An Extended Behavioral Type System for Memory-Leak Freedom

August 19, 2016

1 Abstract

In the previous work, we proposed a behavioral type system for a programming language with dynamic memory allocation and deallocation. The behavioral type system, which uses sequential processes as types where each action is related to an allocation and a deallocation, can estimate an upper bound of memory consumption of a program. However, the previous type system did not deal with path-sensitivity, which results in an imprecise abstraction even for a simple program. In order to address this problem, we propose an extension of the previous type system with dependent types. The dependent type carries more information for a program such that it can handle path-sensitivity and estimate an upper bound of memory cell consumption more precisely. We prove the soundness of the extended type system and propose a type inference algorithm of this type system. We also implemented the algorithm. Our experiment shows that the extended type system is very useful to deal with a practical path-sensitive program by checking whether the if-guard-part is a null pointer or not.

2 Introduction

Manual memory mangagement primitives (e.g. `malloc` and `free` in C language) are a very flexible way to manage computer memory cells. We can write a program which dynamically allocates a memory cell during running and deallocates a memory cell when it is no longer used. However, manual memory management primitives often cause hard-to-find problems, for example, double frees (`free` a deallocated memory cell), memory leaks (forget to deallocate memory cells) and illegal accesses to a dangling pointer. Therefore, many static verification methods have been proposed to guarantee safe memory deallocation. They prove *partial* memory-leak freedom: if a program terminates, all the memory cells are safe deallocated. As we know that nonterminating programs are very common in real-world programmings such as Web servers and operating systems. To guarantee *total* memory-leak freedom, if a program does not consume unbounded number of memory cells during execution, is a very crucial issue.

Example 2.1. 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

```
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.

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.

In order to prove *total* memory deallocation, we proposed a behavioral type system in previous study[]. It can abstract the behavior of a program by using sequential process whose actions represent manual memory management primitives, and our behavior type only consider the the number and order of manual memory management primitives and recursively calls. For example, the abstract behavioral type of function h in Figure 1 is $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{free};\mathbf{free};\alpha$, which represents function h allocates two memory cells, deallocates them, and then recursively call itself again; the behavioral type of function h' is abstracted as $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\alpha;\mathbf{free};\mathbf{free}$, which represents h' allocates two memory cells, call itself again, and then deallocates those two cells. That way we can easily estimate the upper bound of memory cells a program consumed.

Although our previours behavioral type system can abstract the behavior of a program and estimate the upper bound of memory consumption, verification on abstracted behavioral types are failed in some cases. For example, the extracted behavioral type of function foo in Figure 2 is $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{malloc}+\mathbf{0};\mathbf{free}+\mathbf{0};\mathbf{free};\mathbf{free};\alpha$, which expresses that function foo allocates two memory cells, a choice command between allocating one memory cell and skipping, a choice command between deallocating one cell and skipping, deallocates two cells, and then call itself again. Due to the choice behavioral type, the above type may be seen as $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{malloc};\mathbf{0};\mathbf{free};\mathbf{free};\alpha$, which expresses function foo consumes three memory cells but deallocates two memory cells, and then iterates this behavior again. This behavior means function foo consumes unbounded number of memory cells, although the original program is *total* memory-leak freedom.

```
1    foo() =
2            let y = malloc() in
3            let x = malloc() in
4            ifnull (∗y) then skip else let x1 = malloc() in ∗x ← x1;
5            ifnull (∗y) then skip else free(∗x);
6            free(x) ; free(y) ; foo()
```

Figure 2: a nonterminating program with conditionals

Example 2.2. Figure 2 describes that functioin foo is a total memory-leak freedom program, because it consumes at most three memory cells during execution. This function has two conditionals: if $*y$ is not a null pointer, it will allocates one cell at first conditional and deallocates that cell at second conditional, otherwise skips.

[1]We assume that every memory cell allocated by **malloc** is fixed size to simplify our type system introduced in Section 4. Extension with variable-length cells is one of our future work.

Our current idea is to extend previous behavioral type system with dependent types[]. The dependent types takes more precise information than traditional types, for example, the type $(*x)(\mathbf{malloc}, \mathbf{0})$ is a dependent type, because it dependents on the value $(*x)$. See the function foo again, the current behavioral type of it is $\mu\alpha.\mathbf{malloc}; \mathbf{malloc}; (*x)(\mathbf{malloc}, \mathbf{0}); (*x)(\mathbf{free}, \mathbf{0}); \mathbf{free}; \mathbf{free}; \alpha$. Therefore, the part$(*x)(\mathbf{malloc}, \mathbf{0}); (*x)(\mathbf{free}, \mathbf{0})$ can be seen as $\mathbf{malloc}; \mathbf{free}$ or $\mathbf{0}; \mathbf{0}$ if $(*x)$ does not change between these two choices, which we can definitely judge it is a total memory-leak free program.

The reminder of this paper is structured as follows. Section 3 describes an imperative language with allocation and deallocation primitives and its operational semantics. Section 4 introduces the extended behavioral type system with dependent types. Section 5 describes the type reconstruction procedure; Section **??** shows some experiments and give a discussion; Section 6 describes the related works; Section 7 concludes this paper.

3 Language $\mathcal{L}$

In this section we define an imperative language $\mathcal{L}$ with memory allocation and deallocation primitives, and for simplification we only use pointers as values.

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

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

Notation $\vec{x}$ is for a finite sequence $\{x_1, ..., x_n\}$, where we assume that each element is distinct; $[\vec{x'}/\vec{x}]s$ is for a term obtained by replacing each free occurrence of $\vec{x}$ in s with variables $\vec{x'}$.

The **Var** is a countably infinite set of *variables* and each variable is a pointer. The statement **skip** means "does nothing". The statement $s_1; s_2$ is a sequential execution of s_1 and s_2. The statement $*x \leftarrow y$ changes the content of cell which is pointed to by x with the value y. The statement $\mathbf{free}(x)$ deallocates a memory cell which is pointed to by pointer x. 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. The expression **null** evaluates to the null pointer. The expression $*y$ means dereferencing a memory cell pointed to by y. 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 $f(\vec{x})$ expresses a procedure f with arguments $\vec{x}$. The statement $\mathbf{const}(*x)s$ means $(*x)$ is a constant in statement s. The statement $\mathbf{endconst}(*x)$ means from this point $(*x)$ maybe not a constant.

The d represents a procedure definition which maps a procedure name f to its procedure body $(\vec{x})s$; The D represents a set of procedure definitions $\langle d_1 \cup \dots d_n \rangle$, and each definition is distinct; The pair $\langle D, s \rangle$ represents a program, where D is a set of definitions and s is a main statement; the E represents evaluation context.

3.1 Operational semantics

In this section we introduce operational semantics of language $\mathcal{L}$. We assume there is a countable infinite set $\mathcal{H}$ of *heap addresses* ranged over by l.

We use a configuration $\langle H, R, s, n, C \rangle$ to express a run-time state. Each elements in the configuration is as follows.

- 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;
- n is a natural number that represents the number of memory cells available for allocation, which can be formalized to check memory leaks even for nonterminating programs;
- C is a set related to current constant pointers, which contains $\mathbf{const}(*x)$, $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$.

The operational semantics of the language $\mathcal{L}$ is given by a labeled transition relation $\langle H, R, s, n, C \rangle \xrightarrow{\rho}_D \langle H', R', s', n', C' \rangle$. The label ρ is an action, which is as follows.

$$\rho \text{ (label)} \quad ::= \quad \mathbf{malloc}(x') \mid \mathbf{free} \mid \mathbf{null}(*x) \mid \neg\mathbf{null}(*x) \mid \tau$$

The action $\mathbf{malloc}(x')$ expresses an allocation of a new memory cell, and the new cell binds to a fresh variable x'; **free** expresses a deallocation of a memory cell; $\mathbf{null}(*x)$ means $*x$ is a null pointer, and $\neg\mathbf{null}(*x)$ not; τ expresses the other internal actions. For the operational semantics, we often omit τ in $\xrightarrow{\tau}_D$. The metavariable σ is used for a finite sequence of actions $\rho_1 \dots \rho_n$. The $\xrightarrow{\rho_1 \dots \rho_n}_D$ is short for $\xrightarrow{\rho_1}_D \xrightarrow{\rho_2}_D \dots \xrightarrow{\rho_n}_D$. The $\stackrel{\rho}{\Longrightarrow}_D$ means $\longrightarrow_D^* \xrightarrow{\rho}_D \longrightarrow_D^*$. We write $\stackrel{\rho_1 \dots \rho_n}{\Longrightarrow}_D$ for $\stackrel{\rho_1}{\Longrightarrow}_D \dots \stackrel{\rho_n}{\Longrightarrow}_D$.

Notation the $\mathbf{Dom}(f)$ is a mapping from function name f to its domain; for a map f, the $f\{x \mapsto v\}$ and $f \backslash x$ are defined as follows:

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

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

and $filter\_C(C, *x)$ is defined by a pseudcode as follows:

$$\begin{array}{lll} filter\_C(C, *x) & = & let\ C' = C - \mathbf{const}(*x)\ in \\ & & if\ \mathbf{const}(*x) \in C'\ then\ return\ C' \\ & & else\ return\ C' \backslash \{\mathbf{null}(*x), \neg\mathbf{null}(*x)\} \end{array}$$

Figure 3 depicts the relation $\xrightarrow{\rho}_D$. Several important rules are listed as follows.

- SEM-CONSTSEQ: **const**$(*x)$ and **endconst**$(*x)$ together guarantees a pointer pointed to by $*x$ cannot be changed in the statement s. The set C with the new added **const**$(*x)$ describes this status.

- SEM-IFNULLT and SEM-IFCONSTNULLT: these two rules represents if $(*x)$ is a null pointer, the statement s_1 will be executed. the difference of the two is if the **const**$(*x)$ is in set C then**null**$(*x)$ is added to C, which means $(*x)$ is a null pointer and cannot be updated from now on; otherwise $(*x)$ can be changed, like SEM-IFNULLT.

- SEM-FREE: deallocating one memory cell pointed by x is to remove linkage of pointer variable x to heap; this action will release one memory cell space, which increments the number of available memory cells n by one.

- SEM-MALLOC and SEM-OUTOFMEM: allocating one memory cell is described as updating the heap by adding a fresh heap variable l to anywhere v of the heap and adding the linkage of a fresh register variable x' to that l; This action is allowed only if the number of available memory cells is positive; otherwise **OutOfMemory**.

- SEM-ASSIGNEXN, SEM-FREEEXN and SEM-DEREFEXN: these rules express that accessing a null pointer or a dangling pointer will give raise to an exceptional state **MemEx**. However, in this paper we do not see the state **MemEx** is an erroneous state, hence a well-typed program may lead to these states. One thing we should notice the command **free**(x) , if x is a null pointer, raises state **MemEx** in the current semantics, although it is equivalent to **skip** in the C language.

- SEM-ASSIGNCONSTEXN: expressing that if a constant memory cell pointed to by x or its aliases are changed it will raise exceptional state **ConstEx**.

In order to deal with a path-sensitive program to guarantee *total* memory-leak freedom, we redefined the several definitions as follows. defined as follows:

Definition 1 (total memory-leak freedom)**.** *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.*

Definition 2 (Memory leak)**.** *A configuration* $\langle H, R, s, n, C\rangle$ goes overflow *if there is* σ *such that* $\langle H, R, s, n, C\rangle \stackrel{\sigma}{\Longrightarrow}$ **OutOfMemory**. *A program* $\langle D, s\rangle$ consumes at least n cells *if* $\langle \emptyset, \emptyset, s, n, \emptyset\rangle$ *goes overflow.*

$$\frac{C' = filter\_C(C, *x)}{\langle H, R, \mathbf{endconst}(*x), n, C\rangle \to_D \langle H, R, \mathbf{skip}, n, C'\rangle} \quad \text{(Sem-ConstSkip)}$$

$$\langle H, R, \mathbf{const}(*x)s, n, C\rangle \to_D \langle H, R, s; \mathbf{endconst}(*x), n, C \cup \{\mathbf{const}(*x)\}\rangle \quad \text{(Sem-ConstSeq)}$$

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

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

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

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

$$\frac{H(R(x)) = \mathbf{null}, \mathbf{const}(*x) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2,\ n, C\rangle \xrightarrow{\mathbf{null}(*x)}_D \langle H,\ R,\ s_1,\ n, C\rangle} \quad \text{(Sem-IfNullT)}$$

$$\frac{H(R(x)) \neq \mathbf{null}, \mathbf{const}(*x) \notin C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2,\ n, C\rangle \xrightarrow{\neg\mathbf{null}(*x)}_D \langle H,\ R,\ s_2,\ n, C\rangle} \quad \text{(Sem-IfNullF)}$$

$$\frac{H(R(x)) = \mathbf{null}, \mathbf{const}(*x) \in C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2,\ n, C\rangle \xrightarrow{\mathbf{null}(*x)}_D \langle H,\ R,\ s_1,\ n, C \cup \{\mathbf{null}(*x)\}\rangle} \quad \text{(Sem-IfConstNullT)}$$

$$\frac{H(R(x)) \neq \mathbf{null}, \mathbf{const}(*x) \in C}{\langle H,\ R,\ \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2,\ n, C\rangle \xrightarrow{\neg\mathbf{null}(*x)}_D \langle H,\ R,\ s_2,\ n, C \cup \{\neg\mathbf{null}(*x)\}\rangle} \quad \text{(Sem-IfConstNullF)}$$

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

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

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

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

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

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

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

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

$$\frac{\forall z.\mathbf{const}(*z) \in C \text{ and } R(x) = R(z)}{\langle H\{R(x) \mapsto v\}, R, *x \leftarrow y, n, C\rangle \longrightarrow_D \mathbf{ConstEx}} \quad \text{(Sem-AssignConstExn)}$$

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

Figure 3: Operational semantics of $\mathcal{L}$.

4 Type system

4.1 Types

The syntax of the types is as follows.

$$
\begin{array}{llll}
P & \text{(behavioral types)} & ::= & \mathbf{0} \mid P_1; P_2 \mid \mathbf{free} \mid \alpha \mid \mu\alpha.P \\
 & & & \mid \mathbf{let}\ x = y\ \mathbf{in}\ P \mid \mathbf{let}\ x = \mathbf{malloc}\ \mathbf{in}\ P \\
 & & & \mid \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ P \mid \mathbf{let}\ x = *y\ \mathbf{in}\ P \\
 & & & \mid (*x)(P_1, P_2) \mid \mathbf{const}(*x)P \mid \mathbf{endconst}(*x) \\
\Gamma & \text{(variable type environment)} & ::= & \{x_1, x_2, \ldots, x_n\} \\
\Psi & \text{(dependent function type)} & ::= & (\vec{x})P \\
\Theta & \text{(function type environment)} & ::= & \{f_1 : \Psi_1, \ldots, f_n : \Psi_n\} \\
k & \text{(constant values)} & ::= & \mathbf{null}(*x) \mid \neg\mathbf{null}(*x) \mid \mathbf{const}(*x) \\
F & \text{(constant value environment)} & ::= & \{k_1, ..., k_n\}
\end{array}
$$

Behavioral types ranged over by P express the abstaction of behaviors of a program. The type $\mathbf{0}$ represents the does nothing behavior; the type $P_1; P_2$ describes a sequential execution of behavioral type P_1 and P_2; The type **malloc** expresses an allocation of a memory cell; the type **free** represents a deallocation of a pointer; the type $\mu\alpha.P$ represents a recursive substitution of α in P ; the type $(*x)(P_1, P_2)$ represents that P_1 or P_2 is obtained dependent on $*x$, e.g., P_1 is obtained if $*x$ is not a null pointer, otherwise P_2; the type $P_1 + P_2$ represents the choice between P_1 and P_2; the α is a type variable; the type $\mathbf{const}(*x)P$ represents that $*x$ is a constant value in type P ; the type $\mathbf{endconst}(*x)$ represents $*x$ no longer be a constant from this point.

A type environments for variables ranged over by Γ is a set of variables without information about their types, because our focus is the behavior of a program.

Dependent function types ranged over by Ψ represents the behavior of a function. $\vec{x}$ is the formal arguments of the function, and the behavioral type P obtained dependent on $\vec{x}$.

Function types ranged over by Θ is a mapping from function names to dependent function types.

k represents constant values information, where $\mathbf{null}(*x)$ represents $(*x)$ is a null pointer; $\neg\mathbf{null}(*x)$ represents $(*x)$ is not a null pointer; $\mathbf{const}(*x)$ represents $(*x)$ should be a constant.

Constant value environment ranged over by F is a set of constant values information.

Notation $filter\_T(F, *x)$ is defined by a pseudcode as follows:

$$
\begin{array}{lcl}
filter\_T(F, *x) & = & let\ F' = F - \mathbf{const}(*x)\ in \\
 & & if\ \mathbf{const}(*x) \notin F'\ then\ return\ (F' \backslash \{\mathbf{null}(*x), \neg\mathbf{null}(*x)\}) \\
 & & else\ return\ F'
\end{array}
$$

Figure 4 depicts semantics of behavioral types with dependent types, and they are given by the labeled transition system. The relation $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$ means that P can make an action ρ, and P turns into P' after it makes action ρ; F and F' record constant value environment before and after making action ρ respectively.

4.2 Typing rules

The type judgment for statements is of the form $\Theta; \Gamma \vdash s : P$, which represents that under the function type environment Θ and the variable type environment Γ, the abstracted behavioral type of statement s is P.

$$\langle \mathbf{0}; P, F\rangle \to \langle P, F\rangle \qquad \text{(Tr-Skip)}$$

$$\langle \mathbf{free}, F\rangle \xrightarrow{\mathbf{free}} \langle \mathbf{0}, F\rangle \quad \text{(Tr-Free)} \qquad \langle \mu\alpha.P, F\rangle \to \langle [\mu\alpha.P/\alpha]P, F\rangle \text{ (Tr-Rec)}$$

$$\langle P_1 + P_2, F\rangle \to \langle P_1, F\rangle \text{ (Tr-ChoiceL)} \qquad \langle P_1 + P_2, F\rangle \to \langle P_2, F\rangle \text{ (Tr-ChoiceR)}$$

$$\frac{\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle}{\langle P_1; P_2, F\rangle \xrightarrow{\rho} \langle P_1'; P_2, F'\rangle} \qquad \text{(Tr-Seq)}$$

$$\langle \mathbf{let}\ x = \mathbf{malloc\ in}\ P, F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle [x'/x]P, F\rangle \qquad \text{(Tr-LetMalloc)}$$

$$\langle \mathbf{let}\ x = y\ \mathbf{in}\ P, F\rangle \to \langle [x'/x]P, F\rangle \qquad \text{(Tr-LetXY)}$$

$$\langle \mathbf{let}\ x = *y\ \mathbf{in}\ P, F\rangle \to \langle [x'/x]P, F\rangle \qquad \text{(Tr-LetX*Y)}$$

$$\langle \mathbf{let}\ x = \mathbf{null\ in}\ P, F\rangle \to \langle [x'/x]P, F\rangle \qquad \text{(Tr-LetX)}$$

$$\langle \mathbf{const}(*x)P, F\rangle \to \langle P; \mathbf{endconst}(*x), F \cup \{\mathbf{const}(*x)\}\rangle \qquad \text{(Tr-Const)}$$

$$\frac{F' = filter\_T(F, *x)}{\langle \mathbf{endconst}(*x), F\rangle \to \langle \mathbf{0}, F'\rangle} \qquad \text{(Tr-Endconst)}$$

$$\frac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F\rangle} \text{(Tr-NotConst1)} \qquad \frac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, F\rangle} \text{(Tr-NotConst2)}$$

$$\frac{\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_1, F\rangle} \text{(Tr-NullIn)} \qquad \frac{\neg\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \to \langle P_2, F\rangle} \text{(Tr-NNullIn)}$$

$$\frac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F \cup \mathbf{null}(*x)\rangle} \qquad \text{(Tr-NNullNotIn1)}$$

$$\frac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, F \cup \neg\mathbf{null}(*x)\rangle} \qquad \text{(Tr-NNullNotIn2)}$$

Figure 4: semantics of behavioral types with dependent types.

Before showing typing rules for statements in Figure 5, we need explain several important definitions. The first one is $OK_n(P, F)$, a predicate, where P represents the behavior of a program which consumes at most n memory cells under constant value environment F.

Definition 3 ($\sharp_\rho(\sigma)$)**.** *$\sharp_\rho(\sigma)$ is the number of ρ in the sequence σ.*

Definition 4. *$OK_n(P, F)$ holds if, (1) $\forall P'$ and σ. if $\langle P, F\rangle \xrightarrow{\sigma} \langle P', F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$*

Definition 5. *$OK(F)$ holds if F does not contain both $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$.*

Intuitively, $OK_n(P, F)$ represents at very running steps, the number of memory cells a program consumed will not exceed the number of memory cells the program requires.

Definition 6 (Subtyping)**.** *$F \vdash P_1 \leq P_2$ is the largest relation such that, for any P_1', F' and ρ, if $\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle$, then there exists P_2' such that $\langle P_2, F\rangle \stackrel{\rho}{\Longrightarrow} \langle P_2', F'\rangle$ and $F' \vdash P_1' \leq P_2'$. We write $P_1 \leq P_2$ if $F \vdash P_1 \leq P_2$ for any F.*

Figure 5 shows the typing rules. For example, the rule T-IfNull represents the behavior of **ifnull** $(*x)$ **then** s_1 **else** s_2 is abstracted as $(*x)(P_1, P_2)$ where P_1 and P_2 are the behavior of s_1 and s_2 respectively; this conditional statement means that executing s_1 if $(*x)$ is a null pointer, otherwise s_2. The typing rule T-Program represents a program requires at most n memory cells during running under the predication $OK_n(P, F)$, where P is behavioral type of statement s.

4.3 Type soundness

Theorem 4.1. *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.

Definition 7. *we write $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$, if $\Theta; \Gamma \vdash s : P$ and $OK_n(P, F)$ with $C \approx F$.*

Lemma 4.2 (Preservation)**.** *suppose that $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$. If $\langle H, R, s, n, C\rangle \xrightarrow{\rho} \langle H', R', s', n', C'\rangle$ then $\exists P', F'$ s.t. (1) $\Theta; \Gamma \vdash \langle H', R', s', n', C'\rangle : \langle P', F'\rangle$ and (2) $\langle P, F\rangle \stackrel{\rho}{\Longrightarrow} \langle P', F'\rangle$.*

Lemma 4.3 (Lack of immediate overflow)**.** *If $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$, then $\langle H, R, s, n, C\rangle \not\xrightarrow{\mathbf{malloc}}$* **OutOfMemory**.

5 Type Reconstruction

In this section we proposed a constraint-based type reconstruction procedure for the extended type system, which is similiar to the one in Kobayashi et al. [3]. This procedure can generate constraints for a given program to be definite and well-typed by constructing a derivation tree based on the rules in Figure 5. According to this procedure, the generated constraint is either a subtyping constraint $P \leq \alpha$ or $OK_\nu(\alpha, F)$ in which ν represents an unknown number of available memory cells. The predication $OK_n(P, F)$ only appears in the rule T-Program, therefore only one constraint of $OK_\nu(\alpha, F)$ is included in the constraints set.

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

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

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

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

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

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

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

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

$$\Theta;\Gamma, x \vdash \mathbf{endconst}(*x) : \mathbf{endconst}(*x) \qquad \text{(T-Endconst)}$$

$$\frac{\Theta;\Gamma, x \vdash s : P}{\Theta;\Gamma, x \vdash \mathbf{const}(*x)s : \mathbf{const}(*x)P} \qquad \text{(T-Const)}$$

$$\frac{\Theta;\Gamma, x \vdash s_1 : P_1 \qquad \Theta;\Gamma, x \vdash s_2 : P_2}{\Theta;\Gamma, x \vdash \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 : (*x)(P_1, P_2)} \qquad \text{(T-IfNull)}$$

$$\Theta, f : (\vec{y})P; \Gamma, \vec{x} \vdash f(\vec{x}) : P[\vec{x}/\vec{y}] \qquad \text{(T-Call)}$$

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

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

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

Figure 5: typing rules

To solve the constraint $OK_\nu(\alpha, F)$, we used several definitions proposed by Kobayashi et al. [3, Lemma 3.8] which describe that a subtyping constraint $\alpha \geq P$ can be resolved by setting $\alpha = \mu\alpha.P$. Therefore, the generated constraints set can be reduced to a single constraint $OK_\nu(P', F)$ for some behavioral type P'.

By definition, $OK_\nu(P, F)$ holds if (1) for any σ and P', if $\langle P, F\rangle \xrightarrow{\sigma} \langle P', F'\rangle$, then there exists a natural number n such that $\sharp_{\mathbf{malloc}}(\sigma) - \sharp_{\mathbf{free}}(\sigma) \leq n$ and (2) $OK(F)$. To prove this predication, we first fix an upper bound for ν and encoding F by hand. Then, $OK_\nu(P, F)$ can be checked, by using some mode checkers like CPAChecker [1], in finitely many states.

6 Related Works

Memory usage is a crucial issue for real-world program, so lots of static verification method for memory usage have been proposed [2, 5–8, 11]. However, these static methods only guaranteed partial memory-leak freedom and lack of illegal accessing of some pointers like null pointers or dangling pointers. In our previous work [], we proposed a behavioral type system, inspired by Kobayashi et al. [3] which guarantees safety properties of resources usage for concurrent programs, to estimate the upper bound number of memory cells a program consumes. By using our behavioral type system with other static methods mentioned above, we can guarantee memory-leak freedom even for nonterminating programs. But verification failed for path-sensitive programs, therefore we need to extend previous behavioral type system with dependent type system.

The dependent type [**?**, 4, 9, 10] takes more precise information than traditional type

7 Conclusion

In order to deal with path-sensitive problem, which results in an imprecise abstraction even such that verification failed even for a memory-leak free program, we proposed an extension of the previous type system with dependent types. We also described a type reconstruction algorithm for this extended type system, and we conducted several experiments to prove whether our idea can deal with path-sensitivity problem or not.

Our extended type system can deal with path-sensitivity but only for the guard-part of a conditional is a pointer. Therefore verification failed on a program where guard-part of a conditional is not a pointer. Besides, for simplification our extended type system excludes several features of real-world programs. For example, alias pointers and variable-sized memory blocks. Encoding the part where a pointer is a constant one by hand is unrealistic; Our types ignore the size of the allocated block and our system only counts the number of types **malloc** and **free**. Therefore, a program, which contains memory leaks by allocating huge memory blocks, may seem to be a well-typed one in our type system. We need to refine our current type system to solve these problms.

In order to solve the constraint of form $OK_\nu(P, F)$, we fix an upper bound for ν at first, which makes our reconstruction incomplete. For example, a given program consumes at most n numbers of memory cells, but if the ν we chose is great than n, the verification holds; otherwise, if the ν is less than n, the verification failed. The reason is that we have not yet known whether there exits an n s.t. $OK_n(P, F)$.

8 Proof of Lemmas

Lemma 8.1. *If* $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$ *and* $OK(F)$*, then* $OK(F')$

Proof. By induction on $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$.

- Case $P = \mathbf{0}; P'$ and $\langle \mathbf{0}; P', F\rangle \to \langle P', F\rangle$

  We need to prove $OK(F')$. From assumption, we have that $OK(F)$ holds, and in this case F' is the same as F. Therefore, $OK(F')$ holds.

- Case $P = \mathbf{let}\ x = \mathbf{malloc}\ \mathbf{in}\ P'$ and $\langle \mathbf{let}\ x = \mathbf{malloc}\ \mathbf{in}\ P', F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle [x'/x]P', F\rangle$

  Similiar to above.

- Case $P = \mathbf{let}\ x = y\ \mathbf{in}\ P'$ and $\langle \mathbf{let}\ x = y\ \mathbf{in}\ P', F\rangle \to \langle [x'/x]P', F\rangle$

  Similiar to above.

- Case $P = \mathbf{let}\ x = *y\ \mathbf{in}\ P'$ and $\langle \mathbf{let}\ x = *y\ \mathbf{in}\ P', F\rangle \to \langle [x'/x]P', F\rangle$

  Similiar to above.

- Case $P = \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ P'$ and $\langle \mathbf{let}\ x = \mathbf{null}\ \mathbf{in}\ P', F\rangle \to \langle [x'/x]P', F\rangle$

  Similiar to above.

- Case $P = \mathbf{free}$ and $\langle \mathbf{free}, F\rangle \xrightarrow{\mathbf{free}} \langle \mathbf{0}, F\rangle$

  Similiar to above.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1,P_2), F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F\rangle}$

  We need to prove $OK(F)$. From the assumption, $OK(F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\mathbf{const}(*x) \notin F}{\langle (*x)(P_1,P_2), F\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, F\rangle}$

  We need to prove $OK(F)$. From the assumption, $OK(F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1,P_2), F\rangle \to \langle P_1, F\rangle}$

  We need to prove $OK(F)$. From the assumption, $OK(F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\neg\mathbf{null}(*x) \in F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1,P_2), F\rangle \to \langle P_2, F\rangle}$

  We need to prove $OK(F)$. From the assumption, it holds.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1,P_2), F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F \cup \mathbf{null}(*x)\rangle}$

  We need to prove $OK(F \cup \mathbf{null}(*x))$. From the assumption, we have $OK(F)$ and $\neg\mathbf{null}(*x) \notin F$. Therefore $OK(F \cup \mathbf{null}(*x))$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\dfrac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin F \qquad \mathbf{const}(*x) \in F}{\langle (*x)(P_1,P_2), F\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, F \cup \neg\mathbf{null}(*x)\rangle}$

  We need to prove $OK(F \cup \neg\mathbf{null}(*x))$. From the assumption, we have $OK(F)$ and $\mathbf{null}(*x) \notin F$. Therefore $OK(F \cup \neg\mathbf{null}(*x))$ holds.

- Case $P = \mathbf{const}(*x)P'$ and $\langle \mathbf{const}(*x)P', F\rangle \to \langle P'; \mathbf{endconst}(*x), F \cup \{\mathbf{const}(*x)\}\rangle$
  We need to prove $OK(F \cup \{\mathbf{const}(*x)\})$. From the assumption, we have $OK(F)$ holds. Also, $F \cup \{\mathbf{const}(*x)\}$ does not contain both $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$. Therefore,$OK(F \cup \{\mathbf{const}(*x)\})$ holds.

- Case $P = \mathbf{endconst}(*x)$ and $\frac{F' = filter\_T(F,*x)}{\langle \mathbf{endconst}(*x), F\rangle \to \langle \mathbf{0}, F'\rangle}$
  we need to prove $OK(F')$. Form assumption, we have $OK(F)$ which means F does not contain both $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$. By the definition of $filter$ function, we have $F' = F\backslash\{\mathbf{null}(*x), \neg\mathbf{null}(*x)\}$ or $F - \mathbf{const}(*x)$, which means F' does not contain both $\mathbf{null}(*x)$ and $\neg\mathbf{null}(*x)$. Therefore, $OK(F')$ holds.

- Case $P = \mu\alpha.P'$ and $\langle \mu\alpha.P', F\rangle \to \langle [\mu\alpha.P']P', F\rangle$
  We need to prove $OK(F)$. From the assumption, we have that $OK(F)$ holds.

- Case $P = P_1; P_2$ and $\frac{\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle}{\langle P_1; P_2, F\rangle \xrightarrow{\rho} \langle P_1'; P_2, F'\rangle}$
  We need to prove $OK(F')$. By IH, we have $\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F'\rangle$ and $OK(F)$ holds, then $OK(F')$ holds.

□

Lemma 8.2. *If $OK_n(P, F)$ and $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$, then*

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

Proof. By induction on $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$.

- Case $P = \mathbf{0}; P'$ and $\langle \mathbf{0}; P', F\rangle \to \langle P', F\rangle$
  We need to prove $OK_n(P', F)$. Assume that $OK_n(P', F)$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.
  From the definition of that $OK(\mathbf{0}; P', F)$ holds, we have (1) if $\langle \mathbf{0}; P', F\rangle \to \langle P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$, which are in contradiction to the assumption. Therefore, $OK_n(P', F)$ holds.

- Case $P = \mathbf{let}\ x = \mathbf{malloc\ in}\ P'$ and $\langle \mathbf{let}\ x = \mathbf{malloc\ in}\ P', F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle [x'/x]P', F\rangle$
  we need to prove $OK_{n-1}([x'/x]P', F)$. Assume that $OK_{n-1}([x'/x]P', F)$ does not hold. Then we have (1) $\exists\sigma$ and Q s.t. $\langle [x'/x]P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m\sigma - \sharp_f\sigma > n$ or (2) $OK(F)$ does not hold.
  From the definition of $OK_n(P, F)$, we have (1) $\langle \mathbf{let}\ x = \mathbf{malloc\ in}\ P', F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle [x'/x]P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n - 1$ and (2) $OK(F)$ holds. Therefore, we get the contradiction, and the $OK_{n-1}([x'/x]P', F)$ holds.

- Case $P = \mathbf{let}\ x = y\ \mathbf{in}\ P'$ and $\langle \mathbf{let}\ x = y\ \mathbf{in}\ P', F\rangle \to \langle [x'/x]P', F\rangle$
  Similar to the above.

- Case $P = \textbf{let } x = *y \textbf{ in } P'$ and $\langle \textbf{let } x = *y \textbf{ in } P', F\rangle \rightarrow \langle [x'/x]P', F\rangle$

  Similar to the above.

- Case $P = \textbf{let } x = \textbf{null in } P'$ and $\langle \textbf{let } x = \textbf{null in } P', F\rangle \rightarrow \langle [x'/x]P', F\rangle$

  Similar to the above.

- Case $P = \textbf{free}$ and $\langle \textbf{free}, F\rangle \xrightarrow{\textbf{free}} \langle \textbf{0}, F\rangle$

  We need to prove $OK_{n+1}(\textbf{0}, F)$, which means we need to prove (1) $\forall\sigma$ and Q if $\langle \textbf{0}, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds. There is no Q and σ s.t. $\langle \textbf{0}, F\rangle \xrightarrow{\sigma} \langle Q, F\rangle$, so (1) holds. $OK(F)$ holds from Lemma 8.1. Therefore, $OK(\textbf{0}, F)$ holds.

- Case $P = \textbf{endconst}(*x)$ and $\frac{F' = filter\_T(F, *x)}{\langle \textbf{endconst}(*x), F\rangle \rightarrow \langle \textbf{0}, F'\rangle}$

  We need to prove $OK_n(\textbf{0}, F')$, which means we need to prove (1) $\forall\sigma$ and Q if $\langle \textbf{0}, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F')$ holds. There is no Q and σ s.t. $\langle \textbf{0}, F\rangle \xrightarrow{\sigma} \langle Q, F\rangle$, so (1) holds. From the assumption $OK_n(P, F)$, we have $OK(F)$, which means F does not contain both $\textbf{null}(*x)$ and $\neg\textbf{null}(*x)$. By the definition of function $filter\_T$, we have $F' = F \backslash \{\textbf{null}(*x), \neg\textbf{null}(*x)\}$ or $F - \textbf{const}(*x)$. Therefore $OK(F')$ holds. So $OK_n(\textbf{0}, F')$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\textbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\textbf{null}(*x)} \langle P_1, F\rangle}$

  We need to prove $OK_n(P_1, F)$. Assume that $OK_n(P_1, F)$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle P_1, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.

  From the definition of that $OK_n((*x)(P_1, P_2), F)$ holds, we have (1) if $\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\textbf{null}(*x)} \langle P_1, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds, which are in contradiction to the assumption. Therefore, $OK_n(P_1, F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\textbf{const}(*x) \notin F}{\langle (*x)(P_1, P_2), F\rangle \rightarrow \langle P_2, F\rangle}$

  We need to prove $OK_n(P_2, F)$. Assume that $OK_n(P_2, F)$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle P_2, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.

  From the definition of that $OK_n((*x)(P_1, P_2), F)$ holds, we have (1) if $\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\neg\textbf{null}(*x)} \langle P_2, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds, which are in contradiction to the assumption. Therefore, $OK_n(P_2, F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\textbf{null}(*x) \in F \qquad \textbf{const}(*x) \in F}{\langle (*x)(P_1, P_2), F\rangle \rightarrow \langle P_1, F\rangle}$

  We need to prove $OK_n(P_1, F)$. Assume that $OK_n(P_1, F)$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle P_1, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.

  From the definition of that $OK_n((*x)(P_1, P_2), F)$ holds, we have (1) if $\langle (*x)(P_1, P_2), F\rangle \rightarrow \langle P_1, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds, which are in contradiction to the assumption. Therefore, $OK_n(P_1, F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\neg\mathbf{null}(*x)\in F \quad \mathbf{const}(*x)\in F}{\langle (*x)(P_1,P_2),F\rangle \to \langle P_2,F\rangle}$

  We need to prove $OK_n(P_2, F)$. Assume that $OK_n(P_2, F)$ does not hold. Then we have (1) $\exists\sigma$ and Q s.t. $\langle P_2, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.

  From the definition of that $OK_n((*x)(P_1, P_2), F)$ holds, we have (1) if $\langle (*x)(P_1, P_2), F\rangle \to \langle P_2, F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds, which are in contradiction to the assumption. Therefore, $OK_n(P_2, F)$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\mathbf{null}(*x),\neg\mathbf{null}(*x)\notin F \quad \mathbf{const}(*x)\in F}{\langle (*x)(P_1,P_2),F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F\cup\{\mathbf{null}(*x)\}\rangle}$

  We need to prove $OK_n(P_1, F \cup \{\mathbf{null}(*x)\})$. Assume that $OK_n(P_1, F \cup \{\mathbf{null}(*x)\})$ does not hold. Then we have (1) $\exists\sigma$ and Q s.t. $\langle P_1, F \cup \{\mathbf{null}(*x)\}\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F \cup \{\mathbf{null}(*x)\})$ does not hold.

  From the definition of that $OK_n((*x)(P_1, P_2), F)$ holds, we have (1) if $\langle (*x)(P_1, P_2), F\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, F \cup \{\mathbf{null}(*x)\}\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds. By $OK(F)$ and $\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin F$, we have $OK(F \cup \{\mathbf{null}(*x)\})$ holds. Therefore, we get the contradiction and $OK_n(P_1, F \cup \{\mathbf{null}(*x)\})$ holds.

- Case $P = (*x)(P_1, P_2)$ and $\frac{\mathbf{null}(*x),\neg\mathbf{null}(*x)\notin F \quad \mathbf{const}(*x)\in F}{\langle (*x)(P_1,P_2),F\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, F\cup\{\neg\mathbf{null}(*x)\}\rangle}$

  Similar to the above.

- Case $P = \mathbf{const}(*x)P'$ and $\langle \mathbf{const}(*x)P', F\rangle \to \langle P'; \mathbf{endconst}(*x), F \cup \mathbf{const}(*x)\rangle$

  We need to prove $OK_n(P'; \mathbf{endconst}(*x), F \cup \mathbf{const}(*x))$. Assume that $OK_n(P'; \mathbf{endconst}(*x), F \cup \mathbf{const}(*x))$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle P'; \mathbf{endconst}(*x), F \cup \mathbf{const}(*x)\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F \cup \mathbf{const}(*x))$ does not hold.

  From the definition of that $OK_n(\mathbf{const}(*x)P', F)$ holds, we have (1) if $\langle \mathbf{const}(*x)P', F\rangle \to \langle P; \mathbf{endconst}(*x), F \cup \mathbf{const}(*x)\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$ and (2) $OK(F)$ holds, which are in contradiction to the assumption. Therefore, $OK_n(P_1, F)$ holds.

- Case $P = \mu\alpha.P'$ and $\langle \mu\alpha.P', F\rangle \to \langle [\mu\alpha.P'/\alpha]P', F\rangle$

  We need to prove $OK_n([\mu\alpha.P'/\alpha]P', F)$. Assume that $OK_n([\mu\alpha.P'/\alpha]P', F)$ does not hold. Then, we have (1) $\exists\sigma$ and Q s.t. $\langle [\mu\alpha.P'/\alpha]P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n$ or (2) $OK(F)$ does not hold.

  From the definition of that $OK_n(\mu\alpha.P', F)$ holds, we have (1) if $\langle \mu\alpha.P', F\rangle \to \langle [\mu\alpha.P'/\alpha]P', F\rangle \xrightarrow{\sigma} \langle Q, F'\rangle$, then $\sharp_m(\sigma) - \sharp_f(\sigma) \leq n$, which is a contradiction; and (2) $OK(F)$ holds. From the Lemma 8.1, $OK(F \cup \neg\mathbf{null}(*x))$ holds. Therefore, $OK([\mu\alpha.P'/\alpha]P', F)$ holds.

- Case $P = P_1; P_2$ and $\frac{\langle P_1,F\rangle \overset{\rho}{\Longrightarrow} \langle P_1',F'\rangle}{\langle P_1;P_2,F\rangle \overset{\rho}{\Longrightarrow} \langle P_1';P_2,F'\rangle}$

  We need to prove $OK_{n'}(P_1'; P_2, F)$, 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 $OK_{n'}(P_1'; P_2, F')$ does not hold. Then, we have (1) $\exists \sigma$, Q and F'' s.t. $\langle P_1'; P_2, F\rangle \xrightarrow{\sigma} \langle Q, F''\rangle$ and $\sharp_m(\sigma) - \sharp_f(\sigma) > n'$ or (2) $OK(F')$ does not hold.

From the definition of that $OK_n(P_1; P_2, F)$ holds, we have (1) if $\langle P_1; P_2, F\rangle \stackrel{\rho}{\Longrightarrow} \langle P_1'; P_2, F'\rangle \xrightarrow{\sigma} \langle Q, F''\rangle$, then $\sharp_m(\rho\sigma) - \sharp_f(\rho\sigma) \leq n$ and (2) $OK(F)$ holds.

From (1), we get $n' + \sharp_m(\rho) - \sharp_f(\rho) < \sharp_m(\rho) + \sharp_m(\sigma) - \sharp_f(\rho) - \sharp_f(\sigma) \leq n$. For any ρ, the $n' + \sharp_m(\rho) - \sharp_f(\rho) = n$, therefore we get a contradiction. By IH, we have $OK(F')$ holds, which is a contradiction. Therefore, $OK_{n'}(P_1; P_2, F')$ holds.

□

Proof of Lemma 4.2: By induction on the derivation of $\langle H, R, s, n, C\rangle \xrightarrow{\rho} \langle H', R', s', n', C'\rangle$.

- Case: $\langle H, R, \mathbf{const}(*x)s, n, C\rangle \to \langle H, R, s; \mathbf{endconst}(*x), n, C \cup \{\mathbf{const}(*x)\}\rangle$

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{const}(*x)s, n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma \vdash \mathbf{const}(*x)s : P$ and $OK_n(P, F)$. From the inversion of typing rules, we get $\Theta; \Gamma \vdash s : P''$ and $\mathbf{const}(*x)P'' \leq P$ for some P''. By subtyping, we have $P''; \mathbf{endconst}(*x) \leq Q$ and $\langle P, F\rangle \Longrightarrow \langle Q, F \cup \{\mathbf{const}(*x)\}\rangle$ for some Q.

  we need to find P' and F' s.t. $\Theta; \Gamma \vdash s; \mathbf{endconst}(*x) : P'$, $OK_n(P', F')$ and $\langle P, F'\rangle \Longrightarrow \langle P', F'\rangle$. Taking Q as P' and $F \cup \{\mathbf{const}(*x)\}$ as F'. Therefore $\langle P, F\rangle \to \langle P', F'\rangle$ holds, and $OK_n(P', F')$ holds from Lemma 8.2. From $\Theta; \Gamma \vdash s; \mathbf{endconst}(*x) : P''; \mathbf{endconst}(*x)$, $P''; \mathbf{endconst}(*x) \leq Q$ and T-Sub, $\Theta; \Gamma \vdash s; \mathbf{endconst}(*x) : P'$ holds.

- Case: $\langle H, R, \mathbf{endconst}(*x), n, C\rangle \to \langle H, R, \mathbf{skip}, n, C'\rangle$ where $C' = filter\_C(C, *x)$

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{endconst}(*x), n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma \vdash \mathbf{endconst}(*x) : P$ and $OK_n(P, F)$. From the inversion of typing rules, we get $\Theta; \Gamma \vdash \mathbf{endconst}(*x) : \mathbf{endconst}(*x)$ and $\mathbf{endconst}(*x) \leq P$. By subtyping and function $filter\_T(F, *x)$, we get $0 \leq Q$ and $\langle P, F\rangle \to \langle Q, F''\rangle$ for some Q.

  we need to find P' and F' s.t. $\Theta; \Gamma \vdash \mathbf{skip} : P'$, $OK_n(P', F')$ and $\langle P, F\rangle \Longrightarrow P', F'\rangle$. Taking Q as P' and F'' as F' therefore $F' \approx C'$ from functions $filter\_T(F, *x)$ and $filter\_C(C, *x)$; $\langle P, F\rangle \to \langle P', F'\rangle$ and $OK_n(P', F')$ hold. From T-Skip, T-Sub and $0 \leq Q$, then $\Theta; \Gamma \vdash \mathbf{skip} : P'$ holds.

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

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{free}(x)\ , n, C\rangle : \langle P, F\rangle$, we have $OK_n(P, F)$ 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 $\mathbf{free} \leq P$. By the subtyping, we have $\langle P, F\rangle \stackrel{\mathbf{free}}{\Longrightarrow} \langle Q, F\rangle$ and $\mathbf{0} \leq Q$for some Q.

  We need to find P' and F' such that $\langle P, F\rangle \stackrel{\mathbf{free}}{\Longrightarrow} \langle P', F'\rangle$, $\Theta; \Gamma \vdash \mathbf{skip} : P'$, and $OK_{n+1}(P', F')$. Take Q as P' and F as F'. Then, $\langle P, F\rangle \stackrel{\mathbf{free}}{\Longrightarrow} \langle P', F'\rangle$ holds, and $OK_{n+1}(P', F')$ holds from Lemma 8.2. We also have $\Theta; \Gamma \vdash \mathbf{skip} : P'$ from T-Skip, $\mathbf{0} \leq Q$ and T-Sub.

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

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s, n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma \vdash \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s : P$ and $OK_n(P, F)$. By the inversion of typing rules, we have

$\Theta; \Gamma, x \vdash s : P''$ and **let** $x =$ **malloc in** $P'' \leq P$ for some P''. By subtyping, we get $\langle P, F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle Q, F\rangle$ and $[x'/x]P'' \leq Q$ for some Q.

We need to find P' and F' such that $\Theta; \Gamma, x' \vdash [x'/x]s : P'$ and $\langle P, F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle P', F'\rangle$ and $OK_{n-1}(P', F')$. Take Q as P' and F as F'. Then $\langle P, F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle P', F'\rangle$ holds, and $OK_{n-1}(P', F')$ holds by Lemma 8.2. From $\Theta; \Gamma, x \vdash s : P''$ and **let** $x =$ **malloc in** $P'' \leq P$, we have $\Theta; \Gamma, x'' \vdash [x''/x]s : [x''/x]P''$ and **let** $x'' =$ **malloc in** $[x''/x]P'' \leq P$, and then by the definition of subtyping we have $[x''/x]P'' \leq Q'$ for some Q'. Therefore, we get $\Theta; \Gamma, x'' \vdash [x''/x]s : Q'$. Take x'' as x' and Q' as P', then $\Theta; \Gamma, x' \vdash [x'/x]s : P'$ holds.

- Case: $\langle H, R, \mathbf{skip}; s, n, C\rangle \to \langle H, R, s, n, C\rangle$

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{skip}; s, n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma \vdash \mathbf{skip}; s : P$ and $OK_n(P, F)$. From the inversion of the typing rules, we get $\Theta; \Gamma \vdash s : P''$ and $0; P'' \leq P$. From the definition of subtyping, we have $\langle P, F\rangle \Longrightarrow \langle Q, F\rangle$ and $P'' \leq Q$ for some Q.

  We need to find P' and F' such that $\Theta; \Gamma \vdash s : P'$ and $\langle P, F\rangle \to \langle P', F'\rangle$ and $OK_n(P', F')$. Take Q as P' and F as F'. Then $\langle P, F\rangle \Longrightarrow \langle P', F'\rangle$ and $OK_n(P', F')$ hold. We also have $\Theta; \Gamma \vdash s : P'$ from T-Sub, $\Gamma \vdash s : P''$ and $P'' \leq Q$.

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

  From the assumption $\Theta; \Gamma \vdash \langle H, R, *x \leftarrow y, n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma \vdash *x \leftarrow y : P$ and $OK_n(P, F)$. From the inversion of typing rules, we have $0 \leq P$.

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

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

  From the assumption $\Theta; \Gamma \vdash \langle H, R, \mathbf{let}\ x = y\ \mathbf{in}\ s, n, C\rangle : \langle P, F\rangle$, we have $\Theta; \Gamma, y \vdash \mathbf{let}\ x = y\ \mathbf{in}\ s : P$ and $OK_n(P, F)$. From the inversion of typing rules, we have $\Theta; \Gamma, x, y \vdash s : P''$ and **let** $x = y$ **in** $P'' \leq P$ for some P''. By subtying, we have $\langle P, F\rangle \to \langle Q, F\rangle$ and $[x'/x]P'' \leq Q$ for some Q.

  We need to find P' and F' such that $\Theta; \Gamma, x', y \vdash [x'/x]s : P'$, $\langle P, F\rangle \to \langle P', F'\rangle$ and $OK_n(P', F')$. Take Q as P' and F as F'. Then $\langle P, F\rangle \Longrightarrow \langle P', F'\rangle$ and $OK_n(P', F')$ hold. From $\Theta; \Gamma, x, y \vdash s : P''$ and **let** $x = y$ **in** $P'' \leq P$, we have $\Theta; \Gamma, x'', y \vdash [x''/x]s : [x''/x]P''$ and **let** $x'' = y$ **in** $[x''/x]P'' \leq P$, and then by subtying we have $[x''/x]P'' \leq Q'$ for some Q'. Therefore, we have $\Theta; \Gamma, x'', y \vdash [x''/x]s : Q'$. Take x'' as x' and Q' as P', then $\Theta; \Gamma, x', y \vdash [x'/x]s : P'$ holds.

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

  Similar to the above.

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

  Similar to the above.

- Case: $\langle H, R, \mathbf{ifnull}\ (*x)\ \mathbf{then}\ s_1\ \mathbf{else}\ \ s_2, n, C\rangle \xrightarrow{\mathbf{null}(*x)} \langle H, R, s_1, n, C\rangle$ if $H(R(x)) = \mathbf{null}$ and $\mathbf{const}(*x) \notin C$

From assumption $\Theta;\Gamma \vdash \langle H, R, \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n, C\rangle : \langle P, F\rangle$, we have $\Theta;\Gamma \vdash \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2 : P$ and $OK_n(P, F)$. From the inversion of typing rules, we have $\Theta;\Gamma \vdash s_1 : P_1$, $\Theta;\Gamma \vdash s_2 : P_2$ and $(*x)(P_1, P_2) \leq P$. By subtyping and $\textbf{const}(*x) \notin C$, which means $\textbf{const}(*x) \notin F$, we get $\langle P, F\rangle \xRightarrow{\textbf{null}(*x)} \langle Q, F\rangle$ and $P_1 \leq Q$ for some Q.

We need to find P' and F' such that $\Theta;\Gamma \vdash s_1 : P'$, $\langle P, F\rangle \xRightarrow{\textbf{null}(*x)} \langle P', F'\rangle$ and $OK_n(P', F')$. Take Q as P' and F as F'. Then $\langle P, F\rangle \xRightarrow{\textbf{null}(*x)} \langle P', F'\rangle$ and $OK_n(P', F')$ hold. We also have $\Theta;\Gamma \vdash s_1 : P'$ from T-Sub, $\Theta;\Gamma \vdash s_1 : P_1$ and $P_1 \leq Q$.

- Case: $\langle H, R, \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n, C\rangle \xrightarrow{\neg\textbf{null}(*x)} \langle H, R, s_1, n, C\rangle$ if $H(R(x)) \neq \textbf{null}$ and $\textbf{const}(*x) \notin C$

  Similar to the above.

- Case: $\langle H, R, \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n, C\rangle \xrightarrow{\textbf{null}(*x)} \langle H, R, s_1, n, C'\rangle$ if $H(R(x)) = \textbf{null}$, $\textbf{const}(*x) \in C$ and $C' = C \cup \{\textbf{null}(*x)\}$

  From assumption $\Theta;\Gamma \vdash \langle H, R, \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n, C\rangle : \langle P, F\rangle$, we have $\Theta;\Gamma \vdash \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2 : P$ and $OK_n(P, F)$. From the inversion of typing rules, we have $\Theta;\Gamma \vdash s_1 : P_1$, $\Theta;\Gamma \vdash s_2 : P_2$ and $(*x)(P_1, P_2) \leq P$. By subtyping and $\textbf{const}(*x) \in C$, we get $\langle P, F\rangle \xRightarrow{\textbf{null}(*x)} \langle Q, F \cup \{\textbf{null}(*x)\}\rangle$ and $P_1 \leq Q$ for some Q.

  We need to find P' and F' such that $\Theta;\Gamma \vdash s_1 : P'$, $\langle P, F\rangle \xRightarrow{\textbf{null}(*x)} \langle P', F'\rangle$ and $OK_n(P', F')$. Take Q as P' and $F \cup \{\textbf{null}(*x)\}$ as F'. Then $C' \approx F'$, $\langle P, F\rangle \xrightarrow{\textbf{null}(*x)} \langle P', F'\rangle$ and $OK_n(P', F')$ hold. We also have $\Theta;\Gamma \vdash s_1 : P'$ from T-Sub, $\Theta;\Gamma \vdash s_1 : P_1$ and $P_1 \leq Q$.

- Case: $\langle H, R, \textbf{ifnull}\ (*x)\ \textbf{then}\ s_1\ \textbf{else}\ \ s_2, n, C\rangle \xrightarrow{\neg\textbf{null}(*x)} \langle H, R, s_2, n, C'\rangle$ if $H(R(x)) \neq \textbf{null}$, $\textbf{const}(*x) \in C$ and $C' = C \cup \{\neg\textbf{null}(*x)\}$

  Similar to the above proof.

- Case: $\langle H, R, s_1; s_2, n, C\rangle \to \langle H', R', s_1'; s_2, n', C'\rangle$

  From the assumption $\Theta;\Gamma \vdash \langle H, R, s_1; s_2, n, C\rangle : \langle P, F\rangle$, we have $\Theta;\Gamma \vdash s_1; s_2 : P$ and $OK_n(P, F)$ with $C \approx F$. By inversion of typing rules, we have $\Theta;\Gamma \vdash s_1 : P_1$, $\Theta;\Gamma \vdash s_2 : P_2$ and $P_1; P_2 \leq P$ for some P_1 and P_2.

  By IH on $\langle H, R, s_1, n, C\rangle$ with derivation $\langle H, R, s_1, n, C\rangle \xrightarrow{\rho} \langle H', R', s_1', n', C'\rangle$, we have $\exists P_1', F_1'$ s.t. $\Theta;\Gamma \vdash \langle H', R', s_1', n', C'\rangle : \langle P_1', F_1'\rangle$ and $\langle P_1, F\rangle \xrightarrow{\rho} \langle P_1', F_1'\rangle$.

  By subtyping we have $\langle P, F\rangle \xrightarrow{\rho} \langle Q, F_1'\rangle$ and $P_1'; P_2 \leq Q$ for some Q.

  We need to find P' and F' s.t. $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$, $OK_n(P', F')$ and $\Theta;\Gamma \vdash s_1'; s_2 : P'\rangle$. Take Q as P' and F_1' as F', $\langle P, F\rangle \xrightarrow{\rho} \langle P', F'\rangle$ and $OK_n(P', F')$ hold. By T-Sub, $\Theta;\Gamma \vdash s_1'; s_2 : P_1'; P_2$ and $P_1'; P_2 \leq Q$, we have $\Theta;\Gamma \vdash s_1'; s_2 : P'$ holds.

□

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

Lemma 8.3. *If* $\Theta;\Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$ *and* $\langle H, R, s, n, C\rangle \xRightarrow{\rho}$ *and* $\rho \in \{\textbf{malloc}(x'), \textbf{free}, \textbf{null}(*x), \neg\textbf{null}(*x)\}$, *then there exists* P' *and* F' *such that* $\langle P, F\rangle \xRightarrow{\rho} \langle P', F'\rangle$.

Proof. Induction on the derivation of $\Theta; \Gamma \vdash \langle H, R, s, n, C\rangle : \langle P, F\rangle$. □

Proof of Lemma 4.3:

By contradiction. Assume $\langle H, R, s, n, C\rangle \xrightarrow{\rho}$ **OutOfMemory**. Then, n is 0 and $\rho = \mathbf{malloc}(x')$ from SEM-OUTOFMEM. From the assumption we have $\Theta; \Gamma \vdash s : P$ and $OK_0(P, F)$. From Lemma 8.3, there exists P' and F' such that $\langle P, F\rangle \xrightarrow{\mathbf{malloc}(x')} \langle P', F'\rangle$. However, this contradicts $OK_0(P, F)$.

□

Proof of Theorem 4.1:

We have $\Theta; \emptyset \vdash s : P, \vdash D : \Theta$ and $OK_n(P, F)$.

Suppose that there exists σ such that $\langle \emptyset, \emptyset, s, n, C\rangle \xrightarrow{\sigma} \langle H', R', s', n', C'\rangle \xrightarrow{\rho}$ **OutOfMemory**. Then, $n' = 0$ and $\rho = \mathbf{malloc}(x')$. From Lemma 4.2, there exists P' and F' such that $\Theta; \Gamma' \vdash s' : P'$, $\langle P, F\rangle \stackrel{\sigma}{\Longrightarrow} \langle P', F'\rangle$, and $OK_0(P', F')$; hence $\langle H', R', s', 0\rangle \xrightarrow{\mathbf{malloc}(x')}$. However, this contradicts Lemma 4.3.

□

9 Syntax Directed Typing Rules

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

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

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

$$\frac{\Theta; \Gamma; C_1 \vdash x \qquad C = C_1}{\Theta; \Gamma; C \vdash \mathbf{endconst}(*x) : \mathbf{endconst}(*x)} \quad \text{(ST-EndConst)}$$

$$\frac{\Theta; \Gamma; C_1 \vdash x \qquad \Theta; \Gamma; C_2 \vdash s : P_1 \qquad C = C_1 \cup C_2 \cup \{P_1 \leq P\}}{\Theta; \Gamma; C \vdash \mathbf{const}(*x)s : \mathbf{const}(*x)P} \text{(ST-Const)}$$

$$\frac{\Theta; \Gamma; C_1 \vdash x \qquad C = C_1}{\Gamma; C \vdash \mathbf{free}(x) : \mathbf{free}} \quad \text{(ST-Free)}$$

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

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

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

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

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

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

10 Type Inference

References

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

[2] 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.

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

$PT_\Theta(f) =$

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

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

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

$\quad$ **in** $(C_1, \mathbf{free})$

$PT_\Theta(\mathbf{endconst}(*x)) =$

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

$\quad$ **in** $(C_1, \mathbf{endconst}(*x))$

$PT_\Theta(\mathbf{const}(*x)s) =$

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

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

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

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

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

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

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

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

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

$\quad$ **in** $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \mathbf{let}\ x = y\ \mathbf{in}\ \beta)$

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

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

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

$\quad$ **in** $(C_1 \cup C_2 \cup \{P_1 \leq \beta\}, \mathbf{let}\ x = *y\ \mathbf{in}\ \beta)$

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

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

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

$\quad$ **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

[4] U. Norell. Dependently typed programming in agda. In A. Kennedy and A. Ahmed, editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM, 2009.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*, pages 249–257. ACM, 1998.

[10] H. Xi and F. Pfenning. Dependent types in practical programming. In A. W. Appel and A. Aiken, editors, *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, pages 214–227. ACM, 1999.

[11] 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.