Automated Testing & Verification

Interprocedural Dataflow Analysis

Galeotti/Gorla/Rau
Saarland University

How to handle many methods?

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
   [y := divByX(5)]3;
   [y := divByX(1)]4;
}
```

- How do we know "divByX" does not fail?

```
float area(Square c) {
    [result := c.l*c.l]1;
}

void caller1(Square c2) {
   [Square c = new Square()]1;
   [float a1= area(c)]2;
   [return area(c2)+a1]3;
}
```

- How do we know "area" does not fail?

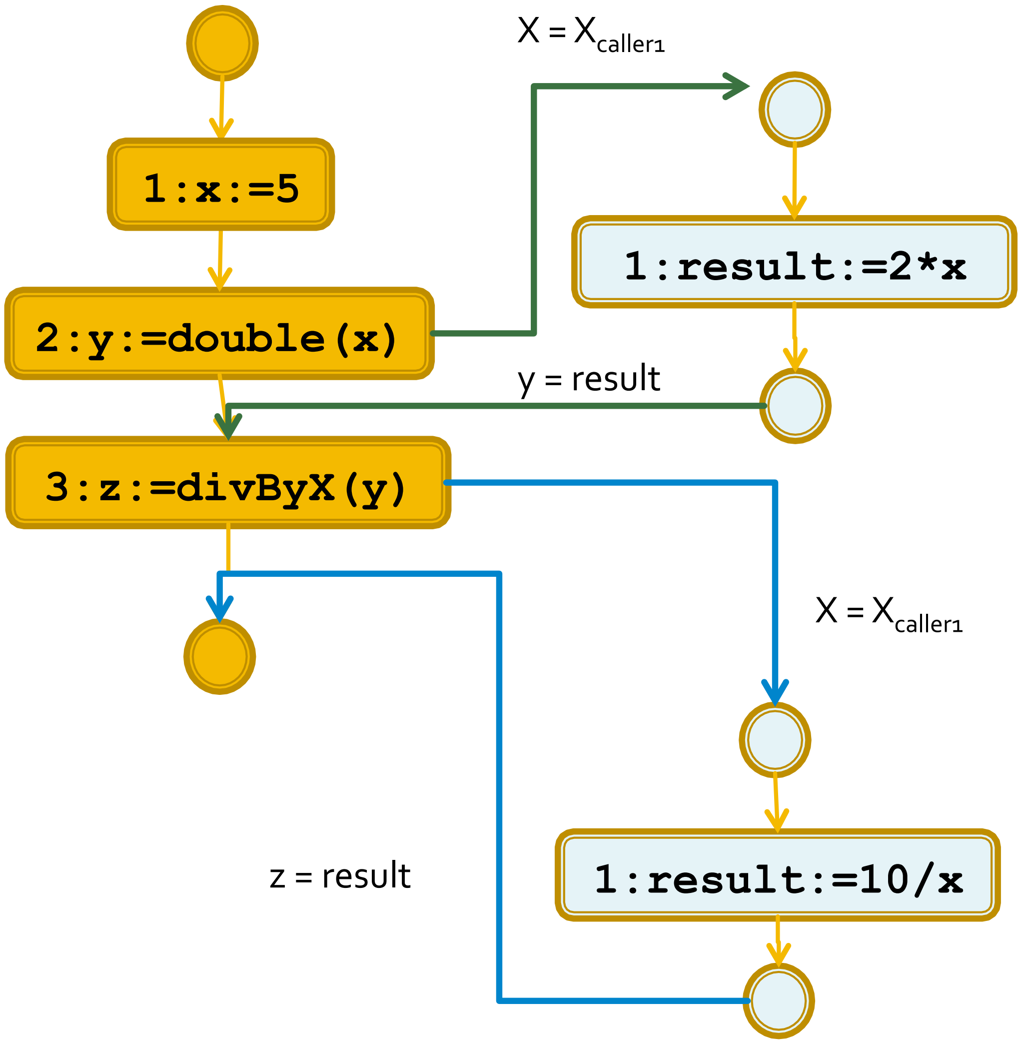
Interprocedural Dataflow Analysis

- Analyze a program with many methods
- Strategies:
  - **Build an interprocedural CFG**
  - Assume/Guarantee
  - Context sensitivity
    - Inlining
    - Call string
    - Compute "summaries"

Interprocedural CFG

- Extend CFG for many procedures
  - Add an edge from the **caller** to the **callee**'s *entry* node
  - Add an edge from the return node to the call's next node

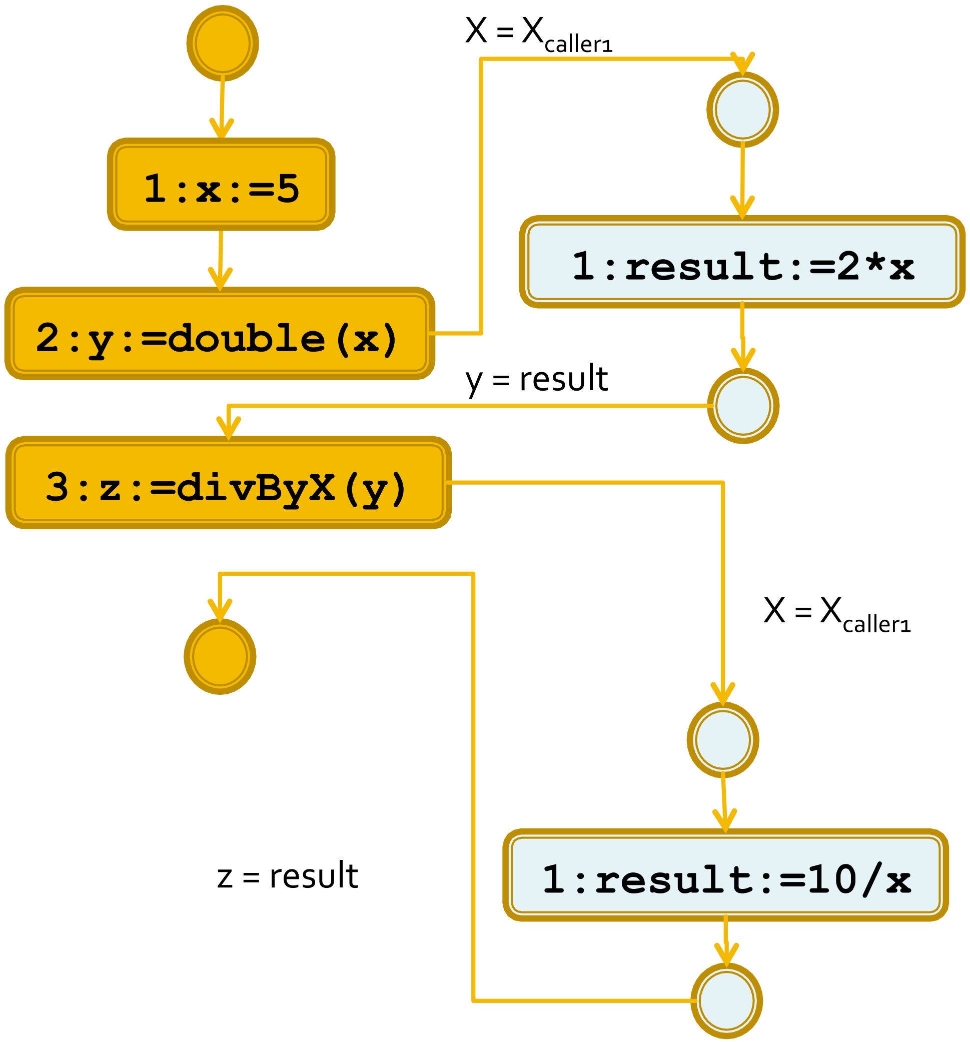
```
int double(int x) {
    [result := 2*x]₁;
}
int divByX(int x) {
    [result := 10/x]₁;
}
void caller1() {
   [x := 5]₁;
   [y := double(x)]₂;
   [z := divByX(y)]₃;
}
```



Interprocedural CFG

- caller 1 Analysis

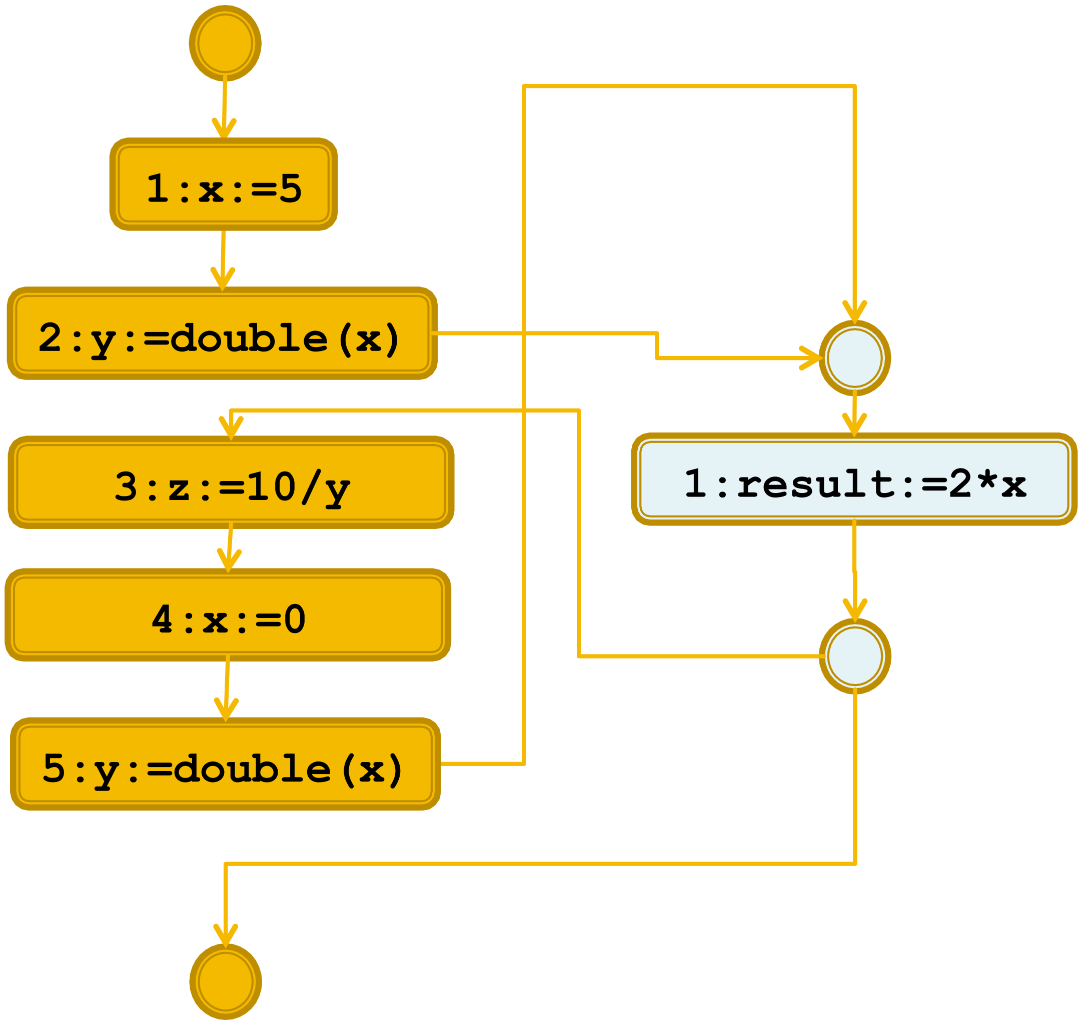
	caller1			double		divByX	
Pos	x	y	z	x	res	x	res



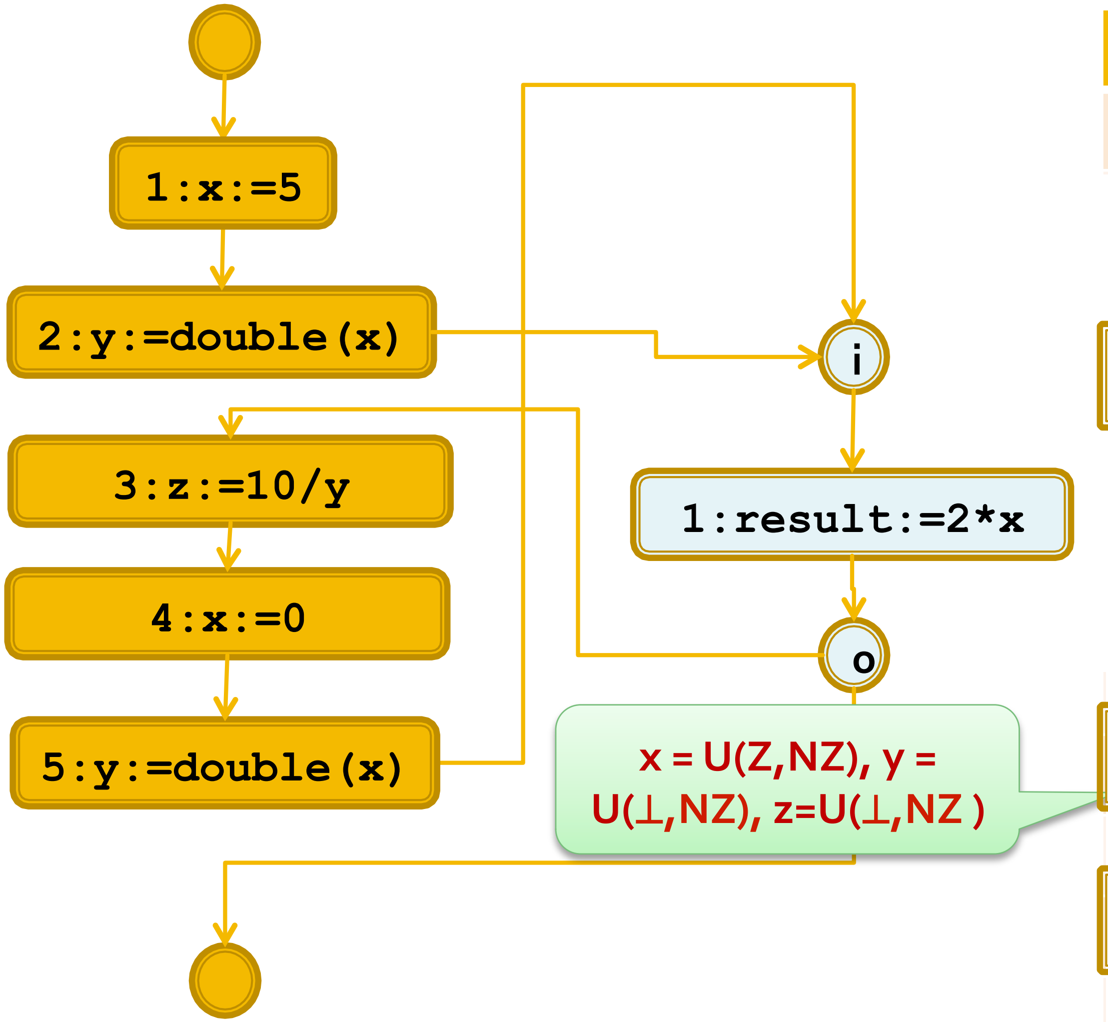
Another example

```
int double(int x) {
     [result := 2*x]₁;
}
void caller1() {
    [x := 5]₁;
    [y := double(x)]₂;
    [z := 10/y]₃
    [x := 0]₄;
    [y := double(x)]₅;
}
```

Any problem with this?



Another example

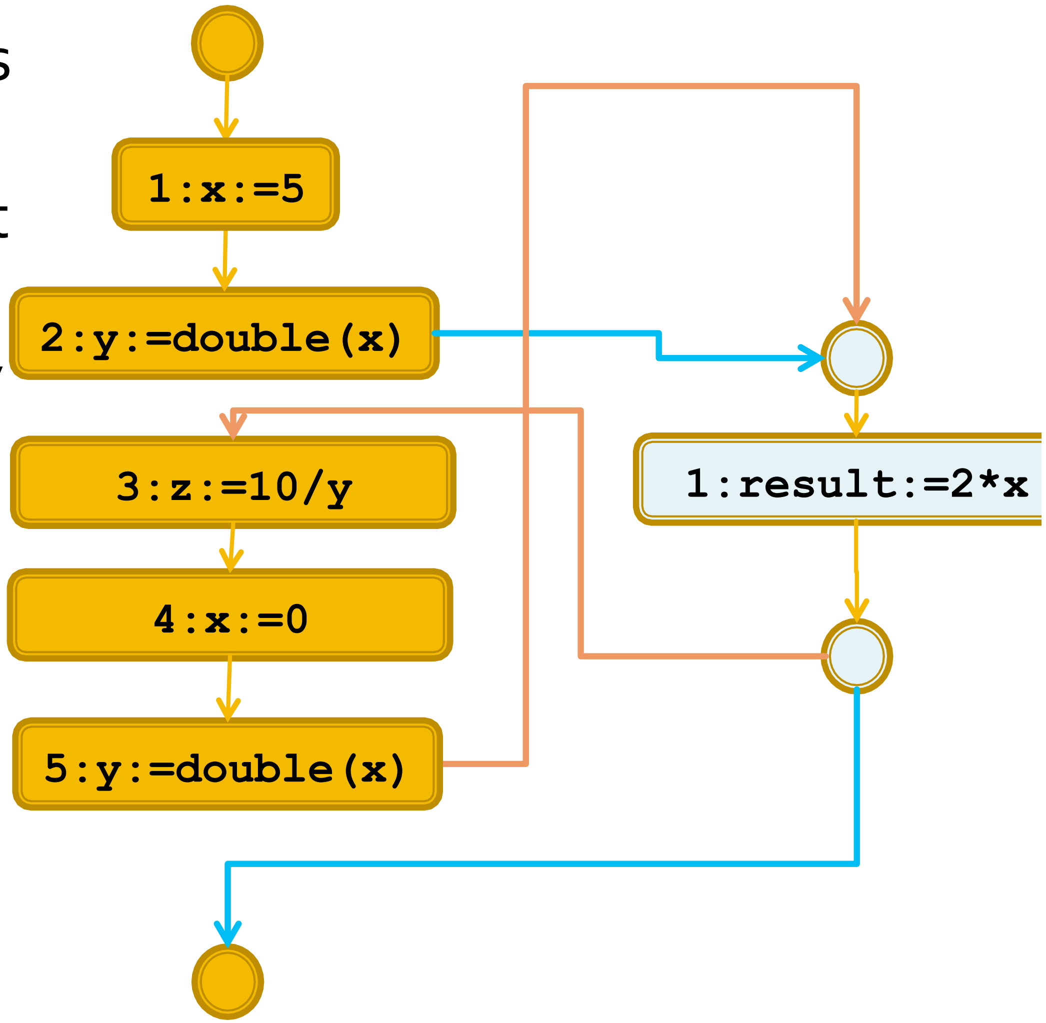


	Caller			double	
Pos	x	y	z	x	Res

- Warning: div by zero!
- False positive!

What was the problem?

- The interprocedural CFG loses precision
- It can not distinguish different context calls.
- In particular, information may flow through infeasible paths



Context Sensibility

- We have to distinguish among several context calls
- Many techniques for this:
  - **Inlining/Cloning**
  - Call chains
  - Assume/ Guarantee
  - Method Summaries

Inliling/Cloning

- Idea: Use dataflow, but rule out incorrect program paths
- How?
  - **Inlining**: Include the callee's code within the caller
    - The program has an unique method now
    - Instantiate input/output parameters
  - **Cloning**: Create a copy of the method for each invocation.
    - Each copy represents a different context

Inlining/Cloning

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
   [y := divByX(5)]3;
   [y := divByX(0)]4;
}
```

Cloning

```
void caller1() {
   [x := 5]1;
   [y := divByX_1(x)]2;
   [y := divByX_2(5)]3;
   [y := divByX_3(0)]4;
}
int divByX_1(int x) {
    [result := 10/x]1;
}
int divByX_2(int x) {
    [result := 10/x]1;
}
int divByX_3(int x) {
    [result := 10/x]1;
}
```

Inlining

```
void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
       y := 10/x;
   [y := divByX(5)]3;
       y := 10/5;
   [y := divByX(0)]4;
      y := 10/0;
}
```

- Advantages:
  - More precision
    - A version of the method for each context
- Disadvantages
  - More computational cost
    - Code explosion

Problems with Inliling/Cloning

- Interface usage

```
Interface I
{
   int getValue();
}
Class A implements I…
Class B implements I…

void process() {
   I i = null;
   if(…)
     i = new A();
   else
     i = new B()

    i.getValue();
}
```

```
Interface I
{
   int getValue();
}

void process(I i) {
   i.getValue();
}
```

- Recursion

```
void process(int v, int x) {
   if(v==0) return x;
   else return process(v-1,x*v)
}
```

Context Sensibility

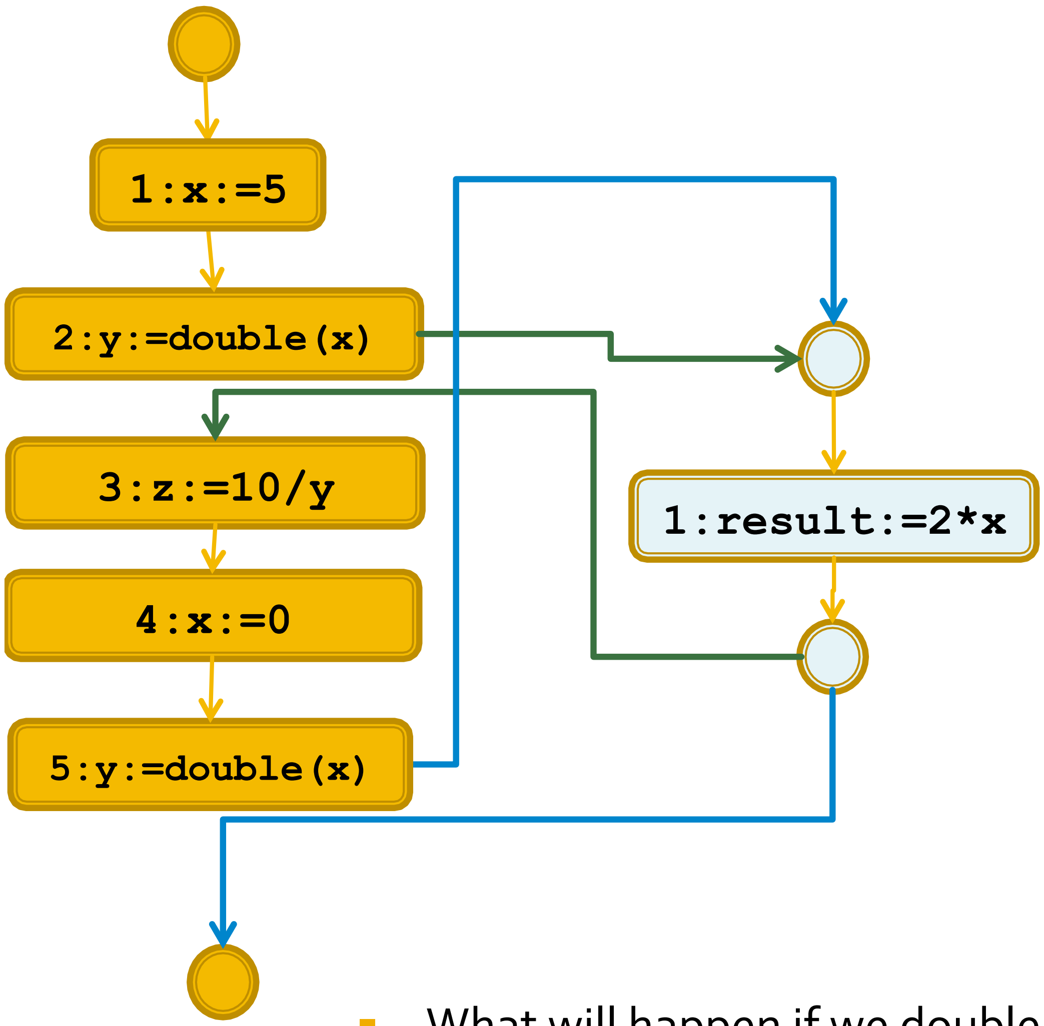
- We have to distinguish among several context calls
- Many techniques for this:
  - Inlining/Cloning
  - **Call chains**
  - Assume/ Guarantee
  - Method Summaries

Call chains

- During runtime, different method invocations are distinguished throught the call stack
  - In particular we might distinguished them throught control labels (example: *m1.4+m2.1+....*)

- Disadvantage
  - The stack might be unbounded
    - Recursion
- Idea:
  - Use only the last ***k*** calls for distinguishing contexts
  - We call this "k-limit"

Revisited example – 1 limit



	Caller				double	
Pos	x	y	z	CC	x	Res

- What will happen if we double calls to another method?
  - `int double(int x) { return g(x); }`
  - `int g(int x) { return (2*x); }`

Recursion & call chains

```
int double(int x) {
     [result := 2*x]1;
}
int g(int v)
{
   if(v>2)
     [return g(v-1)]1T
   else
     [return double(v)]1F
}
void m(int x) {
    [y := g(x)]1;
    [z := 10/y]2;
}
```

- Is it possible to find a “k” that allows us to precisely handle this problem?
  - m_1.g_1F.double
  - m_1. g_1T.g_1F.double
  - m_1. g_1T. g_1T.g_1F.double
  - …
- Call chains require approximation in case of recursive calls

Call chains

Recap:

- We introduce to the abstract value the notion of context using a chain that models the last invokations
  - It is known as ***k*-limiting**
- For non-recursive programs it could be precise enough
  - Although it is not as precise as possible! Why?
    - (loops!)
- For recursive programs contexts we have to approximate
  - Example: m_1. (g_1T)*.g_1F.double
- Although it seems a rather obvious solution, it is not widely used
  - Cost: multiplies the number of states given the number of paths (exponential!)
  - Depends on having a good call-graph approximation

Context Sensibility

- We have to distinguish among several context calls
- Many techniques for this:
  - Inlining/Cloning
  - Call chains
  - **Assume/Verify**
  - Method Summaries

Assume/Verify

- Idea: annotate each method with information about what it assumes and what it guarantees
  - **Precondition**: Initial values for all parameters
  - **Postcondition**: A return value (result)
  - Based on the programmer knowledge
    - Individual knowledge for each method.
  - Or by "default"
    - example: every "equals" implementation

- Verification
  - In the annotated method
    - Assume all values for parameters
    - Verify in the annotated method that the result $\subseteq$ $\text{assumed}_{\text{result}}$
  - In the caller method
    - Verify arg$\subseteq$ $\text{assumed}_{\text{arg}}$
      - Actual parameters satisfy assumptions on formal parameters
    - Use the annotated valued as result.

Assume by default

- Example: Zero analysis
  - Default: **MZ** (top) for all arguments and results
    - Meaning: The method can receive any value and return any value
  - Benefit: All parameters and result satisfy this assumption
  - Cost: Too conservative wrt. the actual input/output
    - Many false positives

Example

- We assume x must be **MZ** and result is **MZ**

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
    [x := 5]1;
    [y := divByX(x)]2;
}
```

- divByX analysis

pos	x	Result
0	**MZ**	$\perp$
1	MZ	**NZ**

  - It holds $\sigma(\text{result}) \subseteq \text{MZ}$
  - Warning: div by zero at #1

- Caller1 Analysis

pos	x	y
0	$\perp$	$\perp$
1	NZ	$\perp$

- It holds $\sigma(x) \subseteq \text{MZ}$

- Problem: div by zero is not possible!

Optimistic assumptions

- We assume x must be **NZ** and result must be **NZ**

```
int divByX(int x) {
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- divByX analysis

pos	x	Result
0	**NZ**	$\perp$
1	NZ	**NZ**

  - No warning s
  - It holds σ(result)$\subseteq$**NZ**

- Caller1 analysis

pos	x	y
0	$\perp$	$\perp$
1	NZ	$\perp$

- It holds σ(x)$\subseteq$NZ

Optimistic assumptions

- We assume x must be **NZ** and result is **NZ**

```
int double(int x) {
    [result := 2*x]₁;
}

void caller1() {
   [x := 0]₁;
   [y := double(x)]₂;
}
```

- Double analysis

pos	x	Result
0	NZ	$\perp$
1	NZ	**NZ**

- It holds $\sigma(\text{result}) \subseteq NZ$

- Caller1 analysis

pos	x	y
0	$\perp$	$\perp$
1	**Z**	$\perp$

- **$\sigma(x) \subseteq NZ$ fails!**
- **It does not satisfy the assumption**
- It is a false positive, although the program is OK.

Annotate & Check

- Instead of defaults, we use method-level annotations
  - The programmer provides these annotations
- Annotation
  - **Precondition**: Initial abstract values for all parameters
  - **Postcondition**: A return value (result)
- Verification
  - In the annotated method
    - Assume all values for all parameters
    - Verify that result $\subseteq$ assumed_result
  - In the caller:
    - Verify that args $\subseteq$ assumed_args
      - Actual parameters satisfy the assumption on formal parameters
    - Use annotated value as result.

Example user-defined annotations

```
@NZ int divByX(@NZ int x)
{
    [result := 10/x]1;
}

void caller1() {
   [x := 5]1;
   [y := divByX(x)]2;
}
```

- divByX Analysis

pos	x	Result
0	NZ	$\perp$
1	NZ	NZ

  - It holds $\sigma(\text{result}) \subseteq NZ$

- Caller1 analysis

pos	x	y
0	$\perp$	$\perp$
1	NZ	$\perp$

- It verifies that $\sigma(x) \subseteq NZ$

Example

```
@NZ int double(@NZ int x)
{
     [result := 2*x]1;
}

void caller1() {
    [x := 0]1;
    [y := double(x)]2;
}
```

- double analysis

pos	x	Result
0	NZ	$\perp$
1	NZ	NZ

  - It holds $\sigma(\text{result}) \subseteq NZ$

- Caller1 analysis

pos	x	y
0	$\perp$	$\perp$
1	Z	$\perp$

- **$\sigma(x) \subseteq NZ$ fails!**
- **The double precondition is not met**
- It is a false positive, although the program is OK

Example

```
@MZ int double(@MZ int x)
{
    [result := 2*x]1;
}

void caller1() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3;
}
```

- Double Analysis

pos	x	Result
0	MZ	$\perp$
1	MZ	**MZ**

  - It holds $\sigma(result) \subseteq MZ$

- Caller1 analysis

pos	x	y	Z
0	$\perp$	$\perp$	$\perp$
1	**NZ**	$\perp$	$\perp$

- **$\sigma(x) \subseteq MZ$ is verified**
- **Warning: div by zero!**
- It is also a false positive!

Context Sensibility

- We have to distinguish among several context calls
- Many techniques for this:
  - Inlining/Cloning
  - Call chains
  - Assume/ Guarantee
  - **Method Summaries**

Context sensitive summaries

- Idea:
  - Compute a <u>unique summary for each method</u>
  - Map dataflow input information to dataflow output information
- Context sensitive
  - Given different input, they output <u>different</u> results
  - Instantiate the summary on each method invokation

Creating Summaries

- Abstract Summaries
  - Represent symbollically the effect of the function over the elements of the lattice

Summaries on demand

- Case x:NZ → result: NZ
- Case x:Z → result: Z

```
int double(int x) {
    [result := 2*x]1;
}
void caller1() {
    [x := 5]1;
    [y := double(x)]2;
    [z := 10/y]3
    [x := 0]4;
    [y := double(x)]5;
}
```

- Caller1 analysis

pos	x	Y	Z
0	⊥	⊥	⊥
1	**NZ**	⊥	⊥
3	NZ	NZ	**NZ**
4	**Z**	NZ	NZ

double: case x:NZ→result:NZ

pos	x	Result
0	NZ	⊥
1	NZ	**NZ**

double: case x:Z→result:Z

pos	x	Result
0	Z	⊥
1	Z	**Z**

Context sensitive annotations

```
@Case("x:NZ -> result:NZ")
@Case("x:Z -> result:Z")
int double(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := double(x)]5;
}
```

- Verify for caller 1
- Case x:NZ

Pos	x	y	Z
1	NZ	⊥	⊥

- Verify for caller 1
- Case x:Z

4	Z	NZ	NZ

Verify: case x:NZ→result:NZ

pos	x	Result
0	NZ	⊥
1	NZ	**NZ**

verify: case x:Z→result:Z

pos	x	Result
0	Z	⊥
1	Z	**Z**

Parametric summaries

Summary:
Case x: A → result: A

```
int double(int x) {
    [result := 2*x]1;
}
void caller1() {
   [x := 5]1;
   [y := double(x)]2;
   [z := 10/y]3
   [x := 0]4;
   [y := double(x)]5;
}
```

double:

pos	x	Result
0	A	⊥
1	A	**A**

- Caller1 Analysis

pos	x	y	Z
0	⊥	⊥	⊥
1	**NZ**	⊥	⊥
3	NZ	NZ	**NZ**
4	**Z**	NZ	NZ

A=NZ

A=Z

Computing summaries

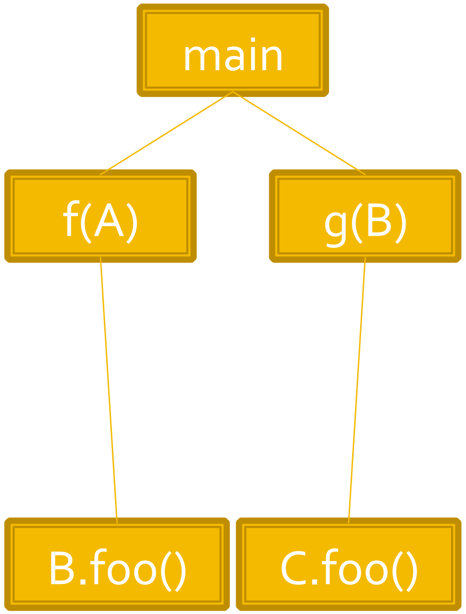
- The call-graph is traversed in a Bottom-Up walk:
  - Starting by leafs (no further method invokations)
    - Perform intraprocedural dataflow analysis
    - Store summary
  - When a method makes a call
    - Look for summary (bottom-up traverse ensures it exists)
    - Instantiate it with actual parameters (top-down)
- For recursive methods (or cycles) requires another fix-point
  - On the recursive subcomponent
    - Cycle in the Call Graph

Call Graph

- A “map” to know which methods to analyze
  - Paramount in object oriente programs

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```

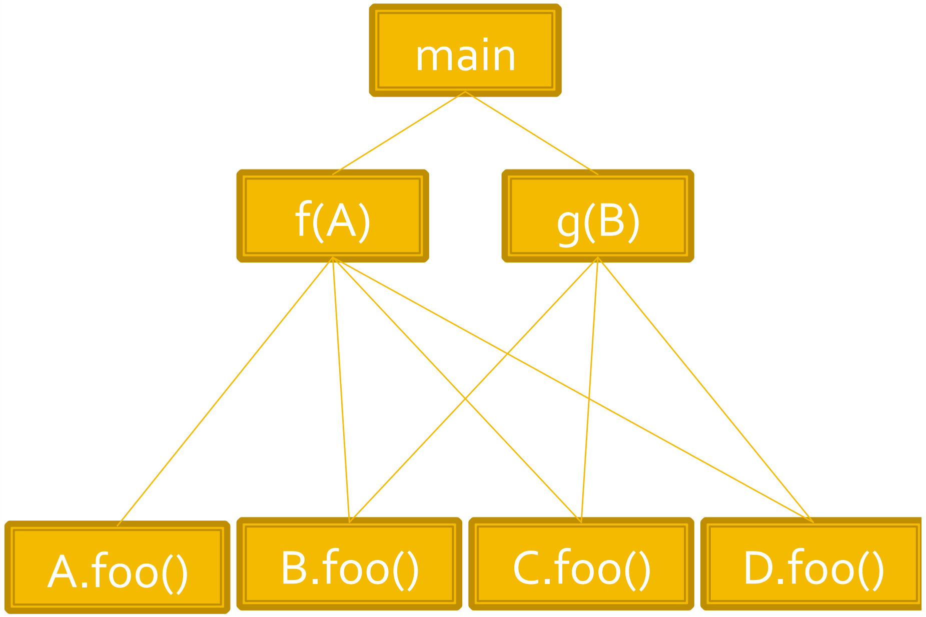


Call Graph

- A "map" to know which methods to analyze
  - Paramount in object oriente programs

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```



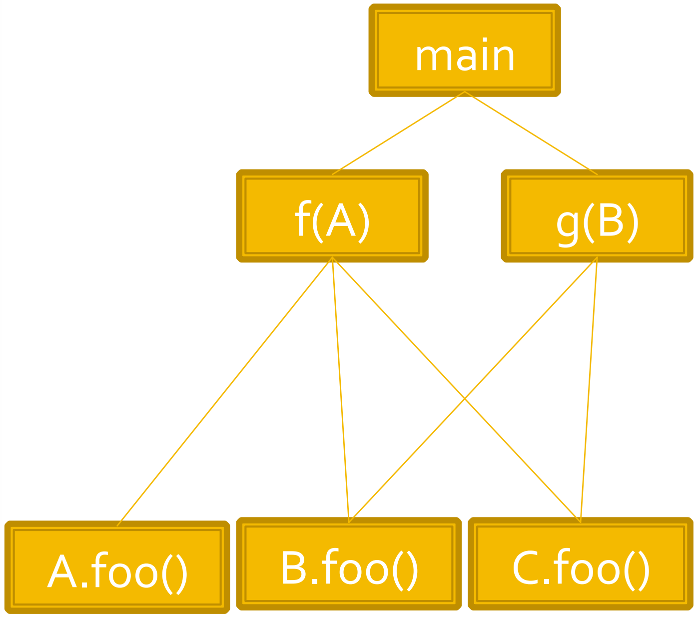
Computed using Class Hierarchy Analysis (CHA)

Call Graph

- A “map” to know which methods to analyze
  - Paramount in object oriente programs

```
static void main(){
   B b1 = new B();
   A a1 = new A();
   f(b1);
   g(b1);
}
static void f(A a2){
   a2.foo();
}
static void g(B b2){
   B b3 = b2;
   b3 = new C();
   b3.foo();
}
```

```
class A {
   foo(){..}
}
class B extends A{
   foo() {…}
}
class C extends B{
   foo() {…}
}
class D extends B{
   foo(){…}
}
```



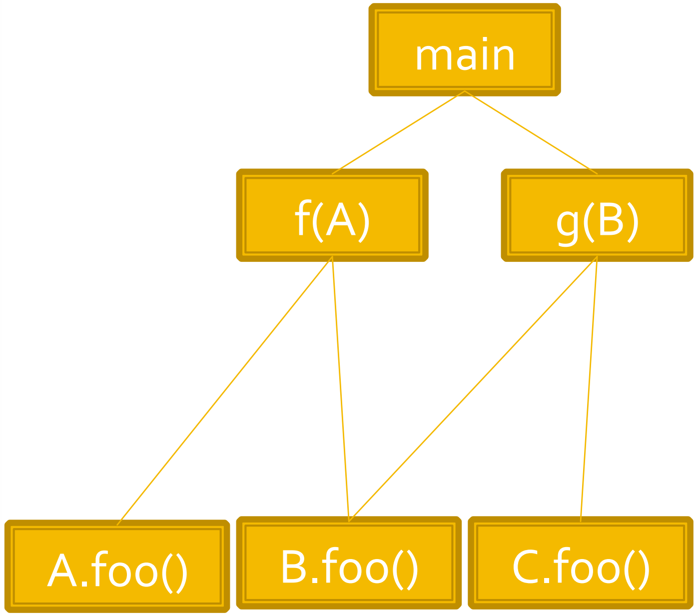
Computed using Rapid Type Analysis (RTA)

Call Graph

- A "map" to know which methods to analyze
  - Paramount in object oriente programs

```
static void main(){
  B b1 = new B();
  A a1 = new A();
  f(b1);
  g(b1);
}
static void f(A a2){
  a2.foo();
}
static void g(B b2){
  B b3 = b2;
  b3 = new C();
  b3.foo();
}
```

```
class A {
  foo(){..}
}
class B extends A{
  foo() {…}
}
class C extends B{
  foo() {…}
}
class D extends B{
  foo(){…}
}
```



Computed using XTA

Final remarks

- Assumptions
  - Simple & efficient
  - Imprecise (too general)
- Annotations
  - Requires effort
  - More precise than assumptions
  - More efficient than interprocedural analysis
- A whole-program analysis is not required!

- CFG Interprocedural
  - Easy to implement
  - Imprecise
  - Can be too expensive
  - As precise as simple
- Summaries
  - Very precise
  - Vert expensive if no abstraction is done
- They require whole-program analysis

Bibliography

- Compilers: Principles, Tecniques &Tools 2nd Edition: Aho, Lam, Sethi, Ullman
- Principles of Program Analysis . Flemming Nielson, Hanne Riis Nielson, Chris Hankin.
- Modern compiler implementation in Java. Andrew Appel. 2nd Edition.