A Behavioral Type System for Memory-Leak Freedom

Qi Tan, Kohei Suenaga
Kyoto University
{tanki, ksuenaga}@fos.kuis.kyoto-u.ac.jp

1. Problem and Motivation

In order to prevent dynamic memory-management related errors such as memory leaks and illegal read/write/free operations to deallocated memory cells, many static verification techniques have been proposed [2, 6–9, 11]. These analyses proposed so far mainly guarantee *partial* memory-leak freedom: All the allocated memory cells are deallocated *if a program terminates*. Hence, these proposed verifiers will say that a nonterminating program is safe.

In real-world programs, *total* memory-leak freedom – if a program consumes bounded number of memory cells during execution – is an important property (e.g., operating system and Web servers). However, analyses for partial memory-leak freedom is not enough for such software that does not terminate.

$h()$=	$h'()$=
$\quad\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}$	$\quad\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}$
$\quad\mathbf{let}\ y = \mathbf{malloc}()\ \mathbf{in}$	$\quad\mathbf{let}\ y = \mathbf{malloc}()\ \mathbf{in}$
$\quad\mathbf{free}(x); \mathbf{free}(y); h()$	$\quad h'(); \mathbf{free}(x);\ \mathbf{free}(y)$

Figure 1. Memory leaks in nonterminating programs.

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

Currently, we are investigating a static verification method to prove, total memory-leak freedom, that a program consumes bounded number of memory cells during execution.

2. Approach

Our method is to abstract the behavior of programs by a *behavioral type system* [3–5]. Behavioral types are described by sequential processes whose actions represent memory allocation and deallocation. For example, our type system can assign a type $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{free};\mathbf{free};\alpha$ to the function h in Figure 1. This type expresses that h can allocate a memory cell twice, deallocate a memory cell twice, and then iterate this behavior. The type assigned to h' in Figure 1 is $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\alpha;\mathbf{free};\mathbf{free}$, which expresses that h' can allocate a memory cell twice, call itself recursively, and then deallocate a memory cell twice. Hence, by inspecting the inferred types (for example, by using some model checkers), one can estimate the upper bound required to execute h and h'.

Notice that, our behavioral type system only about the number and the order of allocations, deallocations and recursive calls; hence, the type system does not guarantee that illegal accesses to a dangling pointer. However, combining safe-memory-deallocation analyses (e.g., one proposed by Suenaga and Kobayashi [7]) with our behavioral type system, we can verify safe-memory-deallocation even for nonterminating programs.

2.1 Language

The definition of the language is as follows; It is a sublanguage of Suenaga and Kobayashi [7].

$$\begin{array}{lll} s & \text{(statements)} & ::= \ \mathbf{skip} \mid s_1; s_2 \mid *x \leftarrow y \mid \mathbf{free}(x) \\ & & \quad\ \mid \mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s \\ & & \quad\ \mid f(\vec{x}) \mid \mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2 \end{array}$$

The command $\mathbf{skip}$ does nothing; the sequence $s_1; s_2$ means the executing order of s_1 and s_2; $*x \leftarrow y$ updates the content of cell pointed by x with the value y; command $\mathbf{free}(x)$ deallocates the memory cell through the pointer x; $\mathbf{let}\ x = \mathbf{malloc}()\ \mathbf{in}\ s$ allocates a cell pointed by x and then executes s; $f(\vec{x})$ means a function call which receives some parameters. Here $\vec{x}$ represents a sequence of variables $\{x_1, ..., x_n\}$; $\mathbf{ifnull}\ (x)\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$ executes s_1 if x is $\mathbf{null}$, otherwise s_2. We omit some commands like dereferenciinig a pointer, let bindings and definitions of variables, program and so on.

A program *leaks* memory if the program consumes unbounded number of memory cells. For example, see Figure 1 again, h does not leak memory, whereas h' does; the former consumes at most two memory cells at once but the latter consumes unbounded number of memory cells.

2.2 Behavioral Type System

We define behavioral types, CCS-like processes that abstract the behavior of programs, as follows:

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

The type **0** is the behavior "does nothing"; $P_1; P_2$ is for sequential execution; $P_1 + P_2$ is for choice; $\mathbf{malloc}$ is the behavior of a program that allocates a memory cell exactly once; $\mathbf{free}$ is for deallocating exactly once; $\mu\alpha.P$ is the recursive type. For example, in Figure 1, function h has the type $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\mathbf{free};\mathbf{free};\alpha$; function h' has the type $\mu\alpha.\mathbf{malloc};\mathbf{malloc};\alpha;\mathbf{free};\mathbf{free}$. The semantics of behavioral types is given by a labeled transition system, which is omitted.

The type judgment of our type system is of the form $\Theta; \Gamma \vdash s : P$. It reads "under environments Θ and Γ the behavior of s is as described in P". We design the type system so that $\Theta; \Gamma \vdash s : P$ implies the following property:

> When s executes $\mathbf{malloc}$ (resp. $\mathbf{free}$), then P is equivalent to $\mathbf{malloc}; P'$ (resp. $\mathbf{free}; P'$) for a type P' such that $\Theta; \Gamma \vdash s' : P'$, where s' is the continuation of s.

[1] We assume that $\mathbf{malloc}()$ allocates a fixed-sized block

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

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

Figure 2. Typing rules for $\mathbf{free}()$ and $\mathbf{malloc}()$

This property guarantees that the behavioral types soundly approximates the upper bound of the consumed memory cells.

We present two typing rules. The rule T-Free represents that the behavior of $\mathbf{free}()$ is $\mathbf{free}$. The rule T-Malloc represents that $\mathbf{let}\ \ x = \mathbf{malloc}()\ \mathbf{in}\ \ s$ has the behavior $\mathbf{malloc}(); P$, where P is the behavior of s.

3. Experiment

To check feasibility of our behavioral type system for verification of total memory-leak freedom and investigate problems in current type system framework, we apply CPAChecker [1] (1) to an original program and (2) to a program which represents the abstracted behavior of original one.

	original programs	abstracted behavior
	Result of verification	Result of verification
poker.c	Success	Fail
database.c	Success	Fail
gen_init_cpio.c	Success	Fail
decompress_unlzo.c	Success	Fail

Figure 3. Verification of total memory-leak free

In Figure 3, the result of verification $Success$ and $Fail$ represent a program consumes bounded or unbounded number of memory respectively. It shows that our type system is too imprecise to verify total memory-leak freedom of the programs. The reason is that our method is not path-sensitive. One typical example is as follows.

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

Figure 4. Success by CPAChecker but Fail by our aproach

The abstracted behavioral type of above program is $\mu\alpha.(\mathbf{0} + \mathbf{malloc}); (\mathbf{0} + \mathbf{free}); \alpha$ which is not enough to check no-memory-leak, although it is memory-leak free if the condition c does not change between the allocation and deallocation.

We have checked many of source codes from Github and Linux kernel and found that extending our behavioral type system with dependencies is useful for our future work.

$$\begin{array}{lll} P & ::= & \ldots \\ & & \mid \text{CASE(x)}\ \{\text{VAL\_1} : P_1; \ldots; \text{VAL\_n}: P_n\ \} \end{array}$$

Figure 5. Extension to dependent types

Figure 5 shows that extending current behavioral type system with dependent types. By using this dependent type, the behavior type of program in Figure 4 is $\mu\alpha.\text{CASE(c)}\{\text{VAL\_1} : \mathbf{malloc}; \text{VAL\_2} : 0\}; \text{CASE(c)}\{\text{VAL\_1} : \mathbf{free}; \text{VAL\_2} : 0\}; \alpha$, which is able to check no-memory-leak.

4. Related Works

Many static verification methods have been proposed [2, 6–9, 11] for memory-leak freedom. These methods guarantee partial memory-leak freedom and no illegal accesses to deallocated cells, whereas our behavioral type system guarantees total memory-leak freedom. By using both their methods and our type system, we can guarantee safe-memory-deallocation even for nonterminationg programs.

Behavioral types are extensively studied in the context of concurrent program verification [3–5, 10]. Our type system is largely inspired by one proposed by Kobayashi et al. [5], which guarantees that a concurrent program accesses resources according to specification.

5. Conclusion

To verify memory-leak freedom for possibly nonterminating programs, we have proposed a behavioral type system which abstracts the behavior of programs with allocation and deallocation. Although we omitted the statement and proofs, we have proved the type soundness and conducted experiments to check feasibility of our approach.

The allocation primitives defined in our type system ignore the size of the allocated block for simplification. Hence, our approach may see a memory-leak program as a well-typed one. Variable-length cells will be included in our future work. Another feature should be considered is path-sensitive, because our behavioral type system is not enough to check memory-leak freedom. For preciseness, we are going to extend our type system with dependencies.

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. ISBN 1-58113-662-5.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In C. Hankin, editor, *ESOP 1998*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. ISBN 3-540-64302-8. . URL `http://dx.doi.org/10.1007/BFb0053567`.

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

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

[6] 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. ISBN 3-540-37756-5. . URL `http://dx.doi.org/10.1007/11823230_26`.

[7] 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. ISBN 978-3-642-10671-2.

[8] 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. ISBN 978-1-4503-1454-1. . URL `http://doi.acm.org/10.1145/2338965.2336784`.

[9] 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.

[10] H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In S. Drossopoulou, editor, *ESOP 2008*, volume 4960 of *Lecture Notes in Computer Science*, pages 269–283. Springer, 2008. ISBN 978-3-540-78738-9. . URL `http://dx.doi.org/10.1007/978-3-540-78739-6_21`.

[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. ISBN 1-59593-014-0.