A Behavioral Type System for Memory-Leak Freedom

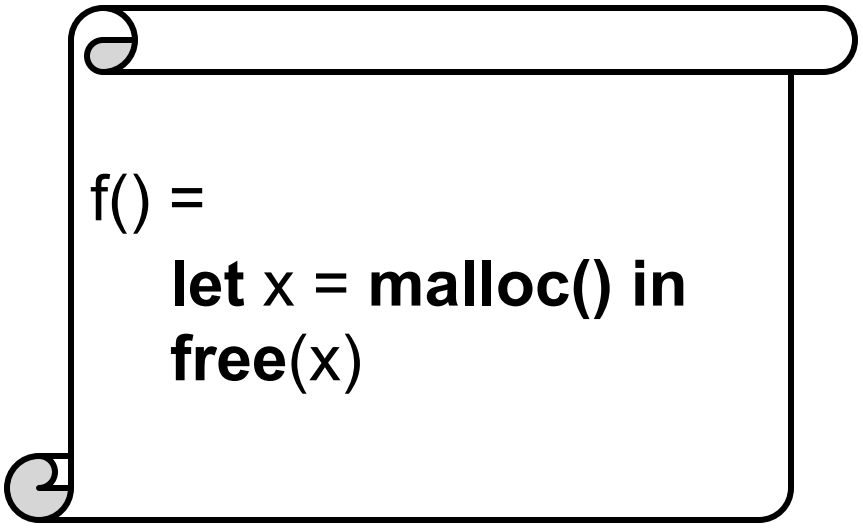
Qi Tan, Kohei Suenaga, and Atsushi Igarashi
Kyoto University

Introduction

- Memory leaks are very serious problems
  - Applications stop working
  - System crashes

Memory-leak freedom

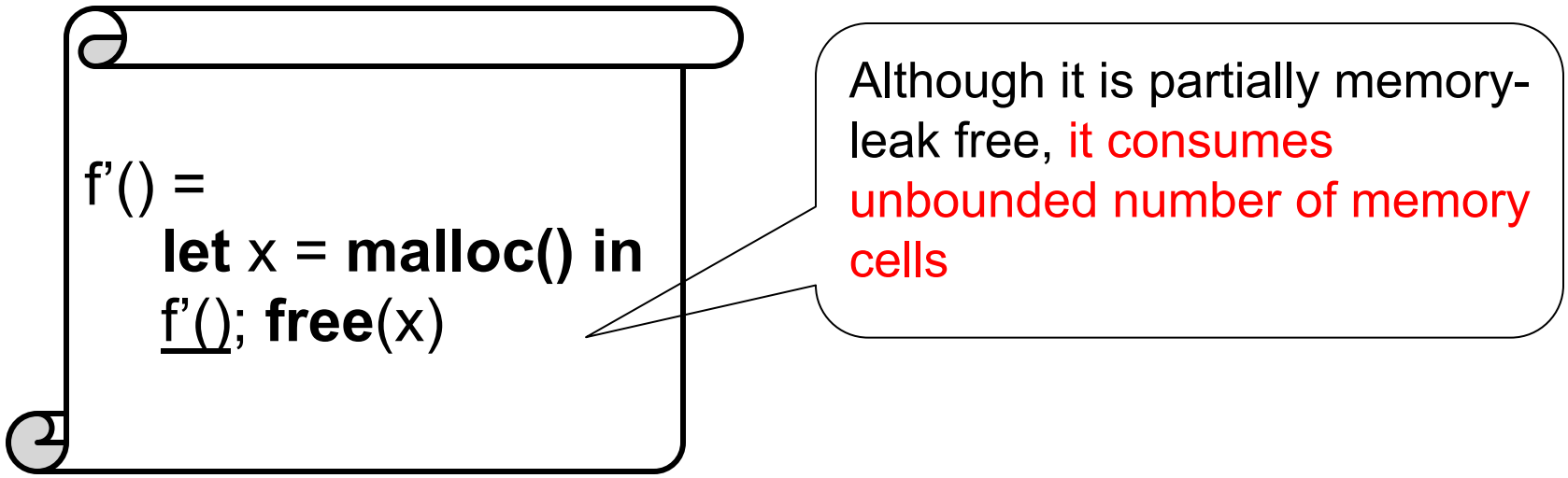
- All the allocated memory cells are eventually deallocated



Example 1: memory-leak free program

Partial memory-leak freedom

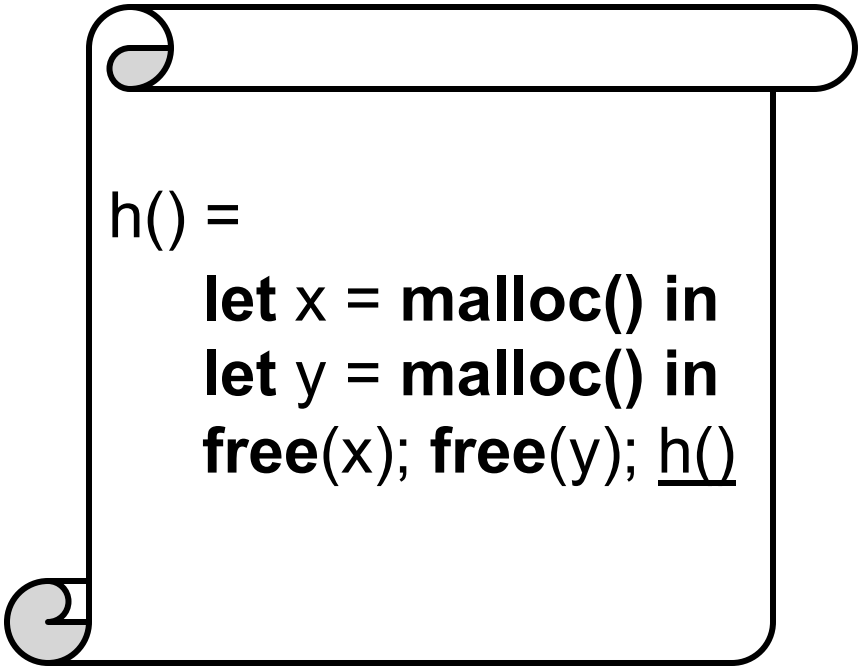
- All the allocated memory cells are eventually deallocated ***if a program terminates***

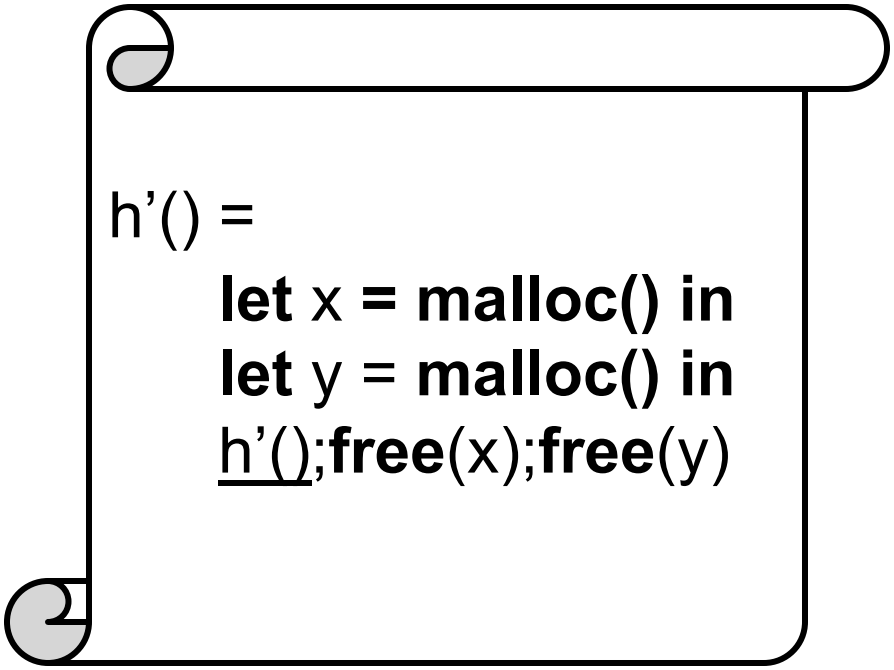


Example 2: partial memory-leak freedom

Total memory-leak freedom

- A program consumes bounded number of memory cells ***even when it does not terminate***



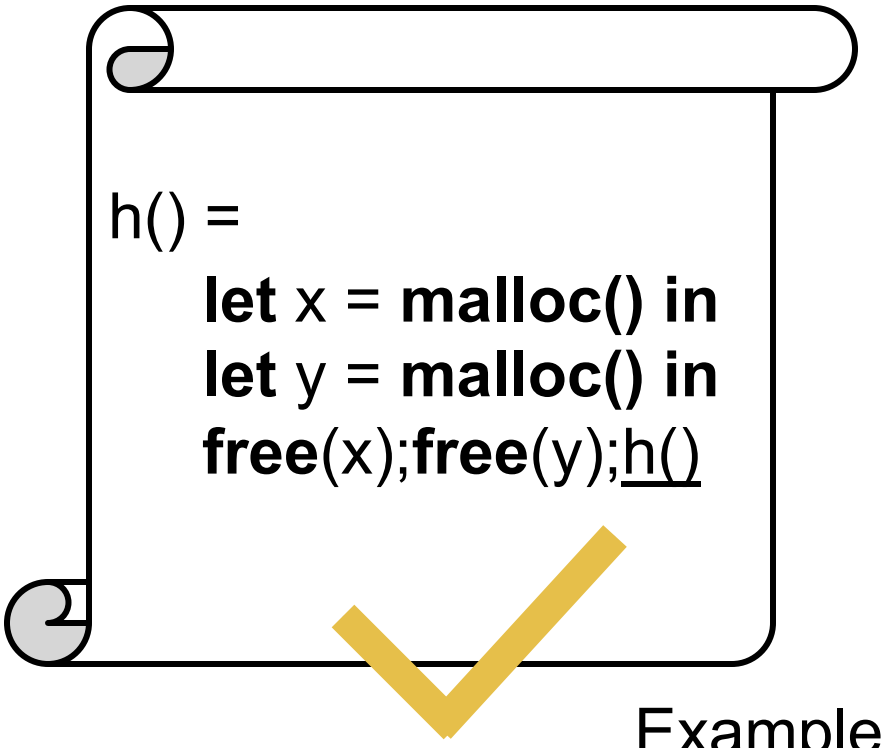


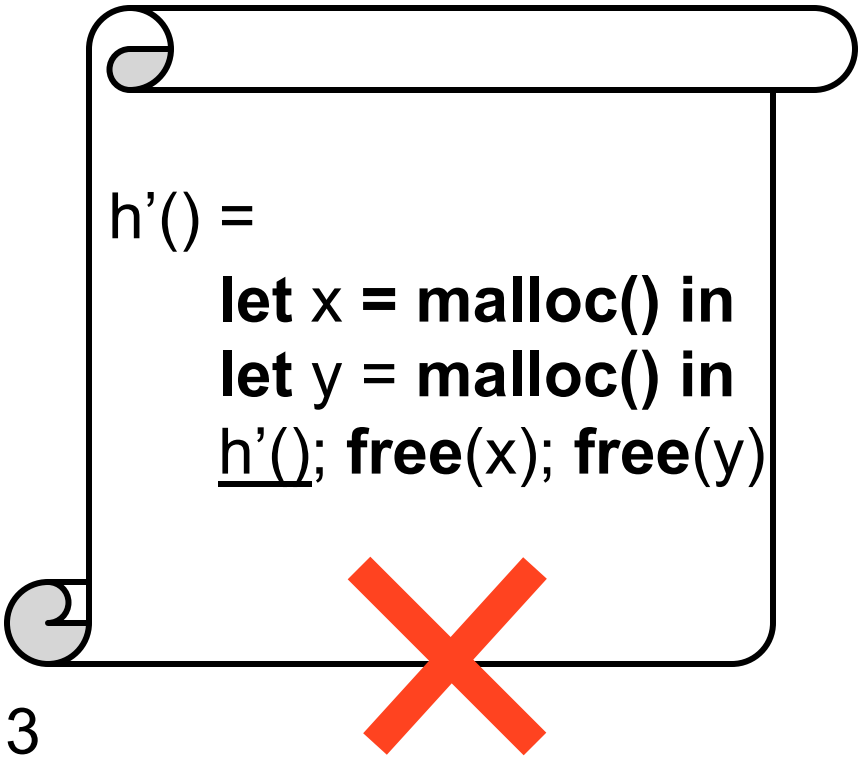
Example 3: both are partially memory-leak free.

h() is totally memory-leak free, but h'() is not.

Goal

- Verification of total memory-leak freedom



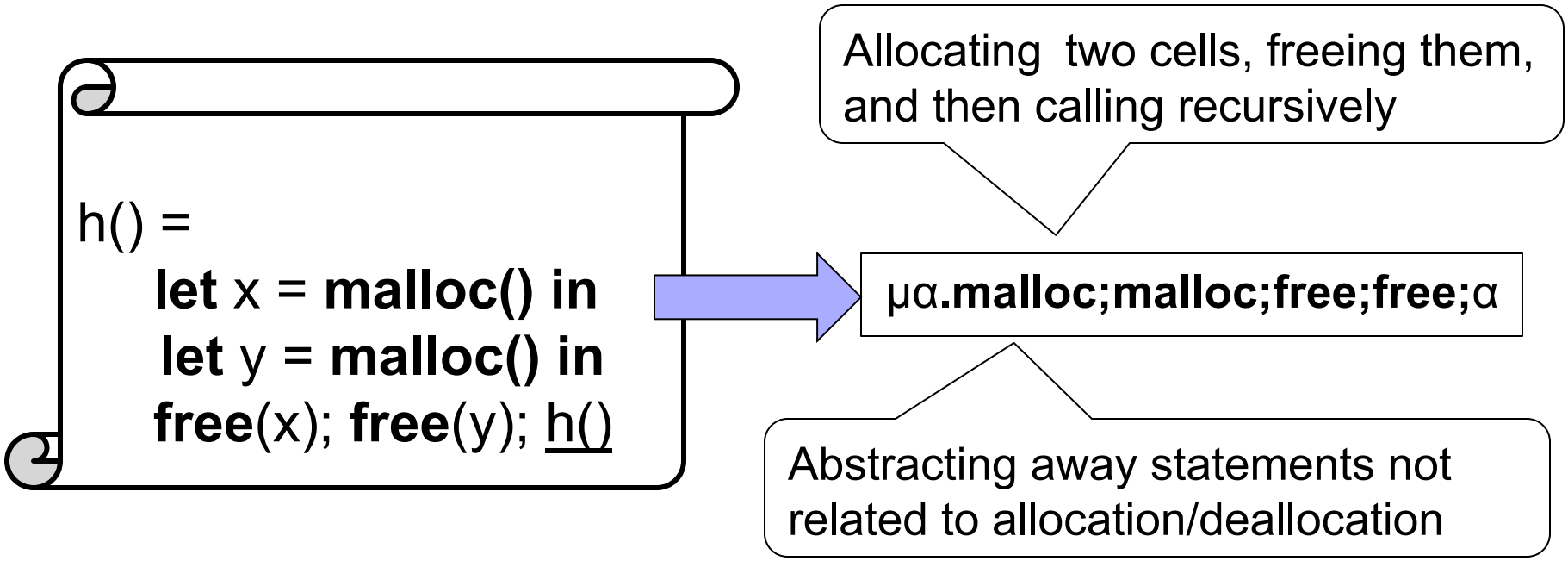


Example 3

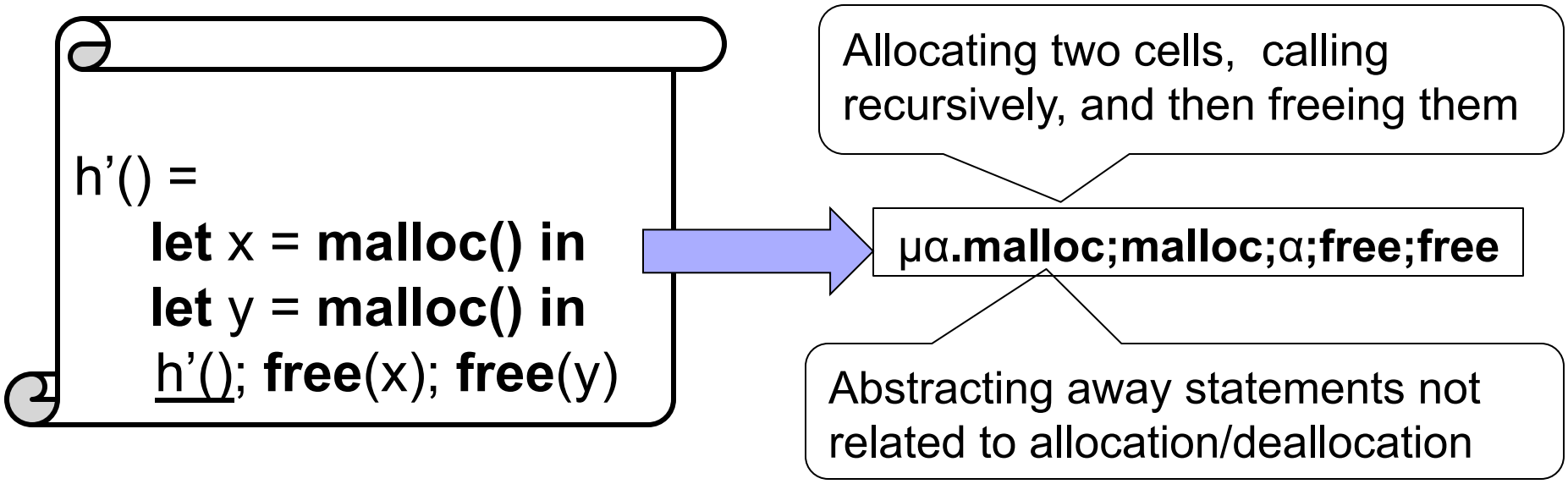
Idea

- Behavioral types to abstract the behavior of a program
  - Sequential processes as types
  - Information about the number and the order of allocations, deallocations, and recursive calls
  - Used to estimate the upper bound of memory consumption of a program

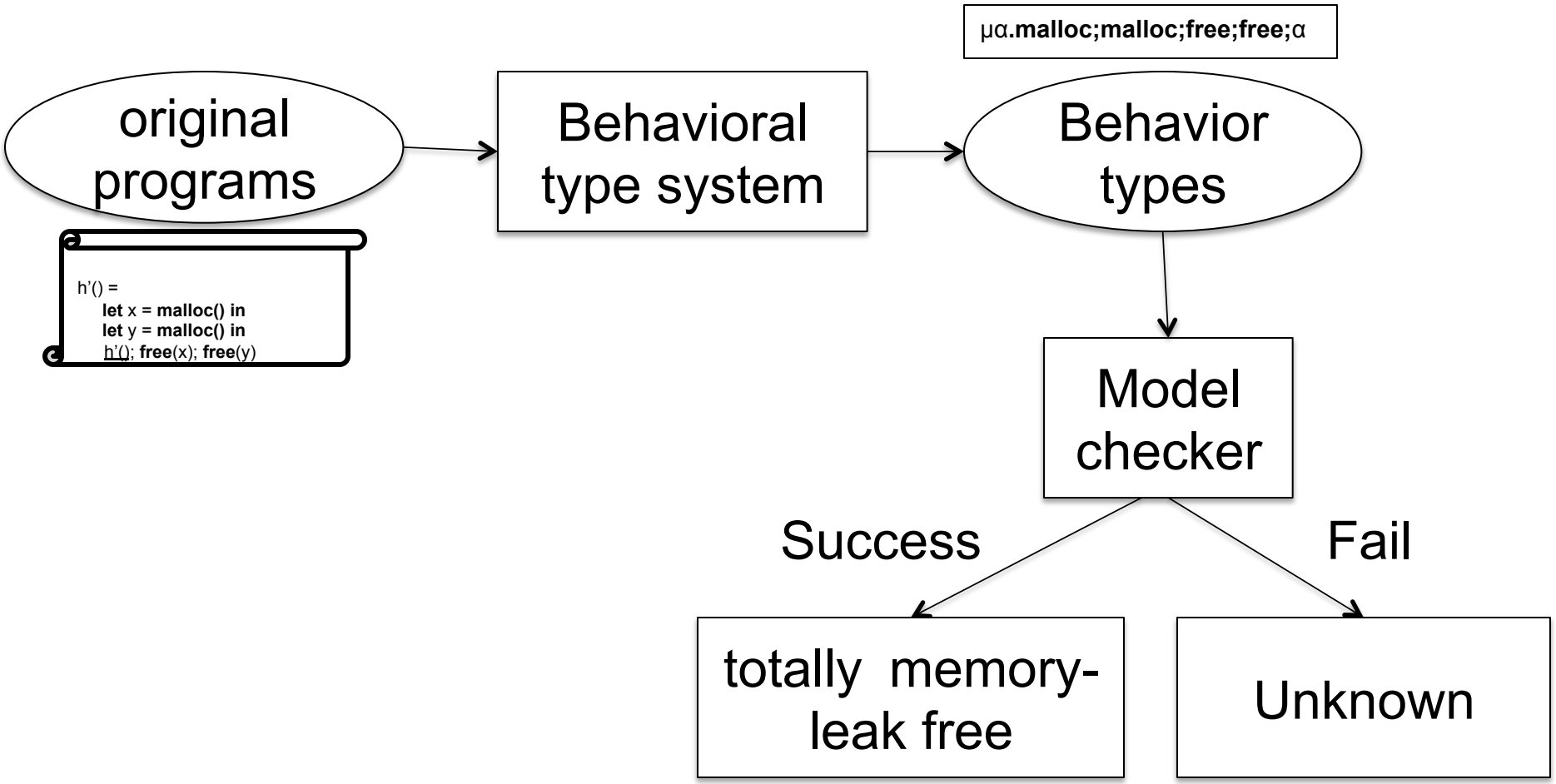
Explanation of the idea



Explanation of the idea



Overview


μα.malloc;malloc;free;free;α
original programs
Behavioral type system
Behavior types
h'() =
let x = malloc() in
let y = malloc() in
h'(); free(x); free(y)
Model checker
Success
Fail
totally memory-leak free
Unknown

Outline

- Language
- Behavioral Type System
- Preliminary Experiments
- Related Work
- Conclusion
- Future Work

Language

x, y, z, ... (variables) $\in$ **Var**

s (statements) ::= **skip** | s_1; s_2
| **let** x = y **in** s | $f(\vec{x})$
| $*x \leftarrow y$ | **let** x = $*$y **in** s
| **let** x = **malloc**() **in** s | **free**(x)
| **ifnull**(x) **then** s_1 **else** s_2
| **let** x = **null** **in** s

d (proc. defs.) ::= { $f \longrightarrow (x_1, ..., x_n)s$}

D (definitions) ::= $\langle d_1 \cup ... \cup d_n \rangle$

P (programs) ::= $\langle D, s \rangle$

Outline

- Language
- Behavioral type system
  - Syntax of behavioral types
  - Type judgment
  - Typing rule for programs
  - $OK_n(P)$
- Preliminary Experiments
- Related Work
- Conclusion
- Future Work

Syntax of behavioral types

- P (behavioral types) ::=

| **0**	do-nothing
| $P_1;P_2$	sequential execution of P_1 and P_2
| P_1+P_2	choice between P_1 and P_2
| **malloc**	allocation of one memory cell
| **free**	deallocation
| α	type variable
| μα.P	recursion

Type judgment

$$\Theta; \Gamma \vdash s : P$$

- Under Θ and Γ, the abstracted behavior of s is *P*
  - Θ (function type environment) ::= $\{f_1:P_1,...,f_n:P_n\}$
  - Γ (variable type environment) ::= $\{x_1,x_2,...,x_n\}$

- For example

  Θ; Γ ⊢ **let** x = **malloc**() **in free**(x) : **malloc**;**free**

Typing rule for programs

During execution, a program will never allocate more than *n* cells

$$\frac{\vdash D : \Theta \quad \Theta; \emptyset \vdash s : P \quad OK_n(P)}{\vdash \langle D, s \rangle : n}$$

- $\vdash \langle D, s \rangle : n$, a program requires at most *n* memory cells when it is executed
- *P* represents the behavioral type of main statement s

In order to guarantee $\vdash \langle D, s \rangle : n$, we use condition $OK_n(P)$

$OK_n(P)$

σ represents a sequence of actions **malloc**, **free**, and other actions τ

The number of **malloc** in σ

The number of **free** in σ

- **Definition:**

$OK_n(P)$ holds if, for any P', if $P \xrightarrow{\sigma} P'$ then $\#_{\mathbf{malloc}}(\sigma) - \#_{\mathbf{free}}(\sigma) \leq n$

- Intuitively, at every running step, the number of memory cells a program consumes never exceeds the number of cells it requires.

- For example

  P_1 = μα.**malloc;malloc;free;free;**α

  $OK_2(P_1)$ holds, that is, at most two memory cells are consumed

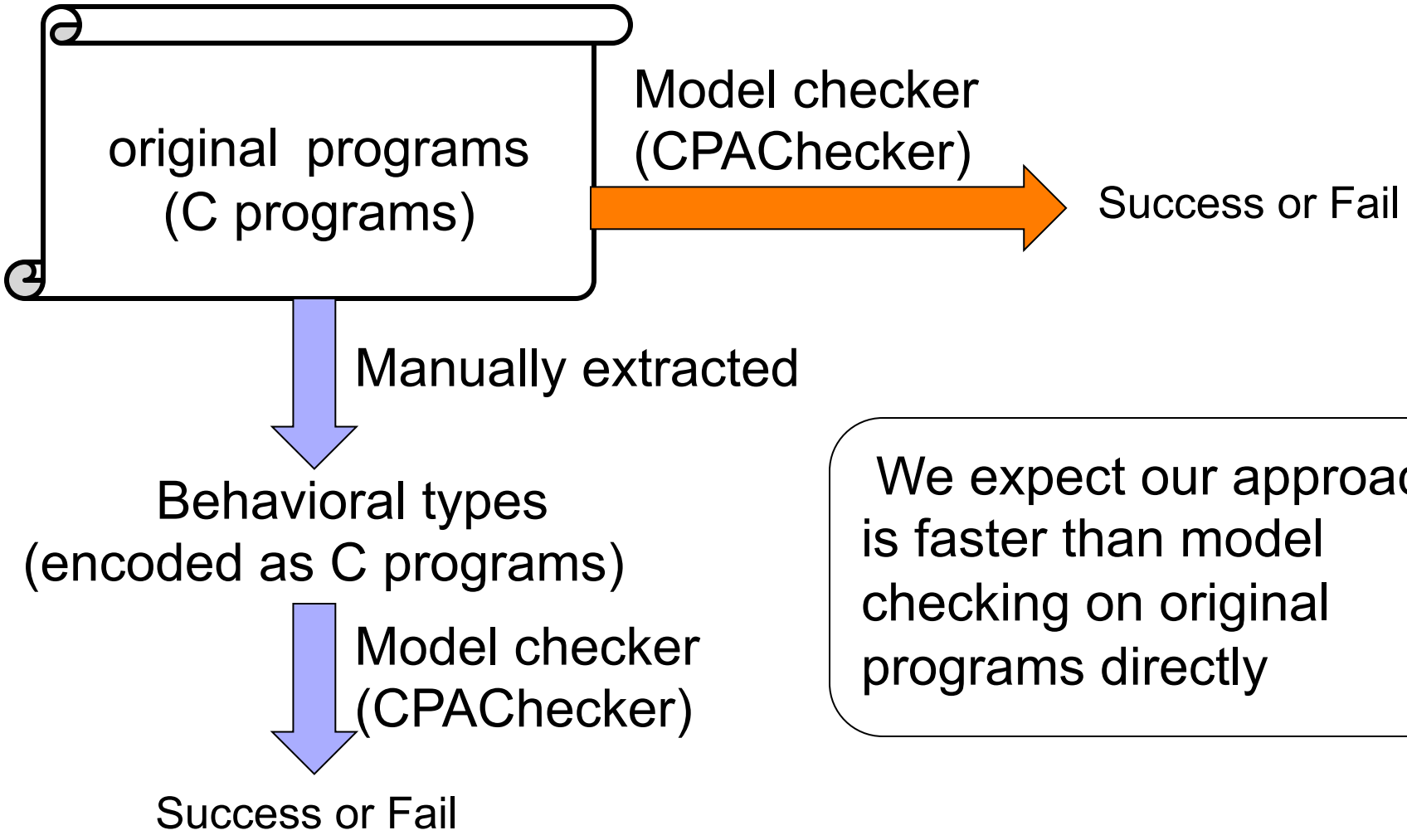
Outline

- Language
- Behavioral Type System
- Preliminary Experiments
  - Objective
  - Comparison
  - Discussion
- Related Work
- Conclusion
- Future Work

Objective

- Checking whether our approach can verify total memory-leak freedom
- Investigating the problems in our current type system

Two ways to verify total memory-leak freedom



We expect our approach is faster than model checking on original programs directly

Comparison

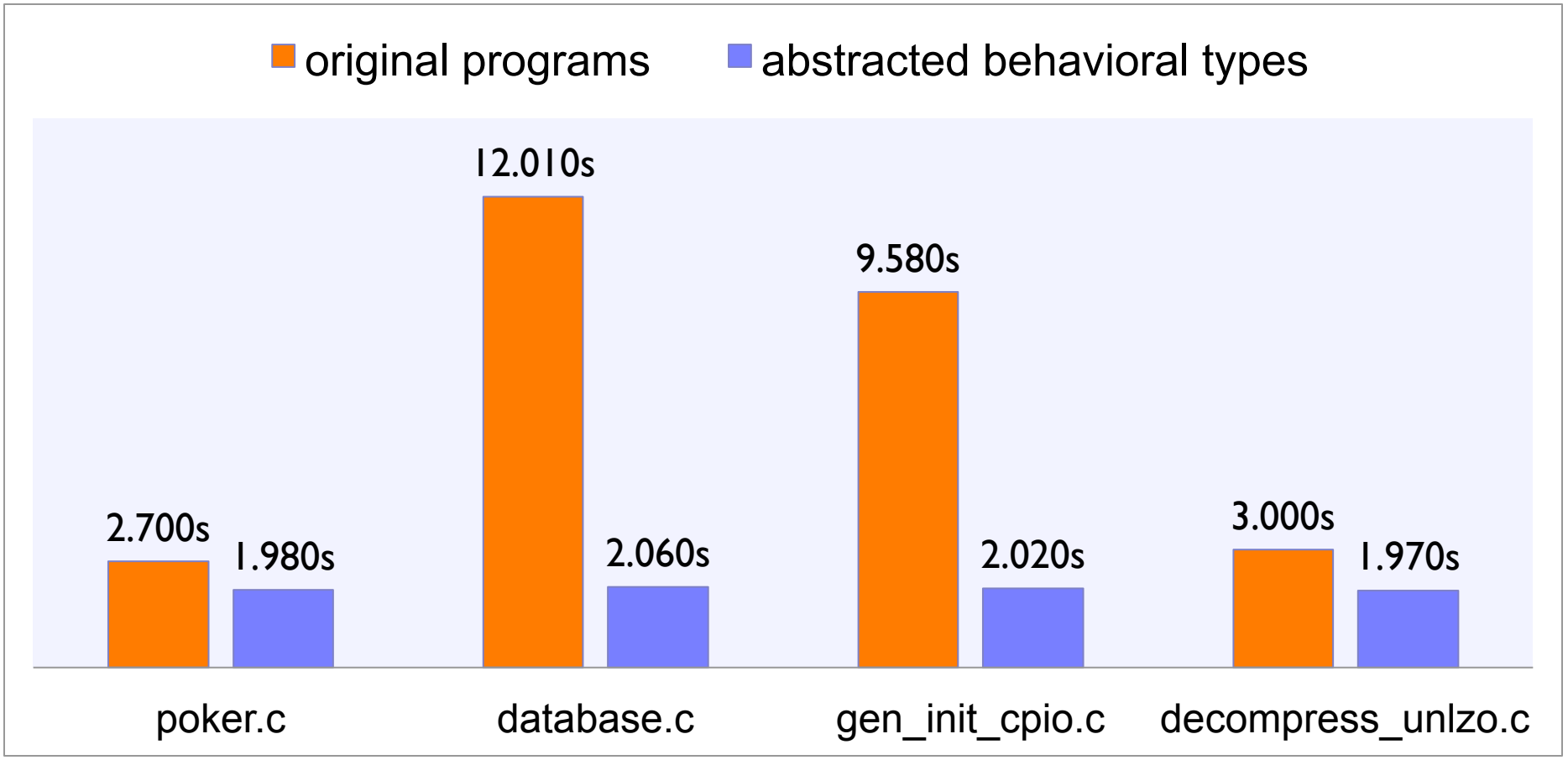


Table 1. Time spent by CPAChecker

Problem: Information in behavioral types is not enough to verify total memory-leak freedom!

	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

Table 2. Result of verification of model checking on original programs and abstracted behavior.

Discussion

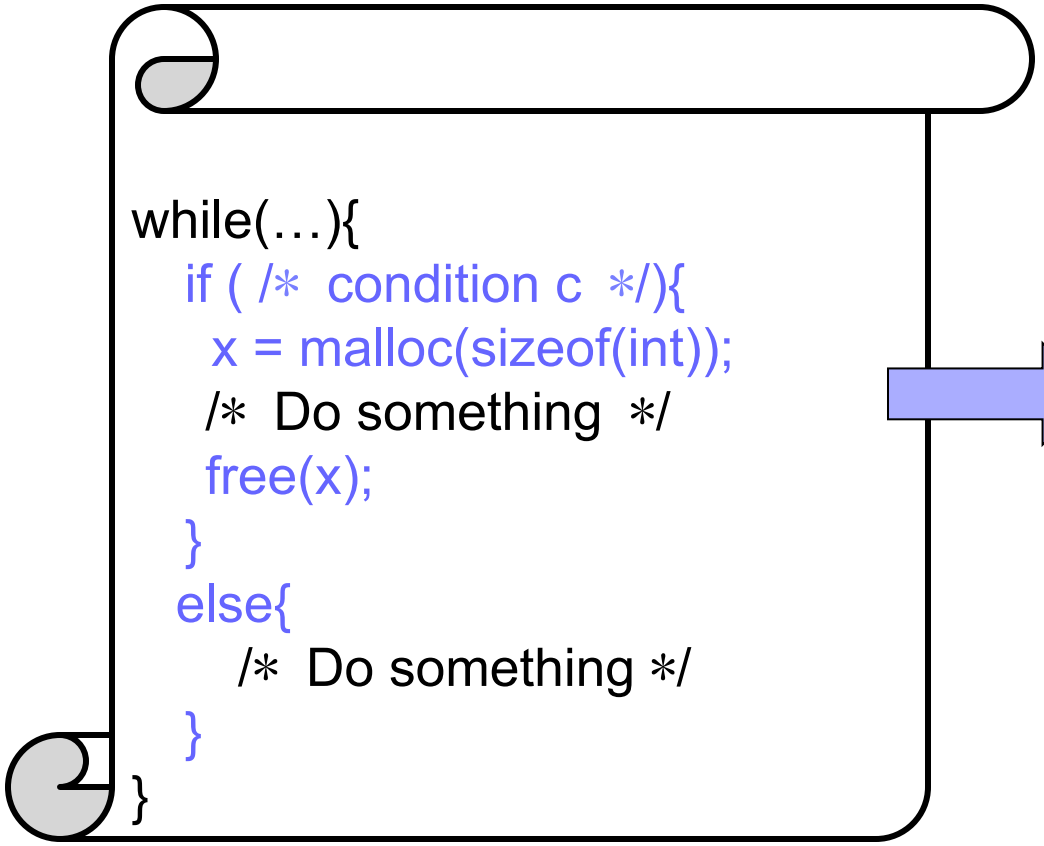
- Verification failed, because our type system is not path-sensitive

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

μα.**malloc**;**0**;α

μα.(**0** + **malloc**);(**0** + **free**);α

Discussion



P : μα.((**malloc**;**free**) + **0**);α

$OK_1(P)$ holds

We confirmed that CPAChecker can verify $OK_n(P)$ for the abstracted behaviroal type of the rewritten programs without much penalty on CPU time

Outline

- Language
- Behavioral type system
- Preliminary Experiments
- Related work
- Conclusion
- Future work

Related work

- Static memory-leak freedom verification [Heine&Lam PLDI'03], [Suenaga&Kobayashi APLAS'09], etc
  - Partial memory-leak freedom
  - Lack of illegal accesses

- Behavioral types are heavily used in concurrent programs [Kobayashi and Suenaga&Wischik LMCS'06], etc
  - Our type system is inspired by one proposed by Kobayashi et al.

Conclusion

- Verification of memory-leak freedom for (possibly) nonterminating programs
- A behavioral type system which abstracts the behavior of programs with allocation and deallocation
- Preliminary experiments
  - Applying CPAChecker on abstracted behavioral types

Future work

- Extension with variable-sized cells
- Improving our type system
  - to make the verification process automatic
  - to verify programs more precisely