A Behavioral Type System for Memory-Leak Freedom

Qi Tan, Kohei Suenaga, 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 CCS-like 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

1.1 Motivation and Problems

Manual memory management primitives (e.g., `malloc` and `free` in C) often cause that forgetting to deallocate memory cells after use, which we call *memory leaks*. It can diminish the performance of the computer by reducing the amount of available memory cells. Memory leaks may not be serious or even detectable by normal means. Normal memory used by an application is released when application terminates. This means that a memory leak in a program that only runs for a short time may not be noticed and is rarely serious. However, in the real-world programs, nonterminating programs such as Web servers and operating systems are very important. If memory leaks in such nonterminating programs, eventually, too much of the available memory cells may become allocated and all or part of the system stops working correctly [**?**].

Example 1.1. The functions h and h' shown in Figure 1 describe memory-leak freedom and memory leaks in nonterminating programs. Function h requires two memory cells at most, whereas function h' requires unbounded number of memory cells to be executed.

1	$h(x)=$	$h'(x)=$
2	**let** $x = \mathbf{malloc}()$ **in**	**let** $x = \mathbf{malloc}()$ **in**
3	**let** $y = \mathbf{malloc}()$ **in**	**let** $y = \mathbf{malloc}()$ **in**
4	$\mathbf{free}(x); \mathbf{free}(y); h(x)$	$h'(x); \mathbf{free}(x);\ \mathbf{free}(y)$

Figure 1. memory leaks in nonterminating programs.

1.2 Approach

2 Language

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

Notation. we write $\vec{x}$ for a sequence $x_1 \ldots x_n$.
$f\{x \to v\}$ denotes
$[v/x]s$ denotes

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

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 to by a pointer x. Command **let** $x = e$ **in** s first evaluates the expression e, binds x to the result, and then executes statement s. The command **let** $x = \mathbf{malloc}()$ **in** s first allocates a memory cell, binds x to the pointer to the cell then executes the statement s. The command **let** $x =$ **null in** s binds x to a null pointer and then executes s. The command **let** $x = y$ **in** s binds x to y and then executes statement s. The command **let** $x = *y$ **in** s binds x to the contents of the cell pointered to by y and executes statement s. The command **ifnull** (x) **then** s_1 **else** s_2 executes the statement s_1 if the pointer x is a null pointer, executes statement s_2 otherwise. The command $f(\vec{x})$ is a function call f with arguments $\vec{x}$; we assume that the sequence $\vec{x}$ is pairwise-distinct. d denotes the definition of function $f(\vec{x})$ which has a body of statement s.

2.1 Operational Semantics

Runtime state is represented by $\langle H, R, s, n \rangle$, where Operational semantics is defined by the relations

$\rightarrow_D, \xrightarrow{\textbf{malloc}}_D, and \xrightarrow{\textbf{free}}_D$ defined by the rules in ....
Here, $\rightarrow_D$ expresses; ....

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Operational Semantics

Definition of memory leak

3 Type System

3.1 Syntax of Types

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

The type **0** abstracts the behavior of **skip** and means "does nothing". $P_1; P_2$ is for sequential execution. $P_1 + P_2$ is abstracted as conditional. **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 $\mu\alpha.\textbf{malloc}; \textbf{malloc}; \textbf{free}; \textbf{free}; \alpha$. 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:

$$\mathbf{0}; P \to P$$

$$\textbf{malloc} \xrightarrow{\textbf{malloc}} 0$$

$$\textbf{free} \xrightarrow{\textbf{free}} 0$$

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

$$P_1 + P_2 \longrightarrow P_1$$

$$P_1 + P_2 \longrightarrow P_2$$

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

The notation $\to$ 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 $\{\textbf{malloc}, \textbf{free}\}$.

3.3 Typing Rules

The type judgment of our type system is given by the form $\Theta; \Gamma \vdash s : P$, where Θ is a mapping from function variables 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 judgment implies the property: when s executes **malloc**(resp.**free**), then P is equivalent to **malloc**; P'(resp.**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 **0**. The rule for **free** represents that the behavior of **free**(x) is **free**. The rule T-Malloc represents that **let** $x =$ **malloc**() **in** s has the behavior **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. $P_2 \stackrel{\alpha}{\Longrightarrow} P_2'$ 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 $\stackrel{\alpha}{\Longrightarrow}$ means that: $\rightarrow^* \xrightarrow{\alpha} \rightarrow^*$.

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 functions 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* $\sharp_{malloc}(\alpha) - \sharp_{free}(\alpha) \leq n$.

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

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

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

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

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

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

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

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

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

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

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

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

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.* $\langle \emptyset, \emptyset, s, n\rangle \twoheadrightarrow^* Error$

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) $P \overset{\alpha}{\Longrightarrow} P'$
(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* $\vdash \langle H', R', s', n'\rangle \nrightarrow Error$

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 Constraint Generation

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. $PT(\langle D, s\rangle) = (C, P)$ denotes that it receives a program and produces a pair (C, P). α_i and β are fresh type variables.

5 Constraints Reduction

6 Preliminary Experiment

7 Related Work

8 Conclusion

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 consumption given a behavioral type and planning to implement a verifier and conduct experiment to see whether our approach is feasible.

$PT_\Theta(f) =$

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

Appendix

1. Proof for Lemma Preservation

By induction on the derivation of evaluation rules.

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

From the assumption, we have known that: ① $OK_n(P)$, and ② $\Theta;\Gamma \vdash \mathbf{free}(x) : P$.
By the inversion lemma on ②, we have: ③ $\mathbf{free} \leq P$.
From the definition of subtyping, ③ and rule $\mathbf{free} \xrightarrow{\mathbf{free}} 0$, we get:

$$\exists P'' \text{ s.t. } ④\ P \overset{\mathbf{free}}{\Longrightarrow} P'', \text{ and } ⑤\ 0 \leq P''$$

We need to prove that there exists P' and Γ' such that:

$$⑥\ \Theta;\Gamma' \vdash \mathbf{skip} : P', \text{ and } ⑦\ P \overset{\mathbf{free}}{\Longrightarrow} P'$$

Take P'' as P'. Then ⑦ holds. By the typing rule T-Skip and ⑤, we get:

$$\frac{\Theta;\Gamma' \vdash \mathbf{skip} : 0 \quad 0 \leq P''}{\Theta;\Gamma' \vdash \mathbf{skip} : P''} \quad \text{(T-Sub)}$$

Therefore, ⑥ holds.

Case: $\langle H, R, \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s_1, n\rangle \xrightarrow{\mathbf{malloc}} \langle H', R', [x'/x]s_1, n-1\rangle$.

From the assumption, we already have ①$\Theta;\Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s_1 : P$,
and ② $OK_n(P)$.
By the inversion lemma and ①, we have ③ $\mathbf{malloc}; P_1 \leq P$, and ④ $\Theta;\Gamma \vdash s_1 : P_1$
We need to find P' and Γ' such that ⑤ $\Theta;\Gamma' \vdash s_1 : P'$, and ⑥$P \overset{\mathbf{malloc}}{\Longrightarrow} P'$
Because of the following derivation:

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

and $0; P_1 \Rightarrow P_1$. Therefore $\mathbf{malloc}; P_1 \xrightarrow{\mathbf{malloc}} P_1$.
By the definition of subtyping and $\mathbf{malloc}; P_1 \xrightarrow{\mathbf{malloc}} P_1$, we have that:

$$\exists P'' \text{ s.t. } ⑦\ P \overset{\mathbf{malloc}}{\Longrightarrow} P'', \text{ and } ⑧\ P_1 \leq P''$$

Taking P'' as P', then ⑥ holds.
And by using subtyping rule T-Sub with premises ④ and ⑧

$$\frac{\Gamma \vdash s_1 : P_1 \quad P_1 \leq P''}{\Gamma \vdash s_1 : P''} \quad \text{(T-Sub)}$$

Therefore we prove that $\Gamma \vdash s_1 : P'$, ⑤ holds.

Case: $\langle H, R, \mathbf{skip}; s_1, n\rangle \to \langle H', R', s_1, n\rangle$.

From the assumption, we have

$$\text{① } \Theta; \Gamma \vdash \mathbf{skip}; s_1 : P, \text{ and ② } OK_n(P)$$

By the inversion lemma on ①, we have

$$\text{③ } \Theta; \Gamma \vdash s_1 : P_1, \text{ and ④ } 0; P_1 \leq P$$

We need to prove that there exists P' and Γ' such that

$$\text{⑤ } \Theta; \Gamma' \vdash s_1 : P', \text{ and ⑥ } P \to^* P'$$

By the definition of subtyping and $0; P_1 \to P_1$, then we get that $\exists P''$

$$\text{⑦ } P \to^* P'', \text{ and ⑧ } P_1 \leq P''$$

Taking P'' as P', we get $P \to^* P'$
And by using rule T-Sub with premises $\Gamma \vdash s_1 : P_1$ and $P_1 \leq P''$, then we have

$$\frac{\Theta; \Gamma \vdash s_1 : P_1 \quad P_1 \leq P''}{\Gamma \vdash s_1 : P''} \quad \text{(T-Sub)}$$

Therefore, we prove that $\Gamma \vdash s_1 : P'$

Case: $\langle H, R, *x \leftarrow y, n\rangle \to \langle H', R', \mathbf{skip}, n\rangle$.

From the assumption, we already have

$$\text{① } \Theta; \Gamma \vdash *x \leftarrow y : P, \text{ and ② } OK_n(P)$$

From the inversion lemma on ①, we have ③ $0 \leq P$.
We need to find P' and Γ' such that

$$\text{④ } \Theta; \Gamma' \vdash \mathbf{skip} : P', \text{ and ⑤ } P \to^* P'$$

Taking P as P', then ⑤ holds.
And because of the following derivation:

$$\frac{\Theta; \Gamma' \vdash \mathbf{skip} : 0 \quad 0 \leq P}{\Theta; \Gamma' \vdash \mathbf{skip} : P} \quad \text{(T-Sub)}$$

therefore ④ holds.

Case: $\langle H, R, \mathbf{let}\ x = y\ \mathbf{in}\ s_1, n\rangle \to \langle H', R', [x'/x]s_1, n\rangle$.

From assumption, we have

$$\textcircled{1}\ \Theta;\Gamma \vdash \textbf{let}\ x = y\ \textbf{in}\ \ s_1 : P, \text{ and } \textcircled{2}\ OK_n(P).$$

From the inversion lemma and ①, we have

$$\textcircled{3}\ \Theta;\Gamma \vdash s_1 : P_1, \text{ and } P_1 \leq P.$$

We need to find P' and Γ' such that:

$$\Theta;\Gamma' \vdash s_1 : P' \ \ and \tag{1}$$
$$P \xrightarrow{\tau}{}^{*} P' \tag{2}$$

Taking P as P'. Therefore (2) holds, because of the definition of $\rightarrow^*$.
And because of the following derivation, (**??**) holds.

$$\frac{\Theta;\Gamma' \vdash s_1 : P_1 \quad P_1 \leq P}{\Theta;\Gamma' \vdash s_1 : P} \quad \text{(T-Sub)}$$

Case: $\langle H, R, \textbf{let}\ x = \textbf{null}\ \ \textbf{in}\ \ s_1, n\rangle \rightarrow \langle H', R', [x'/x]s_1, n\rangle$

From the assumption, we know that

$$\textcircled{1}\Theta;\Gamma \vdash \textbf{let}\ x = \textbf{null}\ \ \textbf{in}\ \ s_1 : P, \text{ and } \textcircled{2}\ OK_n(P).$$

By inversion lemma on (**??**), we get:

$$\Theta;\Gamma \vdash s_1 : P_1, \text{ and } P_1 \leq P.$$

We need to prove that there exists P' and Γ' such that

$$\Theta;\Gamma' \vdash s_1 : P', \text{ and } P \rightarrow^* P'.$$

Taking P as P'. Because of the following derivation, the (**??**) holds.

$$\frac{\Theta;\Gamma' \vdash s_1 : P_1 \quad P_1 \leq P}{\Theta;\Gamma' \vdash s_1 : P} \quad \text{(T-Sub)}$$

And because of the definition of $\xrightarrow{\tau}{}^{*}$, the (**??**) holds.

Case: $\langle H, R, \textbf{let}\ x = *y\ \textbf{in}\ \ s_1, n\rangle \rightarrow \langle H', R', [x'/x]s_1, n\rangle$

From the assumption, we know that

$$\Theta;\Gamma \vdash \textbf{let}\ x = *y\ \textbf{in}\ \ s_1 : P, \text{ and } OK_n(P).$$

By the inversion lemma on (**??**), we get:

$$\Theta;\Gamma \vdash s_1 : P_1, \text{ and } P_1 \leq P.$$

We need to prove there exists P' and Γ' such that:

$$\Theta; \Gamma' \vdash s_1 : P', \text{ and } P \to^* P'.$$

Taking P as P'. Because of following derivation, the (**??**) holds.

$$\frac{\Theta; \Gamma' \vdash s_1 : P_1 \quad P_1 \leq P}{\Theta; \Gamma' \vdash s_1 : P} \quad \text{(T-Sub)}$$

And because of the definition of $\xrightarrow{\tau}{}^*$, (**??**) holds.

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

From the assumption, we have that:

$$\Theta; \Gamma \vdash \textbf{ifnull}\ (x)\ \textbf{then}\ \ s_1\ \textbf{else}\ \ s_2 : P, \text{ and } OK_n(P).$$

By the inversion leamma on (**??**), we get:

$$\Theta; \Gamma \vdash s_1 : P_1, \text{ and } p_1 \leq P'.$$

We need to prove that there exists P' and Γ' such that:

$$\Theta; \Gamma' \vdash s_1 : P_1, \text{ and } P \to^* P'.$$

Taking P as P'. Because of the following derivation, (**??**) holds.

$$\frac{\Theta; \Gamma' \vdash s_1 : P_1 \quad P_1 \leq P}{\Theta; \Gamma' \vdash s_1 : P} \quad \text{(T-Sub)}$$

And by the definition of $\xrightarrow{\tau}{}^*$, (**??**) holds.

Case: $\langle H, R, f(x), n\rangle \to \langle H', R', [x'/x]s_1, n\rangle$ where the body of function $f(x)$ is s_1. we can see $f(x)$ and s_1 as s and s' respectively.

From the assumption, we already have

$$\Gamma \vdash f(x) : P, \text{ and } OK_n(P).$$

By the inversion lemma and (**??**), we have

$$P_1 \leq P, \text{ and } \Gamma \vdash s_1 : P_1.$$

From the definition of subtyping and $P_1 \xrightarrow{0} P_1$, we get $\exists P''$ s.t.

$$P \xrightarrow{0} P'', \text{ and } P_1 \leq P''.$$

Taking the P'' to be P', then we get $P \xrightarrow{0} P'$.
And by using the subtyping rule with premises (**??**) and (**??**), we have

$$\frac{\Gamma \vdash s_1 : P_1 \qquad P_1 \leq P'}{\Gamma \vdash s_1 : P'}$$

Therefore we prove that $\Gamma \vdash s' : P'$ where s' is the command s_1.
Finally to prove $OK_{n'} P'$ that is, $\sharp_m(P') - \sharp_f(P') \leq n'$. And proceed by case analysis.

Case $P = \mathbf{skip}; P'$

According to rule E-Skip, we should prove $\sharp_m(P') - \sharp_f(P') \leq n'$ where n' is n.
Because we have

$$\begin{aligned} OK_n(P) \qquad &= OK_n(\mathbf{skip}; P') \\ &\Rightarrow \sharp_m(\mathbf{skip}; P') - \sharp_f(\mathbf{skip}; P') \leq n \\ &\Rightarrow \sharp_m(P') - \sharp_f(P') \leq n \end{aligned}$$

Then it is proved.

Case $P = \mathbf{malloc}; P'$

Here according to rule E-Malloc, we know the n' is $n - 1$.
Therefore we should prove $\sharp_m(P') - \sharp_f(P') \leq n - 1$

$$\begin{aligned} OK_n(P) \qquad &= OK_n(\mathbf{malloc}; P') \\ &\Rightarrow \sharp_m(\mathbf{malloc}; P') - \sharp_f(\mathbf{malloc}; P') \leq n \\ &\Rightarrow \sharp_m(P') + 1 - \sharp_f(P') \leq n \\ &\Rightarrow \sharp_m(P') - \sharp_f(P') \leq n - 1 \end{aligned}$$

Then it is proved.

Case $P = \mathbf{free}; P'$

According to rule E-Free, we should prove $\sharp_m(P') - \sharp_f(P') \leq n + 1$.

$$\begin{aligned} OK_n(P) \qquad &= OK_n(\mathbf{free}; P') \\ &\Rightarrow \sharp_m(\mathbf{free}; P') - \sharp_f(\mathbf{free}; P') \leq n \\ &\Rightarrow \sharp_m(P') - \sharp_f(P') - 1 \leq n \\ &\Rightarrow \sharp_m(P') - \sharp_f(P') \leq n + 1 \end{aligned}$$

Then it is proved.

Case $P = P_1; P_2$

To prove it by contradiction.

Suppose that $OK_{n'}(P_1'; P_2)$ does not hold. Then we have $P_1; P_2 \xrightarrow{\alpha} P_1'; P_2 \xrightarrow{\exists\sigma} Q$, *s.t.* $\sharp_m(\sigma) - \sharp_f(\sigma) > n'$

From the premise $OK_n(P) = OK_n(P_1; P_2)$, we get

$$\sharp_m(\alpha \cdot \sigma) - \sharp_f(\alpha \cdot \sigma) \leq n \tag{1}$$

From (1), we get

$$\sharp_m(\alpha) + \sharp_m(\sigma) - \sharp_f(\alpha) - \sharp_f(\sigma) \tag{2}$$

and with

$$n' = \begin{cases} n+1, & \alpha = \textbf{free} \\ n-1, & \alpha = \textbf{malloc} \\ n, & otherwise \end{cases}$$

Therefore, we get
$n' + \sharp_m(\alpha) - \sharp_f(\alpha) < \sharp_m(\alpha) + \sharp_m(\sigma) - \sharp_f(\alpha) - \sharp_f(\sigma) \leq n$
When $\alpha = \textbf{free}$, we get that $n + 1 - 1 < n$
When $\alpha = \textbf{malloc}$, we get that $n - 1 + 1 < n$
When $\alpha = other$, we get that $n < n$
All of the three cases are equal to n. Therefore we get the contradiction.

2. Syntax Directed Typing Rules

Typing rules showed in Figure 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 matches the shape of the s [**?**].

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

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

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

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

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

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

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

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

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

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

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

Figure 5. Syntax Directed Typing Rules