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

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

The problem of 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 regions (i.e., accesses to dangling pointers) and that does not dispose memory region when it becomes unnecessary (i.e., memory leak.) In order to detect bugs related to such primitives at the early stage of software development, many static verification methods have been proposed [].

This paper proposes a type-based approach to static verification of memory-leak freedom that works for nonterminating programs. Although memory leaks are relatively more serious in nonterminating programs (e.g., operating systems and Web servers) than terminating ones, the analyses proposed so far put less emphasis to nontermination; they rather verify *partial* memory-leak freedom: if a program terminates, then all the allocated memory cells are deallocated. We say a program is *totally* memory-leak free if it does not consume unbounded amount of memory during execution.

```
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. However, the function h', when it is invoked, consumes unbounded number of memory cells; hence h' is not totally memory-leak free. On the other hand, the function h consumes two cells at most[1]; hence it is totally memory-leak free.

As the first step to the verification of total memory-leak freedom, this paper proposes a *behavioral type system* [] for a programming language with manual memory-management primitives. Our type system captures an abstract behavior related to memory allocation and deallocation as a CCS-like process. 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. On the other hand, 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'.

Notice that our type system alone does not prevent correct 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, there is no accesses to deallocated cells. For such properties, we expect a program to be verified by other no-illegal-access verifiers [].

KS: USE LABEL to refer to other sections, lemmas, and so on. 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. Section 4 describes a type reconstruction algorithm. Section 5 describes current status and future work. **KS: To be revised.**

Notation We write $\vec{X}$ for a finite sequence of X; its length, if we do not explicitly mention, is clear from the context or does not matter.

The following BNF defines our language $\mathcal{L}$.

[1] We assume that every memory cell allocated by **malloc** is fixed size.

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

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* ranged over by w, x, y, and z. 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 **let** $x = e$ **in** s evaluates the expression e, binds x to the result, and executes s. The expression **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 **ifnull** (x) **then** s_1 **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 $d_1 \cup \cdots \cup d_n$. A program is a pair of function definitions D and a main statement s.

1.1 Operational Semantics

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

We give the operational semantics of the language $\mathcal{L}$ as a labeled transition on *configurations* $\langle H, R, s, n \rangle$. 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 available memory cells.

n in a configuration is later used to formalize memory leak caused by nonterminating program.

The operational semantics is given by a labeled translation relation $\langle H, R, s, n \rangle \xrightarrow{\rho}_D \langle H', R', s', n' \rangle$ where ρ, an *action*, is **malloc**, **free**, or τ. The action **malloc** expresses an allocation of a memory cell; **free** expresses a deallocation; τ expresses the other

actions. We often omit τ in $\xrightarrow{\tau}_D$. We use a metavariable σ for a finite sequence of actions $\rho_1 \dots \rho_n$. We write $\xrightarrow{\rho_1 \cdots \rho_n}_D$ 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 $\overset{\rho_1 \cdots \rho_n}{\Longrightarrow}_D$ for $\overset{\rho_1}{\Longrightarrow}_D \dots \overset{\rho_n}{\Longrightarrow}_D$.

Figure 1.1 defines the relation $\xrightarrow{\rho}_D$.

Definition 1. KS: Formally state this definition. *memory leaks: if a program consumes unbounded number of memory cells.*
memory-leak freedom: $\exists n \in \mathbb{N}$ *s.t.* $\langle \emptyset, \emptyset, s, n \rangle \nrightarrow^* Error$

2 Type System

2.1 Syntax of Types

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

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 $\mu\alpha.\mathbf{malloc}; \mathbf{malloc}; \mathbf{free}; \mathbf{free}; \alpha$. 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 $(\tau_1, \dots, \tau_n)P$, which means a function receives some pointers as arguments and its body is abstracted as a behavioral type P.

2.2 Semantics of Behavioral Types

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

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

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

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

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

$$P_1 + P_2 \longrightarrow P_1$$

$$P_1 + P_2 \longrightarrow P_2$$

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

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

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

$$\frac{R(x) \in dom(H)}{\langle H, R, *x \leftarrow y, n\rangle \longrightarrow_D \langle H\,\{R(x) \to R(y)\}, R, \mathbf{skip}, n\rangle} \quad \text{(Tr-Assignment)}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 2. Definition of the labeled transition $\langle H, R, s, n\rangle \xrightarrow{\rho}_D \langle H', R', s', n'\rangle$.

2.3 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 3. 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 2 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 2. $OK_n(P) \iff \forall P',\ P \xrightarrow{\alpha}{}^* P'$ *then* $\sharp_{malloc}(\alpha) - \sharp_{free}(\alpha) \leq n$.

2.4 Type Soundness

This subsection describes some theorems and lemmas for type safety.

Theorem 2.1. *If* $\vdash (D, s) : n$ *then* $\langle \emptyset, \emptyset, s, n \rangle \nrightarrow^* Overflow$

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

Lemma 2.2 (Preservation)**.** *If* $OK_n(P)$, $\Theta; \Gamma \vdash s : P$ *and* $\langle H, R, s, n \rangle \xrightarrow{\rho} \langle H', R', s', n' \rangle$, *then* $\exists P'$ *s.t.*
(1) $\Theta; \Gamma \vdash s' : P'$
(2) $P \stackrel{\rho}{\Longrightarrow} P'$
(3) $OK_{n'}(P')$

Lemma 2.3. *If* $\vdash (D, s)$ *then* $\langle \emptyset, \emptyset, s \rangle \nrightarrow^* Error$*(illegal accesses)*

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3. Typing Rules

3 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.

3.1 Constraints Generation

By syntax directed typing rules, the type inference algorithm has been designed as in Figure 4.

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.

3.2 Constraints Reduction

$PT\langle D, s\rangle$ receives a program as argument and produces a pair which consists of the subtype constraints on behavior types of the form $\alpha \geq A$, and constraints of the form $OK_n(P)$. Thus, we obtain the following constraints:

$$\{\alpha_1 \geq A_1, \ldots, \alpha_n \geq A_n, OK_n(P)\}$$

Here, we can assume that $\alpha_1, \ldots, \alpha_n$ are pairwise-distinct, since $\alpha \geq A_1$ and $\alpha \geq A_2$ can be replaced with $\alpha \geq A_1 + A_2$ by lemma 3.8 in paper [**?**]. we can also assume that $\{\alpha_1, \ldots, \alpha_n\}$ contains all the type variables in the constraints, since otherwise we can always add the tautology $\alpha \geq \alpha$. Each subtype constraints $\alpha \geq A$ can be replaced by $\alpha \geq \mu\alpha.A$, by lemma 3.8(4) [**?**] (substituting α for B in this lemma). Therefore the above constraints can be further reduced to $OK_n([\vec{A}/\vec{\alpha}]P)$. Here, $A'_1, \ldots, A'_n$ are the least solutions for the subtype constraints.

4 Related Work

Many methods for static memory-leak freedom verification have been proposed [**?**, **?**, **?**, **?**, **?**, **?**]. 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 safe memory deallocation for nonterminating programs.

Behavioral type systems.

5 Conclusion

We have described a type system to verify memory-leak freedom for (possibly) nonterminating programs with manual memory-management primitives where every memory cell is fixed size. Our type system abstracts the memory allocation/deallocation behavior of

a program with a CCS-like 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 investigating 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 ignores the size of the allocated block, counting only the number of **malloc** and **free**, which makes the abstraction unsound for actual programs.

Flow sensitivity is another issue we are pursuing. **KS: Complete this paragraph.**

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 $P \xrightarrow{\rho} P'$.

KS: To be revised.

Case $P = \mathbf{0}; P'$

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

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

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 $\sharp_m(P') - \sharp_f(P') \leq n - 1$

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

Then it is proved.

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

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

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

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

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

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 $\alpha = \mathbf{free}$, we get that $n + 1 - 1 < n$
When $\alpha = \mathbf{malloc}$, 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. □

Proof of Lemma **??**: By induction on the derivation of evaluation rules.

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

  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 $\mathbf{free} \leq P$ for some 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_1 such that $P \overset{\mathbf{free}}{\Longrightarrow} P_1$, $\Theta;\Gamma \vdash \mathbf{skip} : P_1$, and $OK_{n+1}(P_1)$. Take P'' as P_1. Then, $P \overset{\mathbf{free}}{\Longrightarrow} P''$ 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.

KS: Rewrite the other cases imitating this case.

- 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

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

  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

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

  By the inversion lemma on ①, we have

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

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

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

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

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

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

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

  From the inversion lemma on ①, we have ③ $0 \leq P$.

  We need to find P' and Γ' such that

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

  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

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

  From the inversion lemma and ①, we have

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

We need to find P' and Γ' such that:

$$\Theta; \Gamma' \vdash s_1 : P' \quad and \tag{3}$$

$$P \xrightarrow{\tau}{}^{*} P' \tag{4}$$

Taking P as P'. Therefore (4) 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: $\langle H, R, \textbf{let } x = \textbf{null in } \ s_1, n\rangle \rightarrow \langle H', R', [x'/x]s_1, n\rangle$

  From the assumption, we know that

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

  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

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

Therefore we prove that $\Gamma \vdash s' : P'$ where s' is the command s_1.

□

B 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

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

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

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

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

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

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

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

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

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

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

Figure 5. Syntax Directed Typing Rules

C Typing Inference Algorithm

$PT_\Theta(f) =$

$\quad$ **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 4. Type Inference Algorithm