Semantic Difference Summarizing

ACROSS PROGRAM VERSIONS

ANNA TROSTANETSKI

2 programs – why?

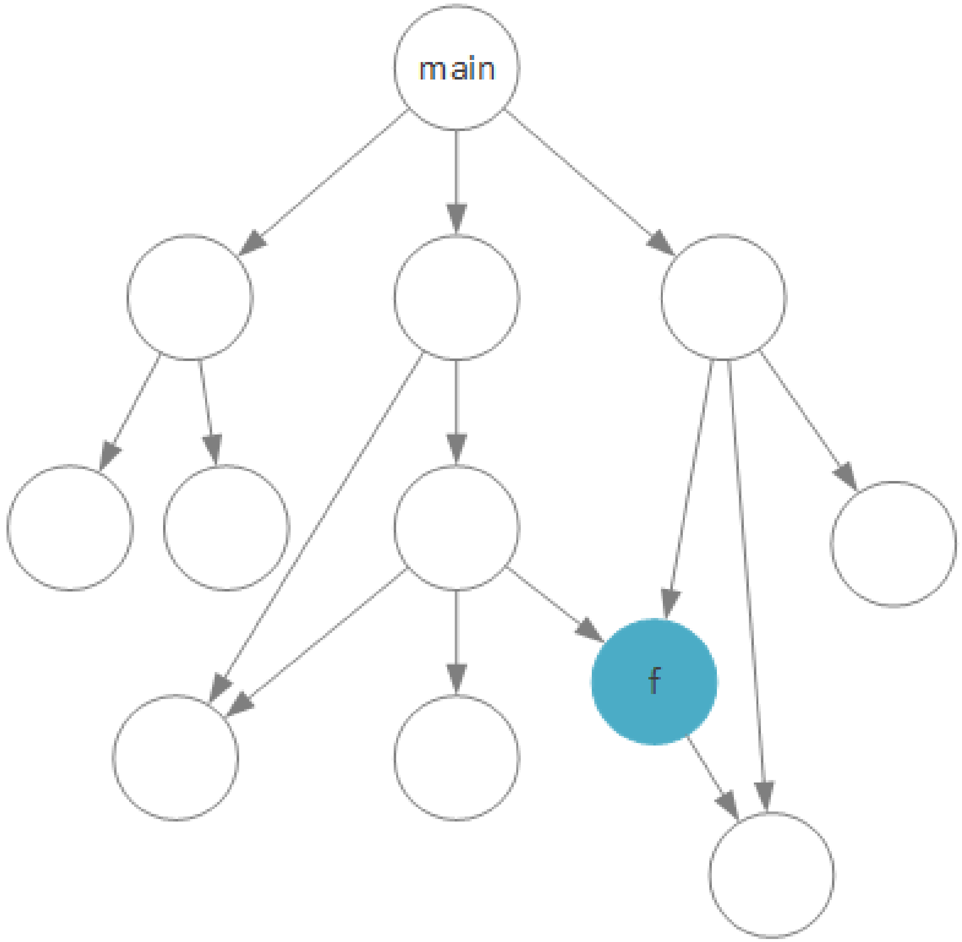
Programs often change and evolve, raising the following interesting questions:

- Did the new version introduced new bugs or security vulnerabilities?
- Did the new version remove bugs or security vulnerabilities?
- More generally, how the behavior of the program change?

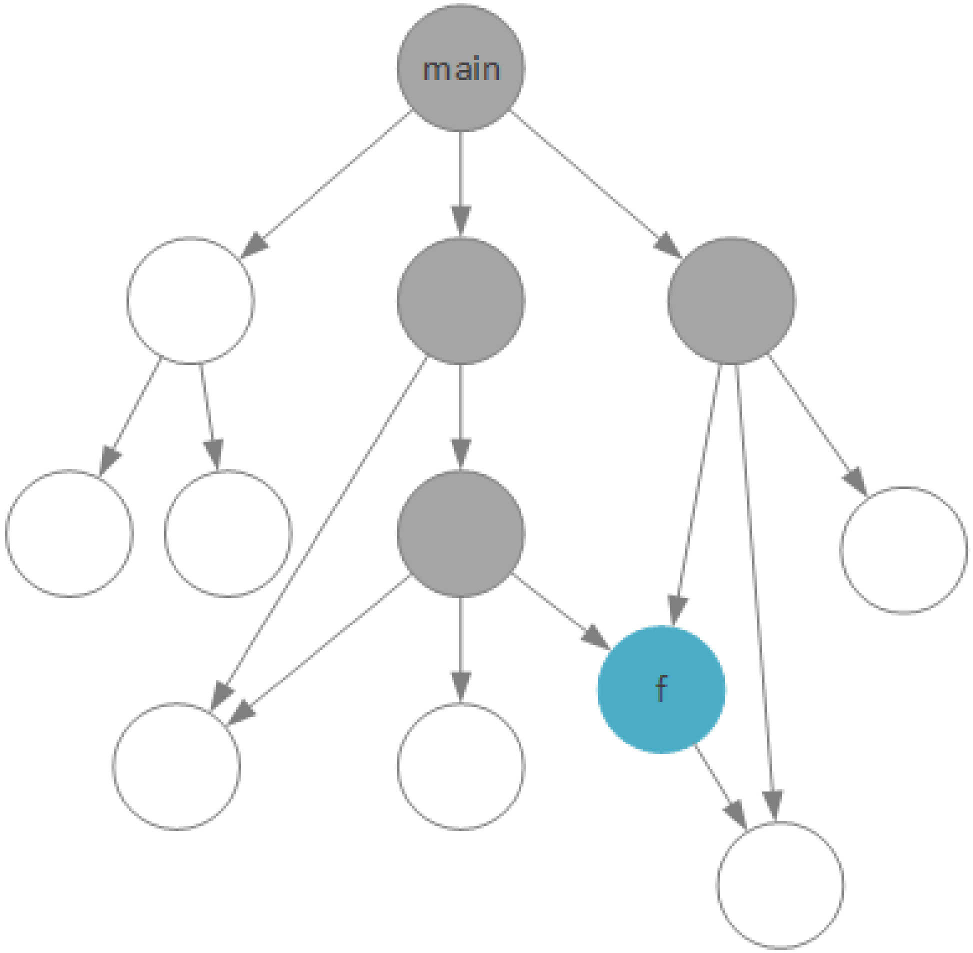
How Programs Change

Changes are small,
programs are big

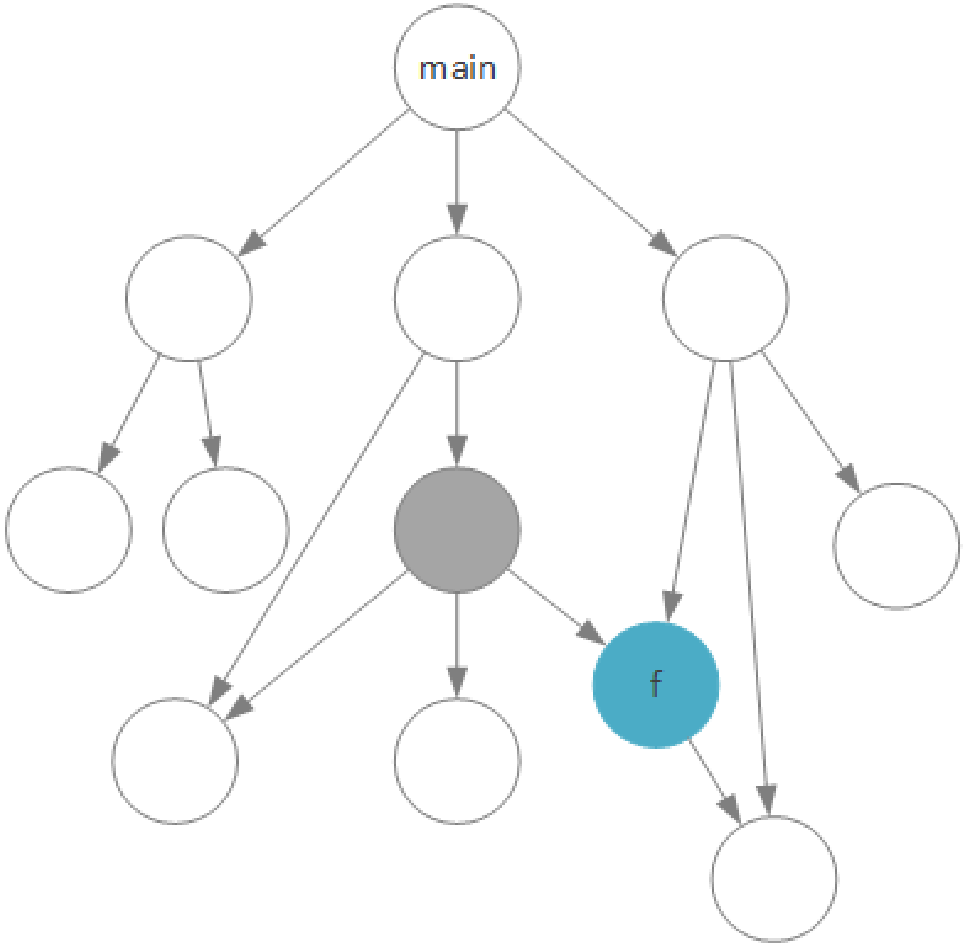
Can our work be
O(change) instead of
O(program)?



Which procedures could be affected



Which procedures are affected



CFGs

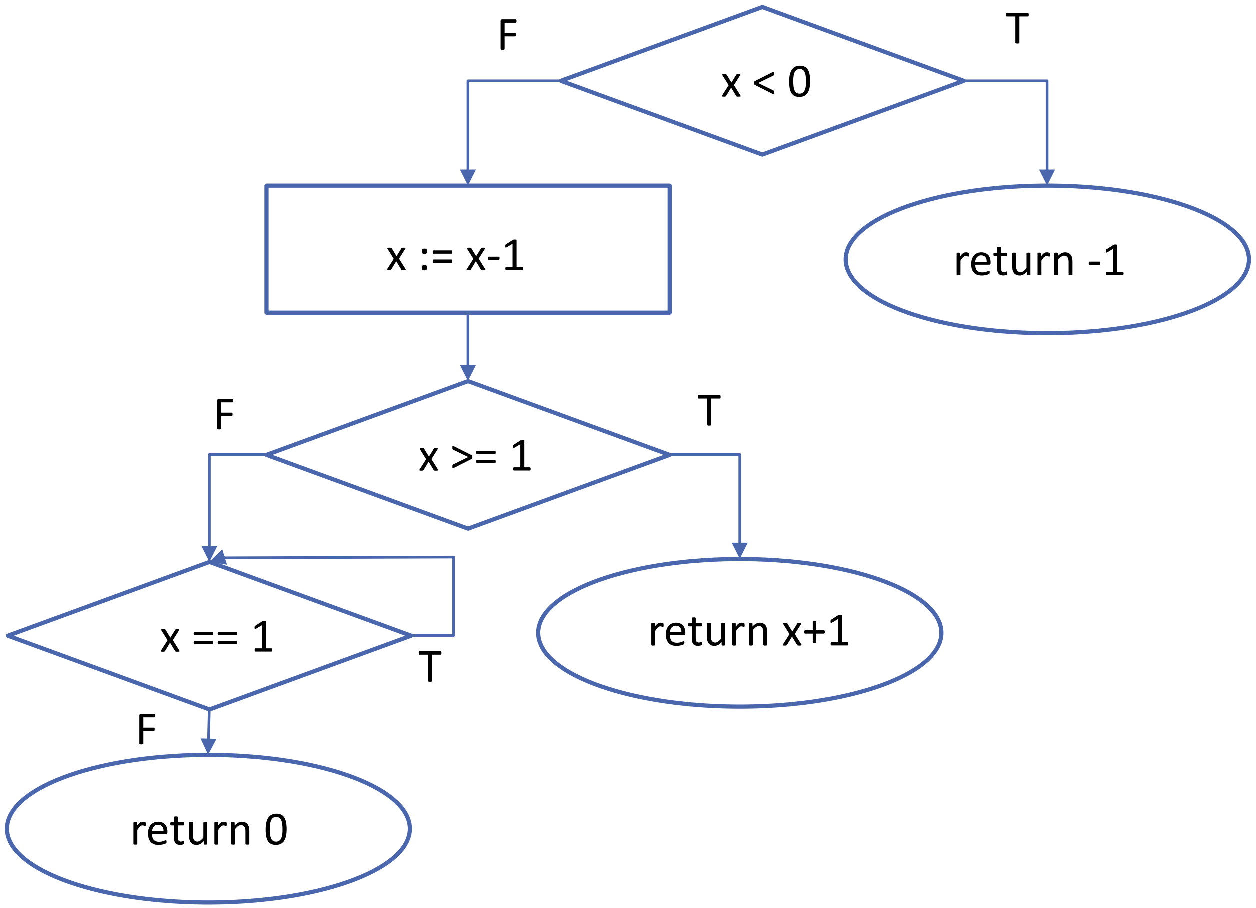
- A control flow graph (CFG) is a directed graph:
  - the nodes represent program instructions (assignments, conditions and returns)
  - the edges represent possible flow of control

- Every procedure is represented by a CFG, and the entire program is represented by a call graph.

Example

```
int p(int x) {

    if (x < 0)
        return -1;
    x--;
    if (x >= 1)
         return x+1;
    else
        while ( x == 1);
     return 0;
}
```



Finite paths

Given a finite path π in a CFG from the entry node to the exit node, define

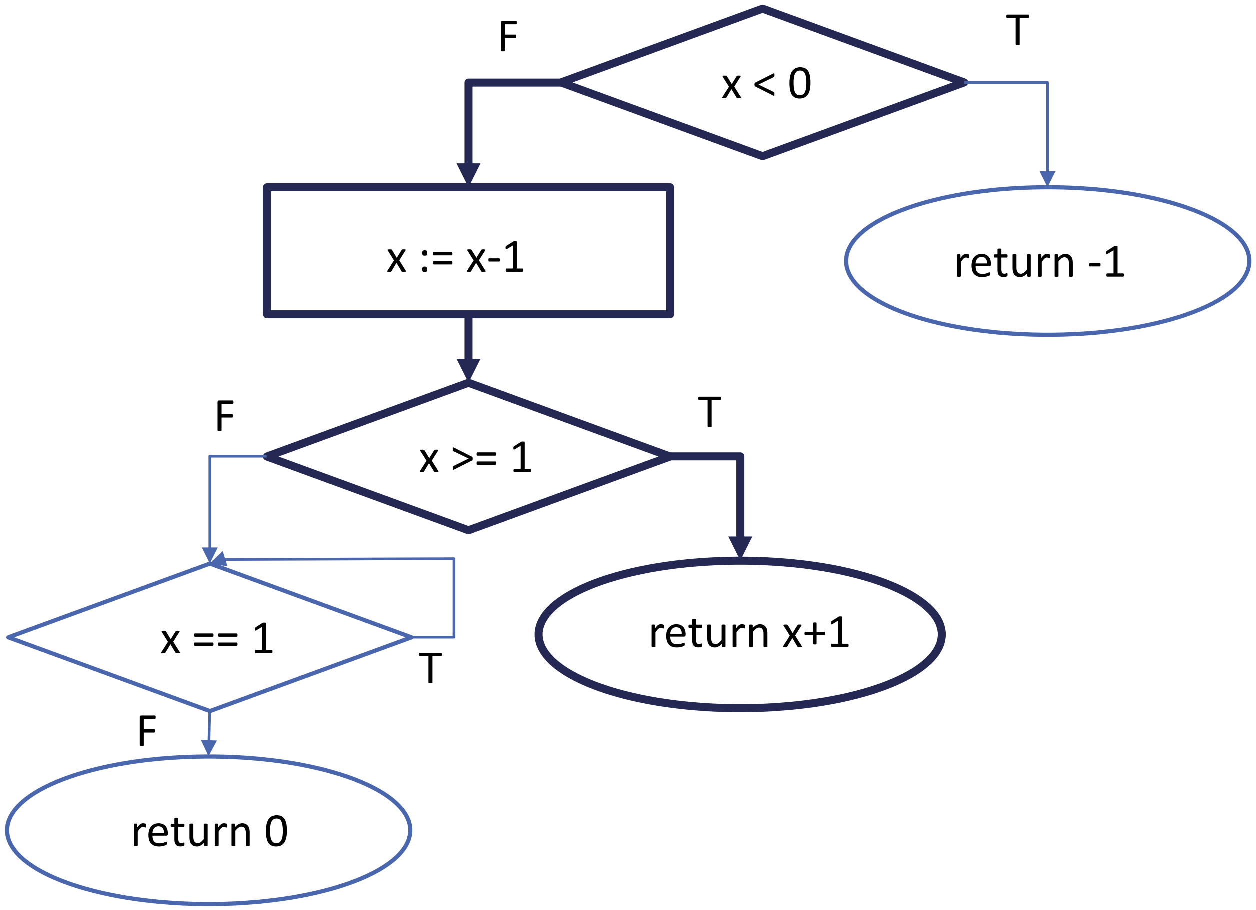
– The **reachability condition**, R_{π}, is an FOL formula which guarantees that control will traverse π.

– The **state transformation**, T_{π}, is a function from initial state $\bar{x}$ to the final state obtained if control traverses π starting with $\bar{x}$.

Example

$$R_\pi(x) = x \geq 0 \wedge x - 1 \geq 1$$
$$\equiv x \geq 2$$

$$T_\pi(x) = x$$



Symbolic execution

- Input variables are given symbolic values
- Every execution path is explored individually (in some heuristic order)
- On every branch, a feasibility check is performed with a constraint solver
- A path constraint is computed for each path
- The reachability condition and state transformation can be constructed out of the path constraint

Procedure summary

- **Path summary** for a finite path π is the pair (R_π, T_π)

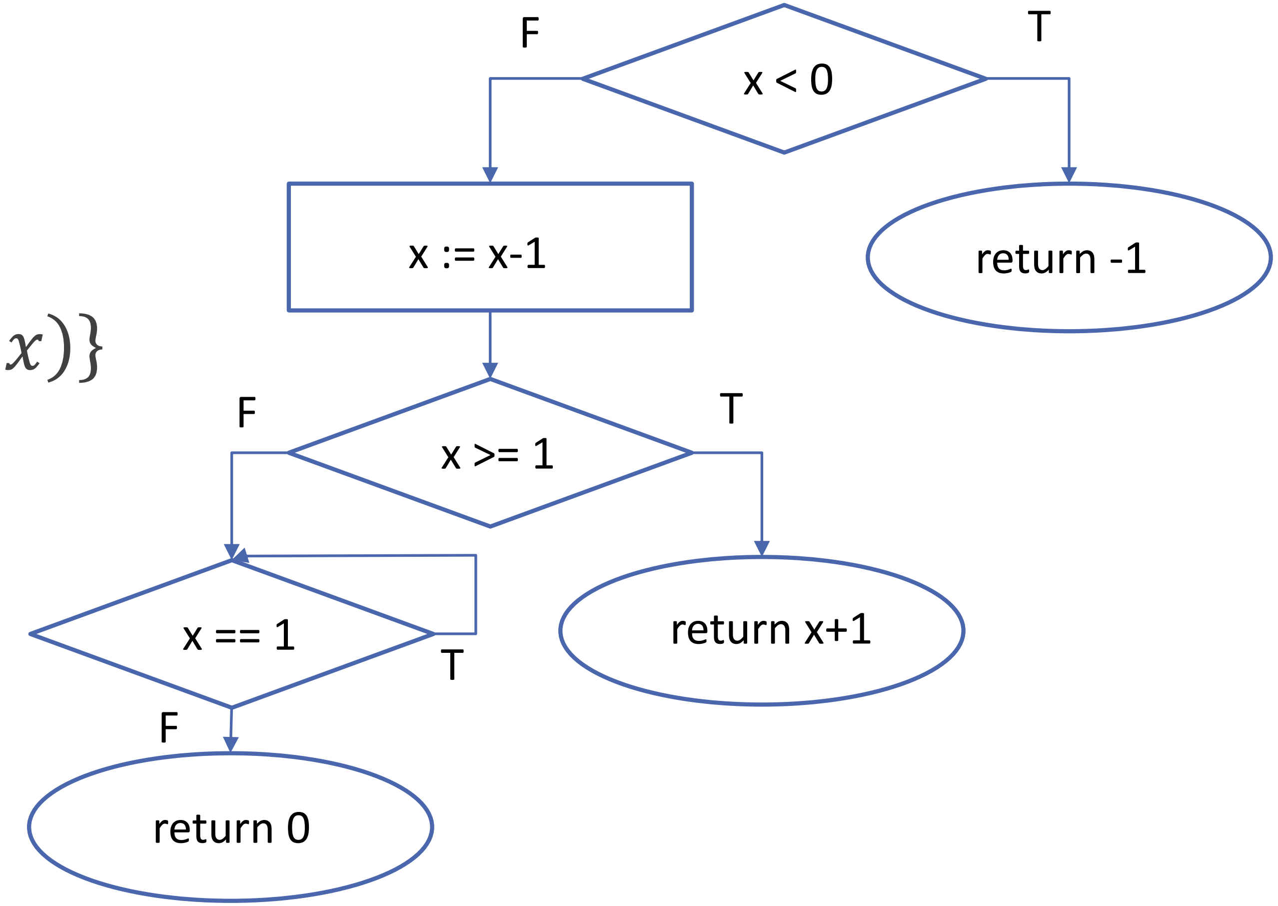
- **Procedure summary** of procedure p, sum_p, is a set of disjoint path summaries

- Given a procedure summary s, its **uncovered part** is $uncovered_p = \neg \bigvee_{(r,t)\in s} r$

Example

A possible summary for this procedure is
$sum_p = \{(x < 0, -1), (x \geq 2, x)\}$

and its uncovered part is
$$x \geq 0 \wedge x < 2$$

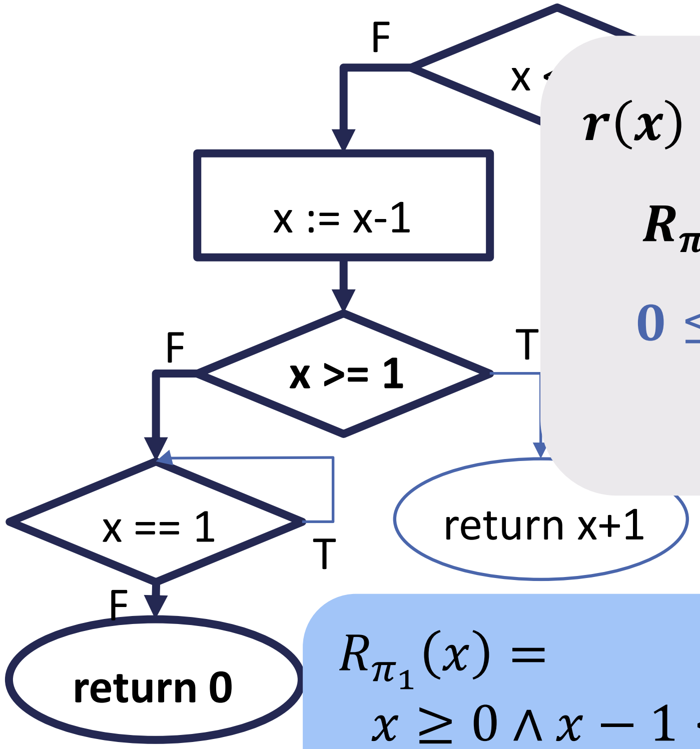


Difference Summary

Path difference for path π_1 in CFG_1 and path π_2 in CFG_2 is a triplet $\left(r, T_{\pi_1}, T_{\pi_2}\right)$ such that:

$$r \leftrightarrow R_{\pi_1} \wedge R_{\pi_2} \wedge T_{\pi_1} \neq T_{\pi_2}$$

Example

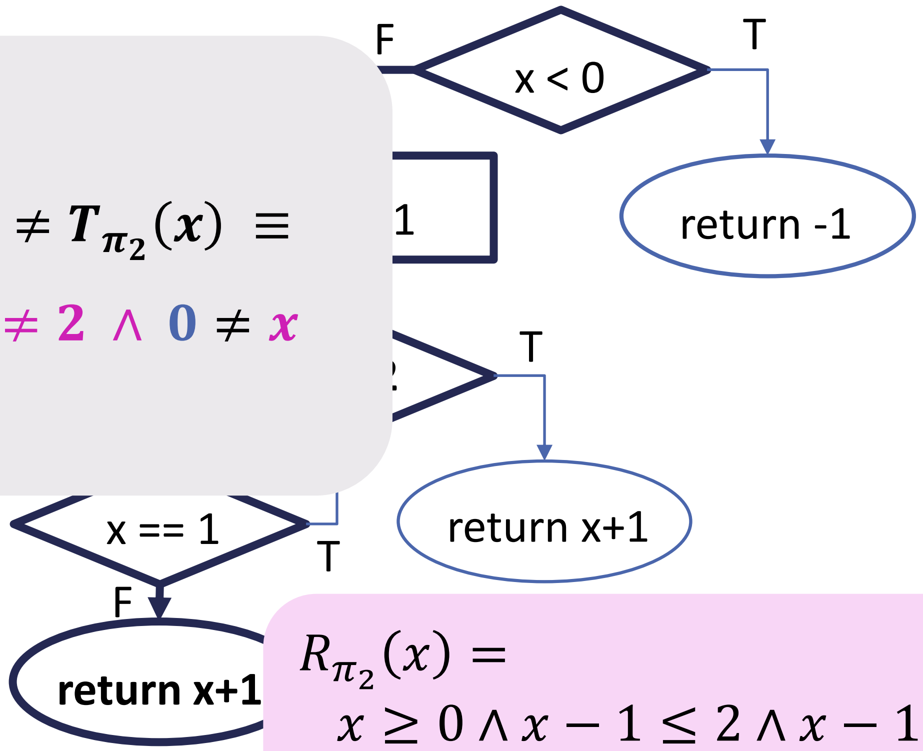


$$\boldsymbol{r(x)} \equiv$$

$$\boldsymbol{R_{\pi_1}(x)} \wedge \boldsymbol{R_{\pi_2}(x)} \wedge \boldsymbol{T_{\pi_1}(x) \neq T_{\pi_2}(x)} \equiv$$

$$\boldsymbol{0 \leq x < 2} \wedge \boldsymbol{0 \leq x \leq 3 \wedge x \neq 2} \wedge \boldsymbol{0 \neq x}$$

$$\equiv \boldsymbol{x = 1}$$



$R_{\pi_1}(x) =$
$$x \geq 0 \wedge x - 1 < 1 \wedge x - 1 \neq 1$$
$$\equiv 0 \leq x < 2$$
$T_{\pi_1}(x) = 0$

$R_{\pi_2}(x) =$
$$x \geq 0 \wedge x - 1 \leq 2 \wedge x - 1 \neq 1$$
$$\equiv 0 \leq x \leq 3 \wedge x \neq 2$$
$T_{\pi_2}(x) = x$

Difference Summary

Difference for a pair of procedures p_1, p_2 is a triplet:

- **changed:** set of path differences

$$\boldsymbol{changed\_condition} = \bigvee_{(r,t_1,t_2)\in changed} r$$

- **termination_changed:** FOL formula for inputs where exactly one procedure terminates.
- **unchanged:** FOL formula for inputs where both procedures terminate with the same outputs, or both do not terminate.

$$\boldsymbol{changed\_condition} \vee \boldsymbol{temination\_changed} \vee \boldsymbol{unchanged} \equiv \boldsymbol{true}$$

Example

```
int p1(int x) {

    if (x < 0)
        return -1;
    x--;
    if (x >= 1)
        return x+1;
    else
        while ( x == 1);
    return 0;
}
```

```
int p2(int x) {

    if (x < 0)
        return -1;
    x--;
    if (x > 2)
        return x+1;
    else
        while ( x == 1);
    return 0;
}
```

Example

The difference summary is:

$$changed_{p_1,p_2} \coloneqq \{(x = 3, x, 0)\}$$

$$termination\_changed_{p_1,p_2} \coloneqq (x = 2)$$

$$unchanged_{p_1,p_2} \coloneqq (x < 2) \lor (x > 3)$$

Difference Summary - computation

- Difference summary is incomputable
- We compute under-approximations of changed and unchanged,

ignoring termination_changed for now:

  - A set $\boldsymbol{computed\_changed} \subseteq changed$
  - A condition $\boldsymbol{computed\_unchanged} \rightarrow unchanged$

Difference Summary - computation

This gives us:

- An **under-approximation** of the difference:

$$\boldsymbol{computed\_changed\_condition} = \bigvee_{(r,t_1,t_2)\in computed\_changed} r$$

- An **over-approximation** of the difference:

$$\boldsymbol{may\_change} = \neg computed\_unchanged$$

Logical relative procedure summary

- To abstract unknown behavior of the program analyzed, we will use uninterpreted functions

- For each pair of matched procedures p_1, p_2 we have
  - a common uninterpreted function UF_{p_1,p_2}
  - individual uninterpreted functions UF_{p_1}, UF_{p_2} as well

Logical relative procedure summary

Given two matching procedures p_1, p_2, their , we can construct their **logical relative summary:**

$$logSum_{p_1}(\bar{x},\bar{x}') = \left(\bigwedge_{(r,t)\in sum_{p_1}} r(\bar{x}) \rightarrow \bar{x}' = t(\bar{x})\right) \wedge \left(computed\_unchanged_{p_1,p_2}(\bar{x}) \rightarrow \bar{x} = UF_{p_1,p_2}(\bar{x})\right) \wedge \left(\neg computed\_unchanged_{p_1,p_2}(\bar{x}) \rightarrow \bar{x} = UF_{p_1}(\bar{x})\right)$$

$$logSum_{p_2}(\bar{x},\bar{x}') = \left(\bigwedge_{(r,t)\in sum_{p_2}} r(\bar{x}) \rightarrow \bar{x}' = t(\bar{x})\right) \wedge \left(computed\_unchanged_{p_1,p_2}(\bar{x}) \rightarrow \bar{x} = UF_{p_1,p_2}(\bar{x})\right) \wedge \left(\neg computed\_unchanged_{p_1,p_2}(\bar{x}) \rightarrow \bar{x} = UF_{p_2}(\bar{x})\right)$$

These summaries will be used to replace each call to the procedures.

Guiding symbolic execution – bottom up

p1=p2=p

```
int p(int x) {
    if (x > 1) {
        int y = -x;
        while (x > 1) {
            x--;
            y = y – x;
        }
    }
    x--;
    if (  x<2 ) {
        if ( f(x) )
            return x+1;
    }
    return 0;
}
```

```
bool f1(int x) {
    return x >= 1;
}
```

$\rightarrow$

```
bool f2(int x) {
    return x > 2;
}
```

$changed_{f_1,f_2} =$

$\{(x = 1 \vee x = 2, true, false)\}$

$termination\_changed_{f_1,f_2} = false$

$unchanged_{f_1,f_2} = x \neq 1 \wedge x \neq 2$

Guiding symbolic execution – bottom up

- For each location in p compute a condition that guarantees that:
  - no different behavior is reachable from here

- Compute the weakest precondition of $computed\_unchanged_{f1,f2}$ from $call\ f$
  - weakest condition guaranteeing f is not called with inputs satisfying $may\_change_{f_1,f_2}$

- Use computed condition to restrict symbolic execution

Guiding symbolic execution – bottom up

p1=p2=p

```
int p(int x) {
    if (x < 2) {
        int y = -x;
        while (x < 1) {
            x++;
            y = y – x;
        }
    }
    x--;
    if (  x<2 ) {
        if ( f(x) )
            return x+1;
    }
    return 0;
}
```

$\boldsymbol{x = 2}$

$\boldsymbol{x = 2}$

$\boldsymbol{x = 2}$

$\boldsymbol{x = 2}$

$\boldsymbol{x = 2}$

$\boldsymbol{(x = 1 \vee x = 2) \wedge x < 2 \equiv x = 1}$

$\boldsymbol{x = 1 \vee x = 2}$

$$unchanged_{f_1,f_2} = x \neq 1 \wedge x \neq 2$$

Demand-driven refinement – top down

Since we are using uninterpreted functions, the discovered difference may not be feasible:

```
int p1(int x) {
    if (x == 5)
        if (abs1(x)==0)
            return 0;

    return x;
}
```

abs1=abs2=abs

```
int p2(int x) {
    if (x == 5)
        if (abs2(x)==0)
            return -1;

    return x;
}
```

$$changed_{p_1,p_2} := \{(x = 5 \wedge UF_{abs_1,abs_2}(x) = 0, 0, -1)\}$$

Summary

We present a differential analysis method that is:

- Modular (analyses each procedure independently of its current use)
- Incremental
- Treats loops
- Computes over-approximation and under-approximation of inputs that produce different behavior
- Introduces abstraction in form of uninterpreted functions, and allows demand driven refinement

Thank you

QUESTIONS?