# A FAST AND LOW-OVERHEAD TECHNIQUE TO SECURE PROGRAMS AGAINST INTEGER OVERFLOWS

RAPHAEL ERNANI RODRIGUES, VICTOR HUGO SPERLE CAMPOS,
FERNANDO MAGNO QUINTAO PEREIRA

fernando@dcc.ufmg.br

# The Objective of the Paper

THE GOAL OF THIS WORK IS TO DESIGN AND IMPLEMENT A NEW RANGE ANALYSIS ALGORITHM THAT WE SHALL USE TO ELIMINATE INTEGER OVERFLOW CHECKS FROM C/C++ PROGRAMS. CONSEQUENTLY, WE WILL INCREASE THE PERFORMANCE OF PROGRAMS THAT ARE GUARDED AGAINST INTEGER OVERFLOWS.

- Due to time constraints, in this presentation we will focus on our new range analysis algorithm.
- We will also talk a little bit about the important of securing programs against integer overflows.

# The Key Contribution

- A Range Analysis algorithm that improves on previous works in several ways:
  1. We handle comparisons between variables, e.g., `x > y`, without expensive relational lattices.
  2. We have a very fast implementation, that relies on techniques such as strongly connected components to analyze half a million constraints in less than 10 seconds.
  3. We use a live range splitting technique to improve the precision of our algorithm.
- And we have shown how to use this algorithm to eliminate some overflow checks in C programs.

# A Brief Overview About Integer Overflows

# What are Integer Overflows

- In many programming languages, integers are made of a finite number of bits.
- When we try to squeeze a value into one of these finite numbers, and the value is larger than the capacity of that type, then something funky may happen:

```
int main() {
  char i = 118;
  while (i < 125) {
    i += 5;
    printf("%8d", i);
  }
  printf("\n");
}
```

```
123   -128   -123   -118   -113   -108   -103    -98
-93    -88    -83    -78    -73    -68    -63    -58
-53    -48    -43    -38    -33    -28    -23    -18
-13     -8     -3      2      7     12     17     22
 27     32     37     42     47     52     57     62
 67     72     77     82     87     92     97    102
107    112    117    122    127
```

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $123 =$ | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | $= 123_{char}$ |
| $128 =$ | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | $= -128_{char}$ |
| $133 =$ | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | $= -123_{char}$ |

# Benign Integer Overflows

- Not every integer overflow is a bug. Programmers may use this semantics to good purposes.
- Integer overflows can be used to implement hash function.
- Integer overflows can be used to implement random number generators.
- In general, integer overflows are a simple and efficient way to implement modular arithmetics:

```
unsigned char c = 120, d = 118;
int i = c, j = d;
c * d = (i * j) % 128
```

# Malign Integer Overflows

```
void read_matrix(int* data, char w, char h) {
  char buf_size = w * h;
  if (buf_size < BUF_SIZE) {
    int c0, c1;
    int buf[BUF_SIZE];
    for (c0 = 0; c0 < h; c0++) {
      for (c1 = 0; c1 < w; c1++) {
        int index = c0 * w + c1;
        buf[index] = data[index];
      }
    }
    process(buf);
  }
}
```

This program has a security bug. Can you identify it?

# Malign Integer Overflows

```
void read_matrix(int* data, char w, char h) {
  char buf_size = w * h;
  if (buf_size < BUF_SIZE) {
    int c0, c1;
    int buf[BUF_SIZE];
    for (c0 = 0; c0 < h; c0++) {
      for (c1 = 0; c1 < w; c1++) {
        int index = c0 * w + c1;
        buf[index] = data[index];
      }
    }
    process(buf);
  }
}
```

BUF_SIZE = $120_{\text{char}}$

strlen(data) = $132_{\text{char}}$

buf_size = $-124_{\text{char}}$

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| w | = | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | = $6_{\text{char}}$ |
| h | = | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | = $22_{\text{char}}$ |
| h * w | = | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | = $-124_{\text{char}}$ |

# How do people handle integer overflows?[♤]

- There are many ways to deal with integer overflows:
  - We can use them conscientiously, taking benefit of its semantics.
  - We can do nothing about them.
  - We can write programs that will never cause integer overflows. This property can be proved statically.
  - We can use programming languages that give programmers resources to handle them dynamically.
  - We can insert checks guarding arithmetic operations.

These checks, of course slow down the program, but not too much[♧]: around 5%

[♤]: Understanding integer overflow in C/C++, ICSE

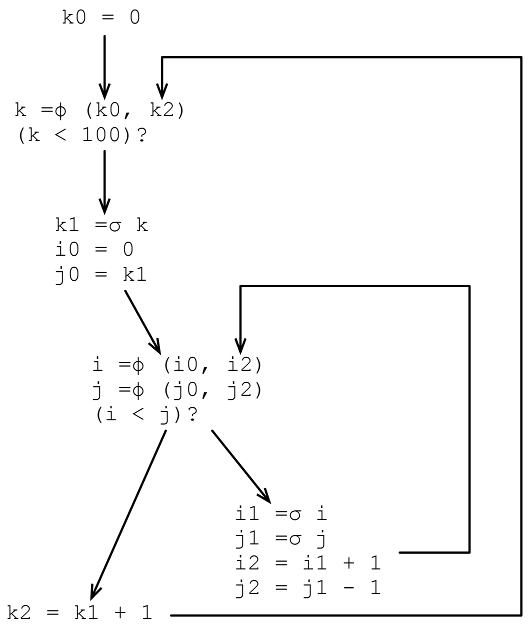
[♧]: RICH: Automatically Protecting Against Integer-Based Vulnerabilities, USENIX

# Who cares about 5%?

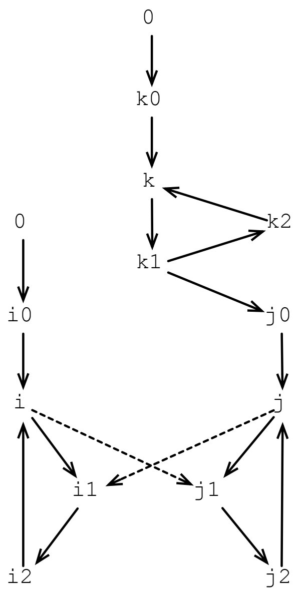
- Brumley *et al*[♧]. have shown that securing a program against integer overflows slows down runtime by no more than 5% on the average.
- Our range analysis technique brings this slow down to under 2.5%.
  - These gains have been verified over large benchmarks.
- These gains are cumulative with improvements in hardware.
- Besides, our range analysis, *which we humbly believe is the nicest in the world today*, can be used to enable many other compiler optimizations.

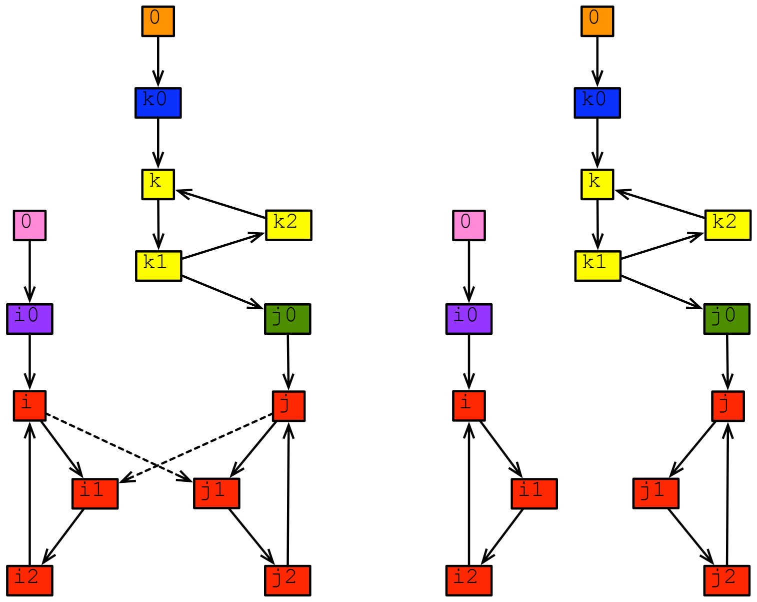
[♧]: RICH: Automatically Protecting Against Integer-Based Vulnerabilities, USENIX

# Range Analysis in One Example



```
k0 = 0

k =ϕ (k0, k2)
(k < 100)?

k1 =σ k
i0 = 0
j0 = k1

i =ϕ (i0, i2)
j =ϕ (j0, j2)
(i < j)?

i1 =σ i
j1 =σ j
i2 = i1 + 1
j2 = j1 - 1

k2 = k1 + 1
```



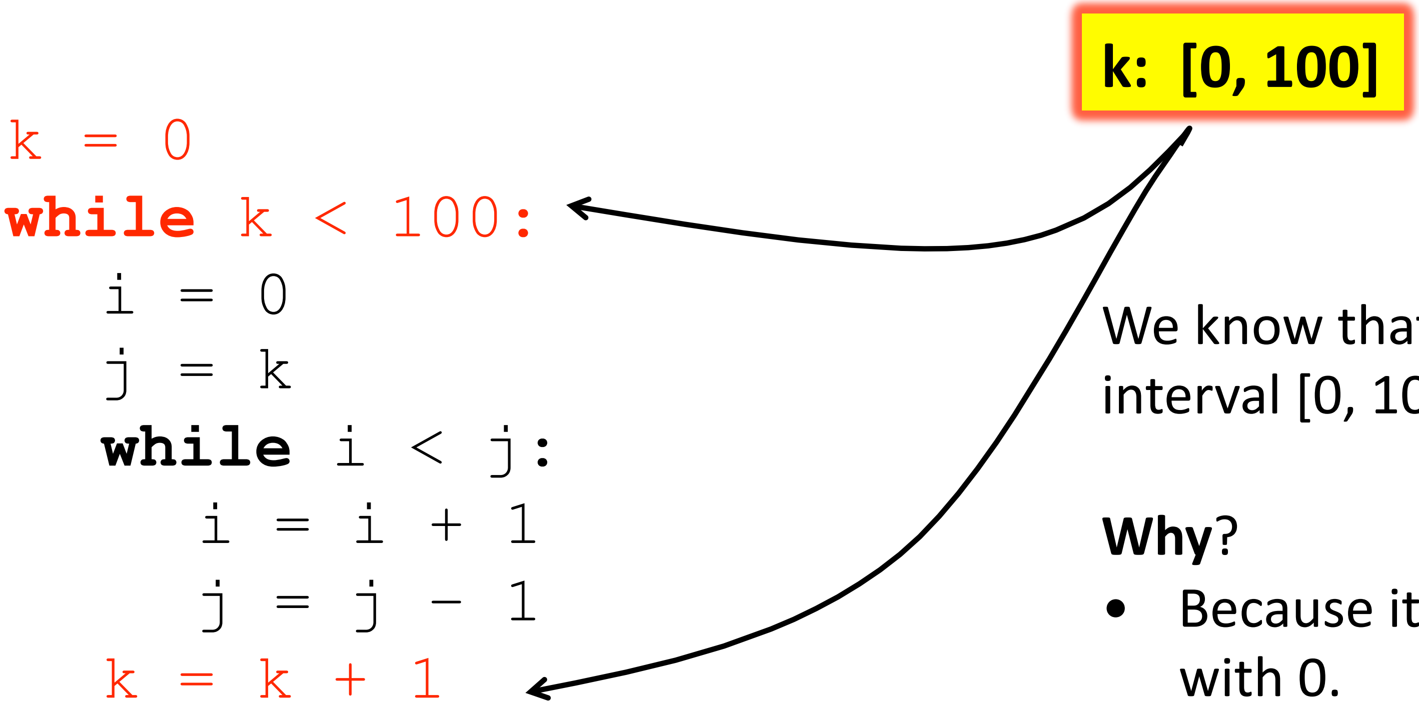


# Example

- We will use a small example to show how our range analysis works.
- This code has been adapted from the nested loop that we find in `quicksort.c`, one of the Stanford benchmarks.

```
k = 0
while k < 100:
  i = 0
  j = k
  while i < j:
    i = i + 1
    j = j - 1
  k = k + 1
```

# Example



```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```

**k: [0, 100]**

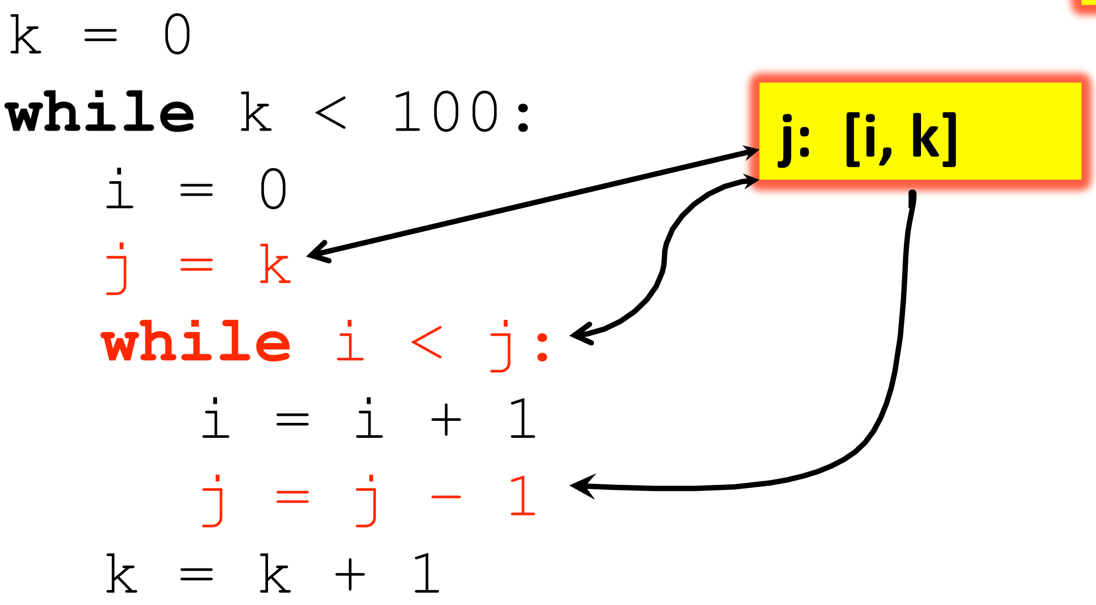
We know that k is in the interval [0, 100].

**Why**?

- Because it is initialized with 0.
- It is only updated through increments.
- It is bounded by k in the loop

# Example

**k: [0, 100]**



```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```

**j: [i, k]**

We know that j is in the interval [i, k].

**Why**?

- Because it is initialized with k.
- It is only updated through decrements.
- It is lower bounded by i in the loop

# Example

**k: [0, 100]**

```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```

**j: [i, 99]**

But we know more about j: it is upper bounded by 99.

**Why**?

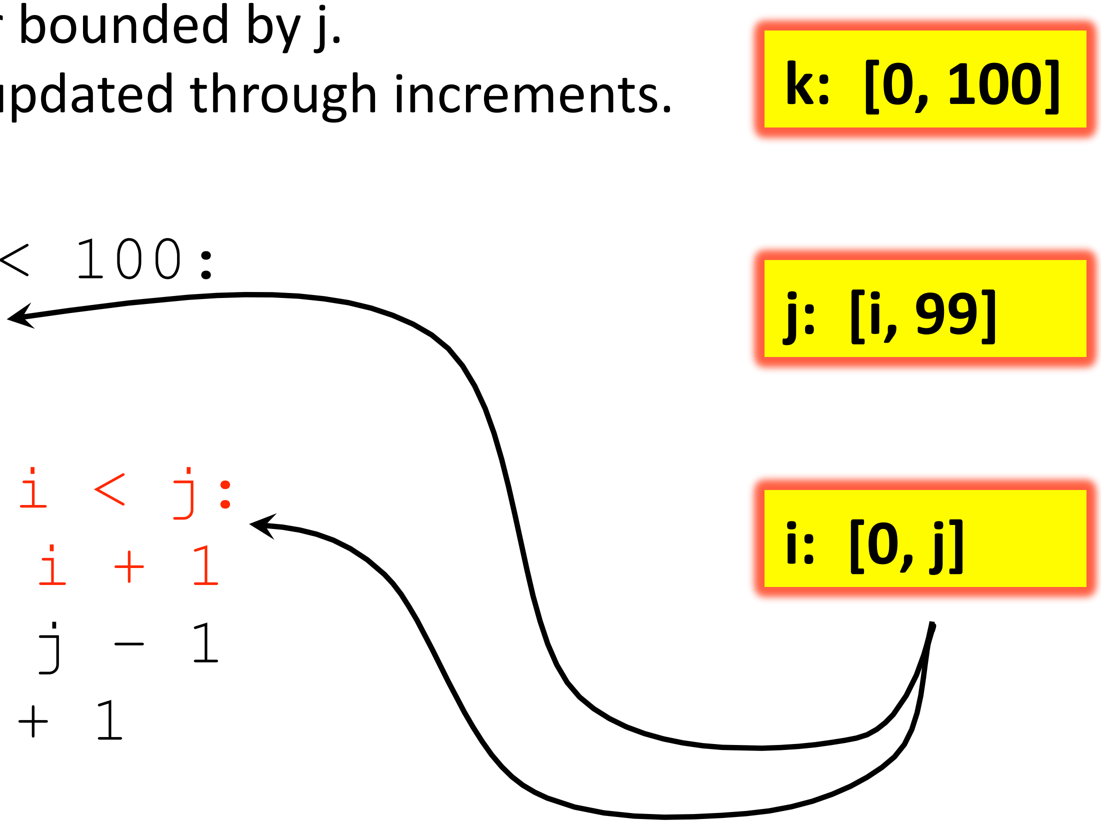
- Because it is upper bounded by k - 1.
- And we already know that k ≤ 100

# Example

We know that i ranges on [0, j]:

- It is initialized with 0.
- It is upper bounded by j.
- It is only updated through increments.

```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```



**k: [0, 100]**

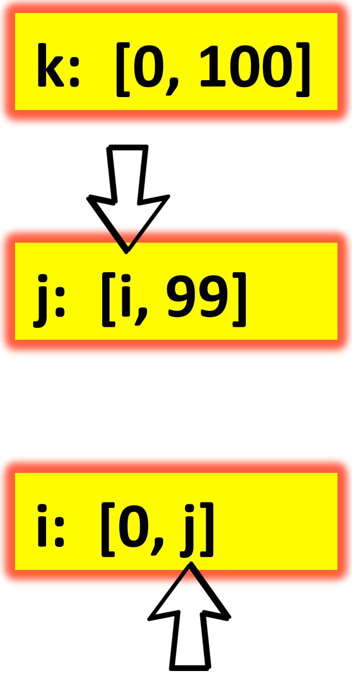
**j: [i, 99]**

**i: [0, j]**

# Example

We use the limits of variables i and j before we know their true ranges. This is the concept that we call **Futures**.

```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```



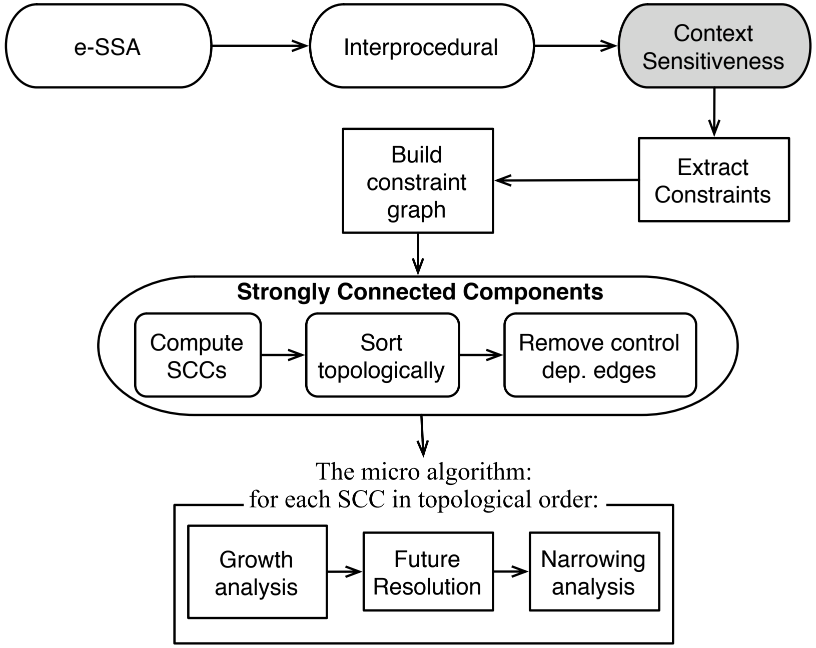
k: [0, 100]

j: [i, 99]

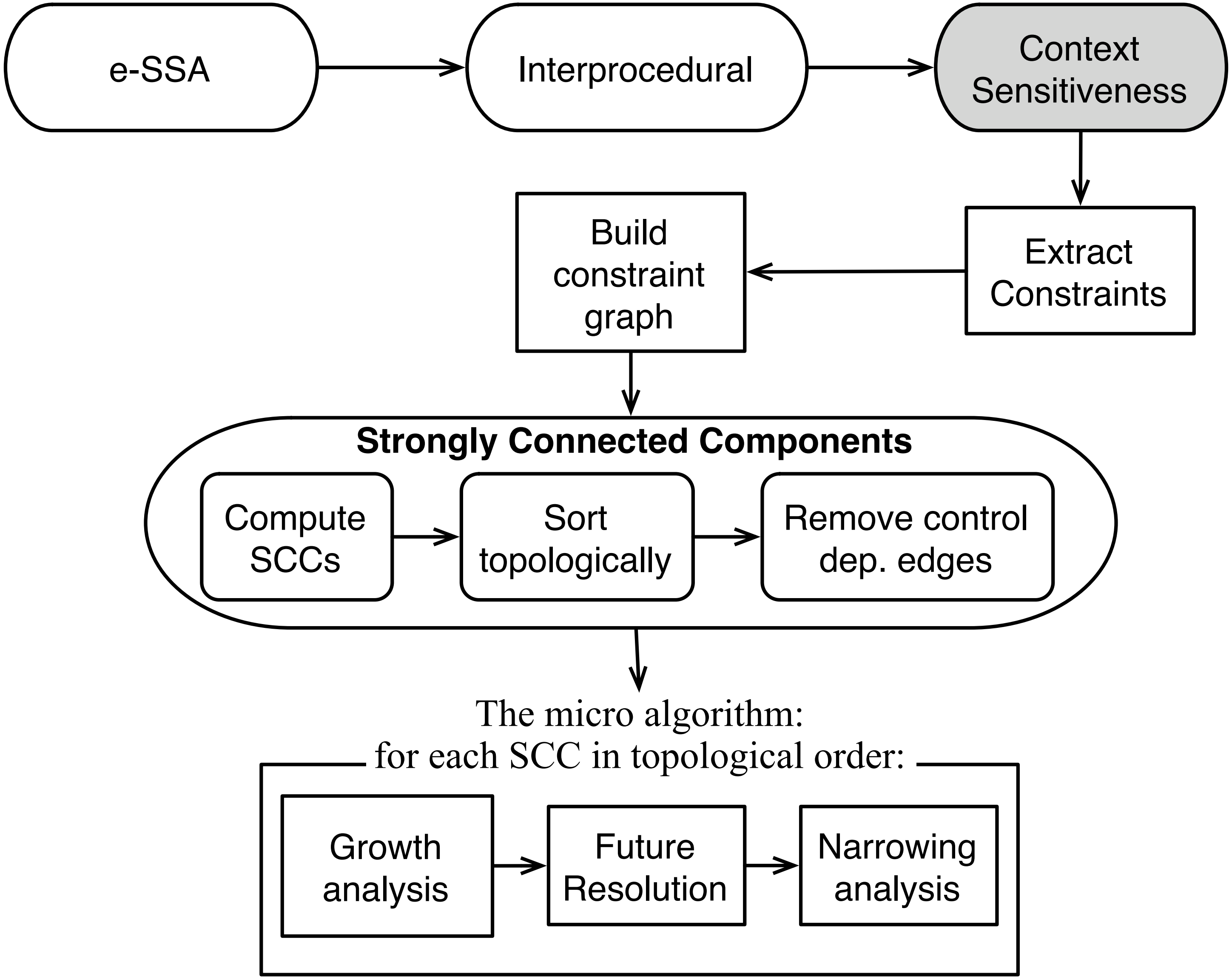
i: [0, j]

A future is like a *promise*: if we can find a good estimation for its value, than we can find a good fixed point solution to the interval analysis.

# THE RANGE ANALYSIS ALGORITHM



# The Overall Structure of the Algorithm



e-SSA

Interprocedural

Context Sensitiveness

Extract Constraints

Build constraint graph

**Strongly Connected Components**

Compute SCCs

Sort topologically

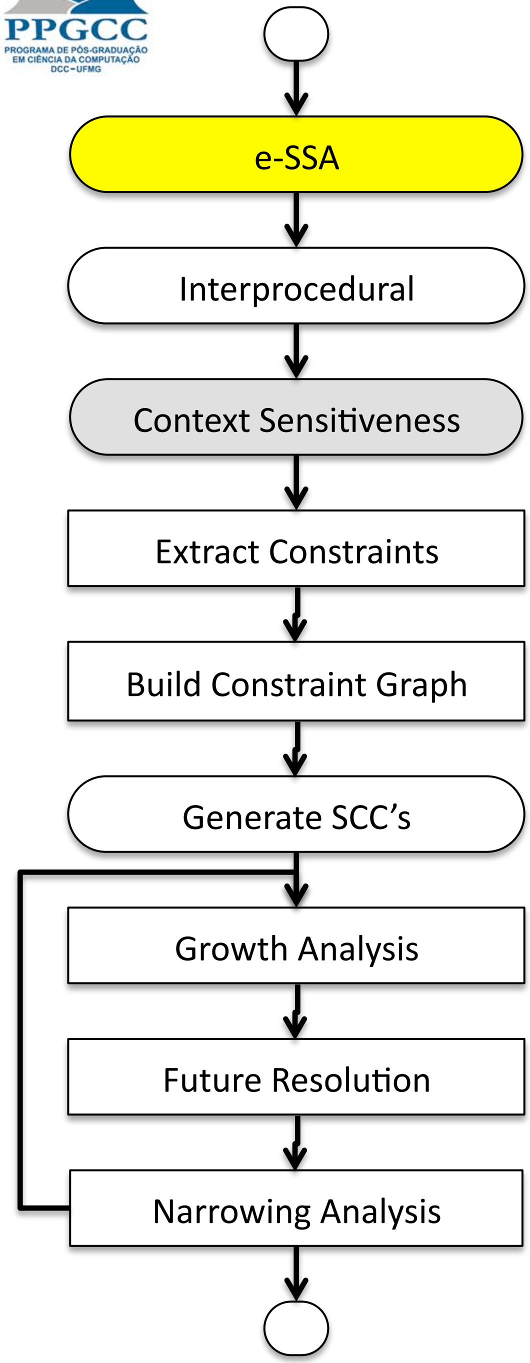
Remove control dep. edges

The micro algorithm:
for each SCC in topological order:

Growth analysis

Future Resolution

Narrowing analysis

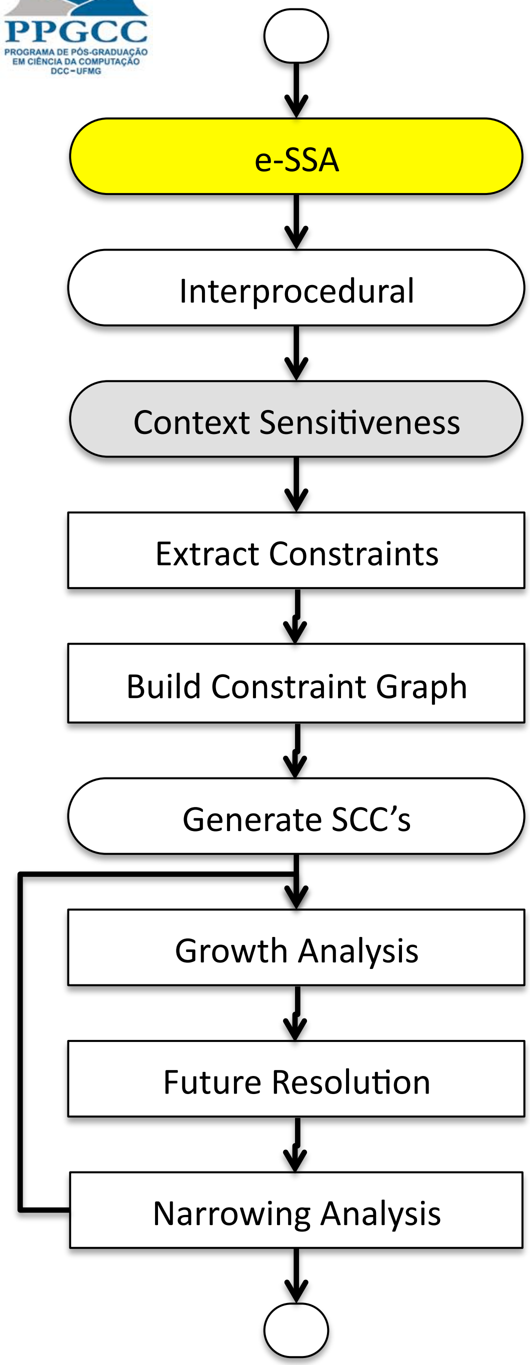


# Extended Static Single Assignment Form

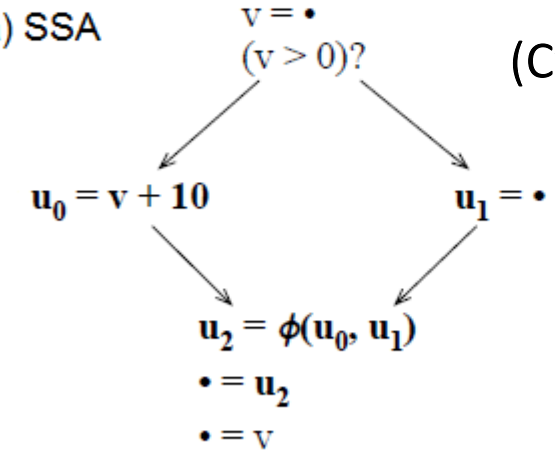
- The first step to solve range analysis is to convert the target program into e-SSA form[◊].
- This intermediate representation let us learn from conditional tests.
  - Hence, it improves precision of the range analysis.
- It increases the program size, but not too much.
  - Less than 10% on the average.

[◊]: ABCD: eliminating array bounds checks on demand (PLDI'00)

# Extended Static Single Assignment Form

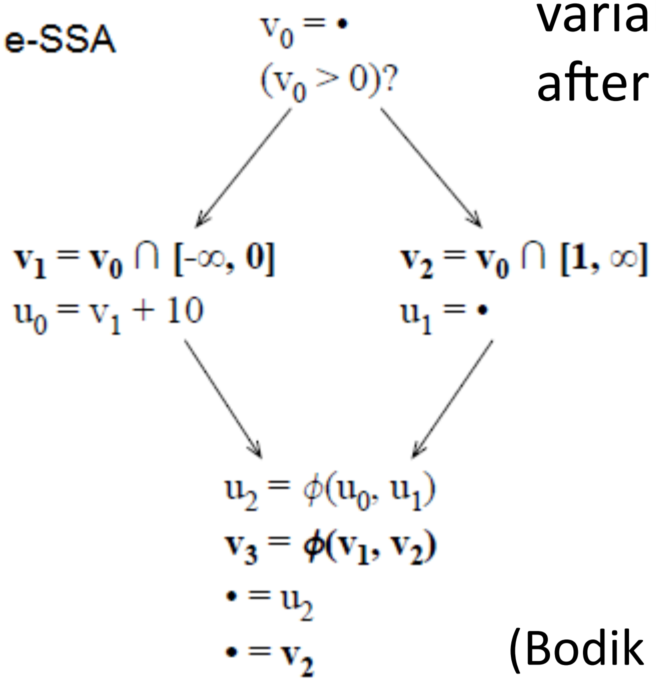


(a) SSA



(Cytron et al, 1991)

(b) e-SSA

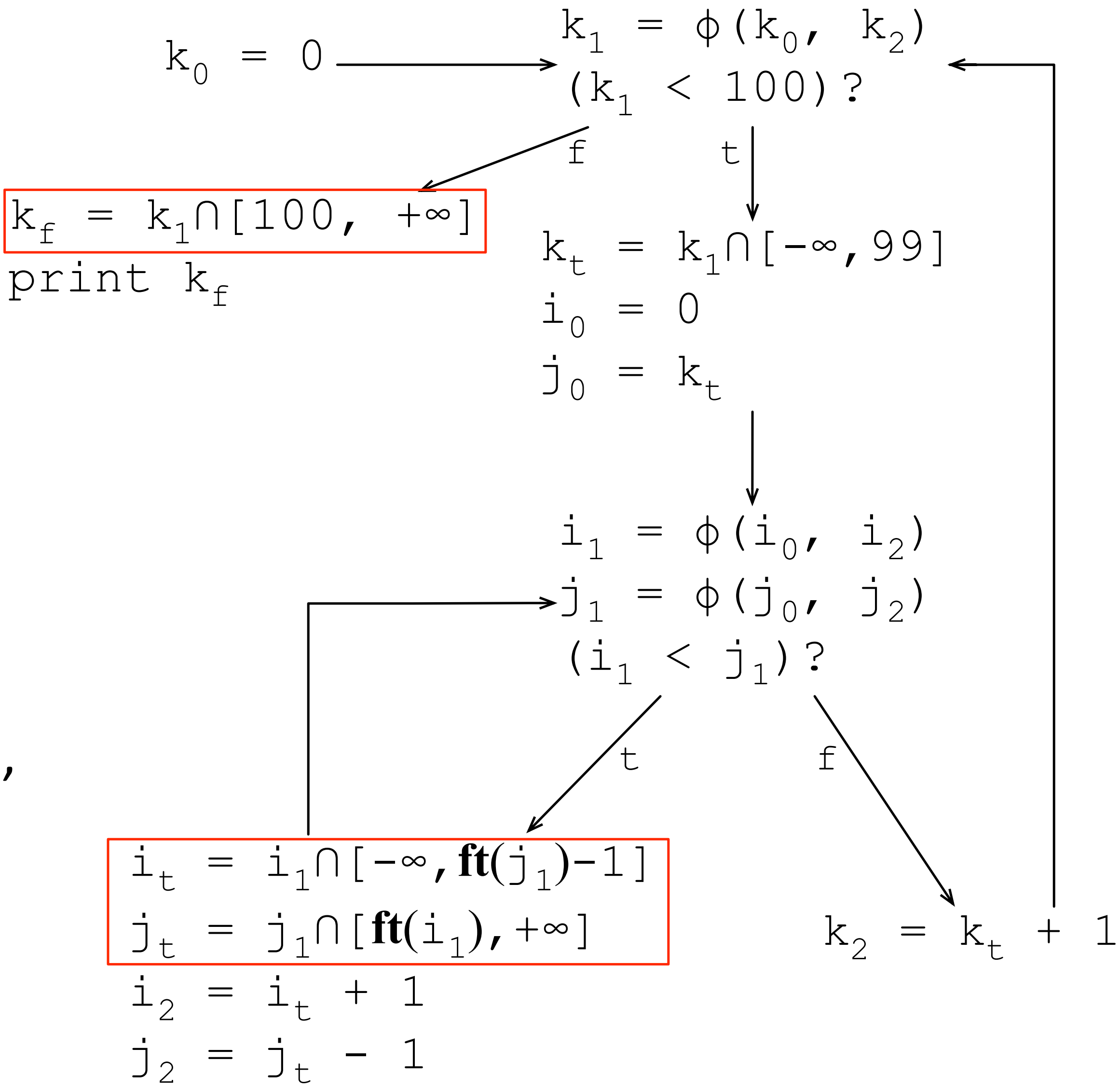


To generate e-SSA form, we split the live ranges of variables that are used right after conditional tests.

(Bodik et al, 2001)

# Extended Static Single Assignment Form

```
k = 0
while k < 100:
    i = 0
    j = k
    while i < j:
        i = i + 1
        j = j - 1
    k = k + 1
```



$k_0 = 0$

$k_1 = \phi(k_0, k_2)$

$(k_1 < 100)?$

f

t

$k_f = k_1 \cap [100, +\infty]$

print $k_f$

$k_t = k_1 \cap [-\infty, 99]$

$i_0 = 0$

$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$

$j_1 = \phi(j_0, j_2)$

$(i_1 < j_1)?$

t

f

$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1) - 1]$

$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$

$i_2 = i_t + 1$

$j_2 = j_t - 1$

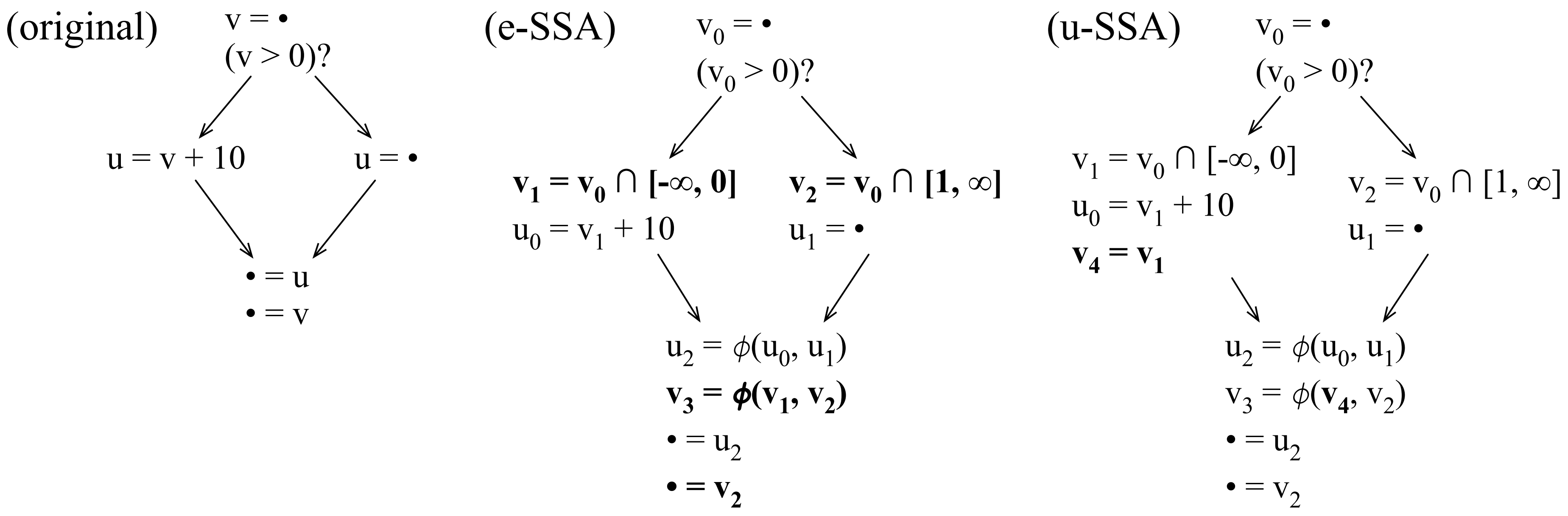
$k_2 = k_t + 1$

We use special instructions, called sigma-functions, to split the live ranges of variables that are tested in conditional branches.

# Live Range Splitting Strategies

We have played with a different program representation (u-SSA), in which we split live ranges after uses. We can only use it if an overflow aborts the program.

If we have an operation such as a = b + c, and an overflow did not happened, then we know that after a is defined, b must be less than c + MAX_INT.



(original)

$v = \bullet$
$(v > 0)?$

$u = v + 10$

$u = \bullet$

$\bullet = u$
$\bullet = v$

(e-SSA)

$v_0 = \bullet$
$(v_0 > 0)?$

$\mathbf{v_1 = v_0 \cap [-\infty, 0]}$
$u_0 = v_1 + 10$

$\mathbf{v_2 = v_0 \cap [1, \infty]}$
$u_1 = \bullet$

$u_2 = \phi(u_0, u_1)$
$\mathbf{v_3 = \phi(v_1, v_2)}$
$\bullet = u_2$
$\mathbf{\bullet = v_2}$

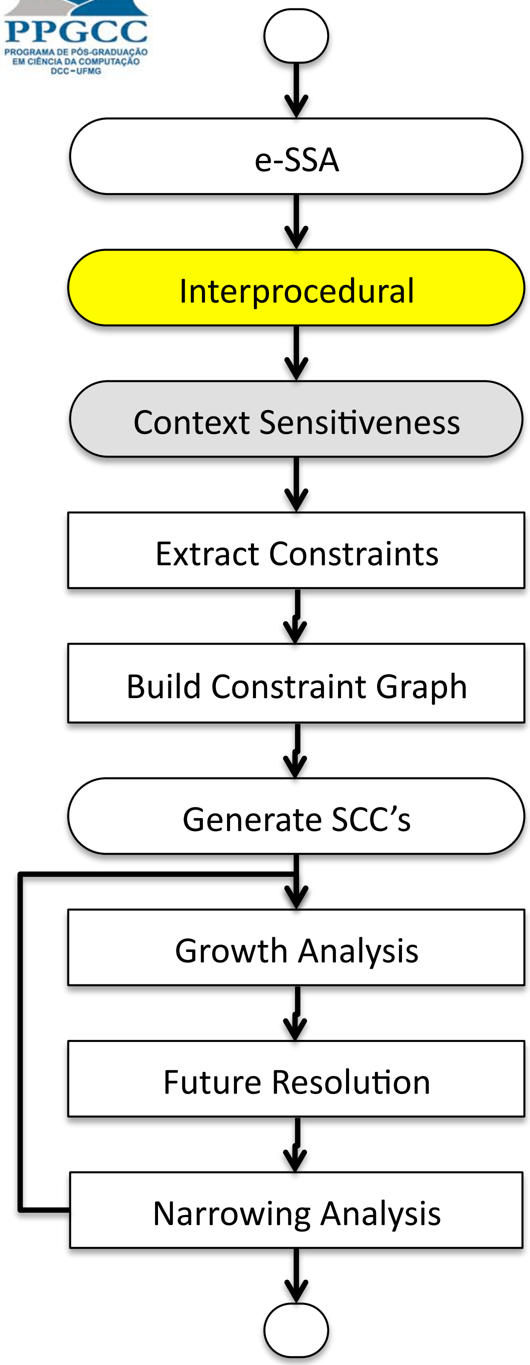
(u-SSA)

$v_0 = \bullet$
$(v_0 > 0)?$

$v_1 = v_0 \cap [-\infty, 0]$
$u_0 = v_1 + 10$
$\mathbf{v_4 = v_1}$

$v_2 = v_0 \cap [1, \infty]$
$u_1 = \bullet$

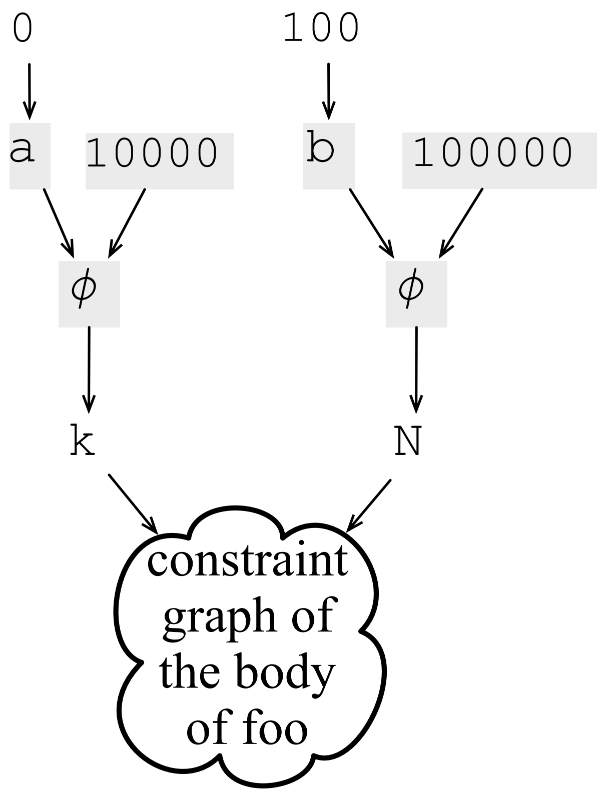
$u_2 = \phi(u_0, u_1)$
$v_3 = \phi(\mathbf{v_4}, v_2)$
$\bullet = u_2$
$\bullet = v_2$

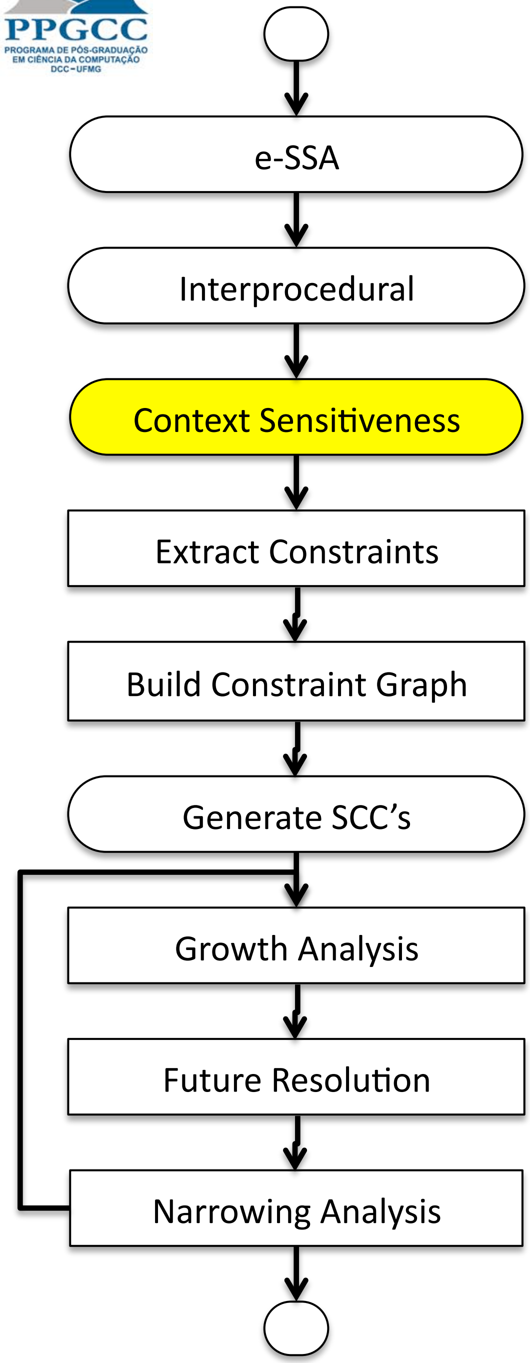


# Inter-procedural Analysis

- Better precision vs slower runtime.

```
main():
    a = 0
    b = 100
    foo(a, b)
    foo(10000, 100000)

foo(k, N):
    while k < N:
        i = 0; j = k
        while i < j:
            i = i + 1
            j = j - 1
        k = k + 1
```




e-SSA

Interprocedural

Context Sensitiveness

Extract Constraints

Build Constraint Graph

Generate SCC's

Growth Analysis

Future Resolution

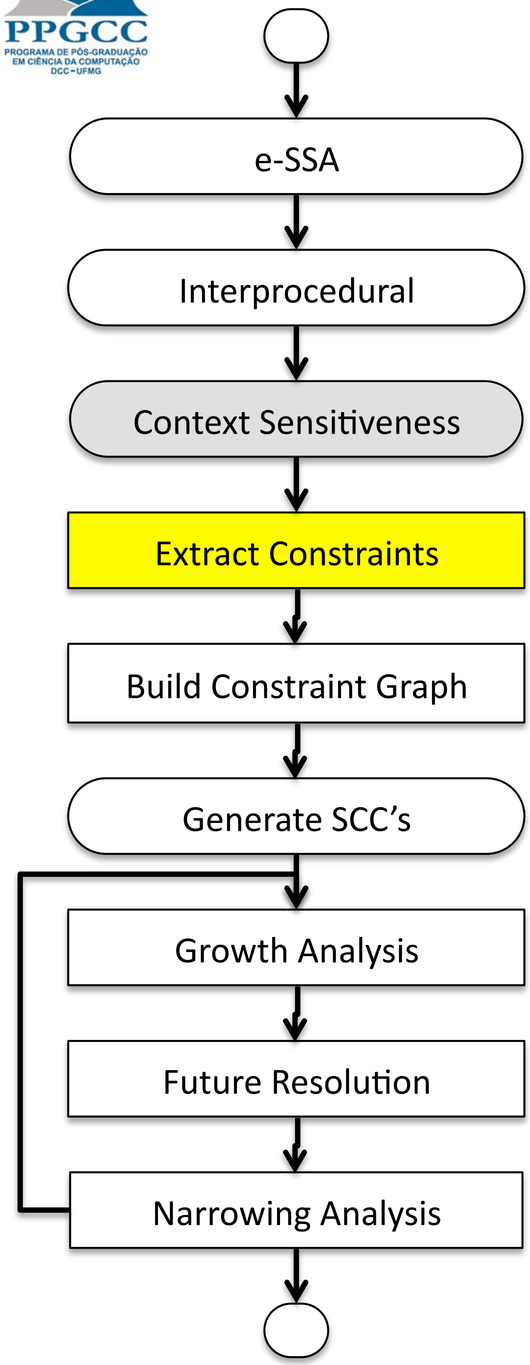
Narrowing Analysis

# Context Sensitiveness

- Function inlining
- Even better precision, even slower runtime

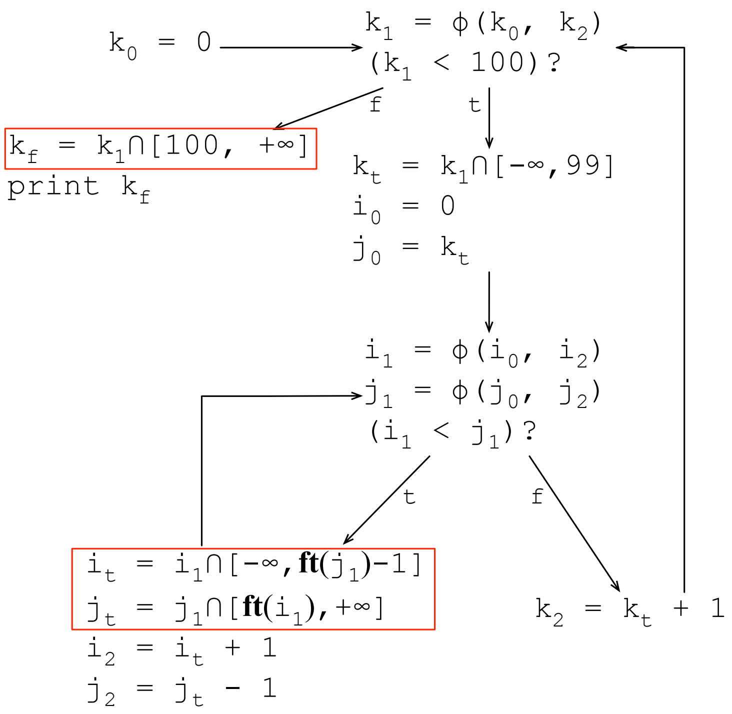
```
main():
    foo(0, 100)
    foo(10000, 100000)

foo(k, N):
    while k < N:
        i = 0; j = k
        while i < j:
            i = i + 1;
            j = j - 1
        k = k + 1
```

```
main():
$k_a = 0$; $N_a = 100$
while $k_a < N_a$:
    $i_a = 0$; $j_a = k_a$
    while $i_a < j_a$:
        $i_a = i_a + 1$
        $j_a = j_a - 1$
    $k_a = k_a + 1$
$k_b = 10000$; $N_b = 100000$
while $k_b < N_b$:
    $i_b = 0$; $j_b = k_b$
    while $i_b < j_b$:
        $i_b = i_b + 1$
        $j_b = j_b - 1$
    $k_b = k_b + 1$
```



# Extraction of Constraints

Each constraint comes out of one instruction in the e-SSA form program. The solution to this constraint system is the solution of our range analysis.



$k_0 = 0$

$k_1 = \phi(k_0, k_2)$
$(k_1 < 100)?$

$\boxed{k_f = k_1 \cap [100, +\infty]}$
print $k_f$

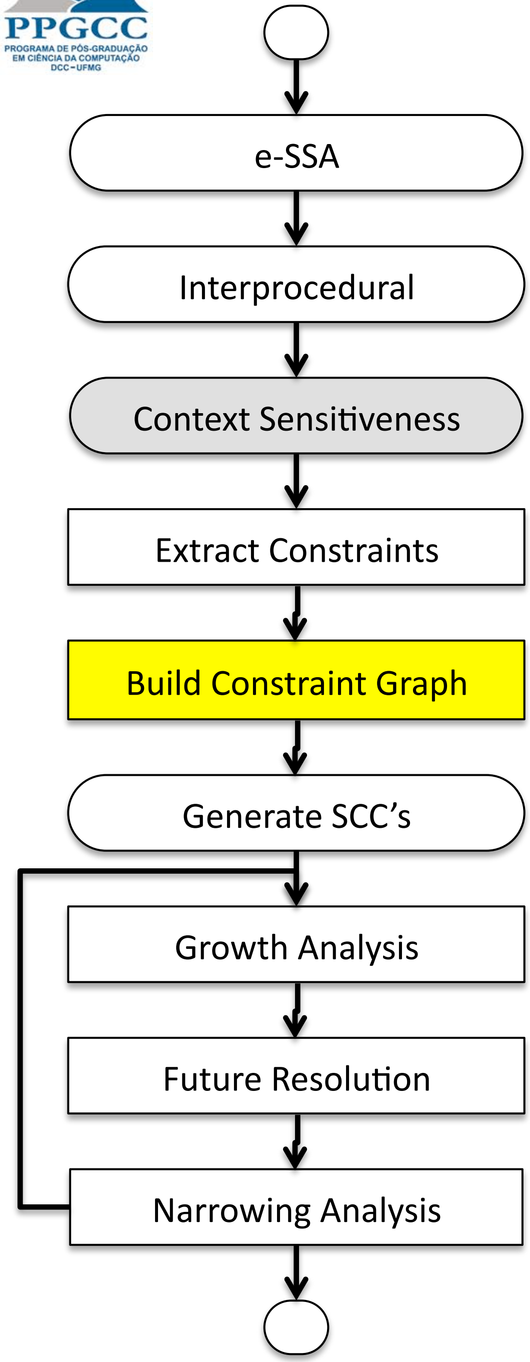
$k_t = k_1 \cap [-\infty, 99]$
$i_0 = 0$
$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$
$j_1 = \phi(j_0, j_2)$
$(i_1 < j_1)?$

$\boxed{\begin{array}{l} i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1)-1] \\ j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty] \end{array}}$
$i_2 = i_t + 1$
$j_2 = j_t - 1$

$k_2 = k_t + 1$

$$
\begin{aligned}
k_0 &= 0 \\
k_1 &= \phi(k_0,\ k_2) \\
k_t &= k_1 \ \cap\ [-\infty, 99] \\
i_0 &= 0 \\
j_0 &= k_t \\
i_1 &= \phi(i_0,\ i_2) \\
j_1 &= \phi(j_0,\ j_2) \\
i_2 &= i_t\ +\ 1 \\
j_2 &= j_t\ -\ 1 \\
k_2 &= k_t\ +\ 1 \\
j_t &= j_1 \ \cap\ [\mathbf{ft}(i_1), +\infty] \\
i_t &= i_1 \ \cap\ [-\infty, \mathbf{ft}(j_1)]
\end{aligned}
$$

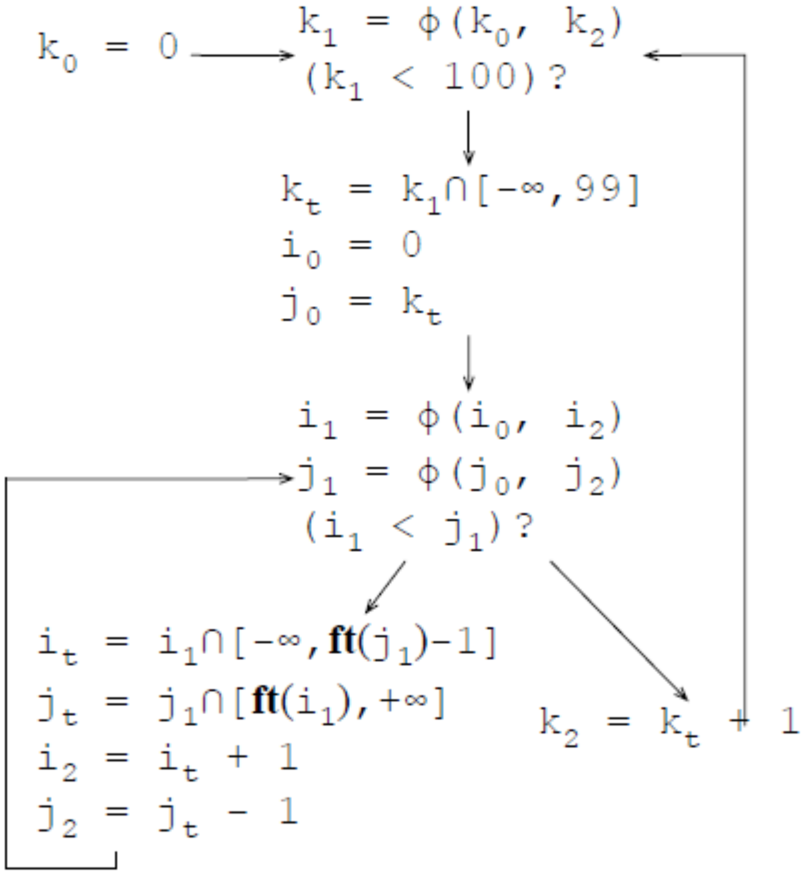


# Constraint Graph

The main data structure that we use in our analysis is the constraint graph.

- This graph has a *data* vertex for each variable in the program.
- The graph has also an *operation* vertex for each constraint in the system.
- Dependence relations determine the *edges*.
  - If constraint C defines variable v, and uses variable u, then we have two edges:
    - u → C
    - C → u

# The Constraint Graph



$k_0 = 0$

$k_1 = \phi(k_0, k_2)$
$(k_1 < 100)?$

$k_t = k_1 \cap [-\infty, 99]$
$i_0 = 0$
$j_0 = k_t$

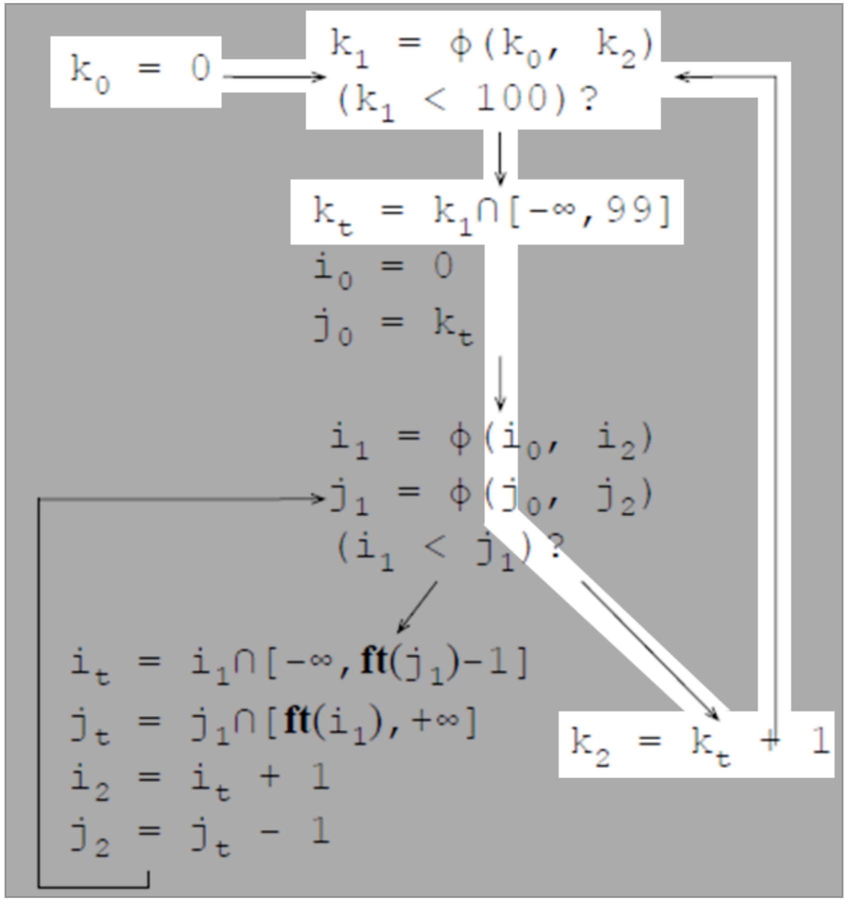
$i_1 = \phi(i_0, i_2)$
$j_1 = \phi(j_0, j_2)$
$(i_1 < j_1)?$

$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1) - 1]$
$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$
$i_2 = i_t + 1$
$j_2 = j_t - 1$

$k_2 = k_t + 1$

- As an example, we shall show how to build the constraint graph for our example program.

# The Constraint Graph



$k_0 = 0$

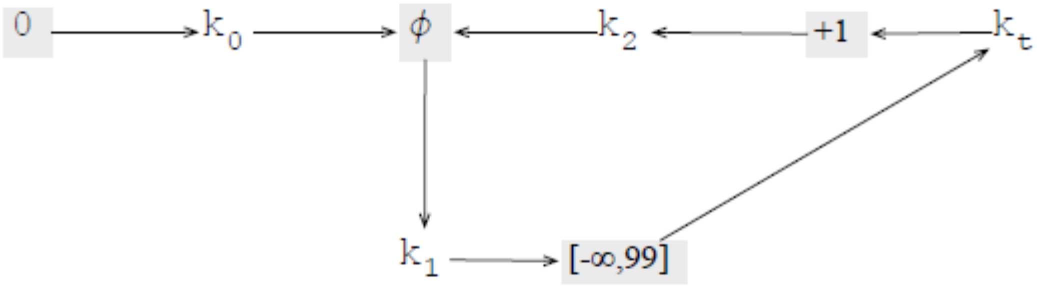
$k_1 = \phi(k_0, k_2)$
$(k_1 < 100)?$

$k_t = k_1 \cap [-\infty, 99]$
$i_0 = 0$
$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$
$j_1 = \phi(j_0, j_2)$
$(i_1 < j_1)?$

$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1)-1]$
$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$
$i_2 = i_t + 1$
$j_2 = j_t - 1$

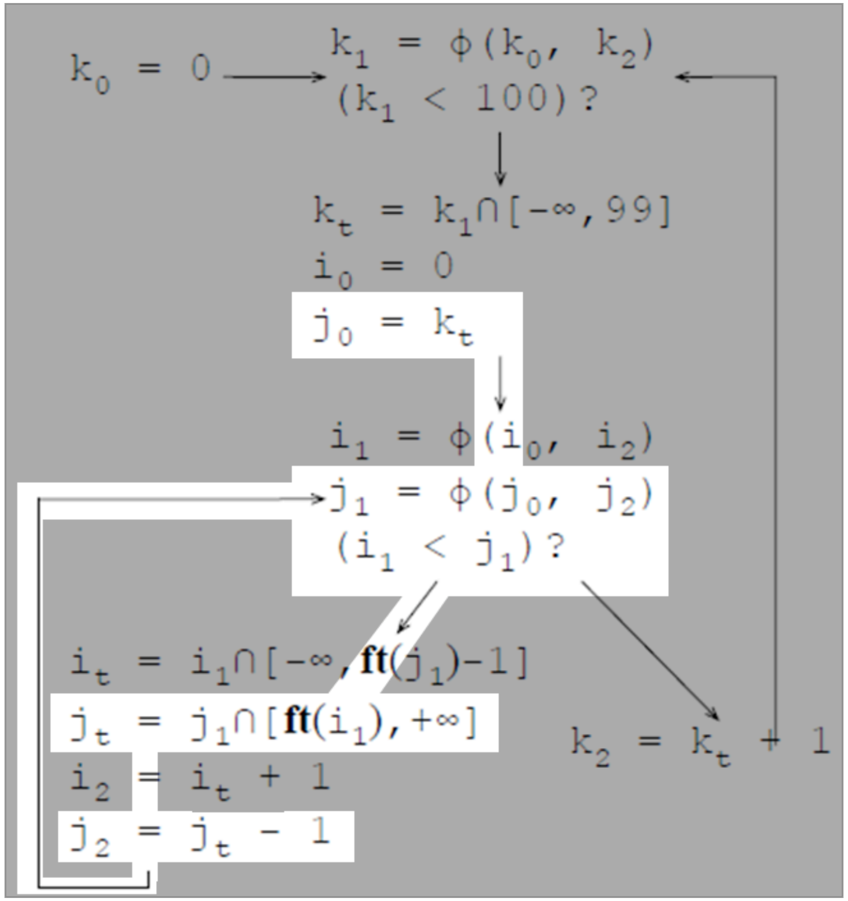
$k_2 = k_t + 1$



We will show the parts of the dependence graph that are related to each variable in our example program.

This is the slice of the graph that deals with the updating of the several versions of variable k.

# The Constraint Graph



$k_0 = 0$

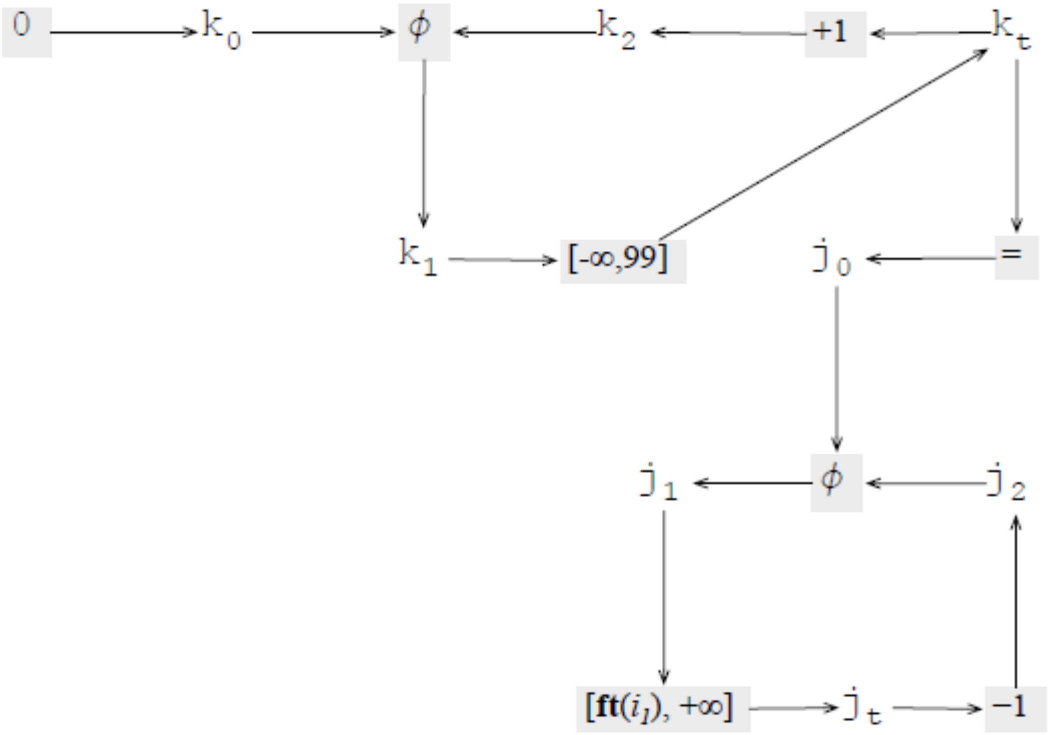
$k_1 = \phi(k_0,\ k_2)$
$(k_1 < 100)?$

$k_t = k_1 \cap [-\infty, 99]$
$i_0 = 0$
$j_0 = k_t$

$i_1 = \phi(i_0,\ i_2)$
$j_1 = \phi(j_0,\ j_2)$
$(i_1 < j_1)?$

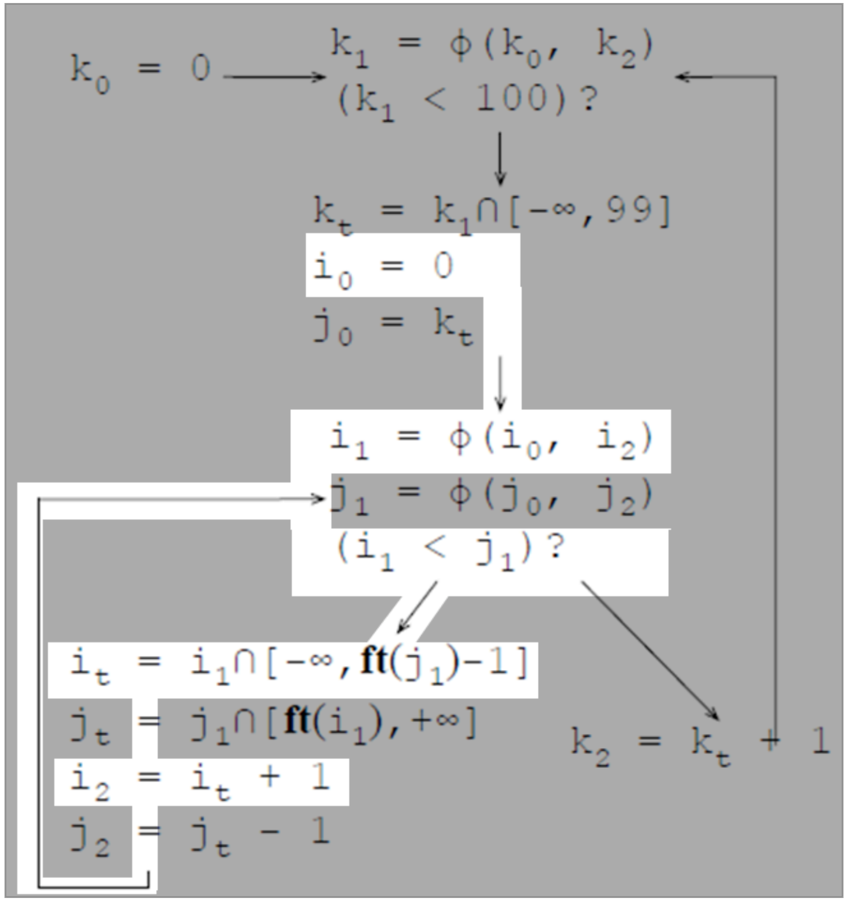
$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1) - 1]$
$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$
$i_2 = i_t + 1$
$j_2 = j_t - 1$

$k_2 = k_t + 1$



This is the slice of the graph that deals with the updating of variable j.

# The Constraint Graph



$k_0 = 0$

$k_1 = \phi(k_0, k_2)$

$(k_1 < 100)?$

$k_t = k_1 \cap [-\infty, 99]$

$i_0 = 0$

$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$

$j_1 = \phi(j_0, j_2)$

$(i_1 < j_1)?$

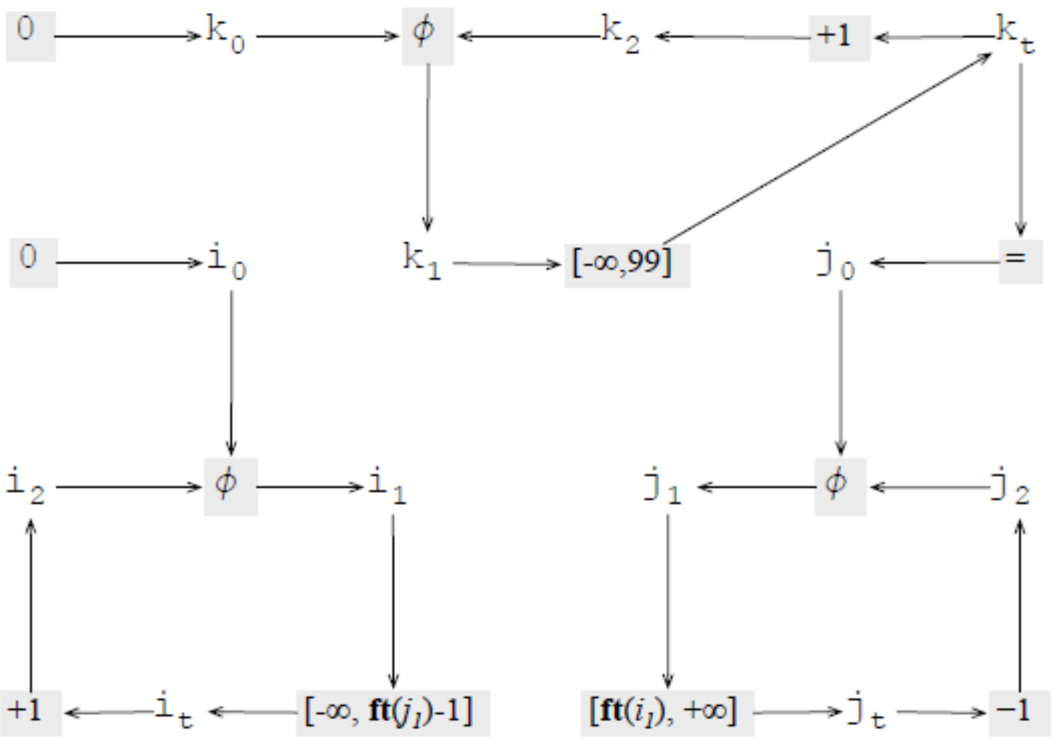
$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1) - 1]$

$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$

$i_2 = i_t + 1$

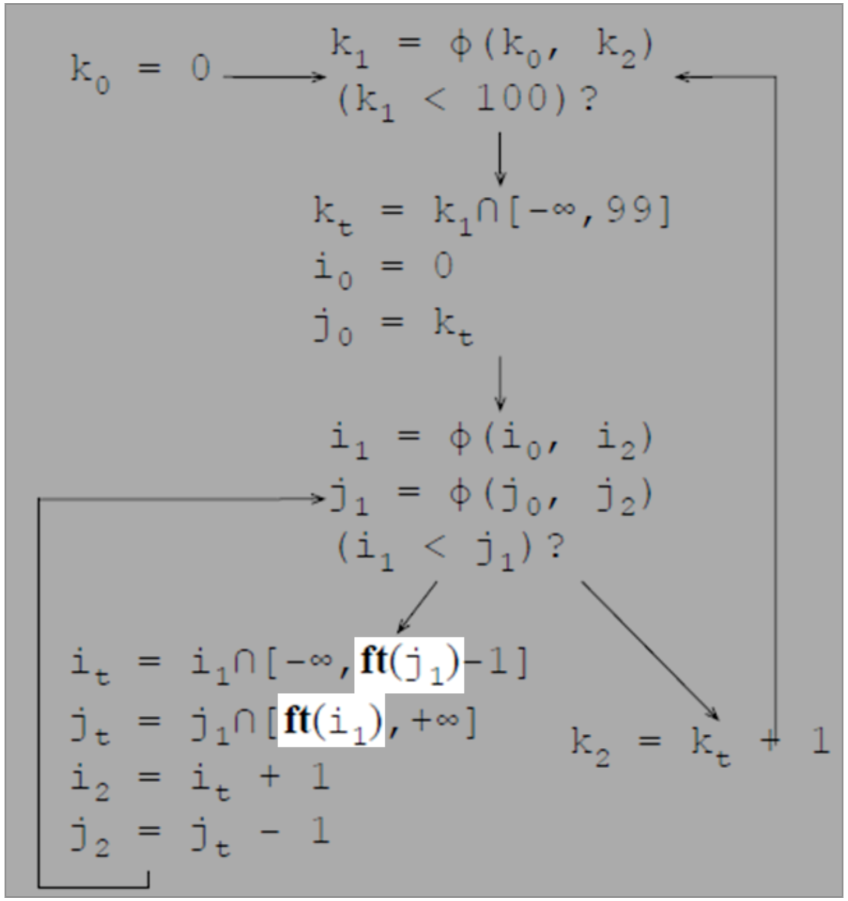
$j_2 = j_t - 1$

$k_2 = k_t + 1$



The slice of variable i.

# The Constraint Graph



$k_0 = 0$

$k_1 = \phi(k_0, k_2)$

$(k_1 < 100)?$

$k_t = k_1 \cap [-\infty, 99]$

$i_0 = 0$

$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$

$j_1 = \phi(j_0, j_2)$

$(i_1 < j_1)?$

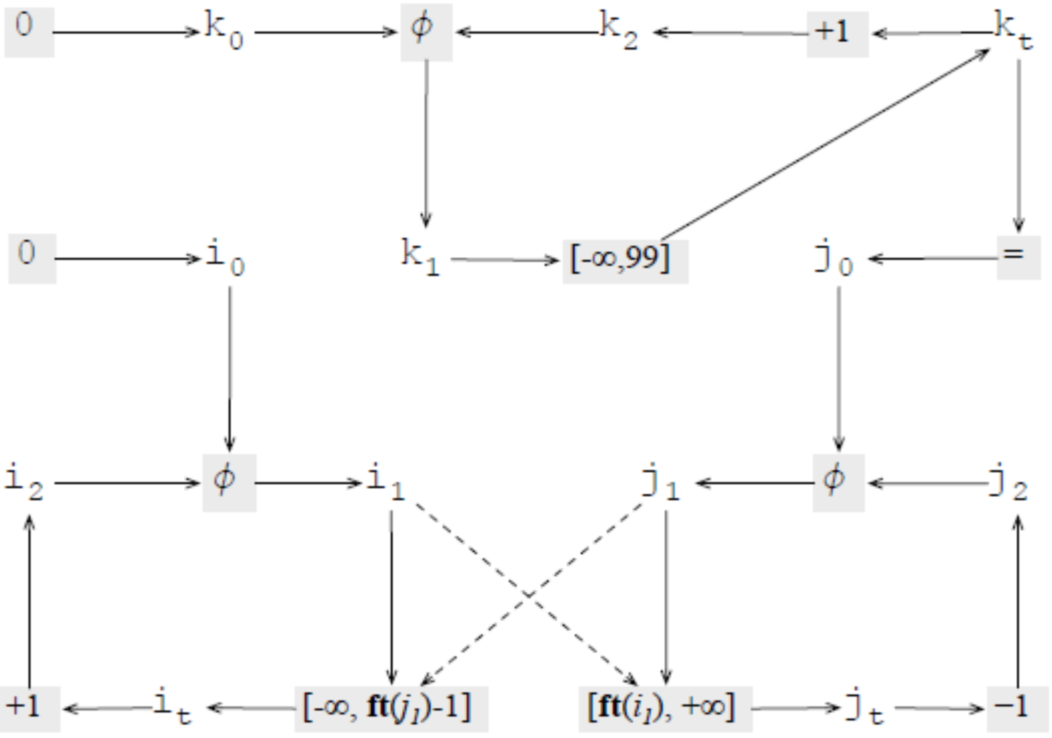
$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1)-1]$

$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$

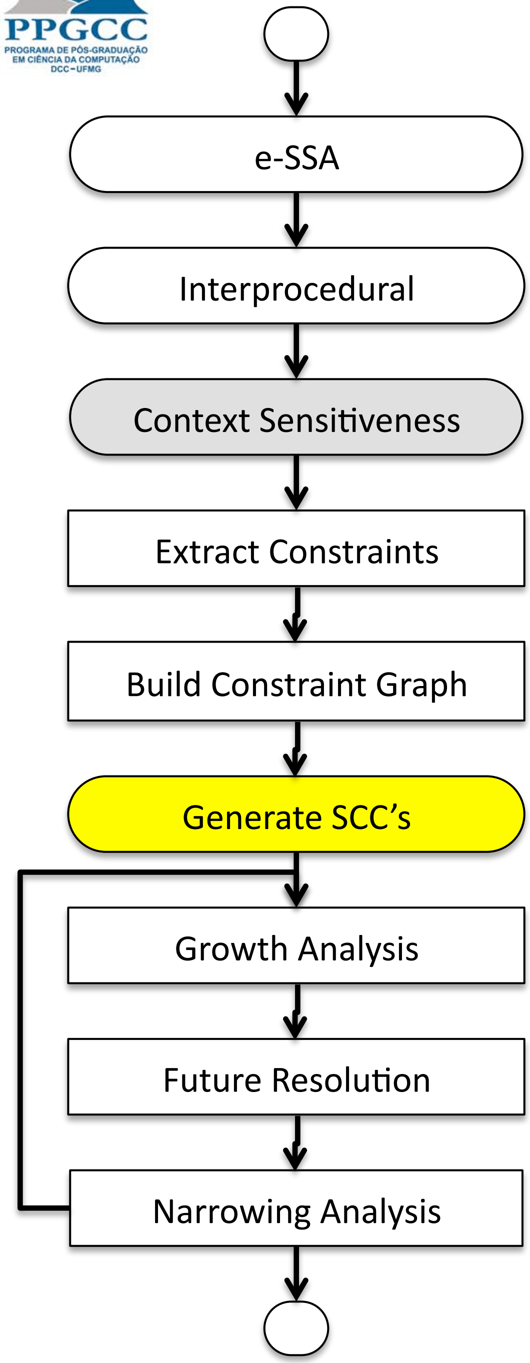
$i_2 = i_t + 1$

$j_2 = j_t - 1$

$k_2 = k_t + 1$



These edges denote control dependences. They will be necessary during the resolution of futures.

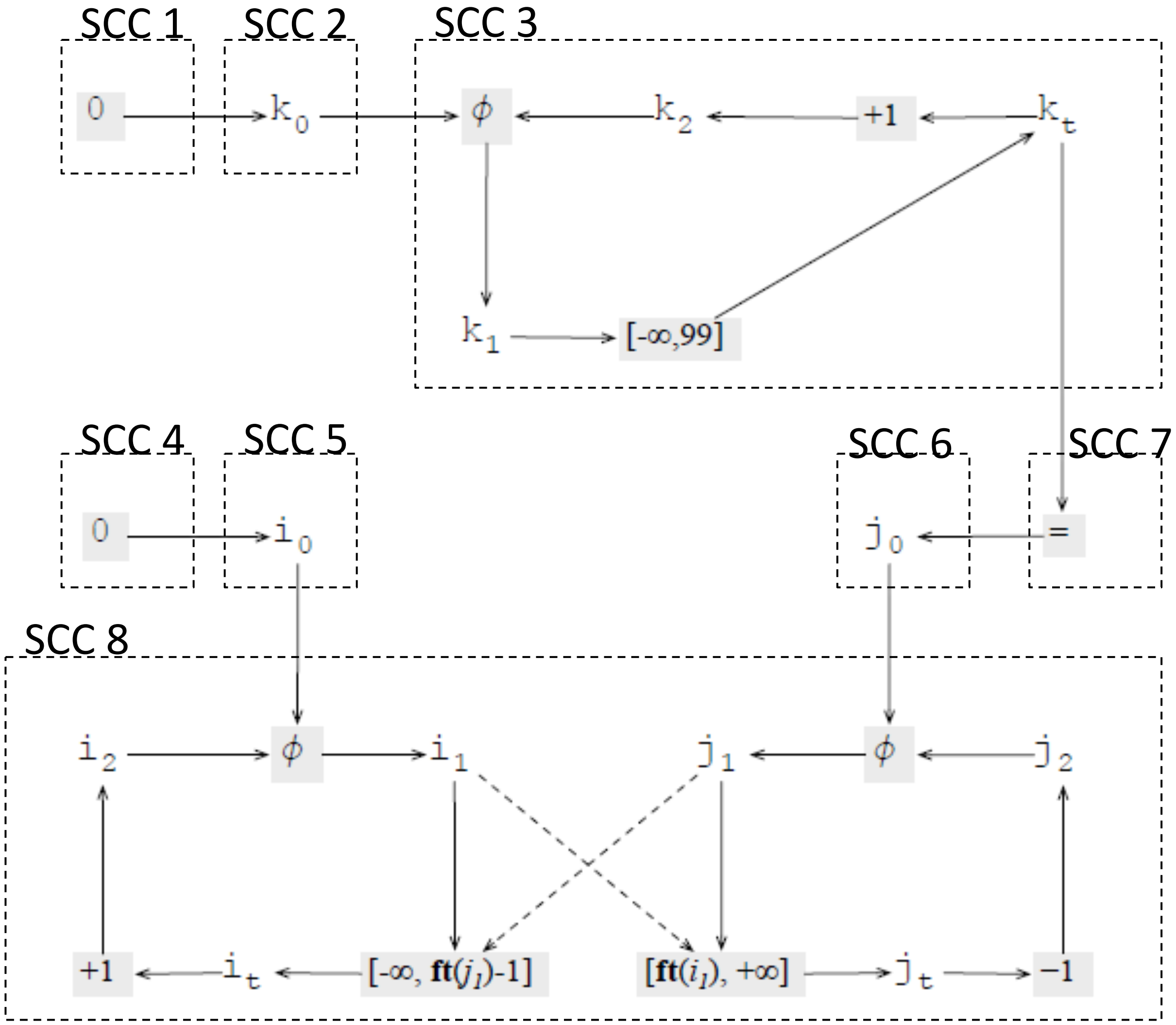


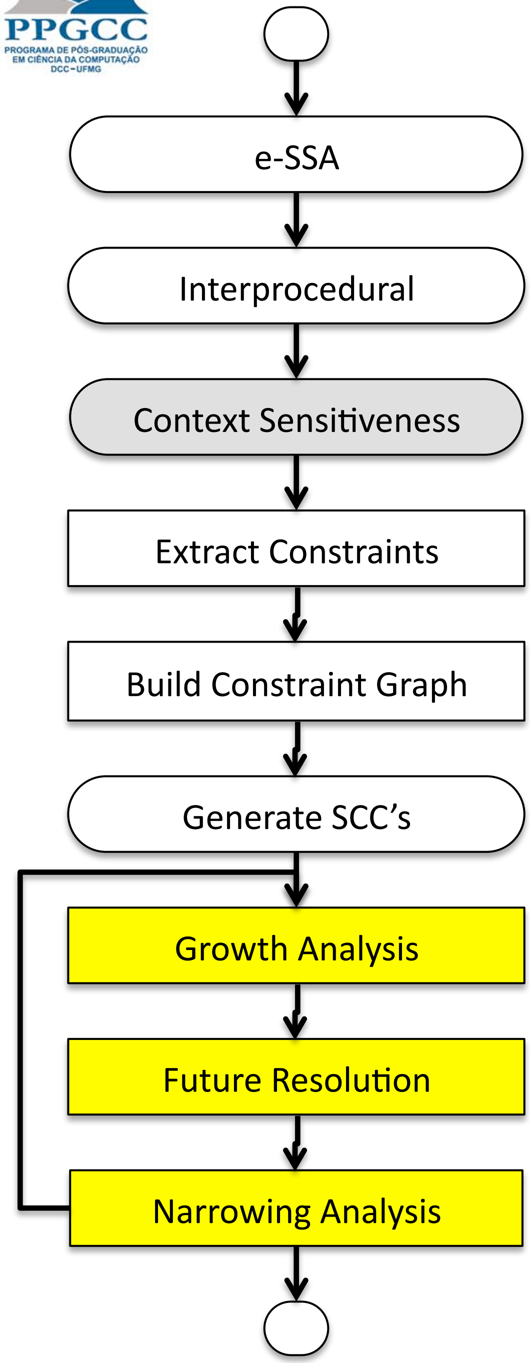
# Strongly Connected Components

- It is well-known in the literature that we can improve the speed of the constraint solver if we process strongly connected components of our constraint graph in topological order.
- But, in our case, SCCs also improves the precision of our results.

# Strongly Connected Components

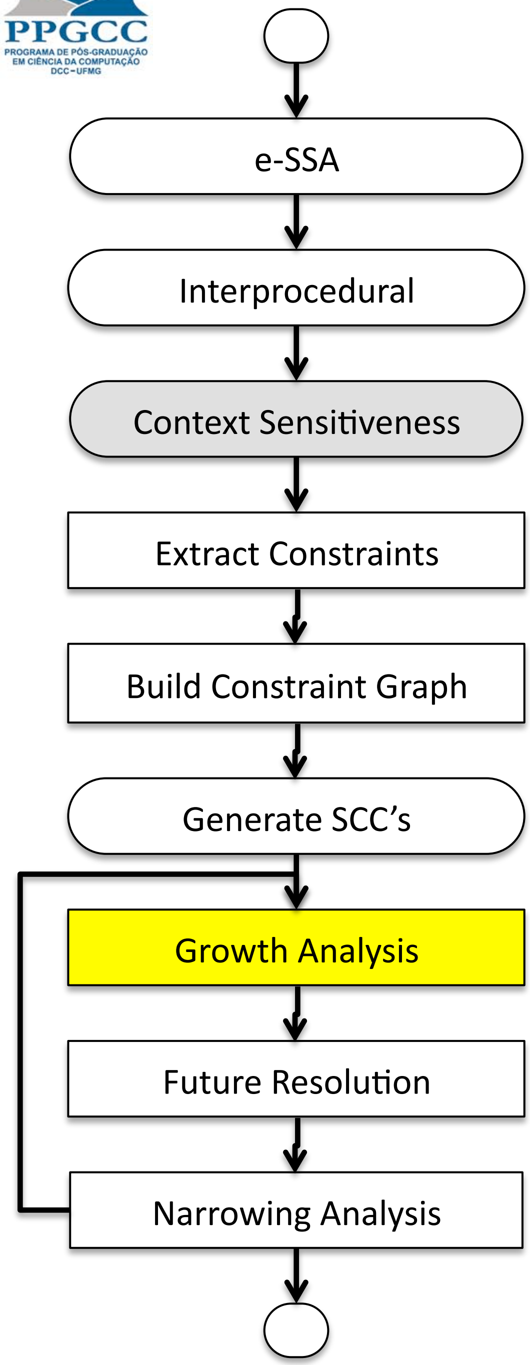
The next phases of our algorithm will be performed once for each SCC in the constraint graph, in topological order.





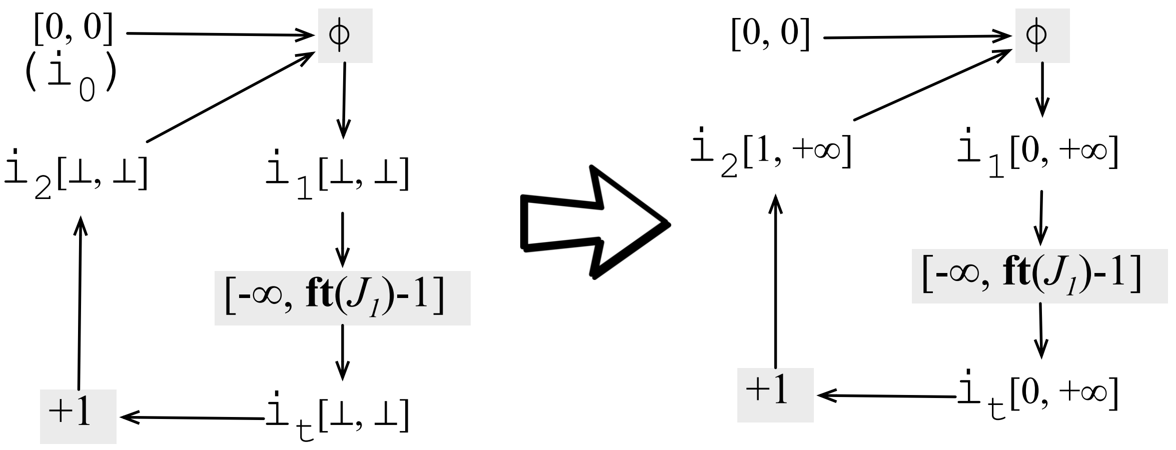
# The Three-Phases Approach

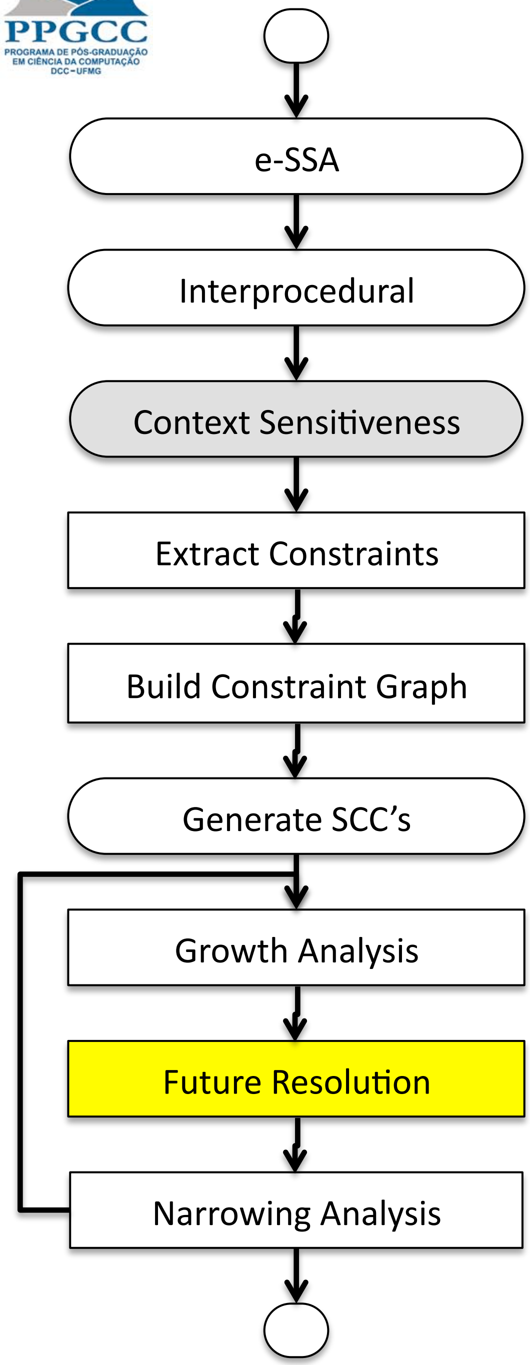
- In the next phase of our algorithm, we iterate three steps, for each strongly connected component.
  - **Widening**: we find how each variable grows (towards -∞, towards +∞, towards both directions, or it remains stable)
  - **Future resolution**: we replace futures by concrete bounds.
  - **Narrowing**: We recover part of the impression of the widening phase by considering the bounds in conditional tests.



# Widening

- We need to know how each variable in the program grows.
  - For instance, if the variable is only updated via increment operations, then it grows towards +∞.
- We do not consider any bounds imposed by conditionals at this point.





e-SSA

Interprocedural

Context Sensitiveness

Extract Constraints

Build Constraint Graph

Generate SCC’s

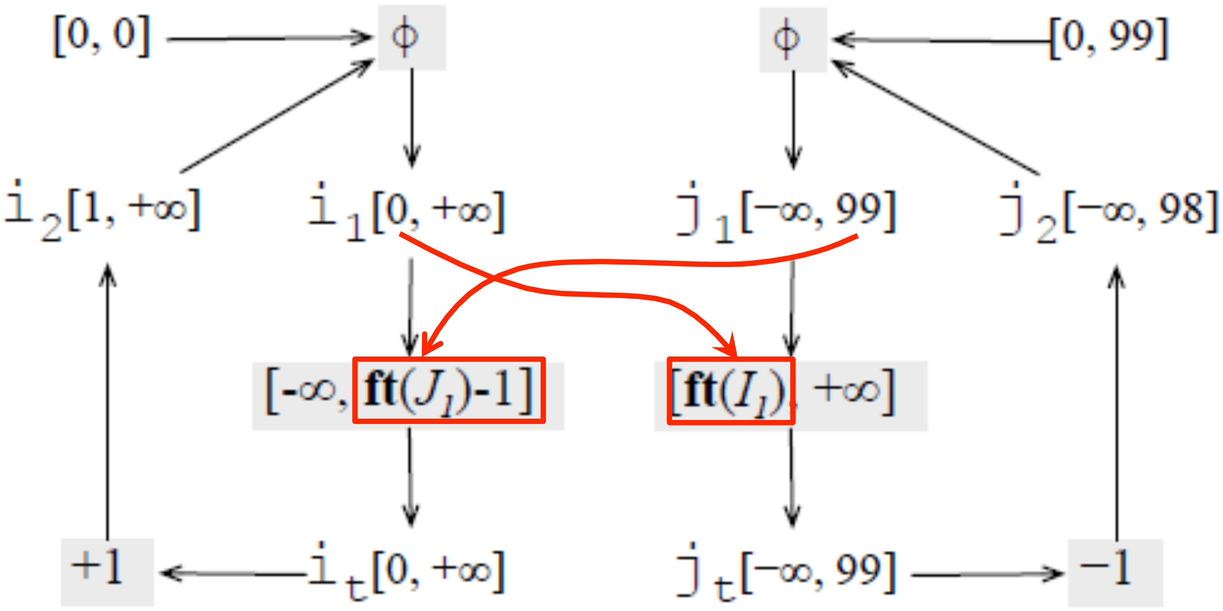
Growth Analysis

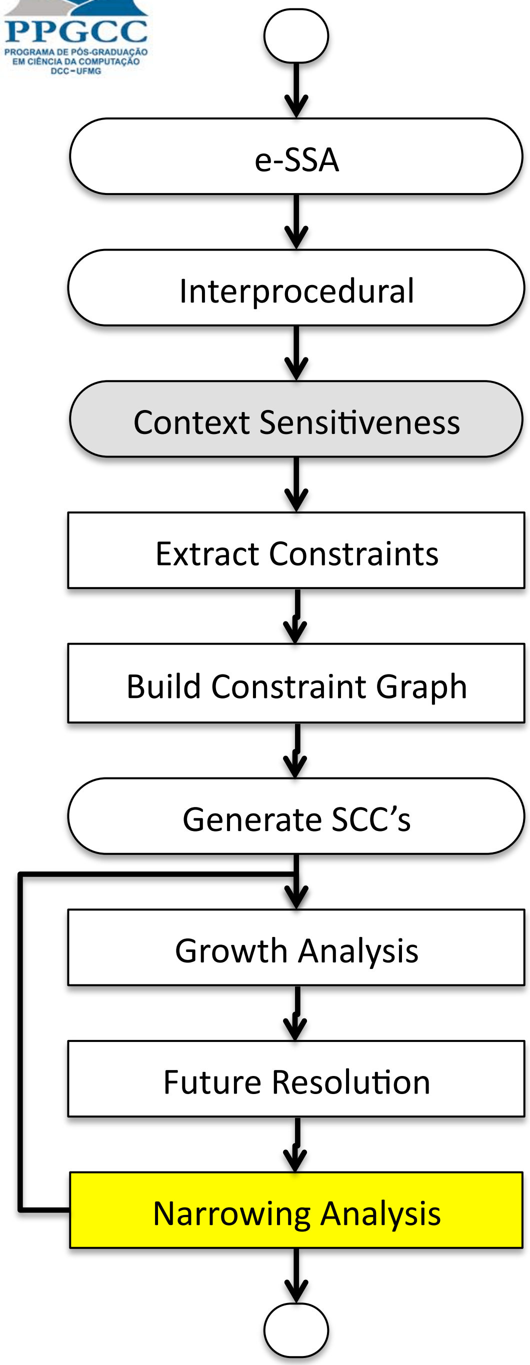
Future Resolution

Narrowing Analysis

# Future Resolution

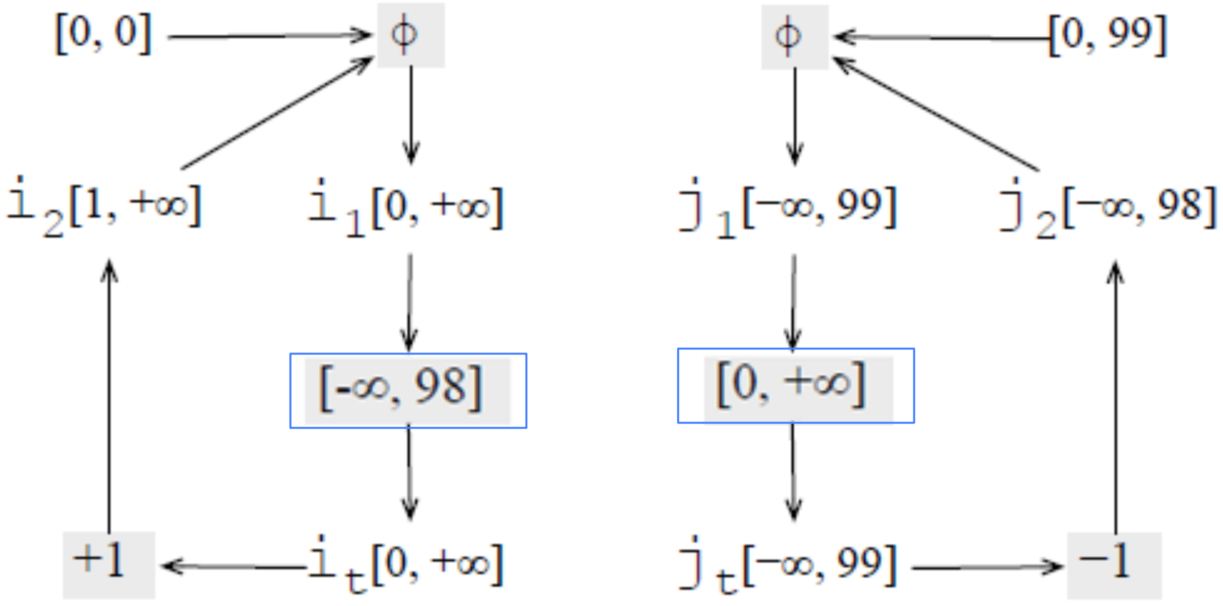
- In the next step, we must replace futures by concrete bounds.
- If a variable's bound remained stable during the widening step, then we know that it can only shrink.

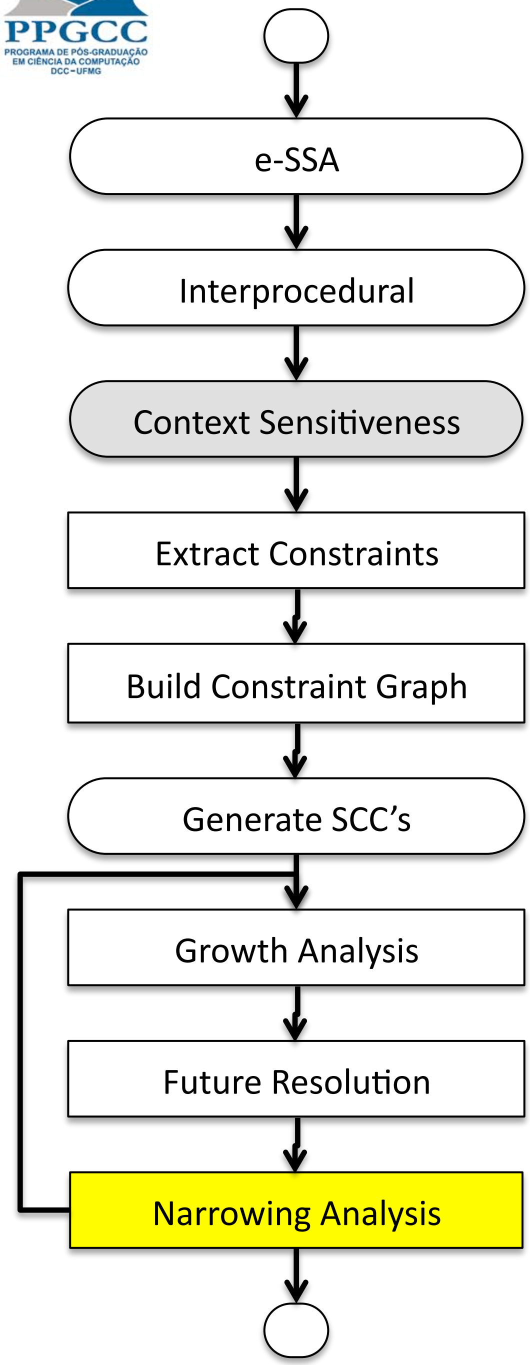




# Narrowing Analysis

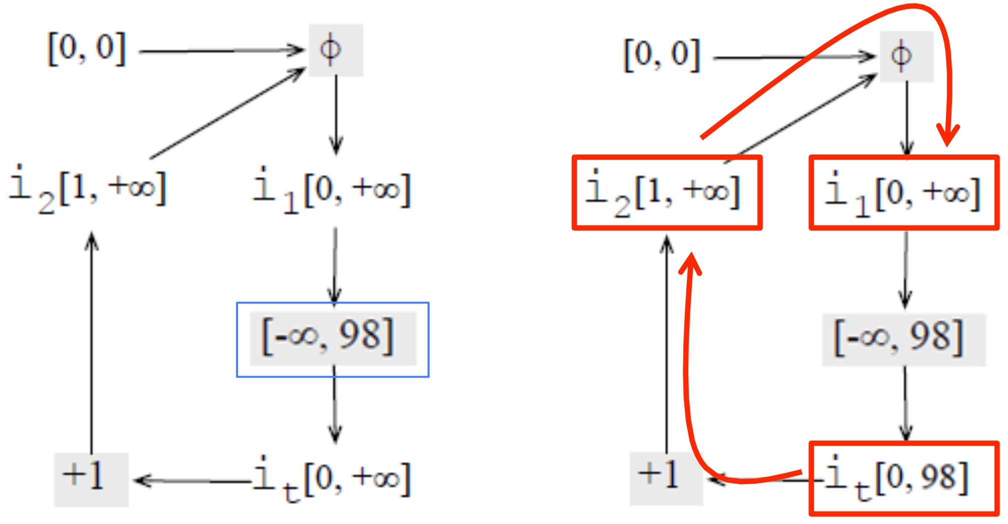
- Once we are done with future resolution, we need to narrow the ranges of the variables.
- This narrowing is guided by the intersection constraints that we have derived from conditional tests.

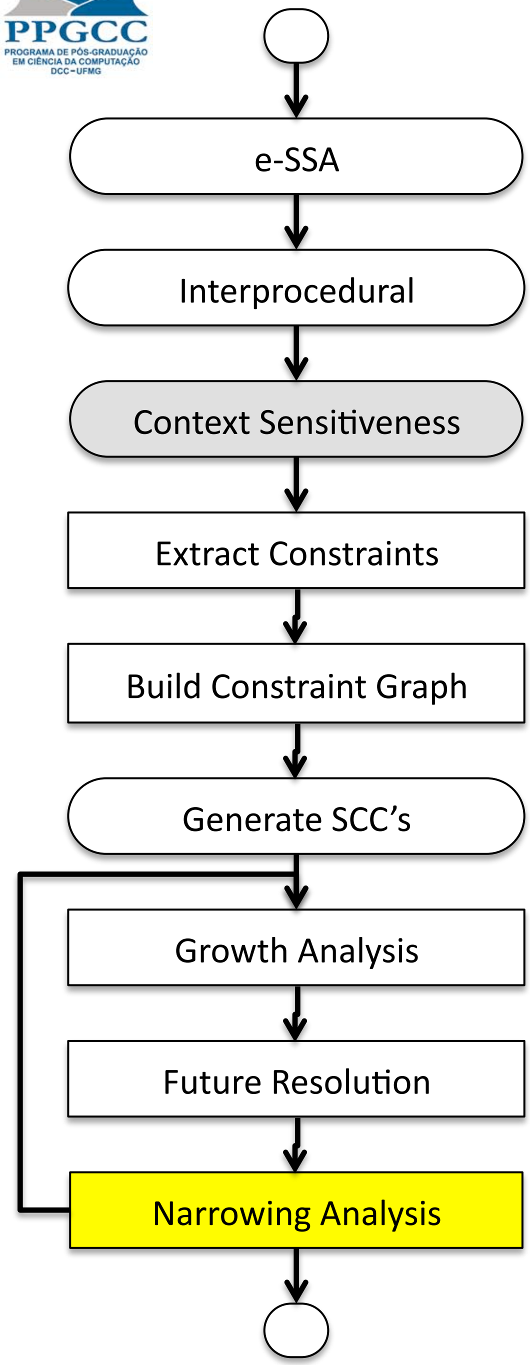




# Narrowing Analysis

- In this example, we know that variable i is less than 99.
- We got this number, 99, out of future resolution.
- We then propagate this restriction throughout every part of the graph that is influenced by i.





# Final Solution

$k_0 = 0$

$k_1 = \phi(k_0, k_2)$

$k_t = k_1 \cap [-\infty, 99]$

$i_0 = 0$

$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$

$j_1 = \phi(j_0, j_2)$

$i_2 = i_t + 1$

$j_2 = j_t - 1$

$k_2 = k_t + 1$

$j_t = j_1 \cap [\mathbf{ft}(i_1), +\infty]$

$i_t = i_1 \cap [-\infty, \mathbf{ft}(j_1)]$

⇨

$I[i_0] = [0, 0]$

$I[i_1] = [0, 99]$

$I[i_2] = [1, 99]$

$I[i_t] = [0, 98]$

$I[j_0] = [0, 99]$

$I[j_1] = [-1, 99]$

$I[j_2] = [-1, 98]$

$I[j_t] = [0, 99]$

$I[k_0] = [0, 0]$

$I[k_1] = [0, 100]$

$I[k_2] = [1, 100]$

$I[k_t] = [0, 99]$

# EXPERIMENTS

# Setup

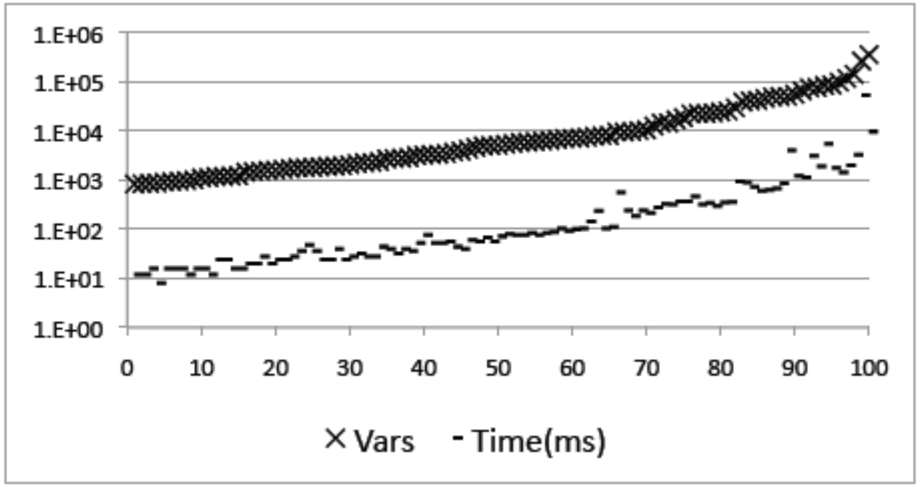
- Implementation (in LLVM) publicly available

- Benchmarks
  - LLVM Test Suite: 400+ programs
  - SPEC 2006

- Hardware
  - x86 ( 2.4 GHZ Core 2 Quad, 3.5GB RAM)

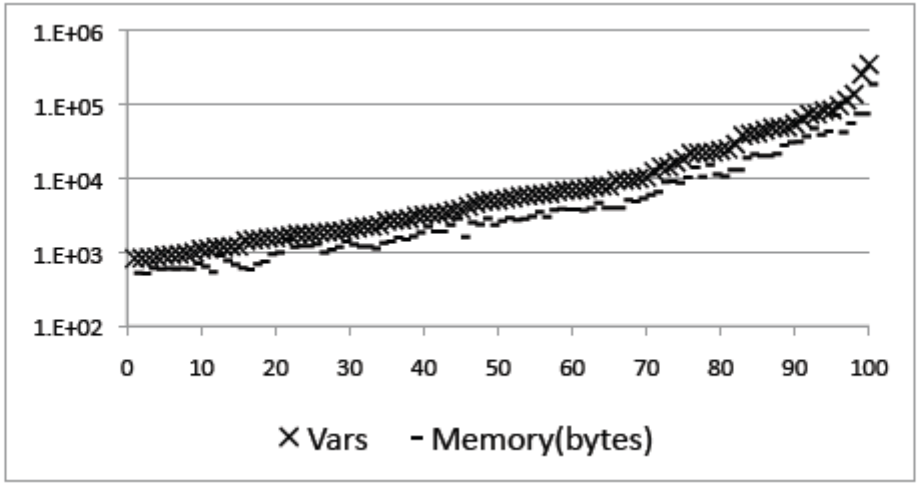
- Average of 15 runs, taking out fastest and slowest

# Linear Asymptotic Complexity in Practice



Runtime

Coefficient of linear determination: 0,967



Memory Consumption

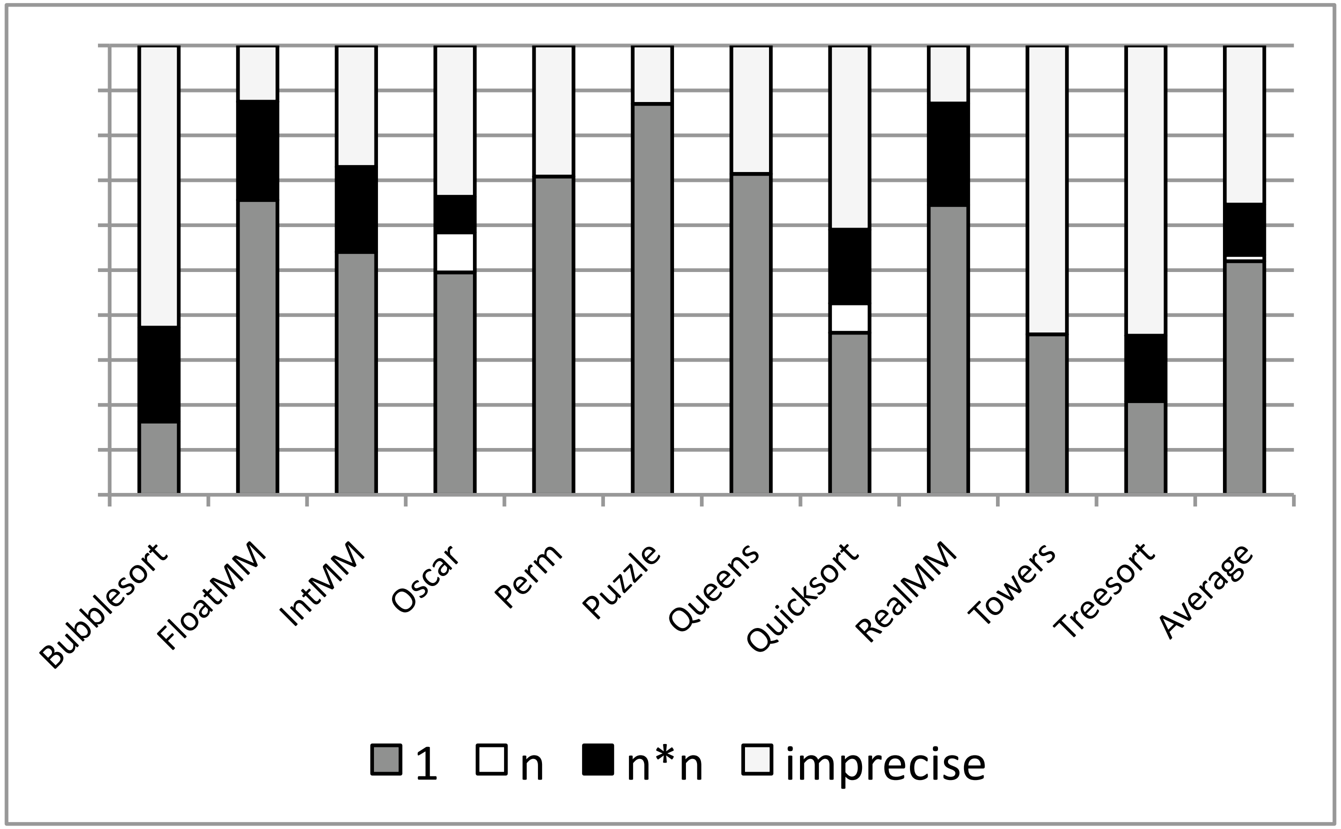
Coefficient of linear determination: 0,994

Each point in each chart represents a benchmark. We have only plotted the 100 largest benchmarks that we have. It includes programs from SPEC 2006.

# Precision

To check the precision of our algorithm, we have implemented a profiler that logs the least and largest values that each integer variable receives throughout the execution of the program. We can then compare these values with the values that we estimate statically.

- The chart below shows the result of this comparison for the Stanford benchmarks. They are a good test, because all the values are hardcoded in the programs.



- "1" means that we got exact bounds.
- "n" means that we got a bound that is less than twice as large as the real value.
- "n*n" means that we are within a quadratic approximation of the variable.

# Overflow Detection

- Instrumentation that checks if an overflow has happened after every `ADD`, `SUB`, `MUL`, `TRUNC` (also bit-casts) or `SHL` (left shift).

- Our implementation can print a warning or stop the execution if  it finds na overflow.

- User can define custom overflow handlers.

- We use range analysis to avoid creating some of these overflow checks.

# A Typical Check

(a)

```
int foo(int x, int y) {
  return x + y;
}
```

(b)

```
entry:
  %add = add nsw i32 %x, %y
  ret i32 %add
```

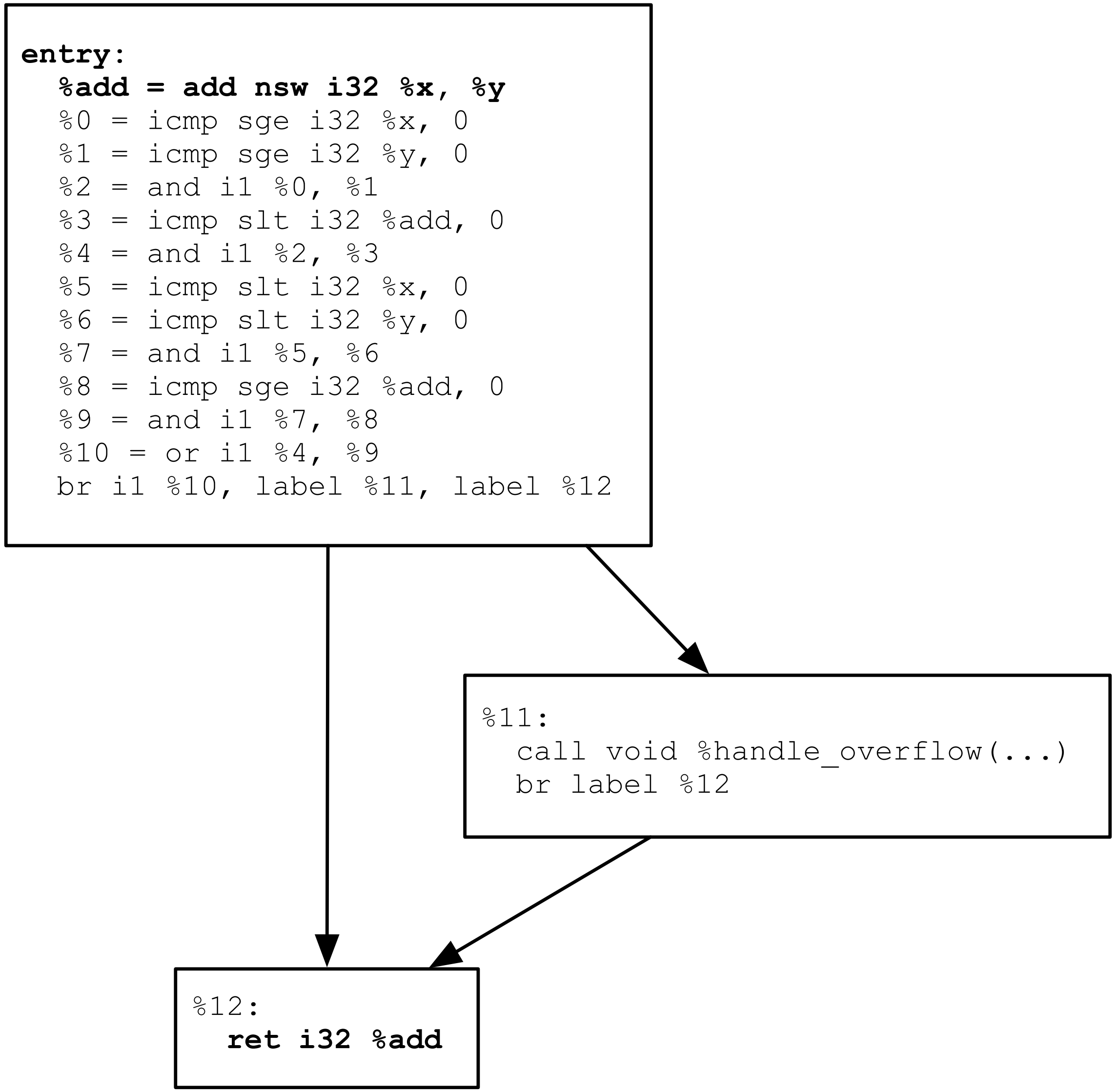
(c)

$$x = o_1 +_s o_2$$

(d)

$$(o_1 > 0 \land o_2 > 0 \land x < 0) \lor$$
$$(o_1 < 0 \land o_2 < 0 \land x > 0)$$

(e)

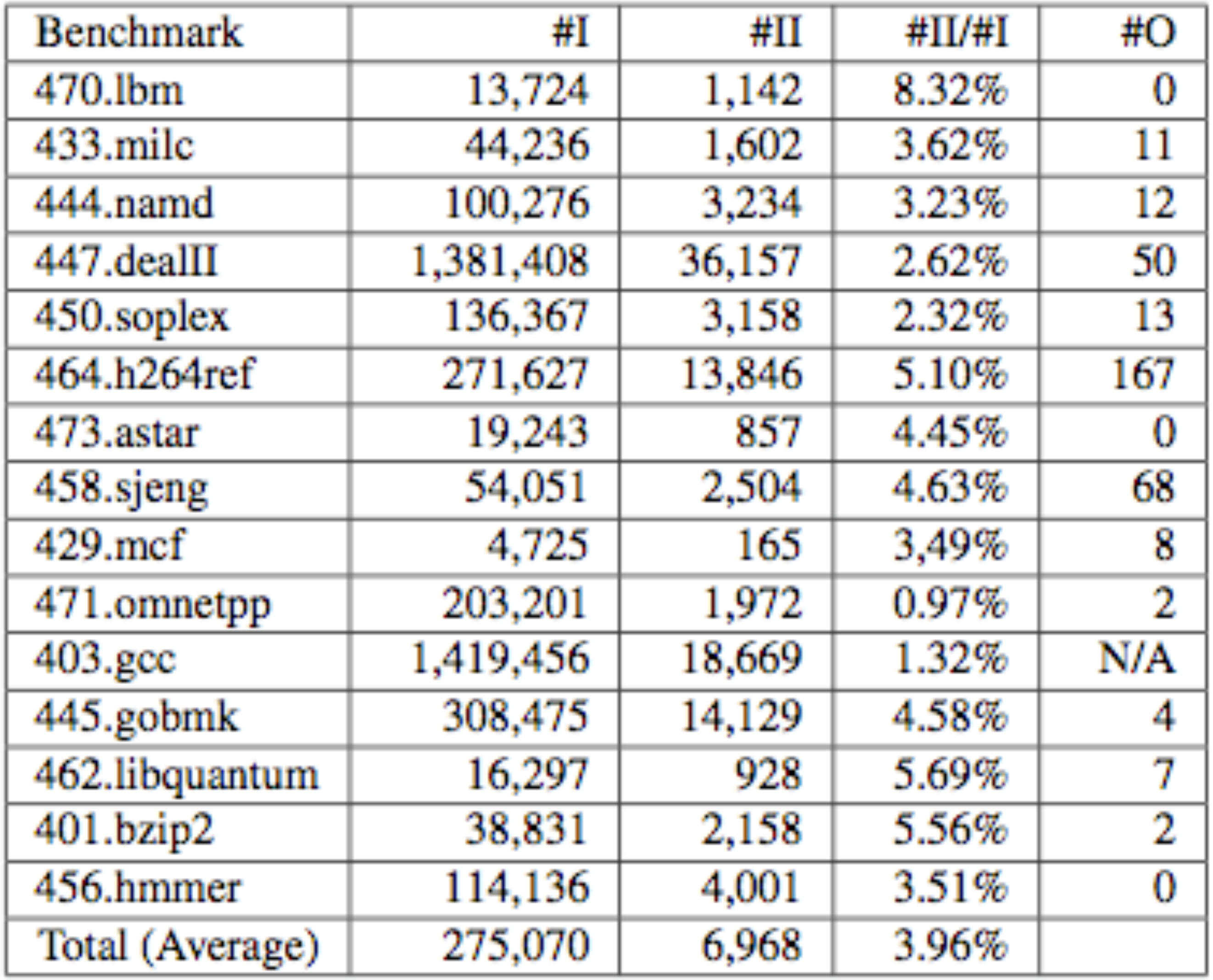


```
entry:
  %add = add nsw i32 %x, %y
  %0 = icmp sge i32 %x, 0
  %1 = icmp sge i32 %y, 0
  %2 = and i1 %0, %1
  %3 = icmp slt i32 %add, 0
  %4 = and i1 %2, %3
  %5 = icmp slt i32 %x, 0
  %6 = icmp slt i32 %y, 0
  %7 = and i1 %5, %6
  %8 = icmp sge i32 %add, 0
  %9 = and i1 %7, %8
  %10 = or i1 %4, %9
  br i1 %10, label %11, label %12
```

```
%11:
  call void %handle_overflow(...)
  br label %12
```

```
%12:
  ret i32 %add
```

# Overflows in Practice

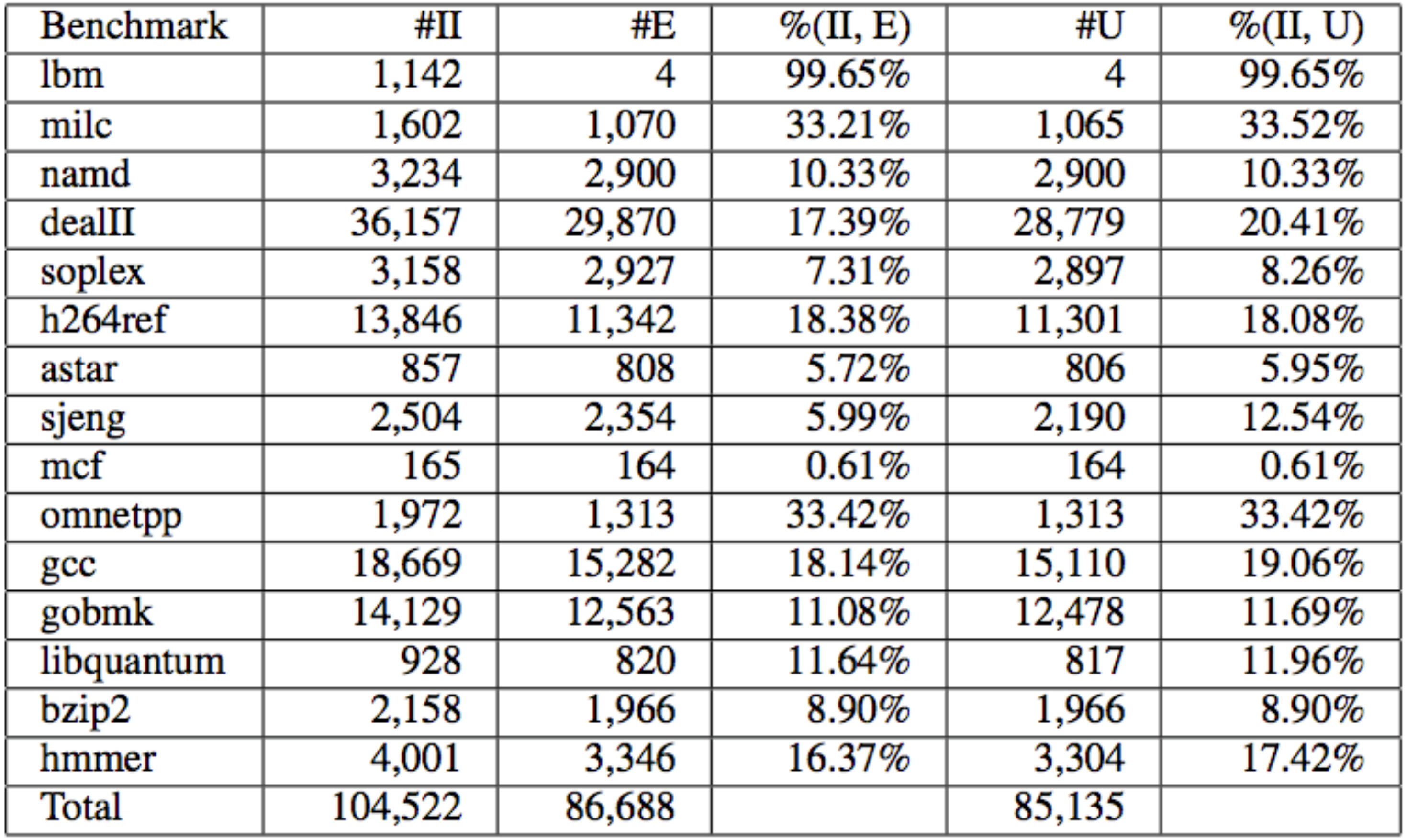


| Benchmark | #I | #II | #II/#I | #O |
|---|---:|---:|---:|---:|
| 470.lbm | 13,724 | 1,142 | 8.32% | 0 |
| 433.milc | 44,236 | 1,602 | 3.62% | 11 |
| 444.namd | 100,276 | 3,234 | 3.23% | 12 |
| 447.dealII | 1,381,408 | 36,157 | 2.62% | 50 |
| 450.soplex | 136,367 | 3,158 | 2.32% | 13 |
| 464.h264ref | 271,627 | 13,846 | 5.10% | 167 |
| 473.astar | 19,243 | 857 | 4.45% | 0 |
| 458.sjeng | 54,051 | 2,504 | 4.63% | 68 |
| 429.mcf | 4,725 | 165 | 3,49% | 8 |
| 471.omnetpp | 203,201 | 1,972 | 0.97% | 2 |
| 403.gcc | 1,419,456 | 18,669 | 1.32% | N/A |
| 445.gobmk | 308,475 | 14,129 | 4.58% | 4 |
| 462.libquantum | 16,297 | 928 | 5.69% | 7 |
| 401.bzip2 | 38,831 | 2,158 | 5.56% | 2 |
| 456.hmmer | 114,136 | 4,001 | 3.51% | 0 |
| Total (Average) | 275,070 | 6,968 | 3.96% | |

- "#I" LLVM bytecodes in the original program.
- "#II" Number of instructions that we had to instrument.
- "#II/#I" Ratio of instrumented instructions.
- "#O": Number of instructions in which we have detected overflows.
- We could not compile gcc, *even without dynamic overflow detection*, because of a incompatible `ctype.h`

# The Benefits of Range Analysis

- "#II": Number of instructions that we had to instrument.
- "#E": Operations instrumented in the e-SSA form programs.
- "#U": Operations instrumented in the u-SSA form programs.
- "%(II, E)" and "%(II, U)": the higher, the better the results of the range analysis.



| Benchmark | #II | #E | %(II, E) | #U | %(II, U) |
|---|---:|---:|---:|---:|---:|
| lbm | 1,142 | 4 | 99.65% | 4 | 99.65% |
| milc | 1,602 | 1,070 | 33.21% | 1,065 | 33.52% |
| namd | 3,234 | 2,900 | 10.33% | 2,900 | 10.33% |
| dealII | 36,157 | 29,870 | 17.39% | 28,779 | 20.41% |
| soplex | 3,158 | 2,927 | 7.31% | 2,897 | 8.26% |
| h264ref | 13,846 | 11,342 | 18.38% | 11,301 | 18.08% |
| astar | 857 | 808 | 5.72% | 806 | 5.95% |
| sjeng | 2,504 | 2,354 | 5.99% | 2,190 | 12.54% |
| mcf | 165 | 164 | 0.61% | 164 | 0.61% |
| omnetpp | 1,972 | 1,313 | 33.42% | 1,313 | 33.42% |
| gcc | 18,669 | 15,282 | 18.14% | 15,110 | 19.06% |
| gobmk | 14,129 | 12,563 | 11.08% | 12,478 | 11.69% |
| libquantum | 928 | 820 | 11.64% | 817 | 11.96% |
| bzip2 | 2,158 | 1,966 | 8.90% | 1,966 | 8.90% |
| hmmer | 4,001 | 3,346 | 16.37% | 3,304 | 17.42% |
| Total | 104,522 | 86,688 | | 85,135 | |

# Final Remarks

- This paper has advanced the state-of-the-art implementations of range analysis.
- Code publicly available. We already have some users.
  - Static range analysis.
  - Different program representations, that *sparsify* several data-flow problems.
  - Dynamic instrumentation to secure programs against integer overflows.
  - Value range profiler.