An Extended Behavioral Type System for Memory-Leak Freedom

Qi Tan, Kohei Suenaga, and Atsushi Igarashi
Kyoto University

Introduction

- Memory leak, forgetting to deallocate an allocated memory cell, is a serious problem
  - Decreasing the performance of computer
  - Unexpected termination of an application
  - System crash

- Verification of *memory-leak freedom* is important

Partial memory-leak freedom

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

Consumes unbounded number of memory cells

```
f() =
    let x = malloc() in
   /* do something */
    free(x)
```

```
f'() =
    let x = malloc() in
    f'(); free(x)
```

Total memory-leak freedom

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

h() =
let x = **malloc() in**
let y = **malloc() in**
free(x); **free**(y); h()

h'() =
let x **= malloc() in**
let y = **malloc() in**
h'();**free**(x);**free**(y)

Both are partially memory-leak free.

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

Goal

- Verification of total memory-leak freedom
  - Important for nonterminating software such as Web servers and OS

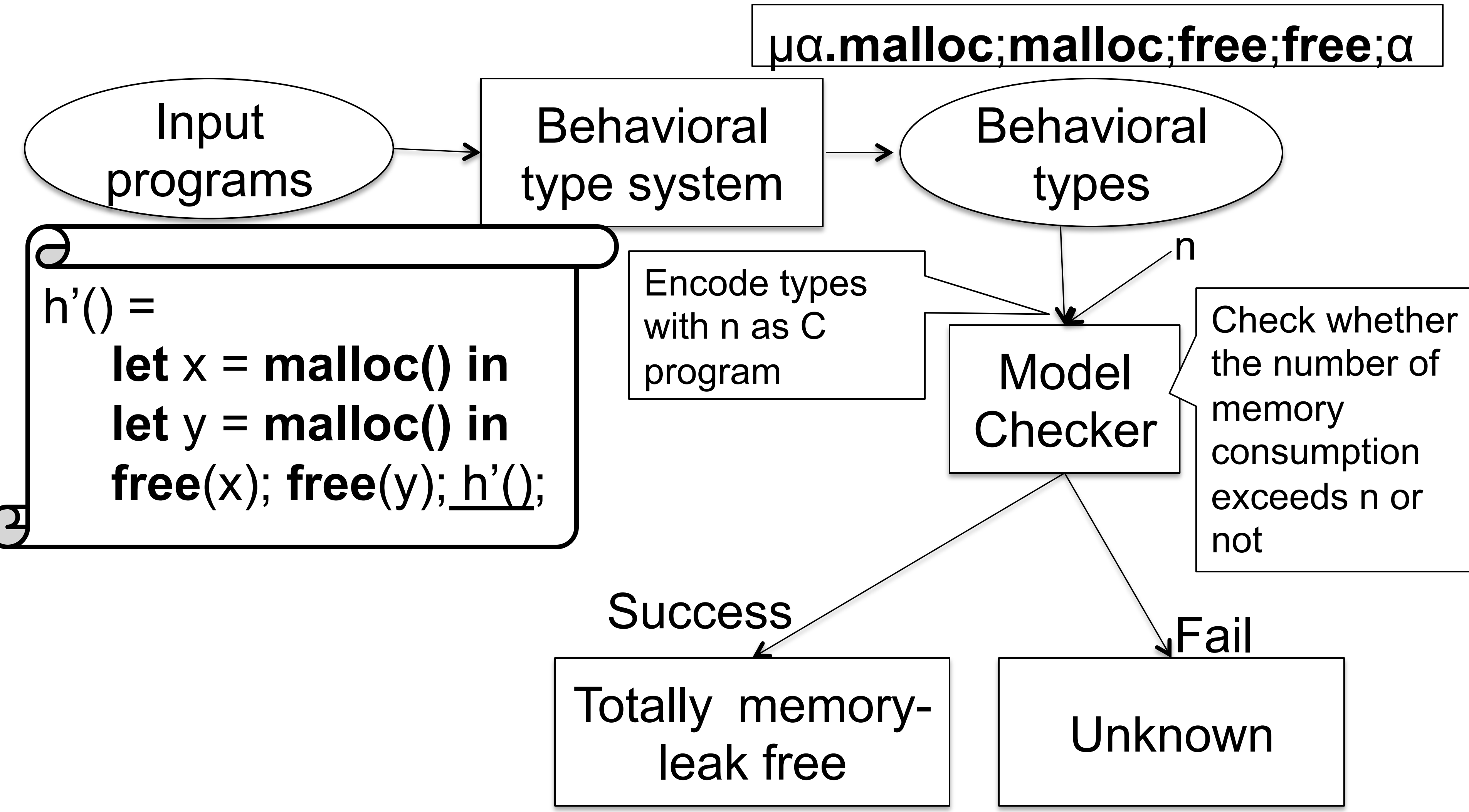
h() =
let x = **malloc() in**
let y = **malloc() in**
free(x);**free**(y);<u>h()</u>

h'() =
let x **= malloc() in**
let y = **malloc() in**
<u>h'()</u>; **free**(x); **free**(y)

Previous work
[Tan&Suenaga&Igarashi PPL2015]

- *Behavioral type system*
  - Type system for abstracting the behavior of a program
  - Sequential processes are used as types
  - Types describe the number and the order of allocations, deallocations, and recursive calls
- Estimating an upper bound of memory cells consumed by a program based on an inferred type

Overview of previous work



Problem in previous work

Imprecise abstraction due to branches

ifnull (*x) then s_1 else s_2: Executes s_1 if *x is null, s_2 otherwise

```
h1(x) =
  let y = malloc() in
    ifnull(*x) then let x1 = malloc() in *y ←x1 else skip;
    ifnull(*x) then free(*y) else skip;
  free(y);
```

previous behavioral type system

malloc;(malloc+0);(free+0); free

malloc+0: Executing malloc *at most* once

Problem in previous work

Although function h1 is a memory-leak free program, we cannot conclude it from this type

malloc;malloc;0;free

malloc;(malloc+0);(free+0); free

This type loses information about if-guard part

Our solution

Introducing *path-sensitive behavioral types* to deal with branch statements

- $(*x)(P_1, P_2)$ means execution of P_1 or P_2 depending on $*x$ is null or not

h1(x) =
  let y = **malloc**() **in**
    ifnull(∗x) **then let** x1 = **malloc**() **in** ∗y ← x1 **else skip**;
    ifnull(∗x) **then free**(∗y) **else skip**;
  free(y);

path-sensitive behavioral type system

malloc;(∗x)(**malloc**, **0**);(∗x)(**free**, **0**);**free**

Our solution

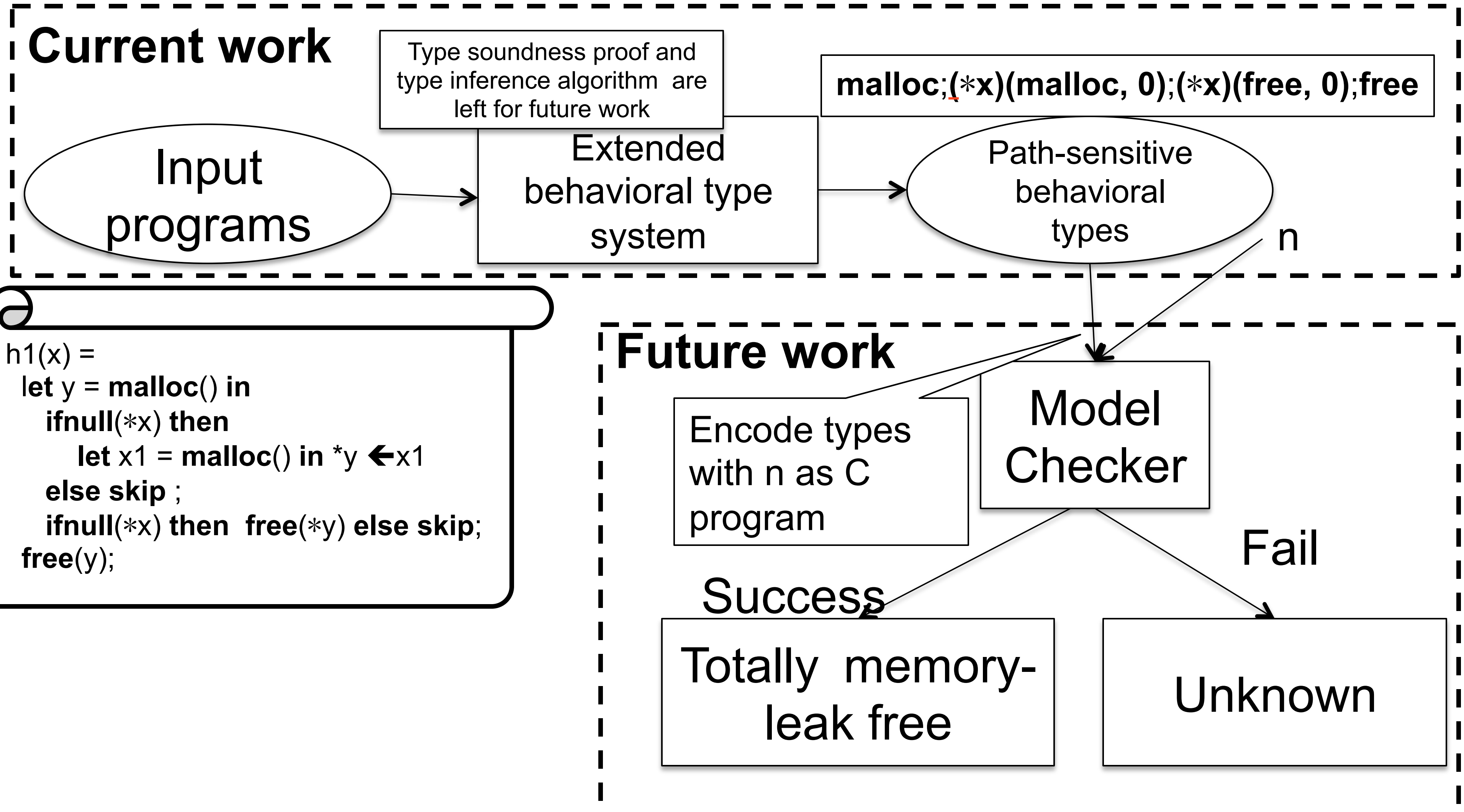
We can conclude h1 is a memory-leak free program from this type

if $*$x is a null pointer, behaves like **malloc**;**malloc**;**free**;**free** otherwise like **malloc**;**0**;**0**;**free**

malloc;($*$x)(**malloc**, **0**);($*$x)(**free**, **0**);**free**

Requires $*$x not to change between two branches

Overview of current work



Outline

- Language
- Extended behavioral type system
- Overview of verification
- Related work
- Conclusion
- Future work

Language

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

s (statements) ::= **skip** | $s_1 ; s_2$ | $*x \leftarrow y$ | $f(\vec{x})$

| **let** x = y **in** s

| **let** x = $*$y **in** s

| **let** x = **null in** s

| **let** x = **malloc**() **in** s

| **free**(x) | **const**($*$x)s

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

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

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

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

Outline

- Language
- Extended behavioral type system
  - Syntax of types
  - Type judgment
  - Typing rule for a program
  - $OK_n(P)$
- Overview of verification
- Related Work
- Conclusion
- Future Work

Syntax of extended behavioral types

- P (behavioral types) ::=

| **0**	do-nothing
| $P_1;P_2$	sequential execution of P_1 and P_2
| **malloc**	allocation of one memory cell
| **free**	deallocation
| $\underline{(*x)(P_1, P_2)}$	execution of P_1 or P_2 depends on $*x$
| $\underline{\mathrm{const}(*x)P}$	execution of P under constantness $(*x)$
| α	type variable
| $\mu\alpha.P$	recursion

Type judgment

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

- Under Θ and Γ, the extracted behavior of s is *P*
  - Θ (function type environment) ::= $\{f_1{:}\psi_1,\ldots,f_n{:}\psi_n\}$
  - Γ (variable type environment) ::= $\{x_1,x_2,\ldots,x_n\}$
  - ψ (dependent function type) ::= $(\vec{x})P$

Example

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

$$\frac{\Theta; \Gamma, x \vdash \textbf{let } x1 = \textbf{malloc() in } *x \leftarrow x1 \textbf{ : malloc;0} \qquad \Theta; \Gamma \vdash \textbf{skip : 0}}{\Theta; \Gamma, x, y \vdash \textbf{ifnull}(*y) \textbf{ then let } x1 = \textbf{malloc() in } *x \leftarrow x1 \textbf{ else skip: }(*y)\textbf{(malloc,0)}}$$

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

Typing rule for a program

$$\frac{\vdash D : \Theta \qquad \Theta; \emptyset \vdash s : P}{\vdash \langle D, s \rangle : P}$$

- *P* represents the behavioral type of main statement s
- In order to guarantee $\langle D, s \rangle$ is totally memory-leak free, we use the predicate $OK_n(P)$

$OK_n(P)$

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

relation $\longrightarrow$ means *P* can perform actions σ and turn into *P'*

- **Definition:**

  $OK_n(P)$ holds, if $P \xrightarrow{\sigma} P'$ implies

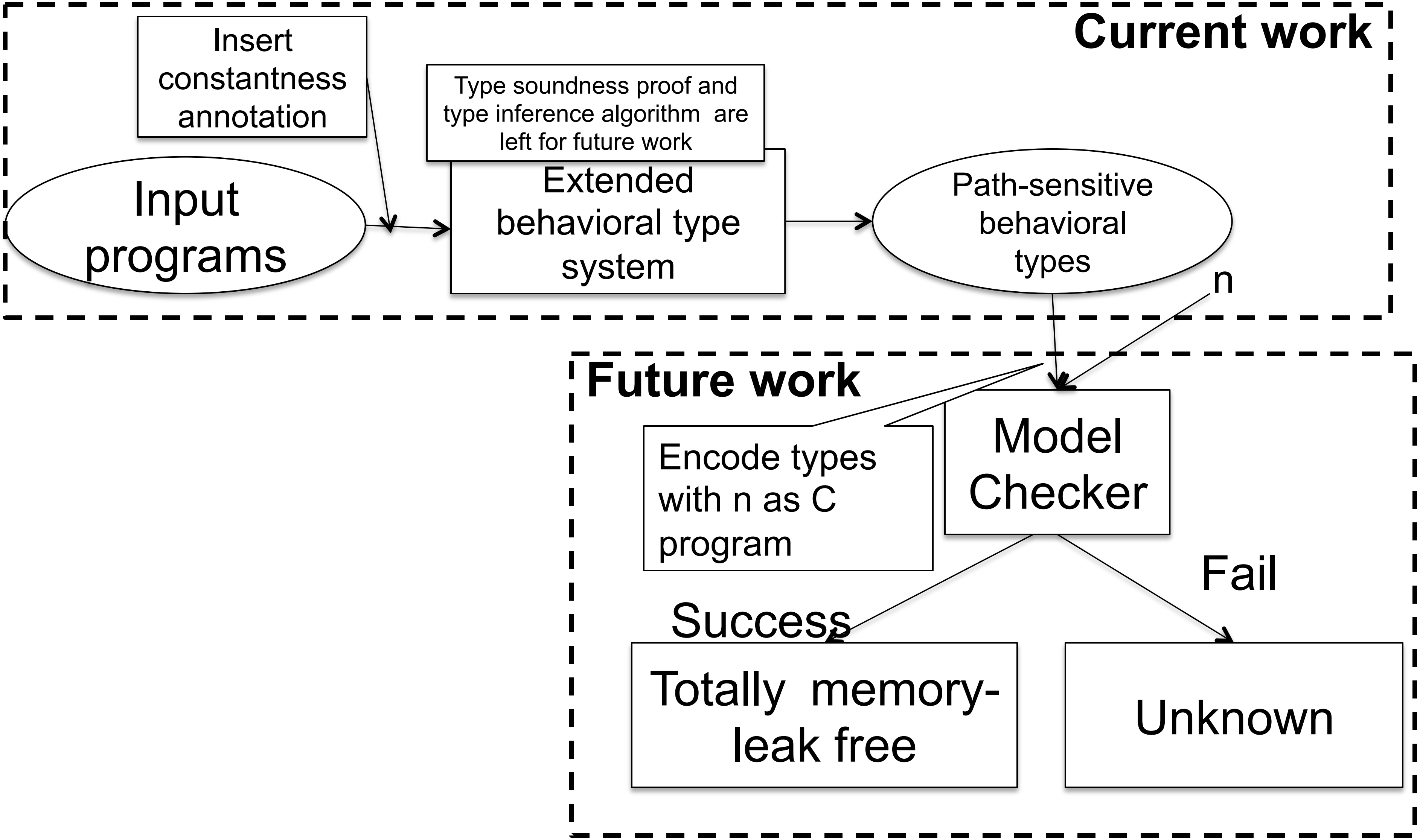
  $\#_{\mathbf{malloc}}(\sigma) - \#_{\mathbf{free}}(\sigma) \leq n$ where $\#_\rho(\sigma)$ is the number of ρ in σ.

- Intuitively, at every running step, the number of memory cells a program consumes cannot exceed the number of cells it requires.

Outline

- Language
- Extended behavioral type system
- Overview of verification
- Related work
- Conclusion
- Future work

Overview of current work



overview of verification

```
let x = malloc() in
let y = malloc() in
    ifnull(∗y) then let x1 = malloc() in ∗x ← x1 else skip;
      /* do-something but not change the value of *y */
    ifnull(∗y) then free(∗x) else skip
free(x); free(y)
```

Manually inserted by a programmer

```
let x = malloc() in
let y = malloc() in
  const(∗y)    /* inserted by a programer */
    (
      ifnull(∗y) then let x1 = malloc() in ∗x ← x1 else skip;
        /* do-something but not change the value of *y */
      ifnull(∗y) then free(∗x) else skip
    );
free(x); free(y)
```

Future work: automated insertion

Overview of verification

```
let x = malloc() in
let y = malloc() in
  const(∗y)
    (
      ifnull(∗y) then let x1 = malloc() in ∗x ← x1 else skip;
        /* do-something but not change the value of ∗y */
      ifnull(∗y) then free(∗x) else skip
    );
free(x); free(y)
```

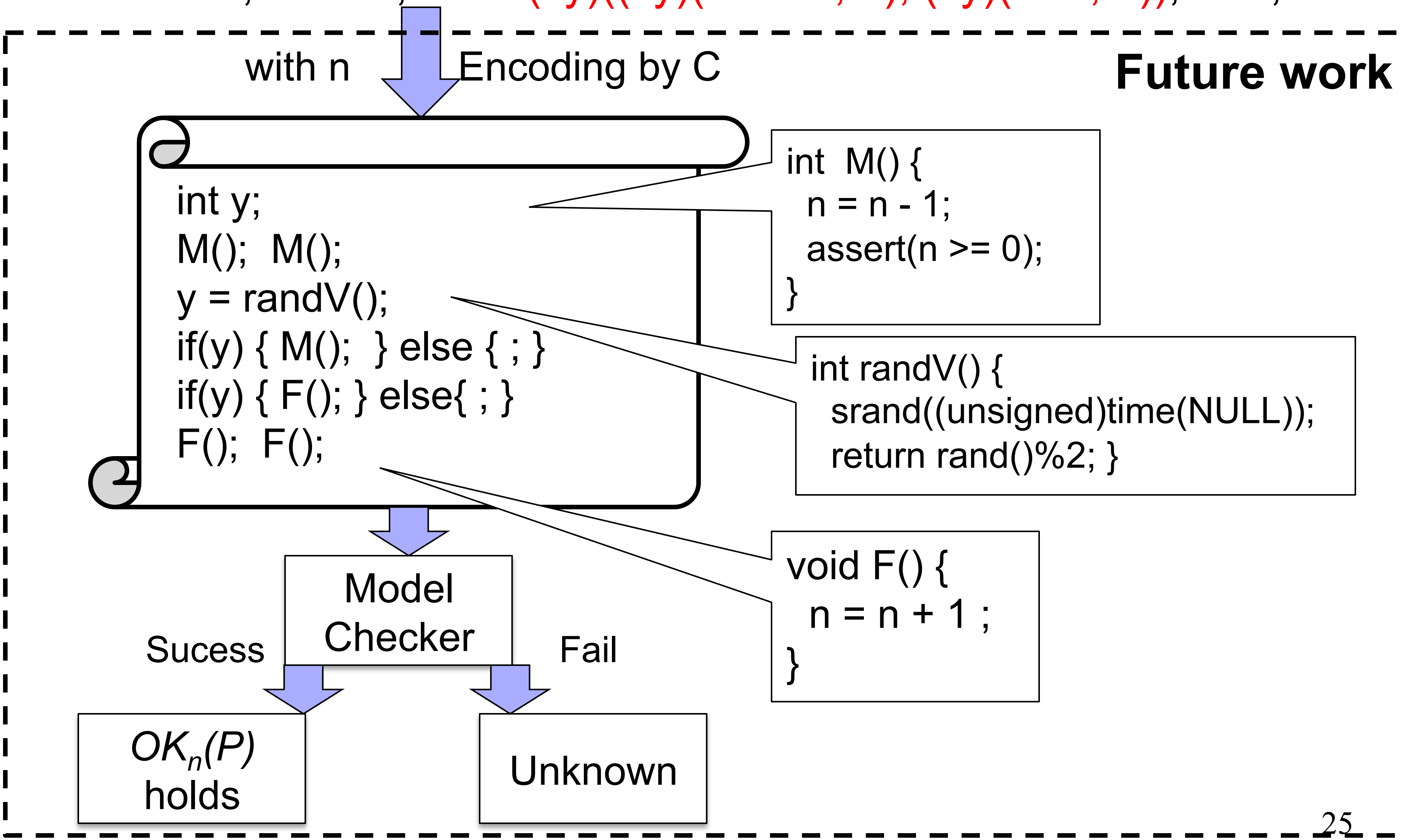
Type inference of our type system

P = malloc; malloc; const(∗y)((∗y)(malloc, 0); (∗y)(free, 0)); free; free

automatically abstracted by our current type system

Overview of verification

P = malloc; malloc; const(∗y)((∗y)(malloc, 0); (∗y)(free, 0)); free; free



Outline

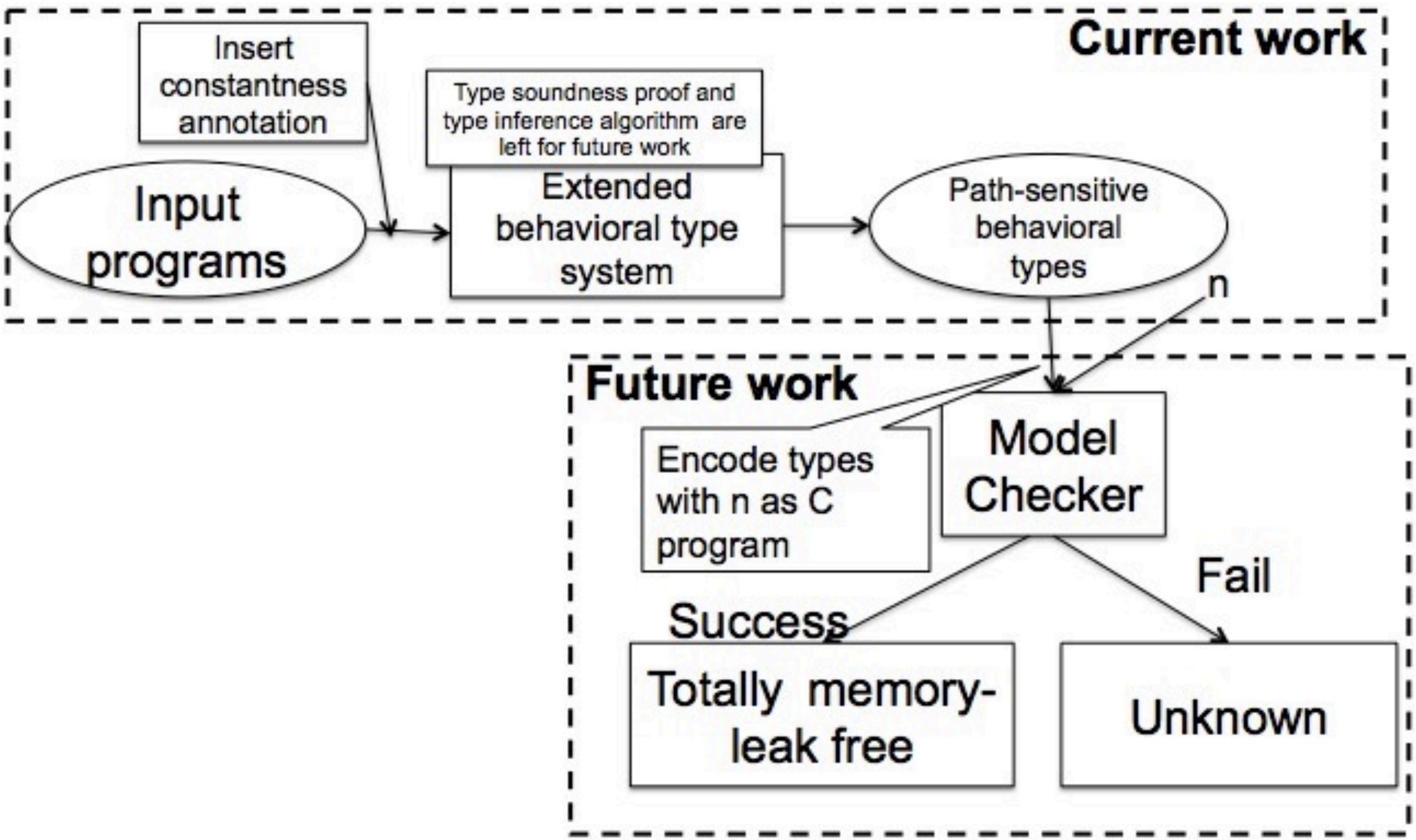
- Language
- Extended behavioral type system
- Overview of verification
- Related work
- Conclusion
- Future work

Related work

- Behavioral type system are widely used for
  - Static verification of resource-usage [Igarashi&Kobayashi ACMTPLS'05],etc
  - Static verification of deadlock-freedom [Kobayashi Acta inf'05]
- Static memory-leak freedom verification [Heine&Lam PLDI'03], [Suenaga&Kobayashi APLAS'09], etc
  - Partial memory-leak freedom
  - Lack of illegal accesses
- Our previous behavioral type system[Tan&Suenaga&Igarashi PPL2015]
  - Total memory-leak freedom

Conclusion

- Path-sensitive behavioral type: describes more information of the behavior of a program



Future work

- Type inference algorithm for our extended type system
- Experiments
- Automated insertion of constantness annotation