An Extended Behavioral Type System for Memory-Leak Freedom

Qi Tan Kohei Suenaga Atsushi Igarashi

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 *path-sensitive behavioral types*. Our extended type system can capture path-sensitive behavior of a program, which enables more precise analysis of the behavior of a program.

1 Introduction

Several programming languages, such as the C language, allow very flexible memory management by providing *manual memory management primitives* (e.g. `malloc` and `free` in the C language). One can write a program that dynamically allocates and deallocates a memory cell during execution.

However, manual memory management primitives often cause hard-to-find bugs. One can write a program that deallocates a memory cell twice, a program that forgets to deallocate a memory cell, and a program that accesses to a memory cell that is already deallocated. These bugs in manual memory management often leads to a serious security problem.

Using a static analysis is one of the approaches to detecting such bugs at the early stage of development. Much effort has been paid to powerful and efficient static analysis of a program with manual memory management [3, 11–14, 16] and development of practical static analyzer such as *Infer* (`http://fbinfer.com/`).

We are interested in statically estimating an upper bound of memory consumption of a program with manual memory management. We are especially interested in the upper-bound analysis of a potentially nonterminating program such as Web servers and operating systems. Such information about an upper bound of memory consumption is useful as a certificate of memory-leak freedom of these important software.

As a first step to this goal, in the previous work, we proposed a *behavioral type system* for imperative programs with manual memory management [15]. In order to show how our previous type system works, consider the following C function `h1`.

```
void h1() {
  int *x, *y;
  x = malloc(sizeof(int));
  y = malloc(sizeof(int));
```

An Extended Behavioral Type System for Memory-Leak Freedom.

Qi Tan, Kohei Suenaga and Atsushi Igarashi, Kyoto University.

```
  free(x); free(y);
}
```

By using our previous type system, we can infer that this function behaves according to a *behavioral type* **malloc**; **malloc**; **free**; **free** that expresses that this function performs allocation twice and then deallocation twice. By inspecting this behavioral type, we can obtain information on how much memory this program may consume at most. Our previous type system can also deal with more complex control statement and recursive function calls.

One problem in the previous type system consists in how we analyze a branch statement. Consider the following function `h2`.

```
void h2(int *x) {
  int *y;
  if (x == NULL) { /* (A) */
    y = malloc(sizeof(int));
  }
  if (x == NULL) { /* (B) */
    free(y);
  }
}
```

This function contains two sequentially executed branch statements. Our previous type system infers $(\mathbf{malloc} + \mathbf{0}); (\mathbf{free} + \mathbf{0})$ as a behavioral type of `h2` ignoring the guard conditions. The symbol $+$ in the behavioral type represents a nondeterministic choice. Therefore, this type expresses that this function may or may not perform an allocation, and then may or may not perform a deallocation. We cannot exclude the possibility that `h2` allocates a memory cell once without a deallocation from this inferred type. However, since the guard `(A)` holds if and only if `(B)`, this program actually performs an allocation and a deallocation sequentially or does nothing.

The current paper tries to address this problem by introducing a *path-sensitive behavioral type* $(*x)(P_1, P_2)$. This type expresses a behavior P_1 if $*x$ is `NULL` and P_2 if $*x$ is not `NULL`. By using path-sensitive behavioral types, the type of `h2` can be expressed by $(*x)(\mathbf{malloc}, \mathbf{0}); (*x)(\mathbf{free}, \mathbf{0})$. With the knowledge that the value of $*x$ does not change throughout this function, which is expressed by another constructor of a behavioral type introduced later, we can conclude that the behavior of `h2` is $(\mathbf{malloc}; \mathbf{free}) + \mathbf{0}$.

In this paper, we define the syntax and the semantics of the behavioral types extended with path-sensitive ones, extend the typing rules. We also discuss how the type inferred by the type system can be used to estimate an upper bound of a program. Although the soundness of the type system is still a conjecture, we expect our extension is an important steppingstone for behavior analysis of a program with manual memory management.

The rest of this paper is structured as follows. Section 2 describes an imperative language with allocation and deallocation primitives and its operational semantics. Section 3 introduces the extended behavioral type system with path-sensitive behavioral types. Section 4 discusses the related work. Section 5 concludes this paper.

2 Language $\mathcal{L}$

This section defines an imperative language $\mathcal{L}$ with memory allocation and deallocation primitives.

In the rest of this paper, we write $\vec{x}$ for a finite sequence $x_1, \ldots, x_n$; we assume that each element is distinct. We write $[\vec{x'}/\vec{x}]s$ for the term obtained by replacing each free occurrence of $\vec{x}$ in s with variables $\vec{x'}$. We write $\mathbf{Dom}(f)$ for the partial function f. We write $f\{x \mapsto v\}$ and $f\backslash x$ as follows.

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

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

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

図 1 Syntax of $\mathcal{L}$.

2.1 Syntax

The syntax of the language $\mathcal{L}$ is defined by the BNF in Figure 1. For simplicity, a value of our language is either a pointer or **null**. The **Var** is a countably infinite set of *variables*. The statement **skip** is a do-nothing statement. 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 to the value y. The statement **free**(x) deallocates a memory cell pointed to by the pointer x. 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. We assume that the size of every memory cell is identical. The expression **null** evaluates to the null pointer. The expression $*y$ evaluates to the content of the memory cell pointed to by y. The statement **ifnull** $(*x)$ **then** s_1 **else** s_2 executes s_1 if $*x$ is **null** and executes s_2 otherwise. The statement $f(\vec{x})$ executes the procedure f with arguments $\vec{x}$. The statement **const**$(*x)s$ is a *constantness annotation*. By writing this annotation, a programmer declares that the value of $*x$ does not change during the execution of s. The statement **endconst**$(*x)$ is a marker for the end of a constantness annotation, which is used only in a runtime state.

A *procedure definition* $f \mapsto (x_1, \ldots, x_n)s$ defines a procedure f that takes $x_1, \ldots, x_n$ as arguments and executes s. We use metavariable D for a set of procedure definitions. A program is a pair $\langle D, s \rangle$, where D is the set of the procedure definitions that may be used in the execution of the main statement s.

2.2 Operational semantics

Definition 1. *A* fact c *is defined by the following BNF:*

$$c ::= \mathbf{const}(*x) \mid \mathbf{null}(*x) \mid \neg\mathbf{null}(*x).$$

We use C for a multiset of facts.

A fact is used in the semantics to keep track of the information about constantness and nullness. The fact **const**$(*x)$ represents that $*x$ is declared to be currently constant by a constantness annotation; **null**$(*x)$ and $\neg$**null**$(*x)$ represents that $*x$ is equal to **null** and not equal to **null** respectively.

For definition of the operational semantics, we designate a countable infinite set $\mathcal{H}$ of *heap addresses* ranged over by l. A configuration is of the form $\langle H, R, s, n, C \rangle$. Each elements in the configuration is as follows:

- H, a *heap*, is a finite mapping from $\mathcal{H}$ to $\mathcal{H} \cup \{\mathbf{null}\}$ representing the state of memory,
- R, an *environment*, is a finite mapping from

Var to $\mathcal{H} \cup \{\mathbf{null}\}$ representing the value bound to each variable,

- s is the statement that is being executed,
- n is the number of the memory cells available for allocation, and
- C is a multiset of *facts* that holds at the current configuration.

The operational semantics of the language $\mathcal{L}$ is given by a labeled transition relation $\langle H, R, s, n, C\rangle \xrightarrow{\rho}_D \langle H', R', s', n', C'\rangle$ where ρ is a label defined by the BNF $\rho ::= \mathbf{malloc} \mid \mathbf{free} \mid \mathbf{null}(*x) \mid \neg\mathbf{null}(*x) \mid \tau$. The label **malloc** expresses an allocation of a new memory cell. The label **free** expresses a deallocation of a memory cell. The label $\mathbf{null}(*x)$ expresses making an assumption that $*x$ is a null pointer; the label $\neg\mathbf{null}(*x)$ expresses not. The label τ expresses an action other than the above; we often omit τ in the relation $\xrightarrow{\tau}_D$. The metavariable σ is used for a finite sequence of actions $\rho_1 \dots \rho_n$. The $\xrightarrow{\rho_1 \cdots \rho_n}_D$ is short for $\xrightarrow{\rho_1}_D \xrightarrow{\rho_2}_D \dots \xrightarrow{\rho_n}_D$. We write $\stackrel{\rho}{\Longrightarrow}_D$ for $\longrightarrow^*_D \xrightarrow{\rho}_D \longrightarrow^*_D$. We write $\stackrel{\rho_1 \cdots \rho_n}{\Longrightarrow}_D$ for $\stackrel{\rho_1}{\Longrightarrow}_D \dots \stackrel{\rho_n}{\Longrightarrow}_D$.

The relation $\xrightarrow{\rho}_D$ is the least relation that satisfies the rules in Figure 2. The operational semantics is, except for the multiset of facts C, almost the same as the previous framework [15]. Important invariants of a configuration is that (1) if C contains $\mathbf{const}(*x)$, then the value of the expression $*x$ is not changed in the next step and (2) if C contains $\mathbf{null}(*x)$ (resp., $\neg\mathbf{null}(*x)$), then the value of $*x$ is equal to **null** (resp., not equal to null).

In order to manipulate the C part of a configuration, we use a function $\mathit{filter}(C, *x)$ in the rules for the operational semantics defined as follows:

$$\begin{array}{l}\mathit{filter}(C, *x) = \\ \quad \mathbf{let}\ \ C' = C - \mathbf{const}(*x)\ \mathbf{in} \\ \quad\quad \mathbf{if}\ \mathbf{const}(*x) \in C'\ \mathbf{then}\ \ C' \\ \quad\quad \mathbf{else}\ \ C' \backslash \{\mathbf{null}(*x), \neg\mathbf{null}(*x)\}.\end{array}$$

The function $\mathit{filter}(C, *x)$ sets C' to the multiset obtained by removing one $\mathbf{const}(*x)$ from C. (Recall that C is a *multiset* of facts.) If C' does not contain $\mathbf{const}(*x)$, then it deletes all the facts related to $*x$. Otherwise, it returns C'.

We explain the rules related to path sensitivity and the constantness annotations; for the other rules, see [15].

- Sem-ConstSeq: The end of the **const** block is marked at the end of the current continuation. The fact $\mathbf{const}(*x)$ is also added to C to record that $*x$ is declared to be constant until $\mathbf{endconst}(*x)$ is encountered.
- Sem-Assign and Sem-ConstAssignExn: These rules handle assignment statements. If constantness of $*z$ is declared (i.e., $\mathbf{const}(*z) \in C$), then a write operation to $*z$ leads to an exception **ConstEx**. This exception is raised if a constantness annotation provided by a programmer is wrong. The type system introduced later does not guarantee unreachability to this exceptional state.
- Sem-IfNullT, Sem-IfNullF, Sem-IfConstNullT, and Sem-IfConstNullF: These rules handle an **if**-statements depending on whether $*x$ is null or not. The semantics records which branch is taken to the multiset of facts C only if C contains $\mathbf{const}(*x)$. Otherwise, even if the semantics recorded $\mathbf{null}(*x)$ or $\neg\mathbf{null}(*x)$ to C, it may not be reliable in the sense of the invariant (2) above since the value of $*x$ may be changed by an assignment via an alias of x.
- Sem-ConstSkip: By using the function *filter* above, this rule removes one $\mathbf{endconst}(*x)$ from C and conducts "garbage collection" of the facts related to $*x$ if necessary.
- Sem-Malloc and Sem-OutOfMem: **malloc** allocates a new memory cell by extending the heap H and decreasing the value of n by 1. If there is no available cell (i.e., $n = 0$), then **malloc** raises **OutOfMemory** exception.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

図 2 Operational semantics of $\mathcal{L}$.

- Sem-Free: **free**(x) removes the entry for $R(x)$ from the heap H and increase the value of n by 1.
- Sem-AssignExn, Sem-FreeExn and Sem-DerefExn: If a program tries to access to a deallocated memory cell, then the exception **MemEx** is raised.

Definition 2. *A program* $\langle D, s\rangle$ *is said to* execute within n memory cells *if* $\langle \emptyset, \emptyset, s, n, \emptyset\rangle \not\longrightarrow_D^* \mathbf{MemEx}$.

P	(behavioral types)	::=	$\mathbf{0} \mid P_1; P_2 \mid \mathbf{malloc} \mid \mathbf{free} \mid (x)P \mid (*x)(P_1, P_2) \mid \mathbf{const}(*x)P \mid \mathbf{endconst}(*x)$ $\mid \alpha \mid \mu\alpha.P$
Γ	(variable type environment)	::=	$\{x_1, x_2, \ldots, x_n\}$
Ψ	(dependent function type)	::=	$(\vec{x})P$
Θ	(function type environment)	::=	$\{f_1 : \Psi_1, \ldots, f_n : \Psi_n\}$

図 3 Syntax of types.

3 Type system

3.1 Types

Figure 3 defines the syntax of *types*. Behavioral types, ranged over by P, specify the behavior of statements. The type $\mathbf{0}$ represents the do-nothing behavior. The type $P_1; P_2$ is the sequential composition of P_1 and P_2. The types **malloc** and **free** represents one allocation and one deallocation, respectively. The type $(x)P$ is the behavior that binds a variable x and behaves as specified by P; the type P may be dependent to x. For example, the behavior of the statement **let** $x = y$ **in skip** is represented by $(x)\mathbf{0}$ since it binds the variable x and does nothing. The type $(*x)(P_1, P_2)$ is the behavior of a branching statement. This type represents the behavior P_1 if $*x$ is null; P_2 otherwise. The type $\mathbf{const}(*x)$ represents the behavior of a statement that works as specified in P under the assumption that the value of $*x$ does not change. The type $\mathbf{endconst}(*x)$ represents the behavior that marks the end of the constantness annotation. The types α and $\mu\alpha.P$ allow to specify a recursive behavior. The type $\mu\alpha.P$ represents the behavior P, in which α represents the behavior of $\mu\alpha.P$ itself. For example, $\mu\alpha.(\mathbf{malloc}; \mathbf{free}; \alpha)$ represents a statement that repeats an allocation followed by a deallocation forever. Γ, a *type environment*, is a set of variables representing the set of free variables that may appear in a statement.

The types for procedures, ranged over by Ψ, is of the form $(\vec{x})P$. This type represents a procedure that takes $\vec{x}$ as arguments and behaves as specified by P, which may be dependent to $\vec{x}$. Procedure type environments, ranged over by Θ, assigns a procedure type to a function name.

Figure 4 defines the operational semantics of the behavioral types. The semantics is defined by a labeled transition system over configuration of the shape $\langle P, C\rangle$. Here, C is the multiset of facts that hold. We add explanation to important rules.

- Tr-Bind: The behavioral type $(x)P$ picks up a fresh name x' and makes transition to $[x'/x]P$. To understand this definition, recall that the rules in Figure 2 for dealing with statements with a binder (i.e., Sem-LetNull, Sem-LetEq, Sem-LetDeref, and Sem-Malloc) generate a fresh variable for the bound variable and add it to the environment R of the configuration. The rule Tr-Bind simulates this behavior.
- Tr-Const and Tr-EndConst: The behavioral type $\mathbf{const}(*x)P$ adds the fact $\mathbf{const}(*x)$ to C and evolves to $P; \mathbf{endconst}(*x)$. If $\mathbf{endconst}(*x)$ is encountered, then all the facts related to $*x$ in C is removed by *filter*. This is parallel to the semantics of $\mathbf{const}(*x)s$ and $\mathbf{endconst}(*x)$ in Figure 2.
- Tr-ConstNondet1 and Tr-ConstNondet2: If the fact $\mathbf{const}(*x)$ is a member of C and there is no $\mathbf{null}(*x)$ nor $\neg\mathbf{null}(*x)$, then the behavioral type $(*x)(P_1, P_2)$ nondeterministically chooses P_1 or P_2. The fact $\mathbf{null}(*x)$ (resp. $\neg\mathbf{null}(*x)$) is added to C if the branch P_1 (resp. P_2) is taken. This addition of the fact is conducted only if $\mathbf{const}(*x)$ is a member of C (cf.,

Tr-NoConst1 and Tr-NoConst2).

- Tr-ConstNull and Tr-ConstNotNull: If $\mathbf{const}(*x)$ is a member of C and if $\mathbf{null}(*x)$ (resp. $\neg\mathbf{null}(*x)$) is a member of C, then the behavioral type $(*x)(P_1, P_2)$ evolves to P_1 (resp. P_2). This happens only if (1) the behavioral type $(*x)(P_1, P_2)$ was inside the block of certain $\mathbf{const}(*x)\{\dots\}$ and (2) there was another $(*x)(P_1', P_2')$ was in the same block before $(*x)(P_1, P_2)$ is reached. Because the transition of $(*x)(P_1', P_2')$ should have registered $\mathbf{null}(*x)$ or $\neg\mathbf{null}(*x)$ to C and both $(*x)(P_1', P_2')$ and $(*x)(P_1, P_2)$ are inside $\mathbf{const}(*x)\{\dots\}$, it is safe to assume that $(*x)(P_1, P_2)$ takes the same branch as $(*x)(P_1', P_2')$.
- Tr-NoConst1 and Tr-NoConst2: If $\mathbf{const}(*x)$ is not in the multiset of facts C, then the behavioral type $(*x)(P_1, P_2)$ nondeterministically evolves to P_1 or P_2.

Example 3.1. *Let P be the behavioral type* $\mathbf{const}(*x)\{(*x)(\mathbf{malloc}, \mathbf{0}); (*x)(\mathbf{free}, \mathbf{0})\}$. *We write P_1 for $(*x)(\mathbf{malloc}, \mathbf{0})$, P_2 for $(*x)(\mathbf{free}, \mathbf{0})$, C_1 for $\{\mathbf{const}(*x)\}$, C_2 for $\{\mathbf{null}(*x)\}$, and C_3 for $\{\neg\mathbf{null}(*x)\}$. Then, the following transition sequence is possible.*

$$
\begin{array}{ll}
 & \langle P, \emptyset\rangle \\
\to & \langle P_1; P_2; \mathbf{endconst}(*x), C_1\rangle \\
\to & \langle \mathbf{malloc}; P_2; \mathbf{endconst}(*x), C_1 \cup C_2\rangle \\
\xrightarrow{\mathbf{malloc}} & \underline{\langle P_2; \mathbf{endconst}(*x), C_1 \cup C_2\rangle} \\
\to & \langle \mathbf{free}; \mathbf{endconst}(*x), C_1 \cup C_2\rangle \\
\xrightarrow{\mathbf{free}} & \langle \mathbf{endconst}(*x), C_1 \cup C_2\rangle \\
\to & \langle \mathbf{0}, \emptyset\rangle.
\end{array}
$$

Notice that, from the underlined configuration, the transition to $\langle \mathbf{0}; \mathbf{endconst}(*x), C_1 \cup C_2\rangle$ is not allowed because the facts include $\mathbf{const}(*x)$ and $\mathbf{null}(*x)$. Another possible transition is

$$
\begin{array}{ll}
 & \langle P, \emptyset\rangle \\
\to & \langle P_1; P_2; \mathbf{endconst}(*x), C_1\rangle \\
\to & \langle \mathbf{0}; P_2; \mathbf{endconst}(*x), C_1 \cup C_3\rangle \\
\to & \underline{\langle P_2; \mathbf{endconst}(*x), C_1 \cup C_3\rangle} \\
\to & \langle \mathbf{0}; \mathbf{endconst}(*x), C_1 \cup C_3\rangle \\
\to & \langle \mathbf{endconst}(*x), C_1 \cup C_3\rangle \\
\to & \langle \mathbf{0}, \emptyset\rangle.
\end{array}
$$

A transition to $\langle \mathbf{free}; \mathbf{endconst}(*x), C_1 \cup C_3\rangle$ is not allowed because $C_1 \cup C_3$ contains $\mathbf{const}(*x)$ and $\neg\mathbf{null}(*x)$. Notice that the path-sensitive behavioral type excludes the possibility of $\langle P, \emptyset\rangle \xrightarrow{\mathbf{malloc}} \langle \mathbf{0}, \emptyset\rangle$ and $\langle P, \emptyset\rangle \xrightarrow{\mathbf{free}} \langle \mathbf{0}, \emptyset\rangle$.

3.2 Type judgment

The type judgment for statements is of the form $\Theta; \Gamma \vdash s : P$. The judgment reads "the behavior of s is P under the function type environment Θ and the variable type environment Γ".

We incorporate subtyping to our type system defined as follows.

Definition 3 (Subtyping)**.** *$C \vdash P_1 \le P_2$ is the largest relation such that, for any P_2', C', and ρ, if $\langle P_2, C\rangle \xrightarrow{\rho} \langle P_2', C'\rangle$, then there exists P_1' such that $\langle P_1, C\rangle \stackrel{\rho}{\Longrightarrow} \langle P_1', C'\rangle$ and $C' \vdash P_1' \le P_2'$. We write $P_1 \le P_2$ if $C \vdash P_1 \le P_2$ for any C.*

The type judgment $\Theta; \Gamma \vdash s : P$ is defined as the least relation that satisfies the rules in Figure 5. We give explanation to several important rules.

- T-Malloc: The statement $\mathbf{let}\ \ x = \mathbf{malloc}()\ \mathbf{in}\ \ s$ performs an allocation once, binds the variable x, and then executes s. The behavior of the whole statement is therefore $\mathbf{malloc}; (x)P$.
- T-LetEq: The statement $\mathbf{let}\ \ x = y\ \mathbf{in}\ \ s$ makes an alias x of y and then executes s. Therefore, we rename x to y in the behavioral type P of s.
- T-LetDeref and T-LetNull: The statements $\mathbf{let}\ \ x = *y\ \mathbf{in}\ s$ and $\mathbf{let}\ \ x = \mathbf{null}\ \mathbf{in}\ \ s$

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

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

$$\langle \mathbf{malloc}, C\rangle \xrightarrow{\mathbf{malloc}} \langle \mathbf{0}, C\rangle \quad \text{(Tr-Malloc)}$$

$$\langle \mathbf{free}, C\rangle \xrightarrow{\mathbf{free}} \langle \mathbf{0}, C\rangle \quad \text{(Tr-Free)}$$

$$\frac{x' \text{ is fresh.}}{\langle (x)P, C\rangle \to \langle [x'/x]P, C\rangle} \quad \text{(Tr-Bind)}$$

$$\langle \mathbf{const}(*x)P, C\rangle \to \langle P; \mathbf{endconst}(*x), C + \mathbf{const}(*x)\rangle \quad \text{(Tr-Const)}$$

$$\frac{C' = \mathit{filter}(C, *x)}{\langle \mathbf{endconst}(*x), C\rangle \to \langle \mathbf{0}, C'\rangle} \quad \text{(Tr-Endconst)}$$

$$\frac{\mathbf{const}(*x) \notin C}{\langle (*x)(P_1, P_2), C\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, C\rangle} \text{(Tr-NoConst1)} \qquad \frac{\mathbf{const}(*x) \notin C}{\langle (*x)(P_1, P_2), C\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, C\rangle} \text{(Tr-NoConst2)}$$

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

$$\frac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin C \qquad \mathbf{const}(*x) \in C}{\langle (*x)(P_1, P_2), C\rangle \xrightarrow{\mathbf{null}(*x)} \langle P_1, C + \mathbf{null}(*x)\rangle} \quad \text{(Tr-ConstNondet1)}$$

$$\frac{\mathbf{null}(*x), \neg\mathbf{null}(*x) \notin C \qquad \mathbf{const}(*x) \in C}{\langle (*x)(P_1, P_2), C\rangle \xrightarrow{\neg\mathbf{null}(*x)} \langle P_2, C + \neg\mathbf{null}(*x)\rangle} \quad \text{(Tr-ConstNondet2)}$$

$$\langle \mu\alpha.P, C\rangle \to \langle [\mu\alpha.P/\alpha]P, C\rangle \quad \text{(Tr-Rec)}$$

図 4 semantics of behavioral types with dependent types.

bind x and then execute s. Therefore, the type of the whole statement is $(x)P$. The type system just ignores that what the variable x is bound to.

- T-IfNull: The statement

  ifnull $(*x)$ **then** s_1 **else** s_2

  branches depending on whether $*x$ is null or not. This behavior is abstracted by the type $(*x)(P_1, P_2)$.
- T-Call: If the type of the procedure f is $(\vec{x})P$, then the type of $f(\vec{y})$ is $[\vec{y}/\vec{x}]P$; notice that P may depend on the arguments of the procedure.

Typing rules for the declarations D and programs $\langle D, s\rangle$ are also shown in Figure 5. These rules are standard.

3.3 Type soundness

Although we have not completed the proof concretely, we conjecture that the following preservation theorem for the judgment $\Theta; \Gamma \vdash s : P$ holds.

Conjecture 3.1 (Preservation)**.** *Suppose that*

- $\vdash D : \Theta$,
- $\Theta; \Gamma \vdash s : P$,
- $\Gamma \subseteq \mathbf{Dom}(R)$,
- $\langle H, R, s, n, C\rangle \xrightarrow{\rho} \langle H', R', s', n', C'\rangle$,

Then, there exist Γ' *and* P' *such that*

- $\Theta; \Gamma' \vdash s' : P'$,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

図 5 Typing rules.

- $\Gamma' \subseteq \mathbf{Dom}(R')$,
- $\langle P, C\rangle \stackrel{\rho}{\Longrightarrow} \langle P', C'\rangle$.

The following fact is a corollary of Conjecture 3.1.

Conjecture 3.2. *If* $\vdash \langle D, s\rangle : P$ *and* $\langle \emptyset, \emptyset, s, n, \emptyset\rangle \stackrel{\sigma}{\Longrightarrow}_D$, *then* $\langle P, \emptyset\rangle \stackrel{\sigma}{\Longrightarrow}$.

3.4 Verification of memory-leak freedom of a program using the inferred behavioral type

Conjecture 3.2 states that the behavior of a program is soundly abstracted by its type. This section describes how the inferred type can be used to estimate an upper bound of memory consumption. We first define "memory consumption" of a behavioral type.

Definition 4. *We write* $OK_n(P, C)$ *if (1)* $\langle P, C\rangle \xrightarrow{\sigma} \langle P', C'\rangle$ *implies* $\sharp_{\mathbf{malloc}}(\sigma) - \sharp_{\mathbf{free}}(\sigma) \leq n$ *where* $\sharp_\rho(\sigma)$ *is the number of* ρ *in* σ.

Conjecture 3.3. *If*

- $\Theta;\Gamma \vdash s : P$,
- $\Gamma \subseteq \mathbf{Dom}(R)$,
- $OK_n(P, C)$, *and*

- $n' \geq n$,

Then, $\langle H, R, s, n', C\rangle \stackrel{\sigma}{\Longrightarrow}^{*}$ **OutOfMemory**.

The conjecture above states that, if $\vdash \langle D, s\rangle : P$ and $OK_n(P, \emptyset)$ hold, then n is an upper bound of the program. Therefore, by conducting type inference to the program, obtaining a type, say, P, and estimating an upper bound of P, we can estimate an upper bound of the memory consumption of the program.

4 Related Work

Behavioral types are widely used in program analysis; a survey of this area can be found in [5]. Several representative uses of the behavioral types are ensuring correctness of the communications conducted among several processes [1, 2, 4, 6], static deadlock-freedom verification [8,9] and static correctness verification of resource-usage patterns (e.g., a file hanlder is closed before termination) [7, 10]. Our current work also can be seen as a static analysis of resource-usage patterns in which memory is a single and unique resource that is accessed via the primitives **malloc** and **free**.

Static verification of memory-leak freedom has been another interesting topic of program verification [3, 11–14, 16]. The main interest of the previous work is guaranteeing the following property: A pointer to every allocated memory cell will be passed to **free** eventually. Our work focuses on the sequence of **malloc** and **free** that may be conducted by a program, forgetting about which **free** deallocates which memory cell.

5 Conclusion

We presented a path-sensitive behavioral type system for an imperative language with manual memory management. Our type system abstracts the behavior of a program concerning the primitives **malloc** and **free** for manual memory management. We stated several conjectures about the type system. We also described how the inferred behavioral types can be used for estimating an upper bound of memory consumption.

We are currently designing and implementing a type inference algorithm for our type system. We can infer a type of a program using essentially the same algorithm as that of Kobayashi et al. [10]. We need to check the feasibility of our type system using practical programs.

The current type system requires a programmer to write constantness annotation **const**$(*x)$. The correctness of these annotations are currently the responsibility of the programmer: If the annotation is wrong, then the type system does not guarantee anything. We suppose that we can verify the correctness of these annotations using our previous work on *fractional ownerships* [12]. We also expect that we can automatically insert constantness annotations using the type inference algorithm of our previous work.

参 考 文 献

[1] Caires, L. and Vieira, H. T.: Conversation types, *Theor. Comput. Sci.*, Vol. 411, No. 51-52(2010), pp. 4399–4440.

[2] Deniélou, P., Yoshida, N., Bejleri, A., and Hu, R.: Parameterised Multiparty Session Types, *Logical Methods in Computer Science*, Vol. 8, No. 4(2012).

[3] Heine, D. L. and Lam, M. S.: A practical flow-sensitive and context-sensitive C and C++ memory leak detector, *PLDI*, Cytron, R. and Gupta, R.(eds.), ACM, 2003, pp. 168–181.

[4] Honda, K., Yoshida, N., and Carbone, M.: Multiparty asynchronous session types, *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*, 2008, pp. 273–284.

[5] Hüttel, H., Lanese, I., Vasconcelos, V. T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H. T., and Zavattaro, G.: Foundations of Session Types and Behavioural Contracts, *ACM Comput. Surv.*, Vol. 49, No. 1(2016), pp. 3.

[6] Igarashi, A. and Kobayashi, N.: A generic type system for the Pi-calculus, *Theor. Comput. Sci.*, Vol. 311, No. 1-3(2004), pp. 121–163.

[7] Igarashi, A. and Kobayashi, N.: Resource usage analysis, *ACM Trans. Program. Lang. Syst.*, Vol. 27, No. 2(2005), pp. 264–313.

[8] Kobayashi, N.: Type-based information flow analysis for the pi-calculus, *Acta Inf.*, Vol. 42, No. 4-5(2005), pp. 291–347.

[9] Kobayashi, N.: A New Type System for Deadlock-Free Processes, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, 2006, pp. 233–247.

[10] Kobayashi, N., Suenaga, K., and Wischik, L.: Resource Usage Analysis for the π-Calculus, *Logical Methods in Computer Science*, Vol. 2, No. 3(2006).

[11] Orlovich, M. and Rugina, R.: Memory Leak Analysis by Contradiction, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, 2006, pp. 405–424.

[12] Suenaga, K. and Kobayashi, N.: Fractional Ownerships for Safe Memory Deallocation, *APLAS*, Hu, Z.(ed.), Lecture Notes in Computer Science, Vol. 5904, Springer, 2009, pp. 128–143.

[13] Sui, Y., Ye, D., and Xue, J.: Static memory leak detection using full-sparse value-flow analysis, *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 254–264.

[14] Swamy, N., Hicks, M. W., Morrisett, G., Grossman, D., and Jim, T.: Safe manual memory management in Cyclone, *Sci. Comput. Program.*, Vol. 62, No. 2(2006), pp. 122–144.

[15] Tan, Q., Suenaga, K., and Igarashi, A.: A Behavioral Type System for Memory-Leak Freedom, *Proceedings of PPL 2015*, March 2015. `http://www.fos.kuis.kyoto-u.ac.jp/~tanki/memoryleak.pdf`.

[16] Xie, Y. and Aiken, A.: Context- and path-sensitive memory leak detection, *ESEC/SIGSOFT FSE*, Wermelinger, M. and Gall, H.(eds.), ACM, 2005, pp. 115–125.