Safe Memory-Leak Fixing for C Programs

Qing Gao, Yingfei Xiong, Yaqing Mi,
Lu Zhang, Weikun Yang, Zhaoping Zhou

Peking University

2014.12.13

报告人介绍

- 熊英飞，北京大学“百人计划”助理教授
- 2009年于日本东京大学获得博士学位
  - 导师：胡振江、武市正人
- 2009-2011年在加拿大滑铁卢大学从事博士后研究
  - 导师：Krzysztof Czarnecki
- 研究领域：程序分析和编程语言设计
- 承担关于安全攸关软件质量、云软件质量、缺陷自动修复等青年973、自然科学基金项目

Memory-Leak: An Example

```c
#include <stdlib.h>
#include <stdio.h>

void f(int *p, int **q){
  *q = p;
}
void g(int *p){
  free(p);
}
int h(int size, int num, int sum){
  int *p = (int*)malloc(sizeof(int)*size);
  int **q = (int**)malloc(sizeof(int*));
  if (size == 0)
    g(p);
  else
    for (int i = 0; i < size; ++i)
      if (p[i] != num){
        f(p, q);
        sum += (*q)[i];
      }
      else
        return i;
  printf("%d", sum);
  return sum;
}
```

Allocation

Free

Use

Leaked

Leaked

Memory-Leak: An Example

```c
#include <stdlib.h>
#include <stdio.h>

void f(int *p, int **q){
  *q = p;
}
void g(int *p){
  free(p);
}
int h(int size, int num, int sum){
  int *p = (int*)malloc(sizeof(int)*size);
  int **q = (int**)malloc(sizeof(int*));
  if (size == 0)
    g(p);
  else
    for (int i = 0; i < size; ++i)
      if (p[i] != num){
        f(p, q);
        sum += (*q)[i];
      }
      else
        return i;
  printf("%d", sum);
  return sum;
}
```

free(q);

free(p);
free(q);

free(p);
free(q);

Ensuring Safety

- Allocation
- Reference
- No double free
- No use after free

Allocation

Free

Use

free(p);
free(q);

```c
#include <stdlib.h>
#include <stdio.h>

void f(int *p, int **q){
  *q = p;
}
void g(int *p){
  free(p);
}
int h(int size, int num, int sum){
  int *p = (int*)malloc(sizeof(int)*size);
  int **q = (int**)malloc(sizeof(int*));
  if (size == 0)
    g(p);
  else
    for (int i = 0; i < size; ++i)
      if (p[i] != num){
        f(p, q);
        sum += (*q)[i];
      }
      else
        return i;
  printf("%d", sum);
  return sum;
}
```

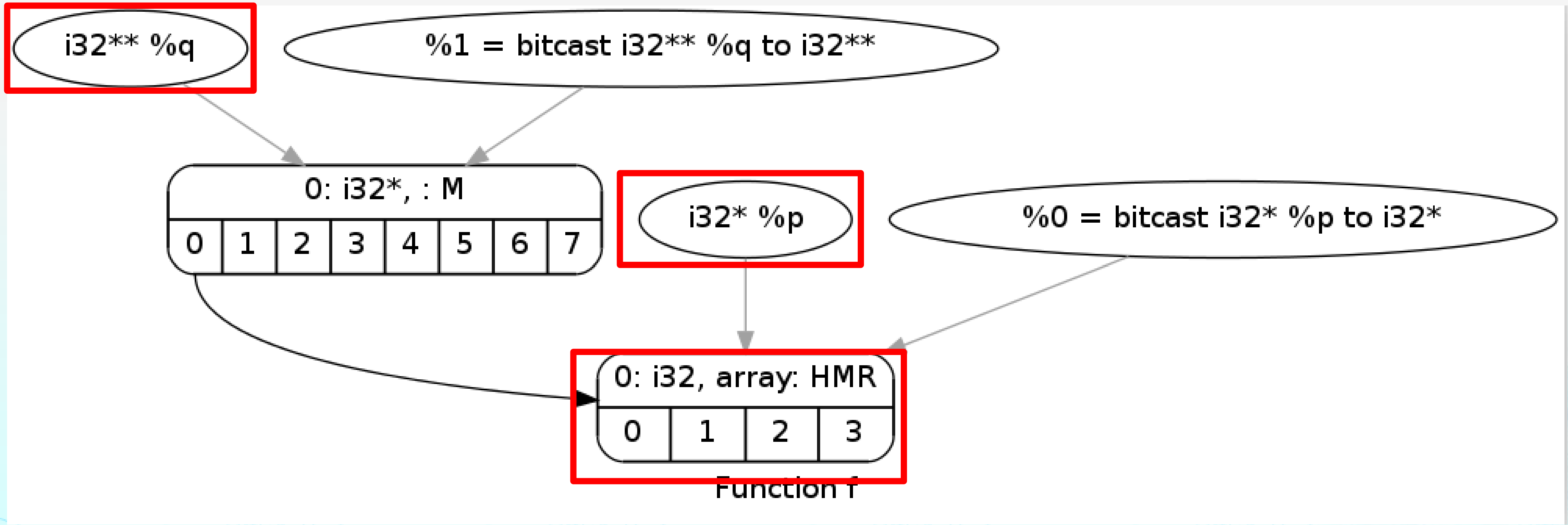
Approach

- Pointer Analysis
- Building Procedural Summaries
- Intraprocedural Analysis

Approach - Pointer Analysis

- Existing pointer analysis algorithms: DSA

```
void f(int *p, int **q){
  *q = p;
}
```

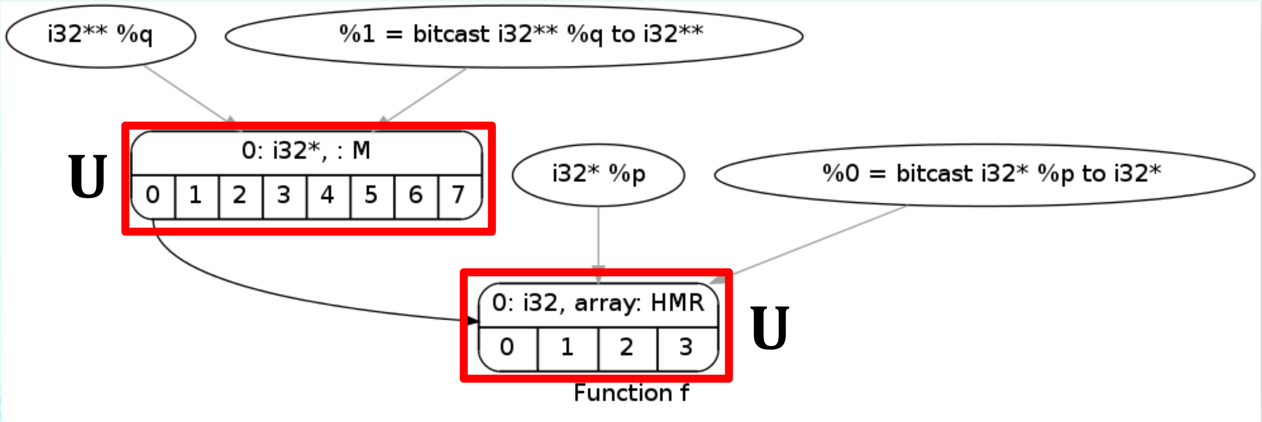


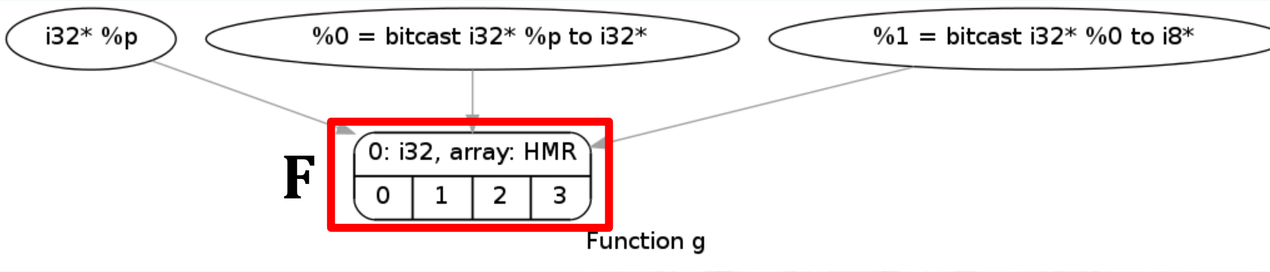
Approach - Building Procedure Summaries

- Procedure type
  - Allocation (A)
  - Free (F)
  - Use (U)
- Iteratively propagate summaries on call graph

```c
void f(int *p, int **q){
  *q = p;
}
```

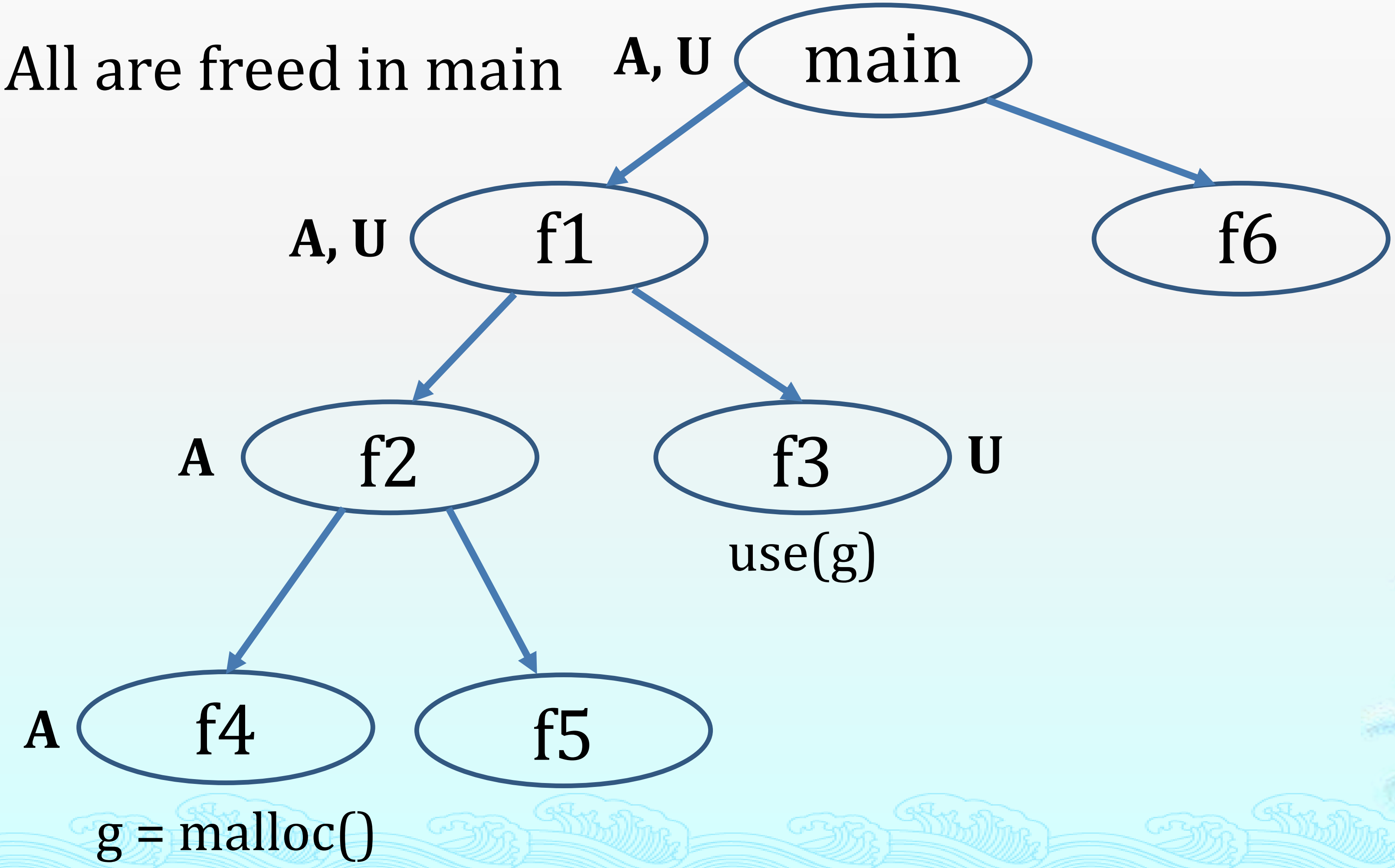
```c
void g(int *p){
  free(p);
}
```





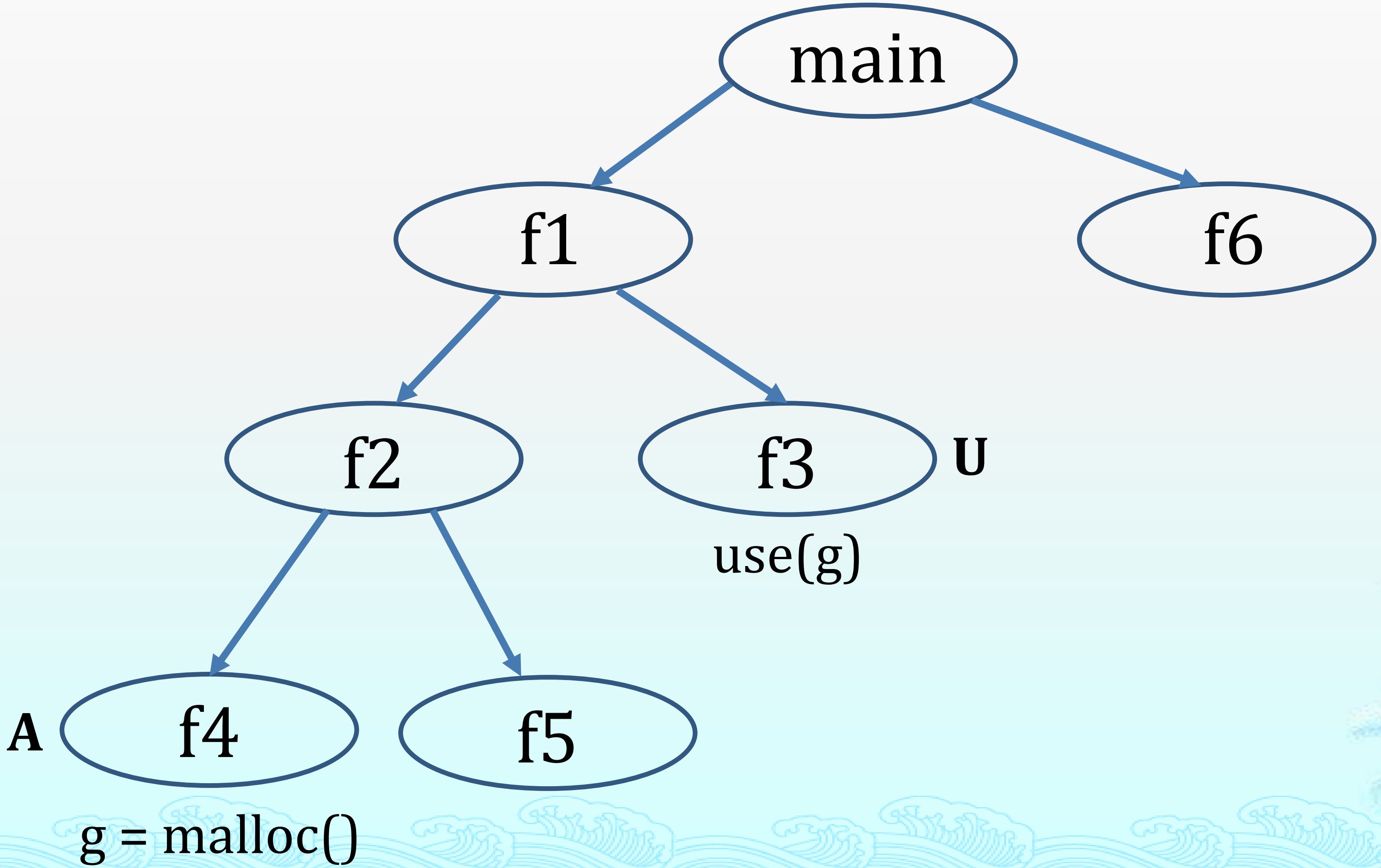
Approach - Building Procedure Summaries

- Global variables
  - All are freed in main



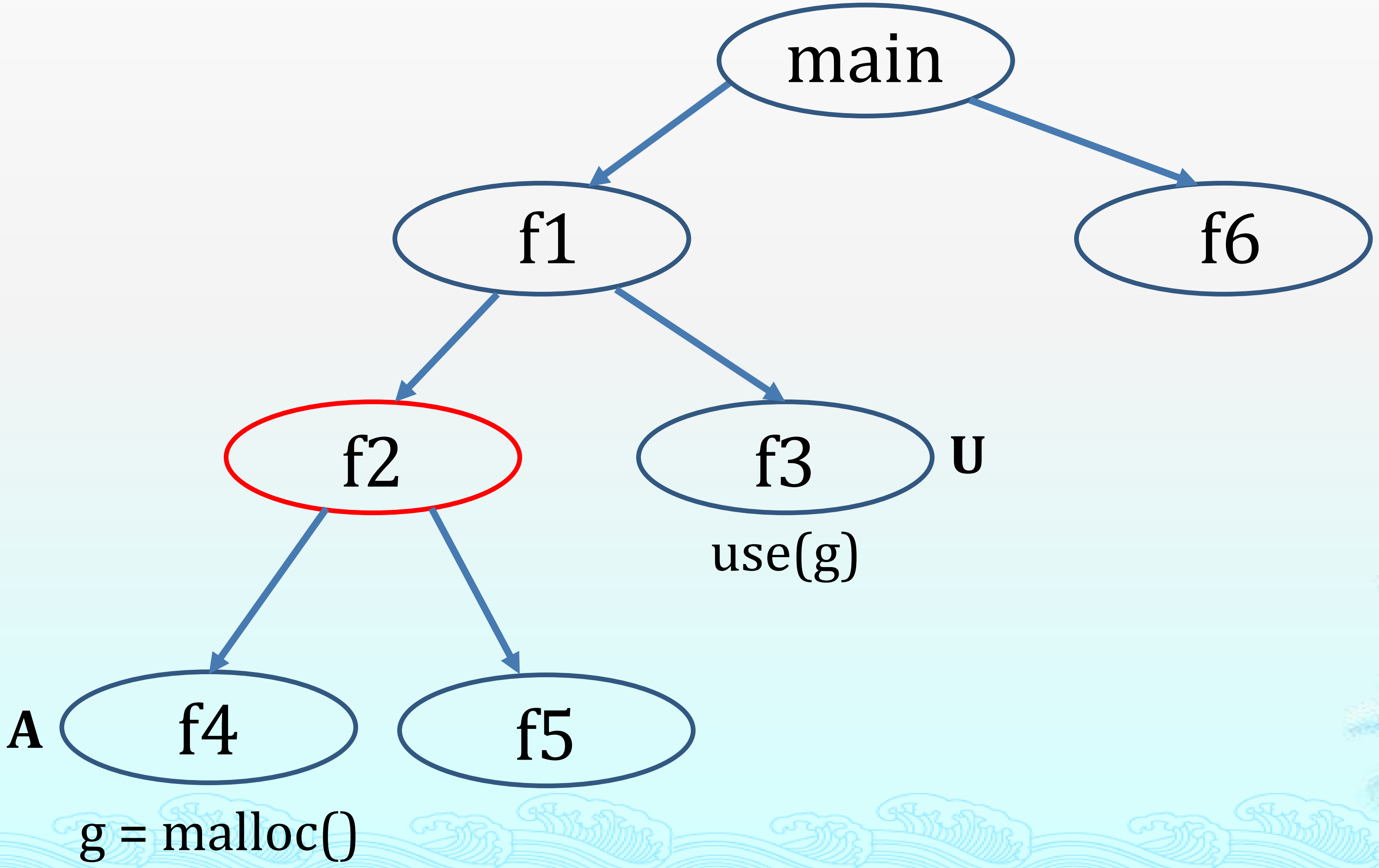
Approach - Building Procedure Summaries

- Global variables



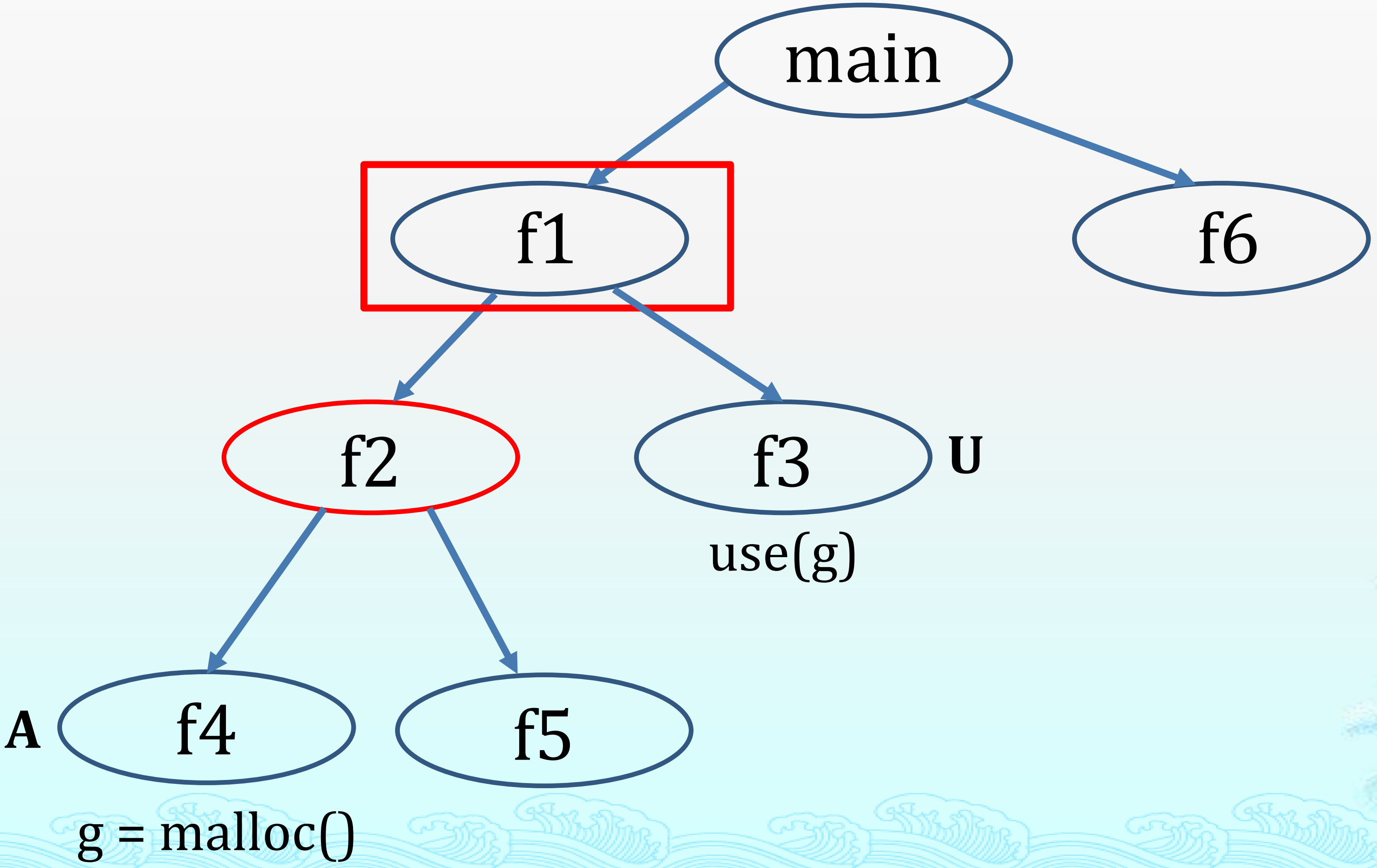
Approach - Building Procedure Summaries

- Global variables



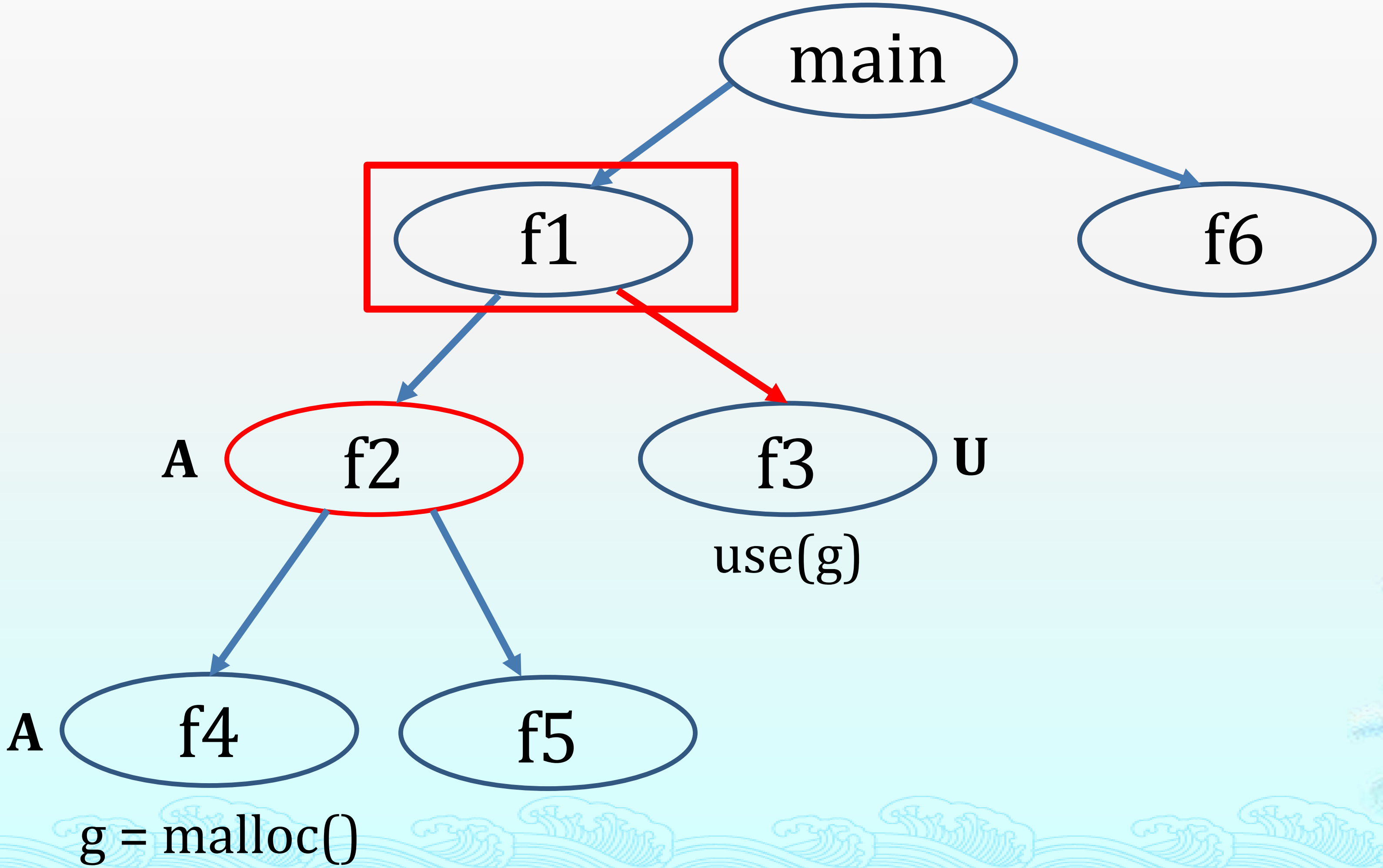
Approach - Building Procedure Summaries

- Global variables



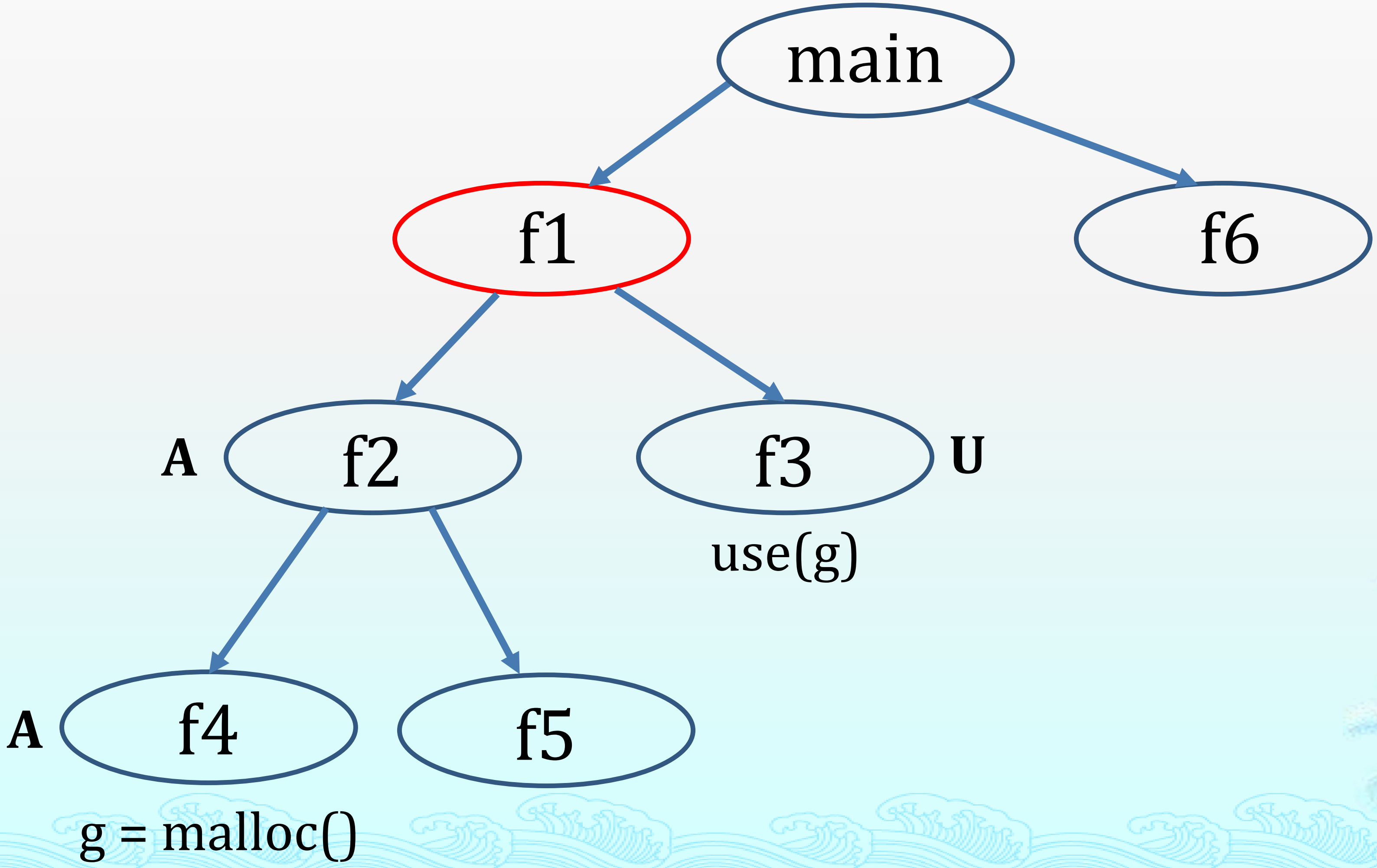
Approach - Building Procedure Summaries

- Global variables



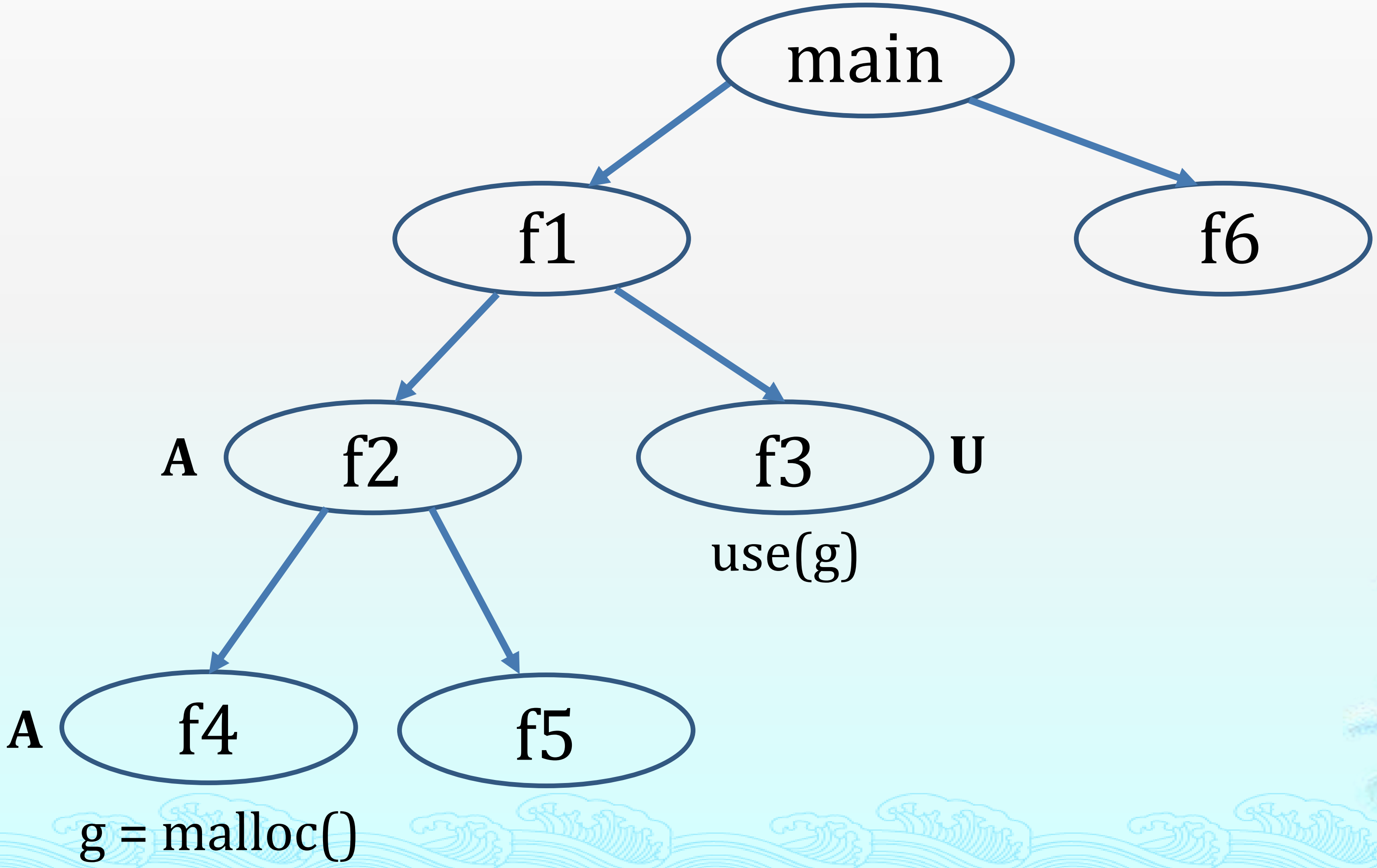
Approach - Building Procedure Summaries

- Global variables



Approach - Building Procedure Summaries

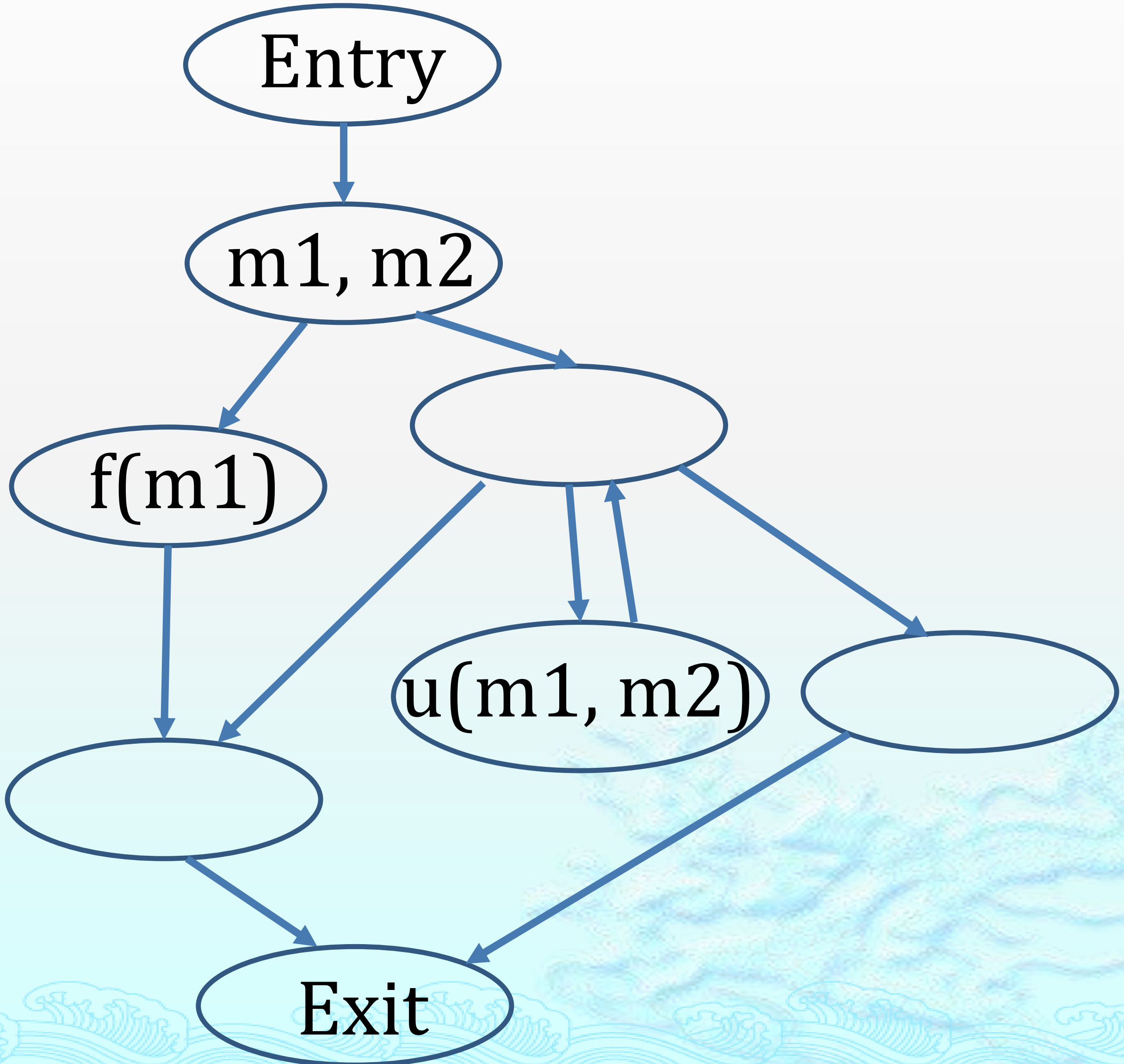
- Global variables



Approach - Intra-procedural Analysis

- Data-flow analysis
- Four passes on CFG

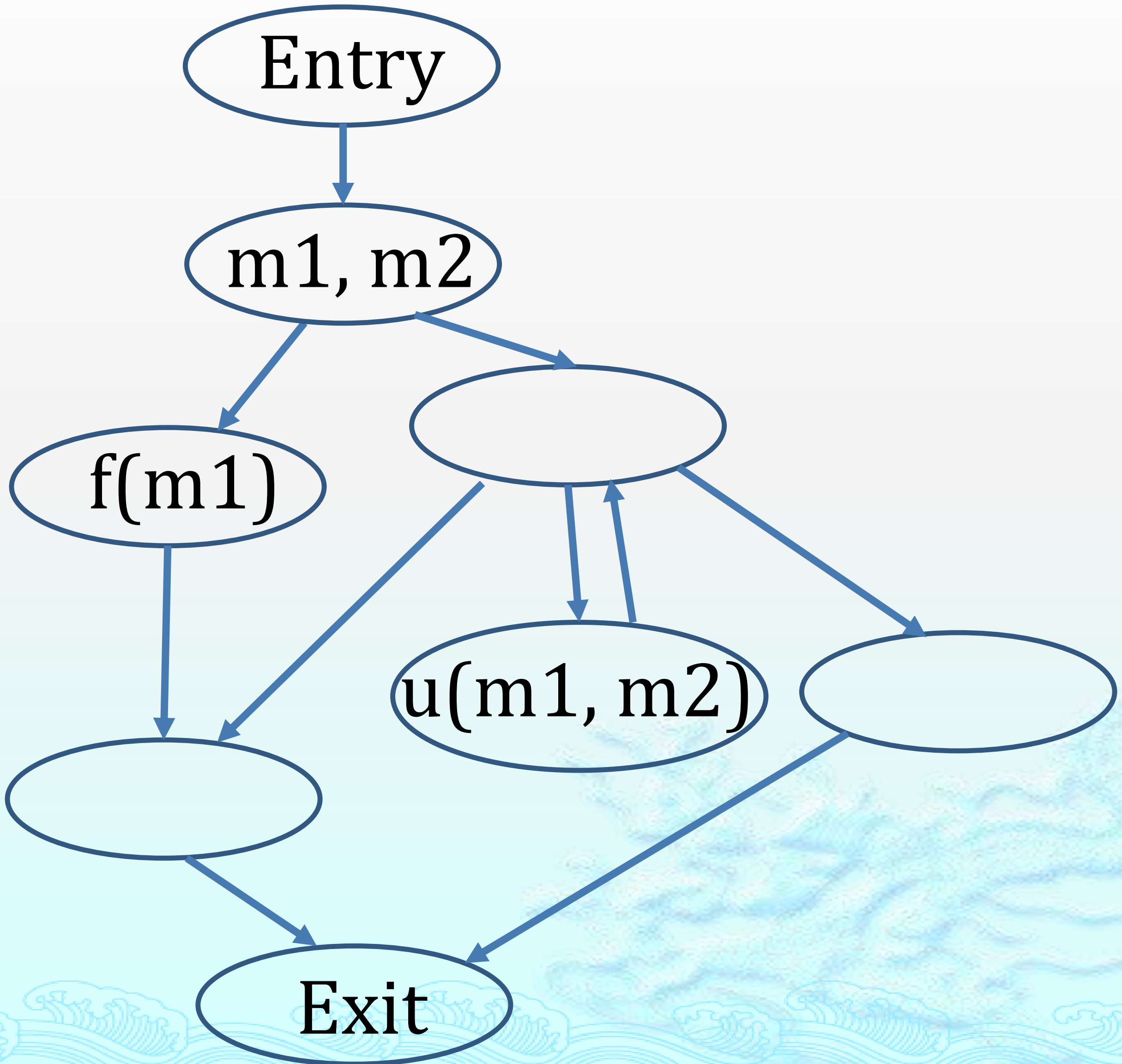
```
int h(int size, int num, int sum){
  int *p = (int*)malloc(sizeof(int)*size);
  int **q = (int**)malloc(sizeof(int*));
  if (size == 0)
    g(p);
  else
    for (int i = 0; i < size; ++i)
      if (p[i] != num){
        f(p, q);
        sum += (*q)[i];
      }
      else
        return i;
  printf("%d", sum);
  return sum;
}
```



Approach - Intra-procedural Analysis

- 1st pass
- Forward analysis

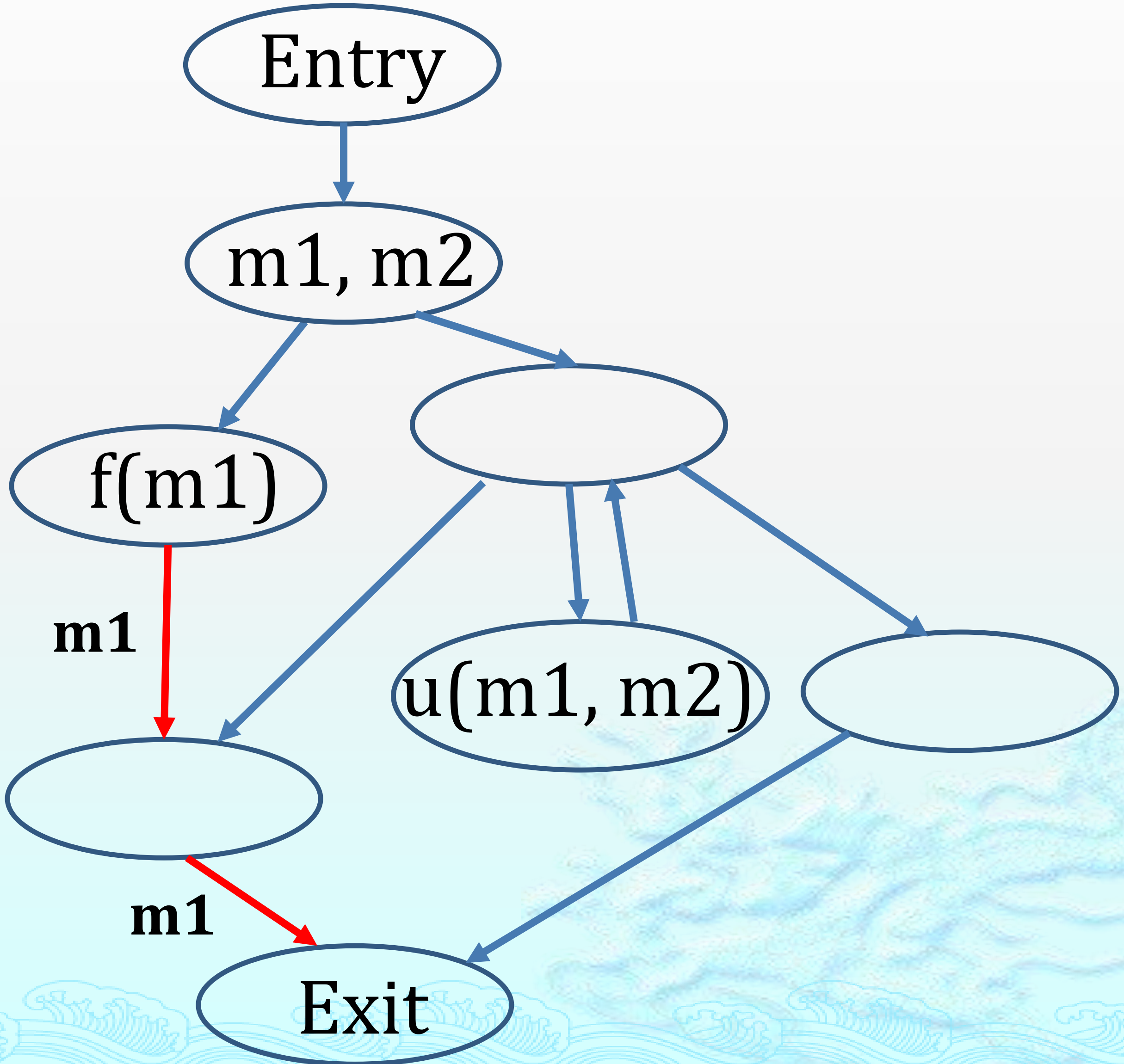
  to avoid insertions after free



Approach - Intra-procedural Analysis

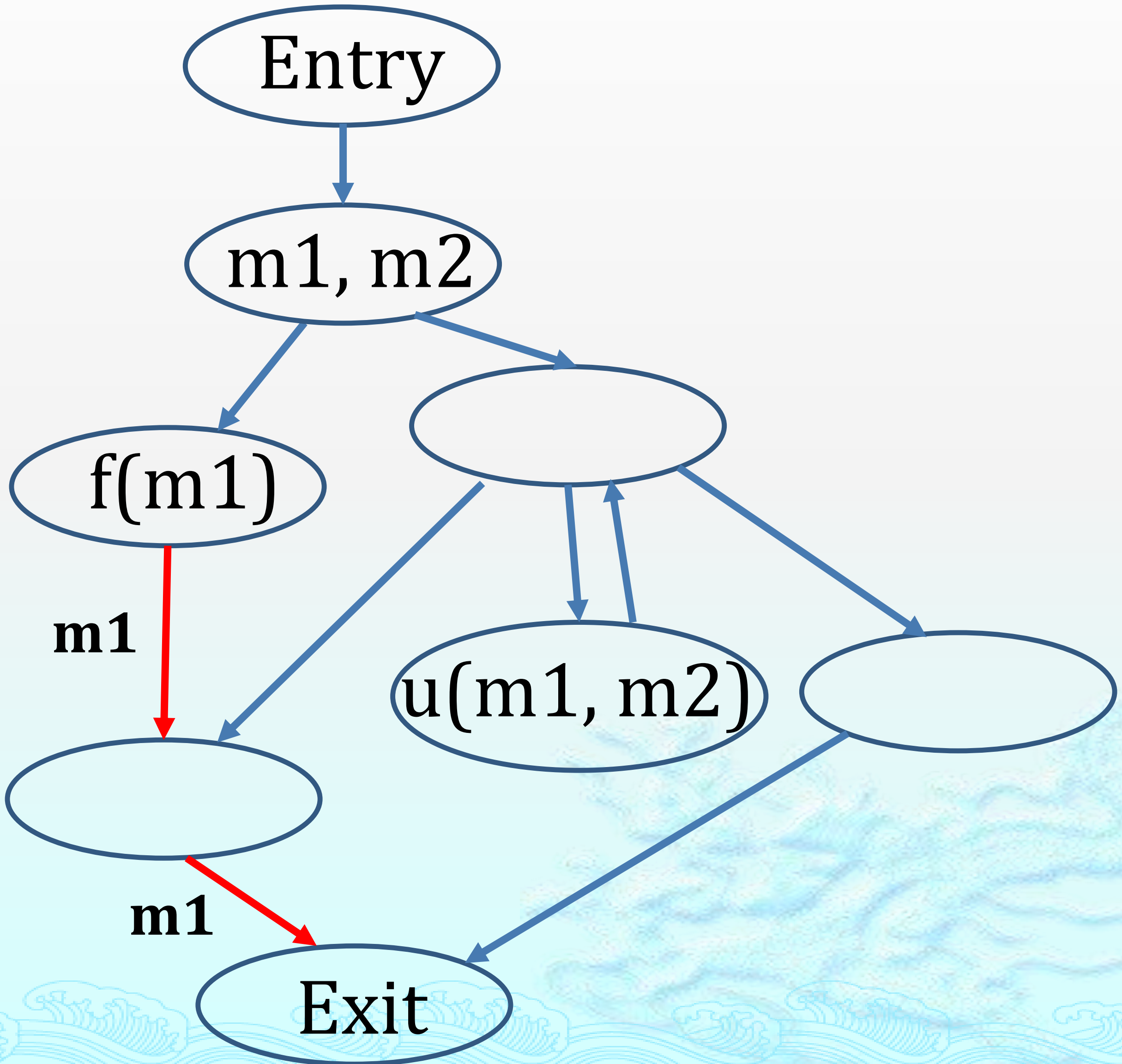
- 1^{st} pass
- Forward analysis

  to avoid insertions after free



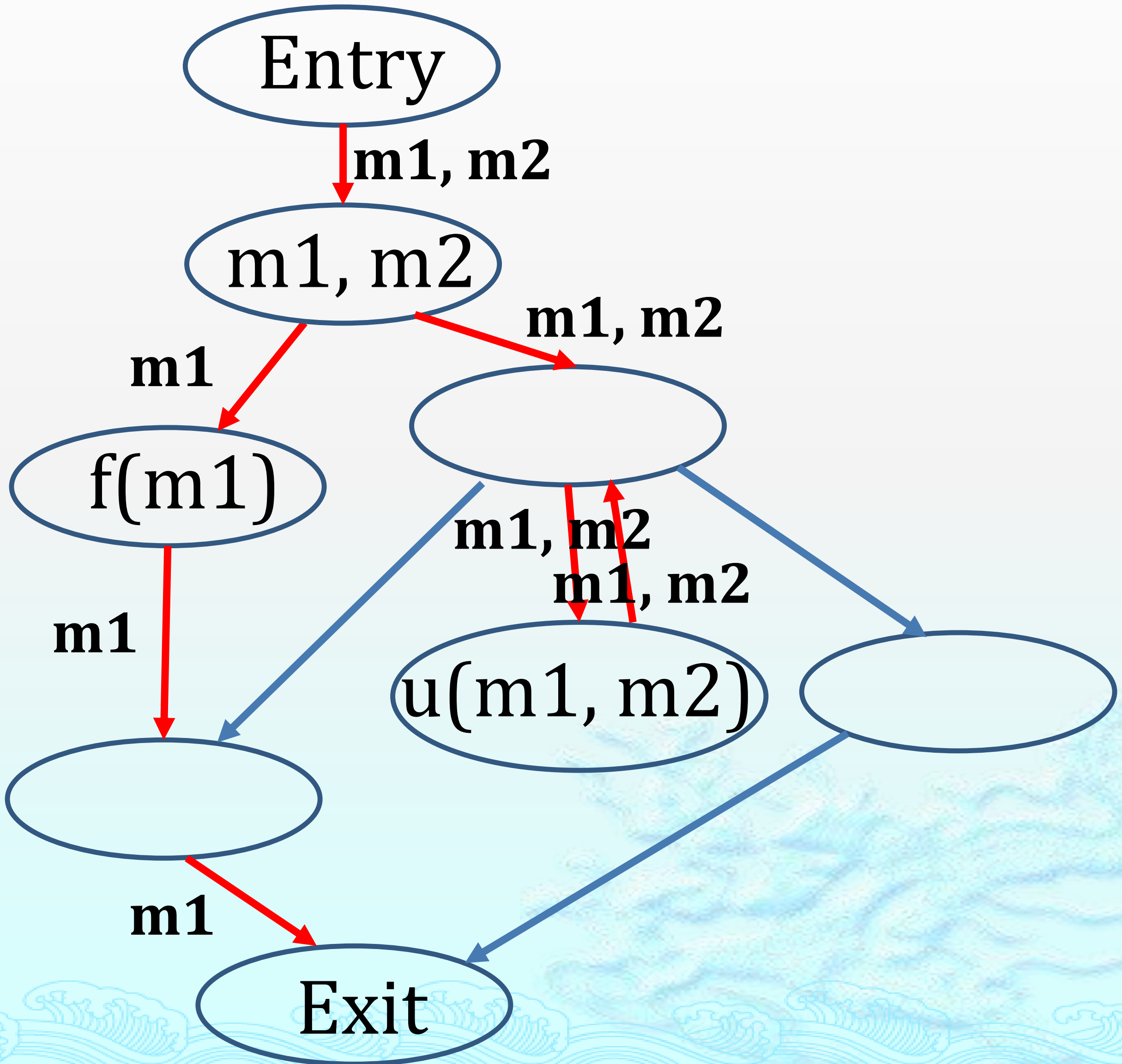
Approach - Intra-procedural Analysis

- 2nd pass
- Backward analysis to avoid insertions before free and use



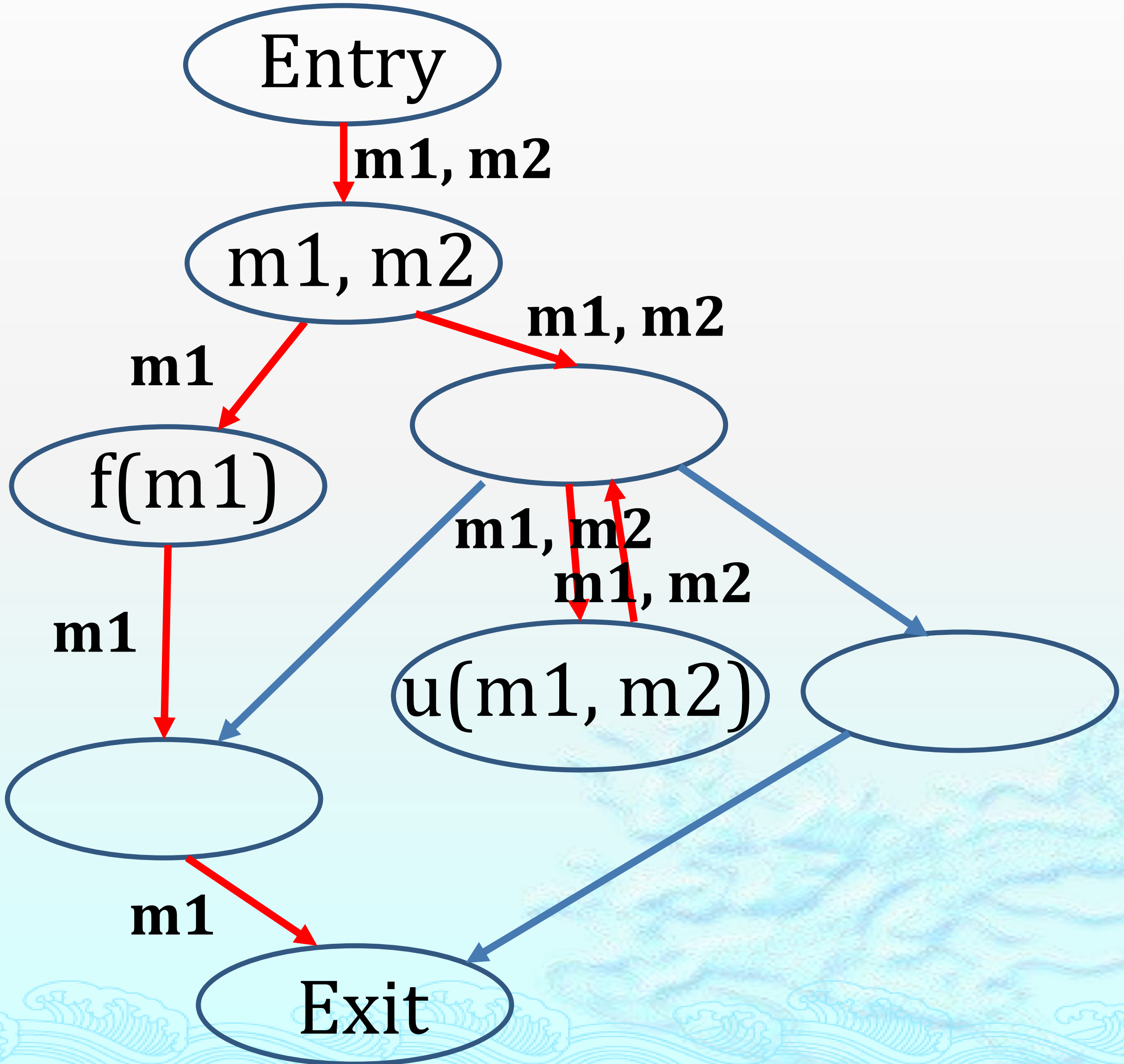
Approach - Intra-procedural Analysis

- 2nd pass
- Backward analysis to avoid insertions before free and use



Approach - Intra-procedural Analysis

- 3rd pass
- Forward analysis to find variables that reference the memory chunk

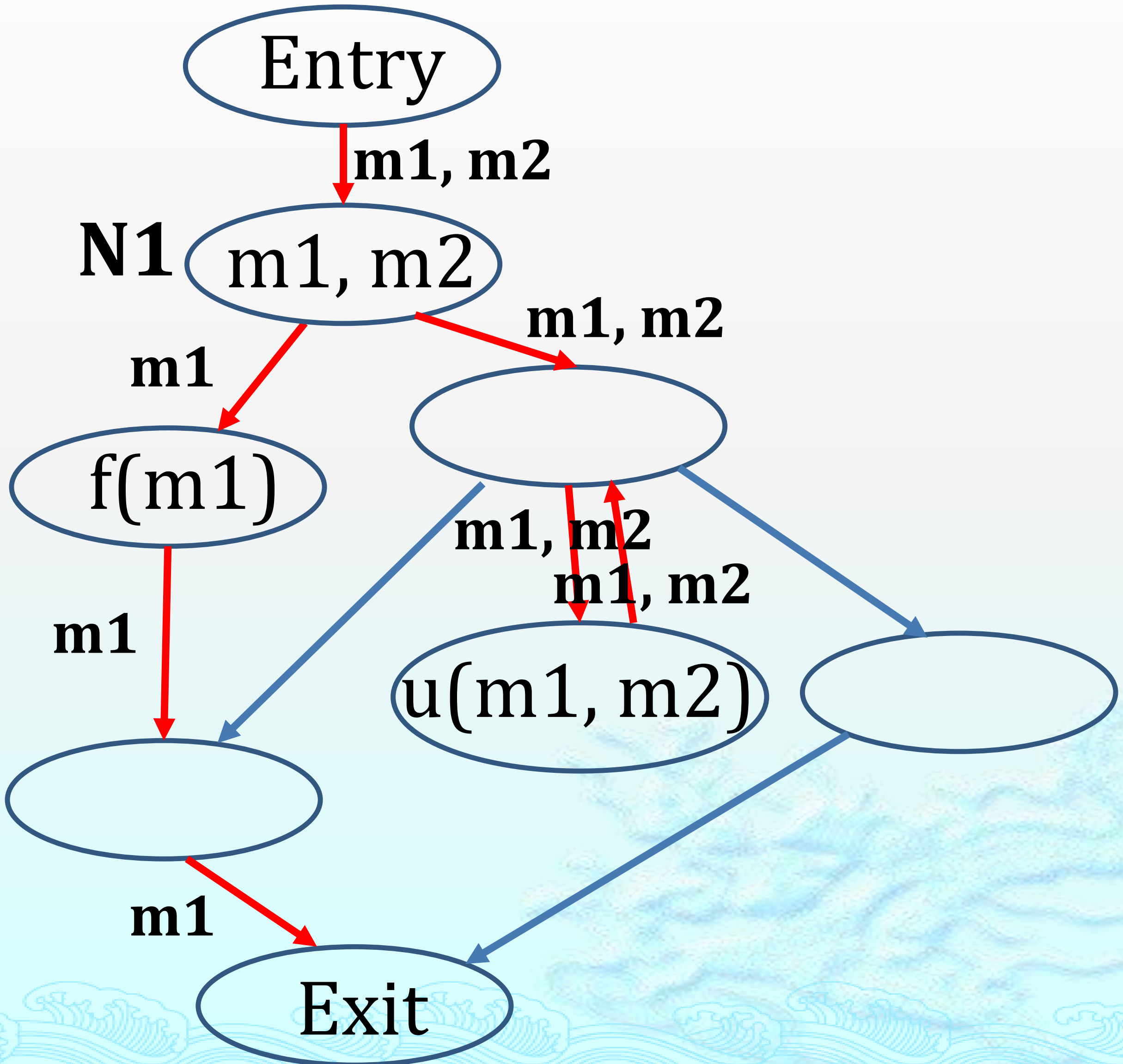


Approach - Intra-procedural Analysis

- 3rd pass
- Forward analysis to find variables that reference the memory chunk

p -> m1
q -> m2

On all edges after N1

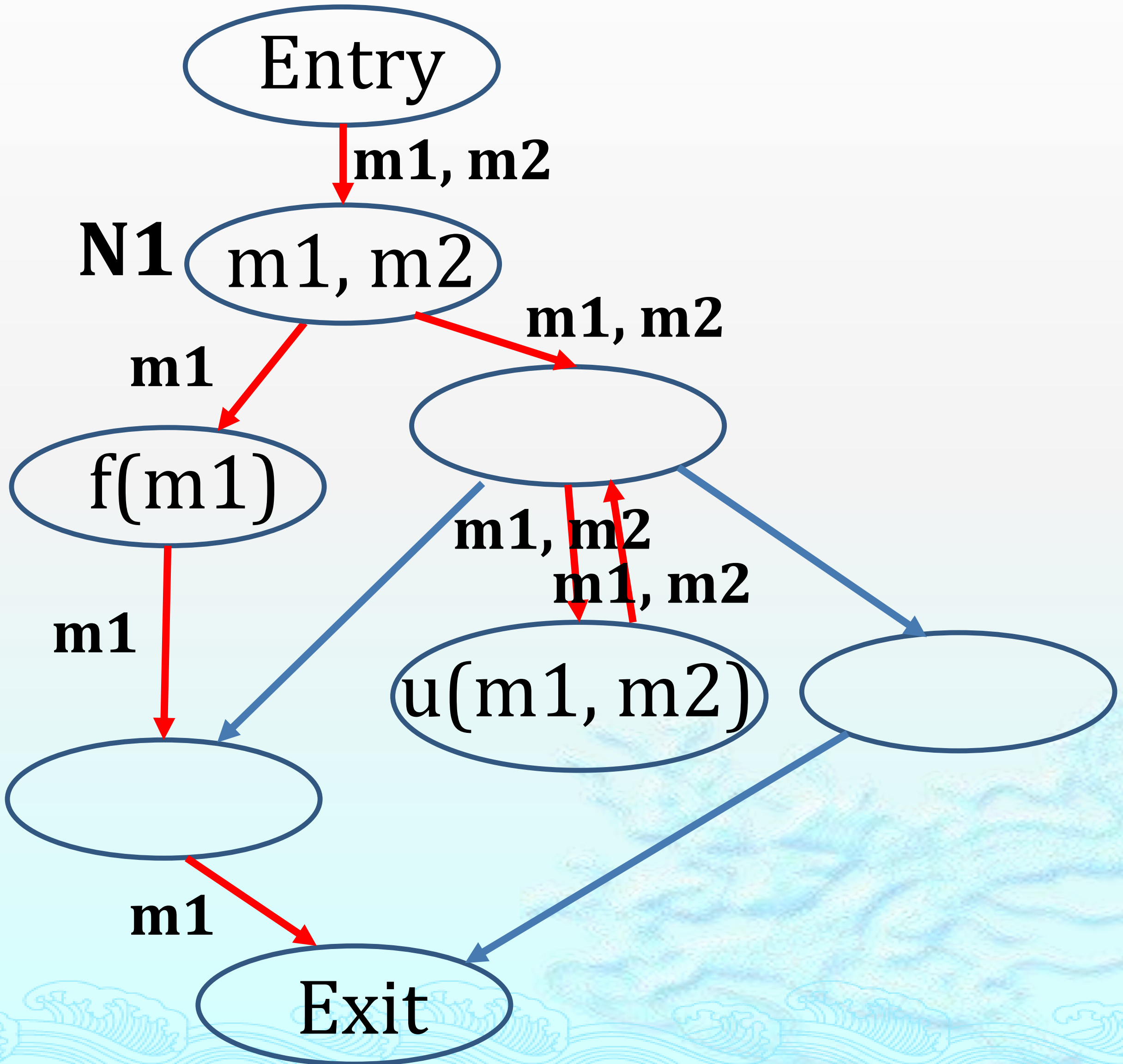


Approach - Intra-procedural Analysis

- 4^{th} pass
- Forward analysis to find early edge for free

p -> m1
q -> m2

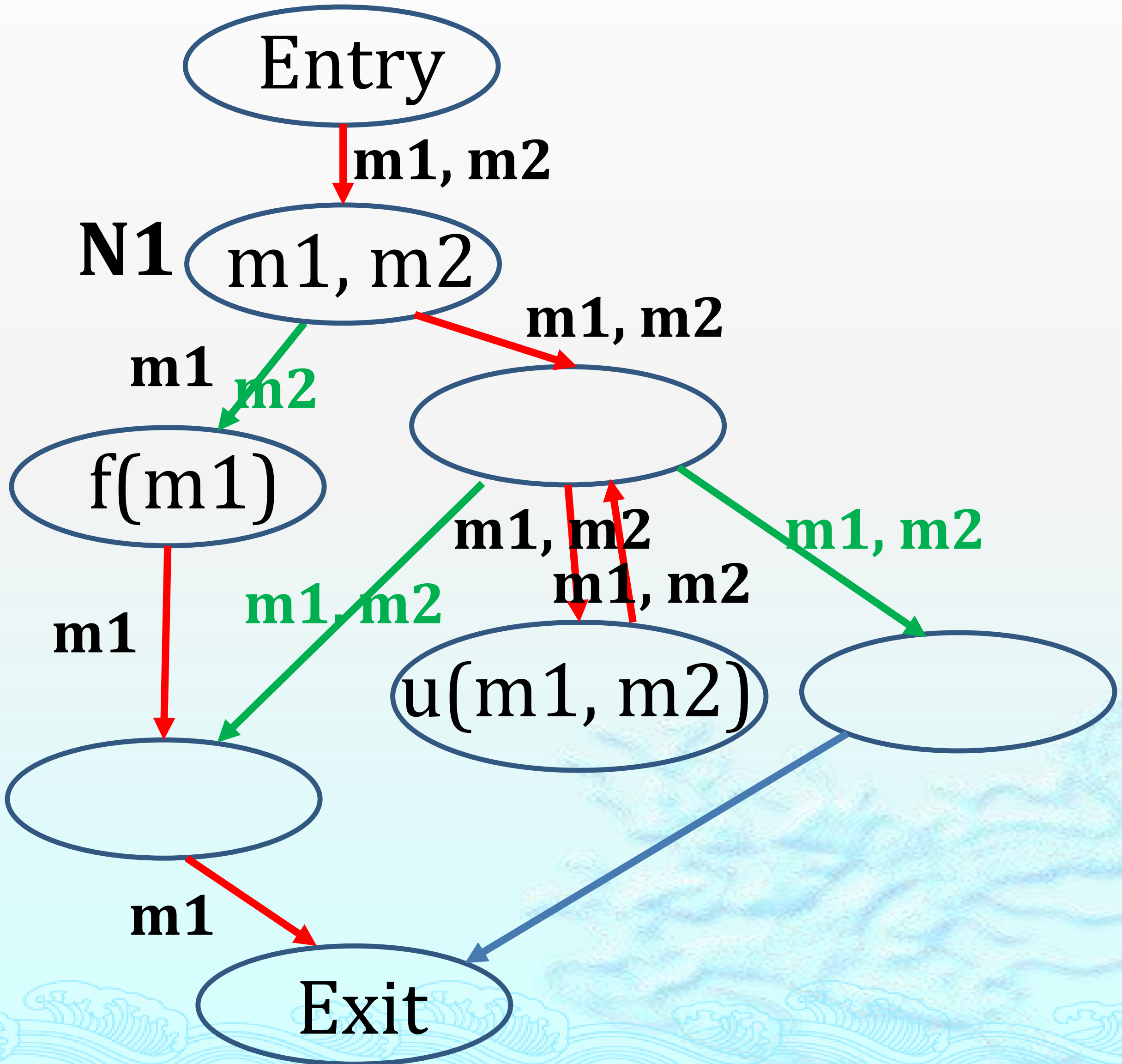
On all edges after N1



Approach - Intra-procedural Analysis

- 4^{th} pass
- Forward analysis to find early edge for free

p -> m1
q -> m2
On all edges after N1

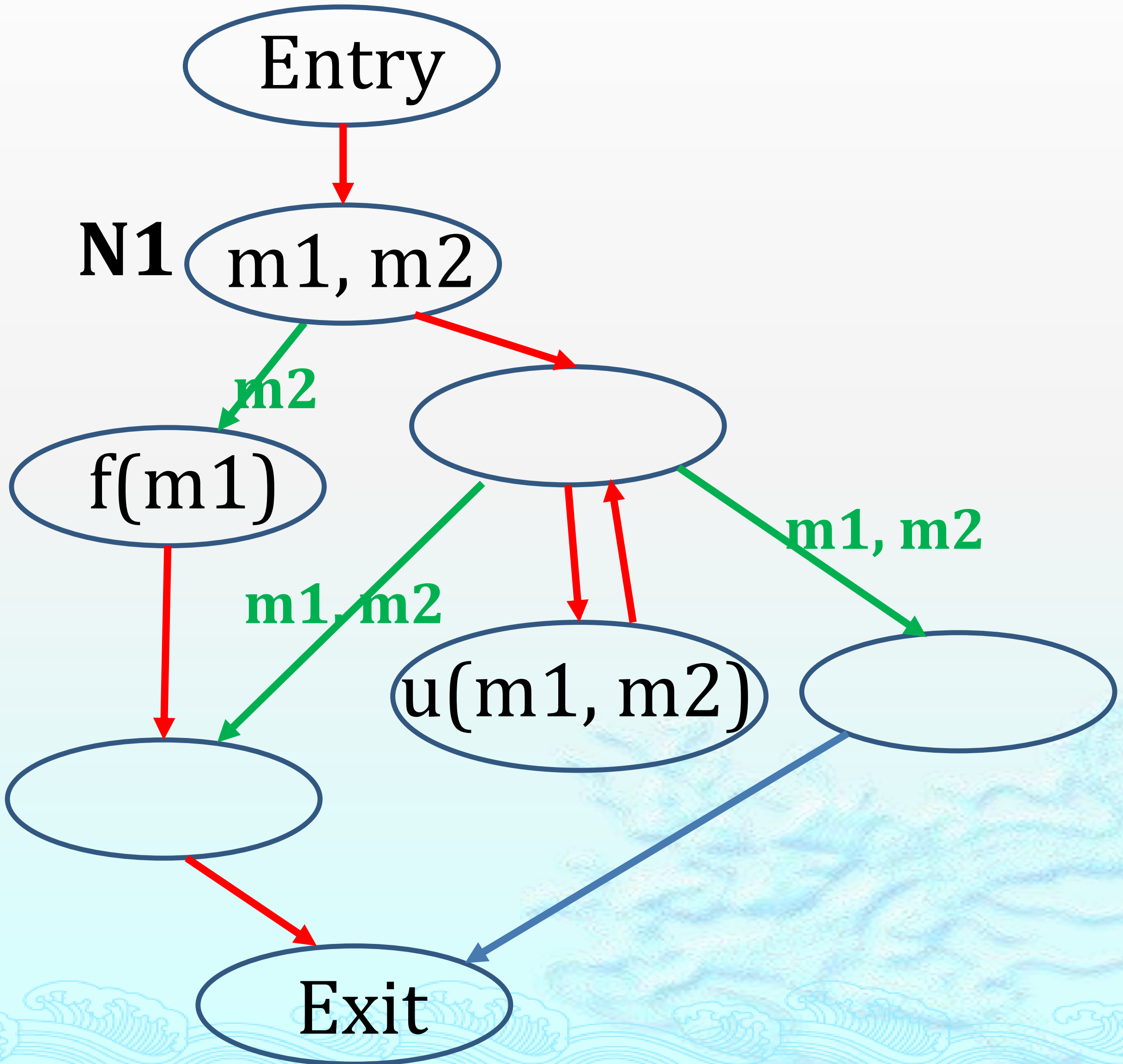


Approach - Intra-procedural Analysis

- Perform fix

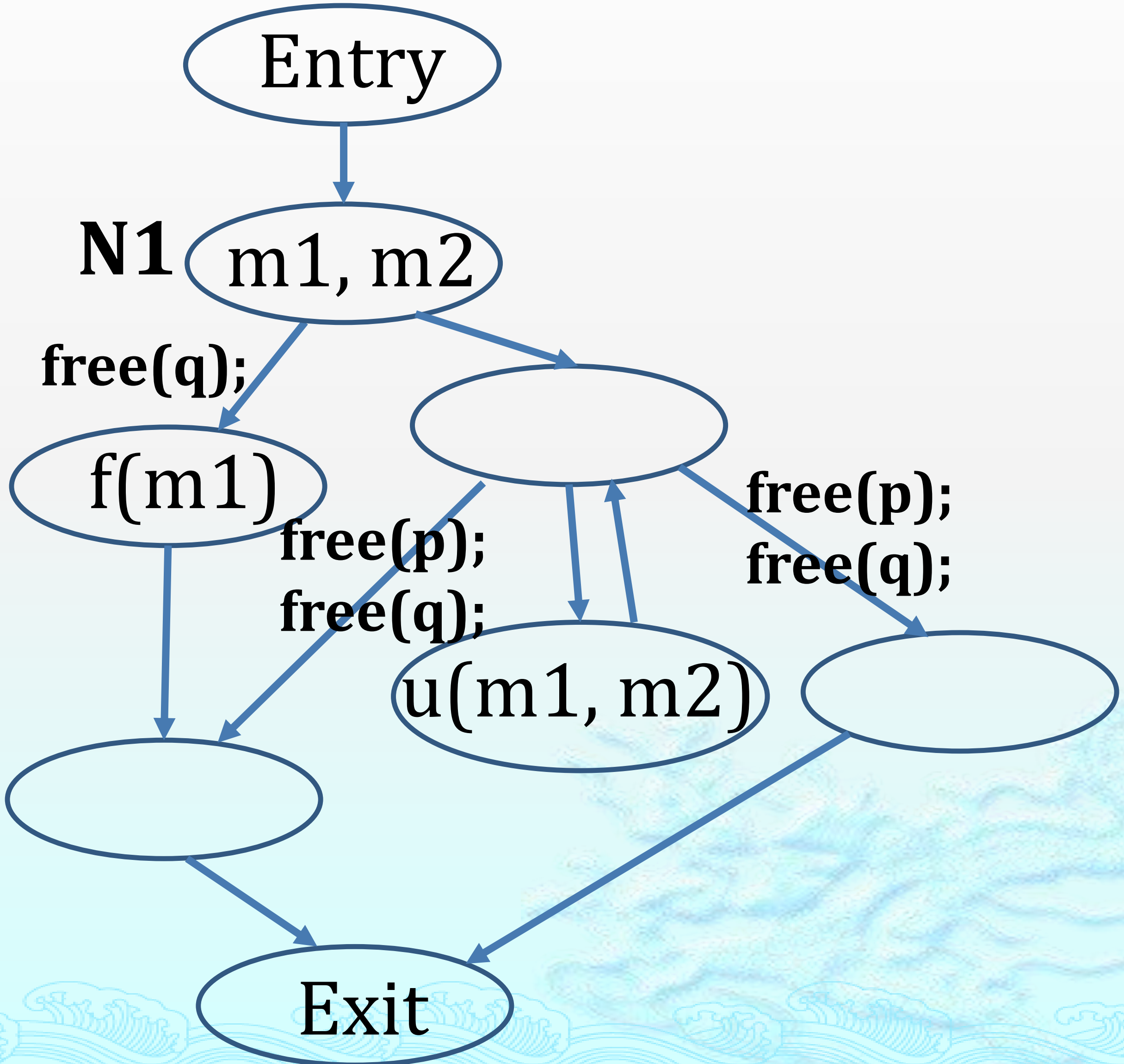
p -> m1
q -> m2

On all edges after N1



Approach - Intra-procedural Analysis

- Perform fix



Challenging Case 1

- Problem: Multiple Allocations

  if (…) p = malloc(); else p = malloc(); Use(p); return;
- Solution
  - Set-based Edge Conditions
  - Always consider a set of allocations that can be freed at the current point

Challenging Case 2

- Problem: allocations in loops cannot be freed

```
for (int i = 0; i < n; i ++) {
  p = malloc();
  use(p);
}
```

- Solution: extract the body of the loop as an independent procedure
  - Need to check the memory chunk is only used within the loop

Empirical Results

- LeakFix: Implemented on LLVM
- Benchmark: SPEC2000

Programs	Size (Kloc)	#Func	#Allocation
art	1.3	44	11
equake	1.5	45	29
mcf	1.9	44	3
bzip2	4.6	92	10
gzip	7.8	128	5
parser	10.9	342	1
ammp	13.3	197	37
vpr	17	290	2
crafty	18.9	127	12
twolf	19.7	209	2
mesa	49.7	1124	67
vortex	52.7	941	8
gap	59.5	872	2
gcc	205.8	2271	53

Existing Detection Techinques

Programs	LC	Fastcheck	SPARROW	SABER
art	1(0)	1(0)	1(0)	1(0)
equake	0(0)	0(0)	0(0)	0(0)
mcf	0(0)	0(0)	0(0)	0(0)
bzip2	1(1)	0(0)	1(0)	1(0)
gzip	1(2)	0(0)	1(4)	1(0)
parser	0(0)	0(0)	0(0)	0(0)
ammp	20(4)	20(0)	20(0)	20(0)
vpr	0(0)	0(1)	0(9)	0(3)
crafty	0(0)	0(0)	0(0)	0(0)
twolf	0(0)	2(0)	5(0)	5(0)
mesa	2(0)	0(2)	9(0)	7(4)
vortex	0(26)	0(0)	0(1)	0(4)
gap	0(1)	0(0)	0(0)	0(0)
gcc	N/A	35(2)	44(1)	40(5)
total	25(34)	58(5)	81(15)	70(14)

LeakFix Effectiveness

Programs	#Fixed	#Maximum Detected	Percentage(%)	#Fixes	#Useless Fixes
art	0	1	0	0	0
equake	0	0	N/A	0	0
mcf	0	0	N/A	0	0
bzip2	1	1	100	1	0
gzip	1	1	100	1	0
parser	0	0	N/A	0	0
ammp	20	20	100	30	0
vpr	0	0	N/A	0	0
crafty	0	0	N/A	0	0
twolf	0	5	0	0	0
mesa	1	9	11	1	0
vortex	0	0	N/A	0	0
gap	0	0	N/A	0	0
gcc	2	44	5	2	0
total	25	85	29	35	0

LeakFix Time Consumption

Programs	Size (Kloc)	Compiling and Linking Time (sec)	LeakFix Time (sec)				Total Time (sec)	Percentage(%)
			Pointer Analysis	Procedure Identification	Detection And Fix	Total		
art	1.3	0.14	0.02	0.01	0.03	0.06	0.20	30.0
equake	1.5	0.18	0.02	0.01	0.08	0.11	0.29	37.9
mcf	1.9	0.68	0.02	0.01	0.32	0.35	1.03	51.4
bzip2	4.6	0.35	0.02	0.01	0.07	0.10	0.45	22.2
gzip	7.8	0.85	0.03	0.01	0.15	0.19	1.04	18.3
parser	10.9	1.421	0.19	0.01	0.30	0.50	1.92	26.0
ammp	13.3	2.42	0.11	0.01	0.57	0.69	3.11	22.2
vpr	17	1.60	0.11	0.01	1.42	1.54	3.14	49.0
crafty	18.9	2.90	0.13	0.01	0.71	0.85	3.75	22.7
twolf	19.7	5.13	0.23	0.01	1.01	1.34	6.47	20.7
mesa	49.7	7.73	5.33	0.16	9.2	14.69	22.42	65.5
vortex	52.7	9.971	1.05	0.06	1.66	2.77	12.74	21.7
gap	59.5	5.901	6.47	0.33	29.7	36.5	42.40	86.0
gcc	205.8	10.99	27.72	4.15	95.9	128.97	139.96	92.1

Thanks!