Program Analysis

13. Modular Analysis

Kihong Heo

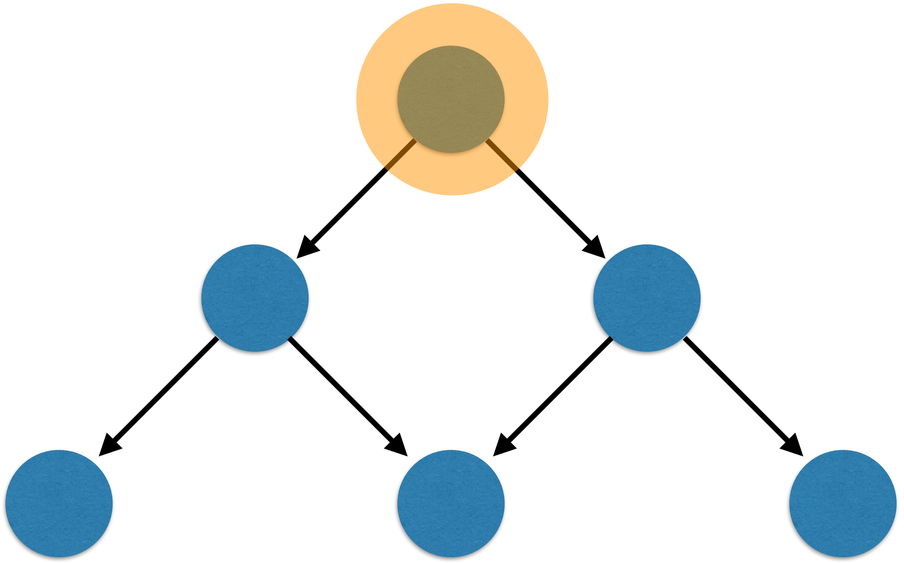
KAIST

Modularity

- Key to build a scalable software system, in general
  - E.g., modular, incremental, and parallel compilation (`make -j`)
- How to make a static analysis modular?

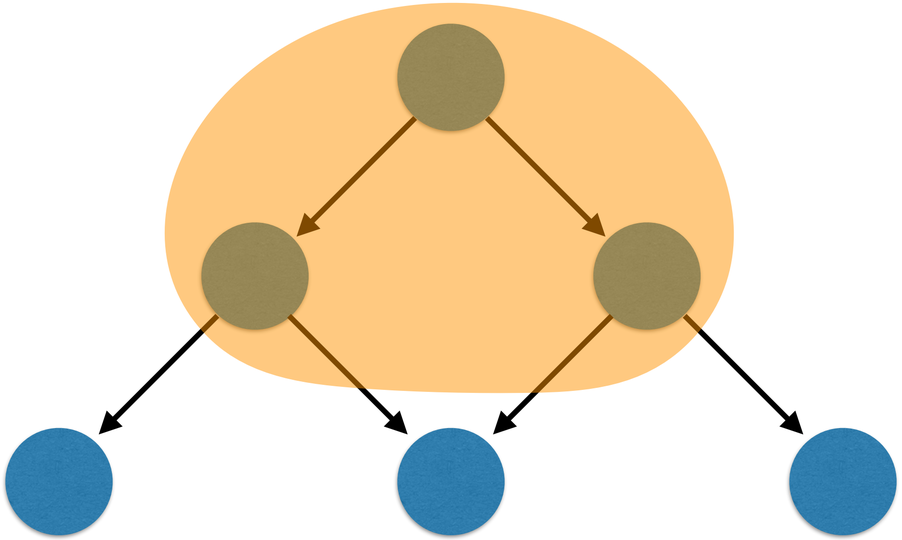
Global Analysis

- Analyze the whole program altogether
  - Starting from a root (e.g., main)
  - Similar to program execution (i.e., interpreter)



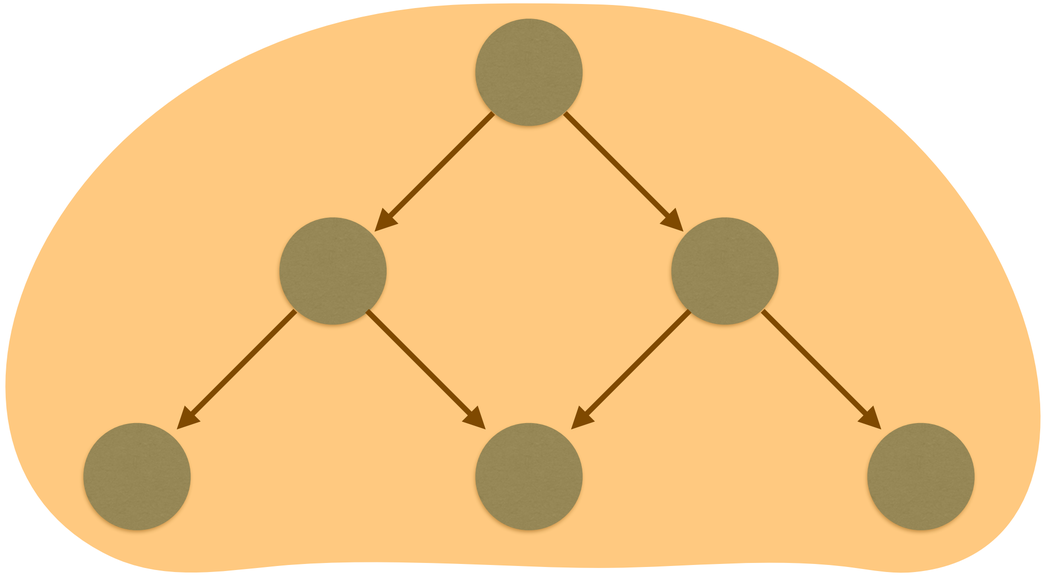
Global Analysis

- Analyze the whole program altogether
  - Starting from a root (e.g., main)
  - Similar to program execution (i.e., interpreter)



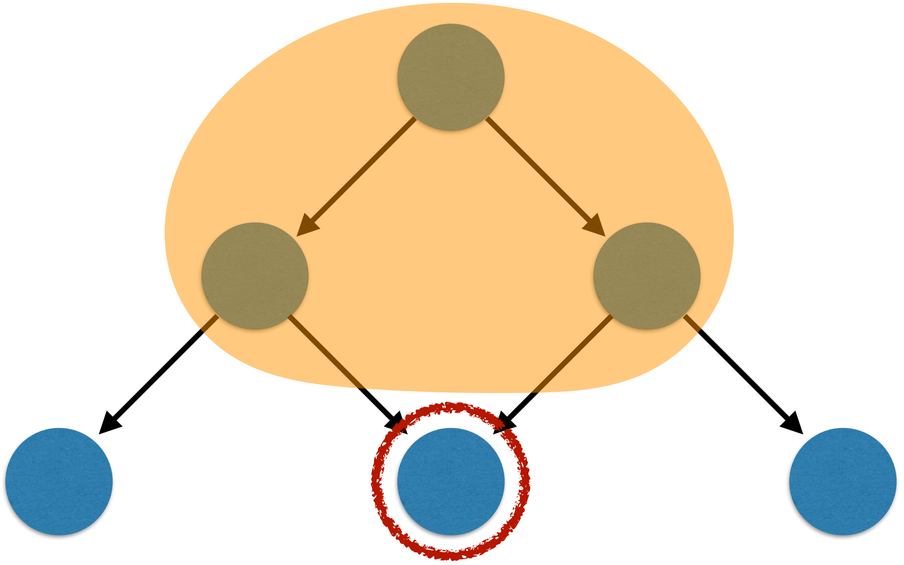
Global Analysis

- Analyze the whole program altogether
  - Starting from a root (e.g., main)
  - Similar to program execution (i.e., interpreter)



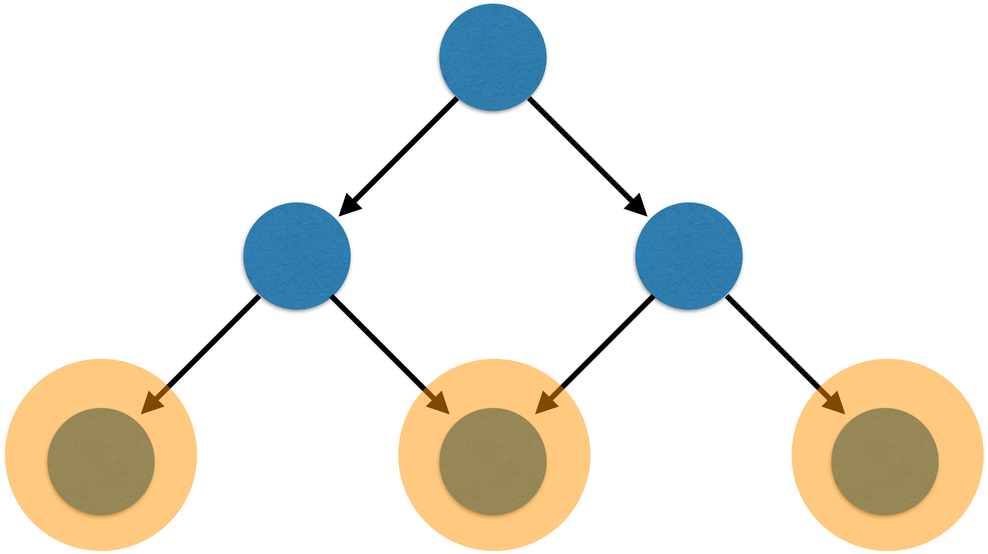
Pros and Cons of Global Analysis

- In general, context-aware but unscalable
  - Pros: aware of calling contexts (e.g., parameters, global variables, etc)
  - Cons: reanalyze the same portion repeatedly



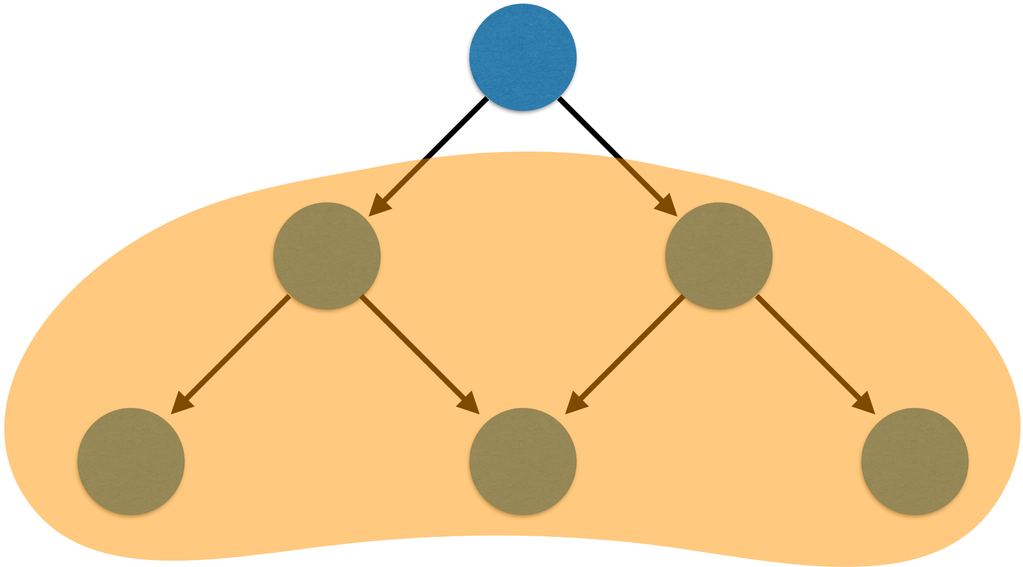
Modular Analysis

- Analyze each subcomponent (e.g., function) independently and compose
  - Starting from leaf elements
  - Similar to compilers



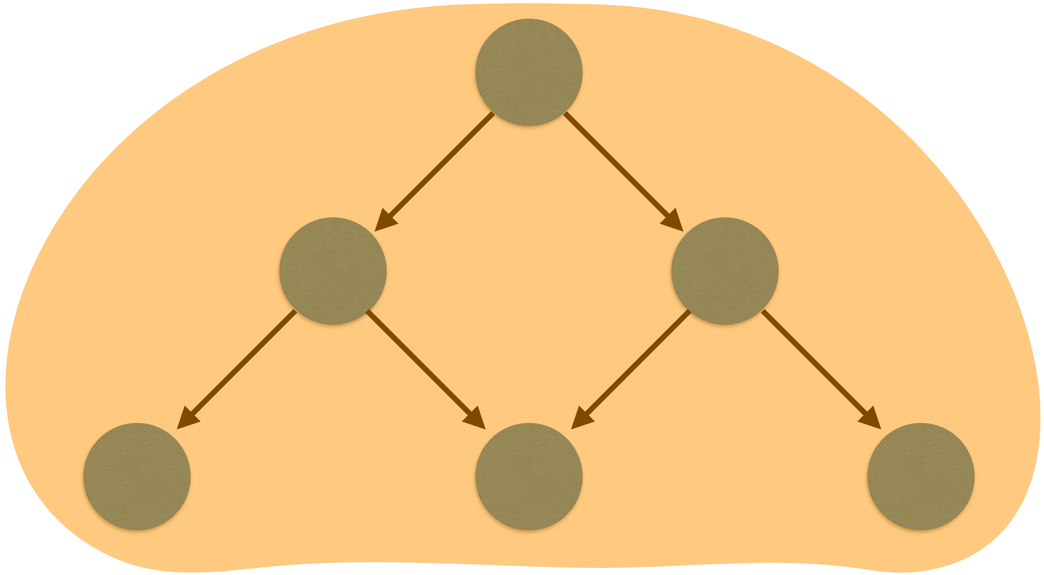
Modular Analysis

- Analyze each subcomponent (e.g., function) independently and compose
  - Starting from leaf elements
  - Similar to compilers



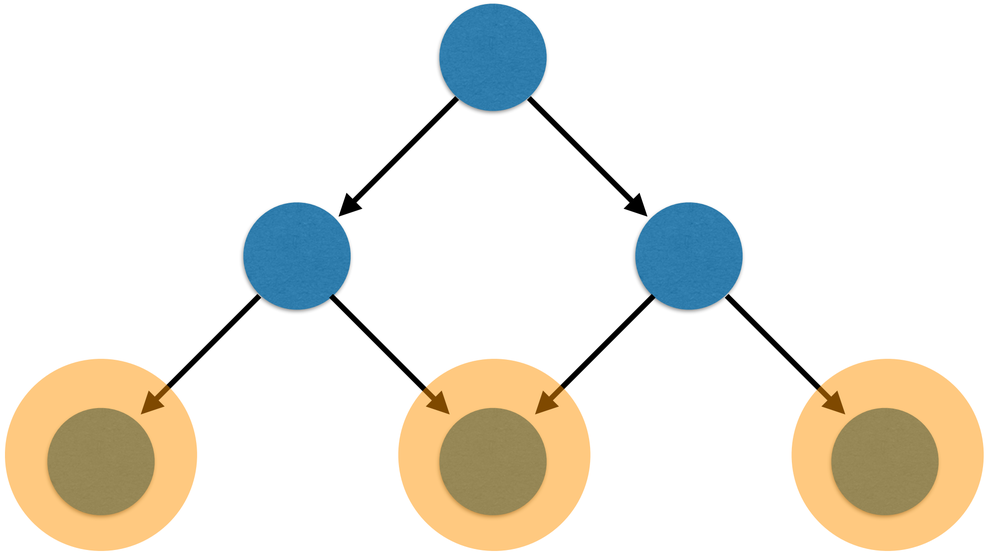
Modular Analysis

- Analyze each subcomponent (e.g., function) independently and compose
  - Starting from leaf elements
  - Similar to compilers



Pros and Cons of Modular Analysis

- In general, scalable but context-unaware
  - Pros: each component is analyzed only once
  - Cons: a mechanism for handling pre-state is needed



Challenges

- How to design **generic yet accurate** modular analysis?
  - Effectively reusable analysis results of subcomponents
- Traditionally, simple or inductively defined properties
  - E.g., nullness, tree, list, etc

How about numerical properties?
(e.g., buffer-overrun)

```
void foo(void* p){
  *p = 0;
}
```

Safe if p is not null

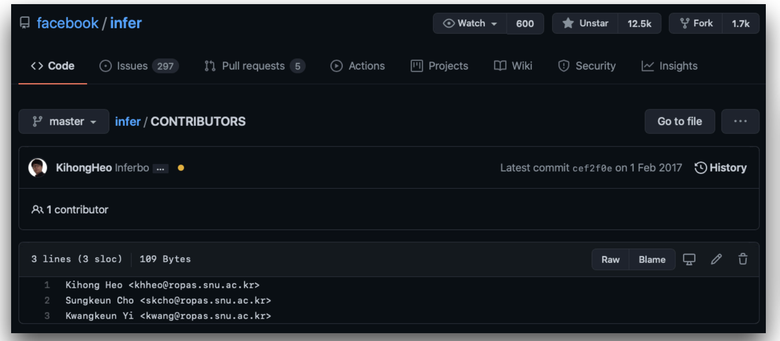
```
void bar(list* p){
  while(!p)
    p = p->next;
}
```

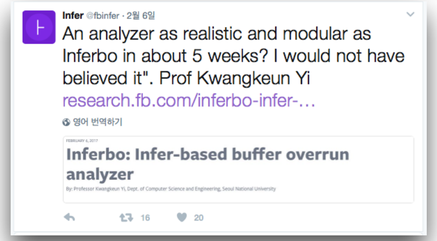
Safely terminate if p is a well-formed singly linked list

Example: Inferbo

- Facebook's **Infer**-based **b**uffer **o**verrun analyzer
- **Function-level modular**: analyze each function and then link them
- **Parameterization**: symbolic parameter for unknown calling context
- **Summary-based**: derive safety conditions for each function
- **Scalability**: enabled by modular and incremental analysis
- **Availability**: provided as a checker of Facebook Infer (https://github.com/facebook/infer)

Impact





*https://research.fb.com/blog/2017/02/inferbo-infer-based-buffer-overrun-analyzer/

Example

```
char* malloc_wrapper(int n) {
  return malloc(n);
}
```

Var	Val
n	[s0, s1]
ret	(offset: [0, 0], size: [s0, s1])

```
void set_i(char* arr, int index){
  arr[index] = 0;
}
```

Var	Val
arr	(offset: [s4, s5], size: [s6, s7])
index	[s8, s9]

Safety Condition
[s4 + s8, s5 + s9] < [s6, s7]

Example (Cont'd)

```
void interprocedural() {
  char *arr = malloc_wrapper(9);
  int i;
  for (i = 0; i < 9; i++) {
    set_i(arr, i);     // safe
    set_i(arr, i + 1); // bug
  }
}
```

Summary of malloc_wrapper

Var	Val
n	[s0, s1]
ret	(offset: [0, 0], size: [s0, s1])

Var	Val
arr	(offset: [0, 0], size: [9, 9])

Example (Cont'd)

```
void interprocedural() {
  char *arr = malloc_wrapper(9);
  int i;
  for (i = 0; i < 9; i++) {
    set_i(arr, i);      // safe
    set_i(arr, i + 1); // bug
  }
}
```

Summary of set_i

Var	Val
arr	(offset: [s4, s5], size: [s6, s7])
index	[s8, s9]

Safety Condition
[s4 + s8, s5 + s9] < [s6, s7]

Var	Val
arr	(offset: [0, 0], size: [9, 9])
i	[0, 8]

Safety Condition
[0 + 0, 0 + 8] < [9, 9]

Example (Cont'd)

```
void interprocedural() {
  char *arr = malloc_wrapper(9);
  int i;
  for (i = 0; i < 9; i++) {
    set_i(arr, i);     // safe
    set_i(arr, i + 1); // bug
  }
}
```

Summary of set_i

Var	Val
arr	(offset: [s4, s5], size: [s6, s7])
index	[s8, s9]

Safety Condition
[s4 + s8, s5 + s9] < [s6, s7]

Var	Val
arr	(offset: [0, 0], size: [9, 9])
i	[0, 8]

Safety Condition
[0 + 1, 0 + 9] < [9, 9]

Unsound Design Choices

- Unsoundness is a **necessary evil** for better scalability and accuracy
- Inferbo is designed to be unsound for the following parts:
  - Aliasing of parameters
  - Global variables
  - Recursive calls

Application: Cost Analysis

```
void loop(ArrayList<Integer> list) {
  for (int i = 0; i <= list.size(); i++) {
  }
}
// Cost: O(|list|)
```

```
+ void foo(int i) {
+  // Cost: O(i)
+ }

  void loop(ArrayList<Integer> list) {
    for (int i = 0; i <= list.size(); i++) {
+     foo(i);
    }
  }
  // Cost: O(|list|²)
```

*https://fbinfer.com/docs/checker-cost

Summary

- Modular analysis: **separately** analyze each subcomponent and then **link**
- Key point: design of **generic and accurate** summary
- Inferbo: a function-level modular analysis with symbolic interval domain
- In practice, unsound design choices may be needed for better performance