)$ holds if there is a natural number $n$ such that, for all $\sigma$ and $P'$, $P \xrightarrow{\sigma} P'$ implies $\sharp_{\mathbf{malloc}}(\sigma) - \sharp_{\mathbf{free}}(\sigma) \leq n$. In order to check this condition soundly, we fix the upper bound of $\nu$ to be checked. Then, $OK_\nu(P)$ can be checked by model-checking a system with finitely many states; hence, model checkers like CPAChecker [2] and SPIN [1, 5] are applicable. More detailed description on how we apply a model-checking algorithm to check $OK_\nu(P)$ is in the forthcoming version.

# 5 Preliminary Experiments

## 5.1 Setup of the experiments

We conducted preliminary experiments to check feasibility and to investigate the problems in the current framework. For the following C programs, we extracted the behavioral types manually in the form of another C programs.

- `poker.c`: A program that models a poker game. It randomly deals cards to two players, compares the decks of the players, and decides the winner.
- `database.c`: A program that models a database management system. This program is taken from an online course of the C language [**?**]. The program allocates memory cells when it opens a database and deallocates the cells when it closes the database.
- `gen_init_cpio.c`: A module for a file system in the Linux kernel v.3.18.1. It allocates a memory cell when it creates a file. Deallocation is conducted in error-handling code.
- `decompress_unlzo.c`: A module in the Linux kernel v.3.18.1 that decompresses LZO files. It allocates a memory cell to store the contents obtained from an input LZO file. Deallocation occurs in error-handling code.

For verifying that each program consumes only fixed amount of memory cells, we applied CPAChecker [2] (1) to the original file and (2) to the file that represents the extracted behavior. All of the experiments are conducted on a machine with an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 8MB cache and 3.76GB memory, running on Debian (kernel version 2.6.32-5-amd64) and CPAchecker (version 1.3.4).

## 5.2 Result

Table 1 and Table 2 show the result. We present the number of lines of each file (`loc`), the number of functions (`nfun`), time spent by CPAChecker in seconds (`cpu time`), the

| | original programs | | | | | |
|---|---|---|---|---|---|---|
| | loc | nfun | cpu time | memory (MB) | fixed num | verified result |
| poker.c | 86 | 4 | 2.700 | 2797 | 4 | TRUE |
| database.c | 153 | 10 | 12.010 | 2907 | 2 | TRUE |
| gen_init_cpio.c | 346 | 19 | 9.580 | 2809 | 2 | TRUE |
| decompress_unlzo.c | 162 | 2 | 3.000 | 2806 | 2 | TRUE |

**Table 1.** Result of the verification of the original C programs.

| | abstracted behavior | | | | | |
|---|---|---|---|---|---|---|
| | loc | nfun | cpu time | memory (MB) | fixed num | verified result |
| poker.c | 16 | 4 | 1.980 | 2803 | 4 | FALSE |
| database.c | 16 | 4 | 2.060 | 2800 | 2 | FALSE |
| gen_init_cpio.c | 16 | 4 | 2.020 | 2802 | 2 | FALSE |
| decompress_unlzo.c | 16 | 4 | 1.970 | 2738 | 2 | FALSE |

**Table 2.** Result of the verification of the extracted behavior.

amount of memory in megabytes (`memory`), the upper-bound of the number of consumed memory cells (`fixed num`), and the result of the verification (`verified result`).

### 5.3 Discussion

CPAChecker, when it is applied to the original programs, was able to verify that the programs are totally memory-leak free. However, the verification failed for experiments with extracted behaviors. This is because our type system is not path-sensitive. For example, a typical pattern where verification with the extracted behaviors fail is as follows.

```
while (...) {
  if (/* some condition c */) {
    x = malloc(sizeof(int));
  }
  /* Do something */
  if (/* condition equivalent to c */) {
    free(x);
  }
}
```

For the program above, extracted behavior is $\mu\alpha.(\mathbf{0} + \mathbf{malloc}); (\mathbf{0} + \mathbf{free}); \alpha$, which is not enough to check no-memory-leak, although it is memory-leak free if the condition `c` does not change between the allocation and the deallocation. Our type system can deal with the program above by rewriting it to the following one.

```
while (...) {
  if (/* some condition c */) {
    x = malloc(sizeof(int));
    /* Do something */
    free(x);
  } else {
```

```
    /* Do something */
  }
}
```

For the rewritten program, the extracted behavior is $\mu\alpha.((\mathbf{malloc}; \mathbf{free}) + \mathbf{0}); \alpha$, for which the predicate $OK_1$ holds. We confirmed that CPAChecker can verify $OK_1$ for the extracted behavior of the rewritten programs without penalty on cpu time.

From the comparison of the Table 1 and Table 2, we can observe that the latter is more efficient than the former.

## 6 Related work

Many methods for static memory-leak freedom verification have been proposed [4, 9–12, 15]. These methods guarantee partial memory-leak freedom and lack of illegal accesses, whereas our type system guarantees total memory-leak freedom. By using both their methods and our type system, we can guarantee that a program correctly uses memory-allocation and memory-deallocation primitives even if the program does not terminate.

Behavioral types are extensively studied in the context of concurrent program verification [6–8, 14]. These type systems guarantee that the communication pattern of concurrent programs are as intended. Our type system is largely inspired by one proposed by Kobayashi et al. [8], which guarantees that a concurrent program accesses resources according to specification.

One possible approach to total memory-leak freedom verification would be checking that a program consumes only bounded number of memory cells by using a model checker [1–3]. We expect, from the result of the experiments, that the resource required for model checking would become smaller if we apply a model checker to the inferred behavior, which focuses on memory allocation and deallocation.

## 7 Conclusion

We have described a type system to verify memory-leak freedom for (possibly) non-terminating programs with manual memory-management primitives. Our type system abstracts the memory allocation/deallocation behavior of a program with a sequential process with actions corresponding to memory allocation and deallocation. We have described a type reconstruction algorithm for the type system.

Our current type system excludes many features of the real-world programs for simplification. We are currently looking at the C programs in the real world to investigate what extension we need to make to the type system. One feature we have already noticed is variable-sized memory blocks. The current behavioral types ignore the size of the allocated block, counting only the number of **malloc** and **free**. Hence, a program that contains memory leaks in usual sense may be well-typed in our type system. We need to refine the abstraction obtained by types in order to address this issue.

Currently, our type reconstruction first fixes an upper bound for $\nu$ in solving constraint $OK_\nu(P)$. This makes our reconstruction incomplete; even if constraint $OK_\nu(P)$

holds for some $\nu \geq n$, our procedure will reject the program if the fixed upper bound is less than $n$. We have not yet figured out whether a constraint of the form $\exists \nu. OK_\nu(P)$, which is needed to be solvable for reconstruction to be complete, is decidable or not.

**Acknowledgment** We thank the comments by the reviewers of PPL 2015. This research is partially supported by KAKENHI 25730040 and 25280024.

## References

[1] M. Ben-Ari. *Principles of the Spin model checker*. Springer, 2008.

[2] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

[3] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[4] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In R. Cytron and R. Gupta, editors, *PLDI*, pages 168–181. ACM, 2003.

[5] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[6] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998.

[7] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[8] N. Kobayashi, K. Suenaga, and L. Wischik. Resource usage analysis for the $\pi$-calculus. *Logical Methods in Computer Science*, 2(3), 2006.

[9] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In K. Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 405–424. Springer, 2006.

[10] K. Suenaga and N. Kobayashi. Fractional ownerships for safe memory deallocation. In Z. Hu, editor, *APLAS*, volume 5904 of *Lecture Notes in Computer Science*, pages 128–143. Springer, 2009.

[11] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In M. P. E. Heimdahl and Z. Su, editors, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, pages 254–264. ACM, 2012.

[12] N. Swamy, M. W. Hicks, G. Morrisett, D. Grossman, and T. Jim. Safe manual memory management in cyclone. *Sci. Comput. Program.*, 62(2):122–144, 2006.

[13] Q. Tan, K. Suenaga, and A. Igarashi. A behavioral type system for memory-leak freedom. Available from `http://www.fos.kuis.kyoto-u.ac.jp/~tanki/memoryleak.pdf`.

[14] H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In S. Drossopoulou, editor, *ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008.

[15] Y. Xie and A. Aiken. Context- and path-sensitive memory leak detection. In M. Wermelinger and H. Gall, editors, *ESEC/SIGSOFT FSE*, pages 115–125. ACM, 2005.

# Appendix

## A Proof of Lemmas

**Lemma A.1.** *If* $OK_n(P)$ *and* $P \xrightarrow{\rho} P'$*, then*

- $OK_{n-1}(P')$ *if* $\rho = \mathbf{malloc}$*,*
- $OK_{n+1}(P')$ *if* $\rho = \mathbf{free}$*, and*
- $OK_n(P')$ *if* $\rho = \tau$*.*

*Proof.* Case analysis on $P \xrightarrow{\rho} P'$.

- Case $P = \mathbf{0}; P'$ and $\mathbf{0}; P' \xrightarrow{\tau} P'$

  We need to prove $OK_n(P')$. Assume that $OK_n(P')$ does not hold. Then, we have $\mathbf{0}; P' \xrightarrow{\tau} P' \xrightarrow{\sigma} Q$ such that $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ for some $\sigma$. From the definition of $OK_n(P)$, we have $\sharp_m(\tau\sigma) - \sharp_f(\tau\sigma) \leq n$. From the definition of $\sharp_\rho(\sigma)$, we get $\sharp_m(\tau) + \sharp_m(\rho) - \sharp_f(\tau) - \sharp_f(\rho) \leq n$ that is equivalent to $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$, which is contradiction.

- Case $P = \mathbf{malloc}$ and $\mathbf{malloc} \xrightarrow{\mathbf{malloc}} \mathbf{0}$

  We need to prove $OK_{n-1}(\mathbf{0})$. From the definition of $OK_n(P)$, we have that $OK_{n-1}(\mathbf{0})$ holds if, for any $P'$, if $\mathbf{0} \xrightarrow{\sigma} P'$ then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n - 1$. There is no such $P'$ s.t. $\mathbf{0} \xrightarrow{\sigma} P'$, therefore, $OK_{n-1}(\mathbf{0})$ holds.

- Case $P = \mathbf{free}$ and $\mathbf{free} \xrightarrow{\mathbf{free}} \mathbf{0}$.

  We need to prove $OK_{n+1}(\mathbf{0})$. From the definition of $OK_n(P)$, we have that $OK_{n+1}(\mathbf{0})$ holds if, for any $P'$, if $\mathbf{0} \xrightarrow{\sigma} P'$ then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n + 1$. There is no such $P'$ s.t. $\mathbf{0} \xrightarrow{\sigma} P'$, therefore, $OK_{n+1}(\mathbf{0})$ holds.

- Case $P = P_1 + P_2$ and $P_1 + P_2 \xrightarrow{\tau} P_1$

  We prove $OK_n(P_1)$ by contradiction. Assume that $OK_n(P_1)$ does not hold. Then, there exists $\sigma$ such that $P_1 + P_2 \xrightarrow{\tau} P_1 \xrightarrow{\sigma} Q$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$. From the definition of $OK_n(P)$, we have $\sharp_m(\tau\sigma) - \sharp_f(\tau\sigma) \leq n$. Then, we have $\sharp_m(\tau) + \sharp_m(\rho) - \sharp_f(\tau) - \sharp_f(\rho) \leq n$. By the definition of $\sharp_\rho(\sigma)$, we get $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$. Therefore, we get the contradiction.

- Case $P = P_1 + P_2$ and $P_1 + P_2 \xrightarrow{\tau} P_2$

  Similar to the case of $P_1 + P_2 \xrightarrow{\tau} P_1$.

- Case $P = P_1; P_2$ and $P_1; P_2 \xrightarrow{\rho} P_1'; P_2$.

  We need to prove $OK_{n'}(P_1'; P_2)$ where $n'$ is determined by

$$n' = \begin{cases} n+1 & \rho = \mathbf{free} \\ n-1 & \rho = \mathbf{malloc} \\ n & \text{Otherwise.} \end{cases}$$

Assume that $OK_{n'}(P_1'; P_2)$ does not hold. Then, there exists $\sigma$ such that $P_1; P_2 \xrightarrow{\rho} P_1'; P_2 \xrightarrow{\sigma} Q$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n'$. From the definition of $OK_n(P)$, we have $\sharp_m(\rho\sigma) - \sharp_f(\rho\sigma) \leq n$; hence we have $\sharp_m(\rho) + \sharp_m(\sigma) - \sharp_f(\rho) - \sharp_f(\sigma) \leq n$. From the assumption $\sharp_m(\sigma) - \sharp_f(\sigma) > n'$, we have

$$n' + \sharp_m(\rho) - \sharp_f(\rho) < \sharp_m(\rho) + \sharp_m(\rho_1) - \sharp_f(\rho) - \sharp_f(\rho_1) \leq n.$$

The leftmost term $n' + \sharp_m(\rho) - \sharp_f(\rho)$ is equal to $n$ for any $\rho$. Hence, we get contradiction.

- Case $P = \mu\alpha.P'$ and $\mu\alpha.P' \xrightarrow{\tau} [\mu\alpha.P'/\alpha]P'$

  We need to prove $OK_n([\mu\alpha.P'/\alpha]P')$. Suppose not. Then, there exists $\sigma$ and $Q$ such that $[\mu\alpha.P'/\alpha]P' \xrightarrow{\sigma} Q$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$. However, this contradicts to $P \xrightarrow{\tau\sigma} Q$ and $OK_n(P)$ as follows.

$$n \geq \sharp_m(\tau\sigma) - \sharp_f(\tau\sigma) = \sharp_m(\sigma) - \sharp_f(\sigma) > n.$$

□

*Proof of Lemma 3.2*:
By induction on the derivation of $\langle H, R, s, n\rangle \xrightarrow{\rho} \langle H', R', s', n'\rangle$.

- Case: $\langle H, R, \mathbf{free}(\mathbf{x})\ , n\rangle \xrightarrow{\mathbf{free}} \langle H', R', \mathbf{skip}, n+1\rangle$.

  We have $OK_n(P)$ and $\Theta; \Gamma \vdash \mathbf{free}(x) : P$. From inversion of the typing rules, we have $\Theta; \Gamma \vdash \mathbf{free}(x) : \mathbf{free}$ and $\mathbf{free} \leq P$. Hence, from the definition of subtyping, we have $\mathbf{0} \leq P''$ and $P \xRightarrow{\mathbf{free}} P''$ for some $P''$. We need to find $P'$ such that $P \xRightarrow{\mathbf{free}} P'$, $\Theta; \Gamma \vdash \mathbf{skip} : P'$, and $OK_{n+1}(P')$. Take $P''$ as $P'$. Then, $P \xRightarrow{\mathbf{free}} P''$ as we stated above. We also have $\Theta; \Gamma \vdash \mathbf{skip} : P''$ from T-Skip, $\mathbf{0} \leq P''$, and T-Sub. $OK_{n+1}(P'')$ follows from Lemma A.1.

- Case: $\langle H, R, \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, n\rangle \xrightarrow{\mathbf{malloc}} \langle H', R', [x'/x]s, n-1\rangle$.

  From the assumption, we have $\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s : P$ and $OK_n(P)$. By the inversion of typing rules, we have $\mathbf{malloc}; P_1 \leq P$ and $\Theta; \Gamma \vdash s : P_1$ for some $P_1$. We have the following derivation:

$$\frac{\mathbf{malloc} \xrightarrow{\mathbf{malloc}} 0}{\mathbf{malloc}; P_1 \xrightarrow{\mathbf{malloc}} 0; P_1}$$

  and $0; P_1 \to P_1$, then we have $\mathbf{malloc}; P_1 \xRightarrow{\mathbf{malloc}} P_1$. Hence, By the definition of subtyping, we have $P \xRightarrow{\mathbf{malloc}} P''$ and $P_1 \leq P''$ for some $P''$.

  We need to find $P'$ and $\Gamma'$ such that $\Theta; \Gamma' \vdash [x'/x]s : P'$ and $P \xRightarrow{\mathbf{malloc}} P'$. Take $P''$ as $P'$ and $[x'/x]\Gamma$ as $\Gamma'$. (Note that $[x'/x]\Gamma$ is well-formed because $x'$ is fresh.)

Then $P \xRightarrow{\textbf{malloc}} P''$ as we state above. We also have $\Theta; [x'/x]\Gamma \vdash [x'/x]s : P''$ from T-Sub, $\Theta; [x'/x]\Gamma \vdash [x'/x]s : P_1$, and $P_1 \leq P''$. $OK_{n-1}(P'')$ follows from Lemma A.1.

- Case: $\langle H, R, \textbf{skip}; s, n\rangle \rightarrow \langle H, R, s, n\rangle$.

  we have $\Theta; \Gamma \vdash \textbf{skip}; s : P$ and $OK_n(P)$. From the inversion of the typing rules, we have $\Theta; \Gamma \vdash s : P_1$ and $0; P_1 \leq P$. Hence, from the definition of subtyping, we have $P \xRightarrow{\tau} P''$ and $P_1 \leq P''$ for some $P''$.

  We need to find $P'$ such that $\Theta; \Gamma \vdash s : P'$ and $P \xRightarrow{\tau} P'$. Take $P''$ as $P'$. Then $P \xRightarrow{\tau} P''$ as we stated above. We also have $\Theta; \Gamma \vdash s : P''$ from T-Sub, $\Gamma \vdash s : P_1$ and $P_1 \leq P''$. $OK_n(P'')$ follows from Lemma A.1

- Case: $\langle H, R, *x \leftarrow y, n\rangle \rightarrow \langle H', R', \textbf{skip}, n\rangle$.

  We have $\Theta; \Gamma \vdash *x \leftarrow y : P$ and $OK_n(P)$. From the inversion of typing rules, we have $0 \leq P$.

  We need to find $P'$ such that $\Theta; \Gamma \vdash \textbf{skip} : P'$, $P \xRightarrow{\tau} P'$ and $OK_n(P')$. Take $P$ as $P'$. Then, $P \xRightarrow{\tau} P'$ and $OK_n(P')$ hold. We also have $\Theta; \Gamma \vdash \textbf{skip} : P'$ from T-Skip, $0 \leq P$ and T-Sub.

- Case: $\langle H, R, \textbf{let } x = y \textbf{ in } s, n\rangle \rightarrow \langle H', R', [x'/x]s, n\rangle$.

  We have $\Theta; \Gamma \vdash \textbf{let } x = y \textbf{in } s : P$ and $OK_n(P)$. From the inversion of typing rules, we have $\Theta; \Gamma \vdash s : P_1$ and $P_1 \leq P$.

  We need to find $P'$ and $\Gamma'$ such that $\Theta; \Gamma' \vdash [x'/x]s : P'$ , $P \xRightarrow{\tau} P'$ and $OK_n(P')$. Take $P$ as $P'$ and $[x'/x]\Gamma$ as $\Gamma'$. Then $P \xRightarrow{\tau} P'$ and $OK_n(P')$ hold. We also have $\Theta; \Gamma \vdash [x'/x]s : P$ from T-Sub, $\Theta; \Gamma \vdash [x'/x]s : P_1$ and $P_1 \leq P$.

- Case: $\langle H, R, \textbf{let } x = \textbf{null in } \ s, n\rangle \rightarrow \langle H', R', [x'/x]s, n\rangle$

  We have $\Theta; \Gamma \vdash \textbf{let } x = \textbf{null in } \ s : P$ and $OK_n(P)$. From the inversion of typing rules, we have $\Theta; \Gamma \vdash s : P_1$ and $P_1 \leq P$.

  We need to find $P'$ and $\Gamma'$ such that $\Theta; \Gamma' \vdash [x'/x]s : P'$, $P \xRightarrow{\tau} P'$ and $OK_n(P')$. Take $P$ as $P'$ and $[x'/x]\Gamma$ as $\Gamma'$. Then, $P \xRightarrow{\tau} P'$ and $OK_n(P')$ hold. We also have $\Theta; \Gamma \vdash [x'/x]s : P$ from T-Sub, $\Theta; \Gamma \vdash [x'/x]s : P_1 P_1 \leq P$.

- Case: $\langle H, R, \textbf{let } x = *y \textbf{ in } \ s, n\rangle \rightarrow \langle H', R', [x'/x]s, n\rangle$

  We have $\Theta; \Gamma \vdash \textbf{let } x = *y \textbf{ in } \ s : P$ and $OK_n(P)$. From the inversion of typing rules, we have $\Theta; \Gamma \vdash s : P_1$ and $P_1 \leq P$.

  We need to find $P'$ and $\Gamma'$ such that $\Theta; \Gamma' \vdash [x'/x]s : P'$, $P \xRightarrow{\tau} P'$ and $OK_n(P')$. Take $P$ as $P'$ and $[x'/x]\Gamma$ as $\Gamma'$. Then, $P \xRightarrow{\tau} P'$ and $OK_n(P')$ hold. We also have $\Theta; \Gamma' \vdash [x'/x]s : P$ from T-Sub, $\Theta; \Gamma' \vdash [x'/x]s : P_1$ and $P_1 \leq P$.

- Case: $\langle H, R, \textbf{ifnull } (x) \textbf{ then } s_1 \textbf{ else } \ s_2, n\rangle \rightarrow \langle H, R, s_1, n\rangle$

We have $\Theta;\Gamma \vdash \textbf{ifnull}\ (x)\ \textbf{then}\ \ s_1\ \textbf{else}\ \ s_2 : P$ and $OK_n(P)$. From the inversion of typing rules, we have $\Theta;\Gamma \vdash s_1 : P_1$ and $P_1 \leq P$.

We need to find $P'$ such that $\Theta;\Gamma \vdash s_1 : P'$ , $P \stackrel{\tau}{\Longrightarrow} P'$ and $OK_n(P')$. Take $P$ as $P'$. Then, $OK_n(P')$ and $P \stackrel{\tau}{\Longrightarrow} P'$ hold. We also have $\Theta;\Gamma' \vdash s_1 : P$ from $\Theta;\Gamma \vdash s_1 : P_1$, $P_1 \leq P$ and T-Sub.

- Case(Tr-IfNullF): $\langle H, R, \textbf{ifnull}\ (x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n\rangle \rightarrow \langle H, R, s_2, n\rangle$.

  The proof is similar to case of $\langle H, R, \textbf{ifnull}\ (x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n\rangle \rightarrow \langle H, R, s_1, n\rangle$.

- Case: $\langle H, R, f(\vec{x}), n\rangle \rightarrow \langle H, R, [\vec{x}/\vec{y}]s, n\rangle$

  From the inversion of T-Call, we have $\Theta;\Gamma \vdash f(\vec{x}) : \Theta(f)$ and $\Theta(f) \leq P$. From Sem-Call, we have $D(f) = (\vec{y})s$. From the assumption $\vdash D : \Theta$, we have $\Theta;\vec{y} \vdash s : \Theta(f)$; hence, with $\Theta(f) \leq P$, we have $\Theta;\vec{y} \vdash s : P$. We then have $\Theta;\vec{x} \vdash [\vec{x}/\vec{y}]s : P$ as required.

□

**Corollary A.2.** *If $OK_n(P)$, $\Theta;\Gamma \vdash s : P$, $\vdash D : \Theta$, and $\langle H, R, s, n\rangle \stackrel{\sigma}{\longrightarrow} \langle H', R', s', n'\rangle$, then there exists $P'$ such that (1) $\Theta;\Gamma \vdash s' : P'$, (2) $P \stackrel{\sigma}{\Longrightarrow} P'$, and (3) $OK_{n'}(P')$.*

We write $\langle H, R, s, n\rangle \stackrel{\rho}{\longrightarrow}$ if there is a transition $\stackrel{\rho}{\longrightarrow}$ from $\langle H, R, s, n\rangle$.

**Lemma A.3.** *If $\Theta;\Gamma \vdash s : P$ and $\langle H, R, s, n\rangle \stackrel{\rho}{\longrightarrow}$ and $\rho \in \{\textbf{malloc}, \textbf{free}\}$, then there exists $P'$ such that $P \stackrel{\rho}{\Longrightarrow} P'$.*

*Proof.* Induction on the derivation of $\Theta;\Gamma \vdash s : P$. □

*Proof of Lemma 3.3*:
By contradiction. Assume $\langle H, R, s, n\rangle \stackrel{\rho}{\longrightarrow} \textbf{OutOfMemory}$. Then, $n$ is 0 and $\rho = \textbf{malloc}$ from Sem-OutOfMem. From the assumption $\Theta;\Gamma \vdash s : P$ and $OK_0(P)$. From Lemma A.3, there exists $P'$ such that $P \stackrel{\textbf{malloc}}{\Longrightarrow} P'$. However, this contradicts $OK_0(P)$.

□

*Proof of Theorem 3.1*:
We have $\Theta;\emptyset \vdash s : P, \vdash D : \Theta$ and $OK_n(P)$. Suppose that there exists $\sigma$ such that $\langle\emptyset, \emptyset, s, n\rangle \stackrel{\sigma}{\longrightarrow} \langle H', R', s', n'\rangle \stackrel{\rho}{\longrightarrow} \textbf{OutOfMemory}$. Then, $n' = 0$ and $\rho = \textbf{malloc}$. From Lemma A.2, there exists $P'$ such that $\Theta;\Gamma' \vdash s : P'$, $P \stackrel{\sigma}{\Longrightarrow} P'$, and $OK_0(P')$; hence $\langle H', R', s', 0\rangle \stackrel{\textbf{malloc}}{\longrightarrow}$. However, this contradicts Lemma 3.3.

□

# B Syntax Directed Typing Rules

Figure 5 shows the syntax-directed version of the typing rules in Figure 4. Based on these rules, we design a constraint generation algorithm for the type reconstruction, which is presented in Figure 6.

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

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

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

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

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

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

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

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

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

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

**Figure 5.** Syntax Directed Typing Rules.

$PT_\Theta(f) =$

$\quad$ **let** $\alpha = \Theta(f)$

$\quad$ **in** $(C = \{\alpha \leq \beta\}, \beta)$

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

$PT_\Theta(s_1; s_2) =$

$\quad$ **let** $(C_1, P_1) = PT_\Theta(s_1)$

$\quad\quad (C_2, P_2) = PT_\Theta(s_2)$

$\quad$ **in** $(C_1 \cup C_2 \cup \{P_1; P_2 \leq \beta\}, \beta)$

$PT_\Theta(*x \leftarrow y) =$

$\quad$ **let** $(C_1, \emptyset) = PT_v(*x)$

$\quad\quad (C_2, \emptyset) = PT_v(y)$

$\quad$ **in** $(C_1 \cup C_2, 0)$

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

$PT_\Theta(\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s) =$

$\quad$ **let** $(C_1, P_1) = PT_v(s)$

$\quad$ **in** $(C_1 \cup \{P_1 \leq \beta\}, \mathbf{malloc}; \beta)$

$PT_\Theta(\mathbf{let}\ x = y\ \mathbf{in}\ s) =$

$\quad$ **let** $(C_1, \emptyset) = PT_v(y)$

$\quad\quad (C_2, P_1) = PT_\Theta(s)$

$\quad$ **in** $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)$

$PT_\Theta(\mathbf{let}\ x = *y\ \mathbf{in}\ s) =$

$\quad$ **let** $(C_1, \emptyset) = PT_v(y)$

$\quad\quad (C_2, P_1) = PT_\Theta(s)$

$\quad$ **in** $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \beta)$

$PT_\Theta(\mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2) =$

$\quad$ **let** $(C_1, P_1) = PT_\Theta(s_1)$

$\quad\quad (C_2, P_2) = PT_\Theta(s_2)$

$\quad\quad (C_3, \emptyset) = PT_v(x)$

$\quad$ **in** $(C_1 \cup C_2 \cup C_3 \cup \{P_1 \leq \beta, P_2 \leq \beta\}, \beta)$

$PT(\langle D, s\rangle) =$

$\quad$ **let** $\Theta = \{f_1 : \alpha_1, \ldots, f_n : \alpha_n\}$

$\quad\quad$ *where* $\{f_1, \ldots, f_n\} = dom(D)$ *and* $\alpha_1, \ldots, \alpha_n$ *are fresh*

$\quad$ **in let** $(C_i, P_i) = PT_\Theta(D(f_i))$ *for each* $i$

$\quad$ **in let** $C_i^{'} = \{\alpha_i \leq P_i\}$ *for each* $i$

$\quad$ **in let** $(C, P) = PT_\Theta(s)$

$\quad$ **in** $(C_i \cup C_i^{'}) \cup C \cup \{OK(P)\}, P)$

**Figure 6.** Type Inference Algorithm