RANGE ANALYSIS

PROGRAM ANALYSIS AND OPTIMIZATION – DCC888

Fernando Magno Quintão Pereira

fernando@dcc.ufmg.br

The material in these slides have been taken from the paper "Speed and Precision in Range Analysis", by Campos *et al*.

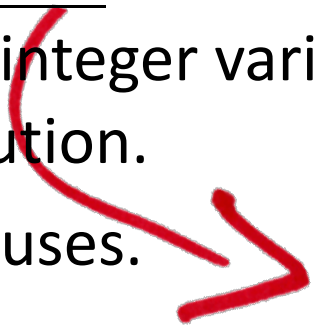
The Range Analysis Problem

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

1) Consider this simple python program on the left. What is the minimum value that k can receive?
2) What is the maximum value that this variable can receive?
3) What about variables i and j?
4) Why is this knowledge important, anyway?

Why does Range Analysis Matter?

- Range Analysis is the problem of finding lower and upper bounds to the values that integer variables can assume throughout program execution.
- This knowledge has many uses.
  - Dead code elimination
  - Array bounds checking elimination
  - Overflow check elimination
  - Static branch prediction



What is the output of this problem? I mean, how should we write its solution?

Dead Code Elimination

- Range analysis gives us the opportunity to do a better conditional constant propagation.
  - Instead of constants, we have ranges.
  - Ranges may degenerate into constants, e.g., [a, b], a = b

```
unsigned int foo(unsigned int v) {
  unsigned int u;
  if (v < 100) {
    u = v & 0x000000FF;
  }
  return u;
}
```

How could range analysis help us to improve this code?

Array Bounds Check Elimination

- Type safe languages, such as Java, JavaScript and C# must guarantee that every array access is within bounds.
- Runtime tests are necessary to check each access.
- Range analysis can be used to eliminate some of these tests.

```
int[] a = new int[100];
int i = 0;
while (i < 100) {
  if (i >= 0 && i < 100) {
    a[i] = 0;
  } else {
    throw new ArrayIndexOutOfBoundsException()
  }
  i++;
}
```

How could range analysis help us to avoid the array bounds check in this example?

Integer Overflow Check Elimination

- 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
```

$$123 = 0\ 1\ 1\ 1\ 1\ 0\ 1\ 1 = 123_{\text{char}}$$

$$128 = 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0 = -128_{\text{char}}$$

$$133 = 1\ 0\ 0\ 0\ 0\ 1\ 0\ 1 = -123_{\text{char}}$$

Malign Integer Overflows

```
1  void read_matrix(int* data, char w, char h) {
2    char buf_size = w * h;
3    if (buf_size < BUF_SIZE) {
4      int c0, c1;
5      int buf[BUF_SIZE];
6      for (c0 = 0; c0 < h; c0++) {
7        for (c1 = 0; c1 < w; c1++) {
8          int index = c0 * w + c1;
9          buf[index] = data[index];
10       }
11     }
12     process(buf);
13   }
14 }
```

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

Malign Integer Overflows

```
1  void read_matrix(int* data, char w, char h) {
2    char buf_size = w * h;
3    if (buf_size < BUF_SIZE) {
4      int c0, c1;
5      int buf[BUF_SIZE];
6      for (c0 = 0; c0 < h; c0++) {
7        for (c1 = 0; c1 < w; c1++) {
8          int index = c0 * w + c1;
9          buf[index] = data[index];
10       }
11     }
12     process(buf);
13   }
14 }
```

Imagine that w = 6 and h = 22. In this case, we have that w * h = 132. But this number is too large to fit into a char. So, -124 ends up being represented instead. If BUF_SIZE = 120, then the test at line 3 is true, and all the 6 * 22 iterations of commands at lines 8 and 9 end up happening. In the end, we have 132 – 120 = 12 bad memory writes. That could produce, for instance, a buffer overflow!

Dynamic Integer Overflow Detection

- Many programming languages guard arithmetic operations against integer overflows.
  - Ada, due to its semantics.
  - JIT compiled JavaScript code, to achieve speed.
  - Secure C, to prevent overflow vulnerabilities♤.

```
unsigned i = 0;
while (i < 100) {
  if (i > 255) {
    throw new IntegerOverflowException();
  } else {
    i++;
  }
}
```

How could we use the results of range analysis to eliminate the **overflow test**?

♤: Understanding integer overflow in C/C++, ICSE, *and*
RICH: Automatically Protecting Against Integer-Based Vulnerabilities, USENIX

Branch Prediction

- Once upon a time, there was a very important research, whose goal was to predict the outcome of branches statically.
- Range analysis was found to be useful in making branch prediction more precise.
- The algorithm described in that paper became, eventually, the base of the implementation of range analysis in the Gnu C compiler♧.

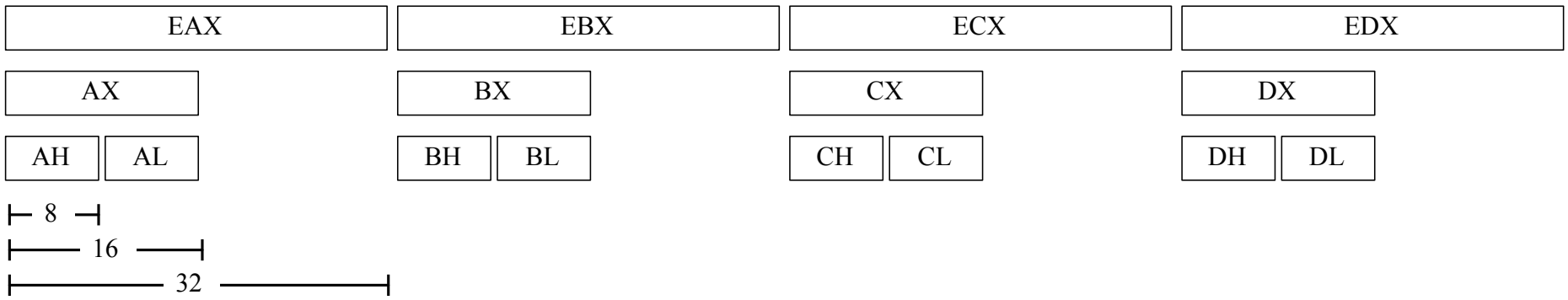
How can range analysis help us to predict the outcome of **this** branch?

```
int i = 0;
int N = 100;
while (i < N) {
  if (i < 99) {
    printf("Taken!\n");
  }
  i++;
}
```

♧: Accurate Static Branch Prediction by Value Range Propagation, PLDI, 1995

Bitwidth Aware Register Allocation

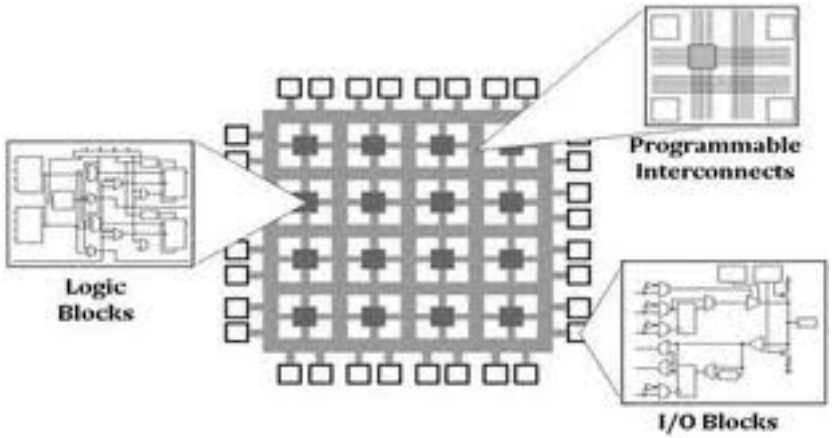
- Some computer architectures have registers of different sizes.
  - As an example, the x86 has registers of eight, 16 and 32 bits (and now 64-bits as well).
  - We can either place two eight-bit variables in registers, or use this register to hold larger values.
- Range analysis frees the developer from worrying about the sizes of variables declared in programs.



High-Level Synthesis

- It is possible to create digital hardware to implement a given algorithm.
- Bit accurate information is important:
  - Less gates to implement registers
  - Less gates to implement arithmetic units
  - Less wires implementing the interconnects

- Range analysis can be more aggressive in this setting, as we are allowed to assume that the input program is devoid of undefined behavior, i.e., we can assume the absence of integer overflows.



Solving Range Analysis

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

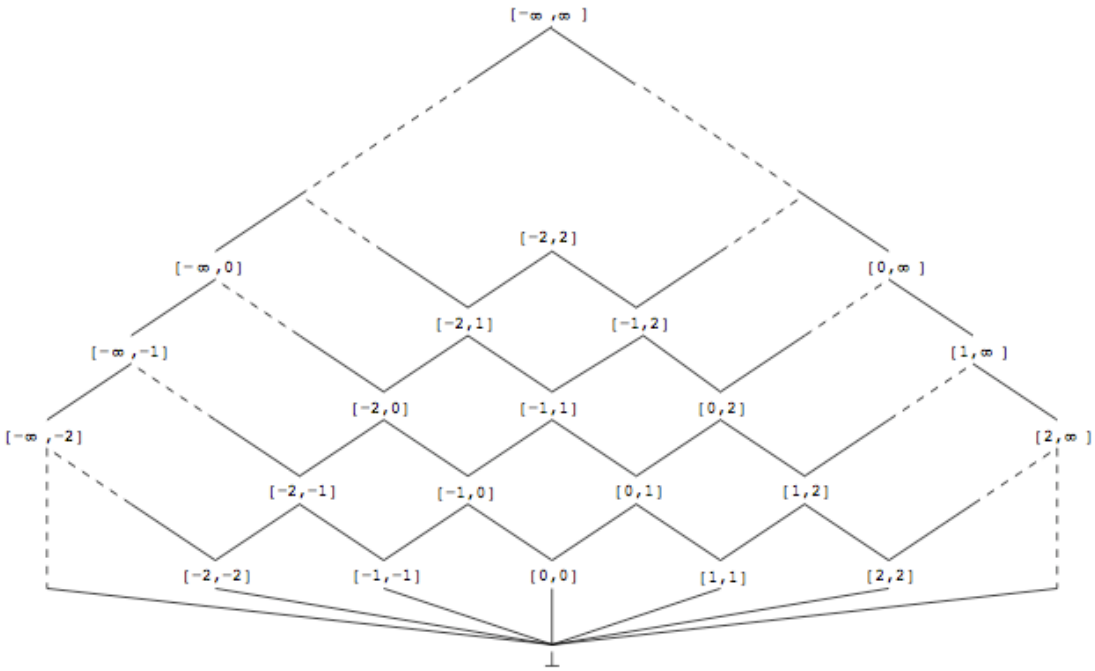
k: [0, 100]
i: [0, 99]
j: [0, 99]

1) Can you design an analysis that solves the range analysis problem?
2) Does your analysis terminate?
3) Which lattice do you use?
4) Is it fast?
5) Is it precise?

Infinite Lattices and Non-Termination

- Interpreting the program may not be a good solution, as the program may not terminate.
- And we must be careful if we go for a data-flow analysis: the range analysis lattice may be infinite.

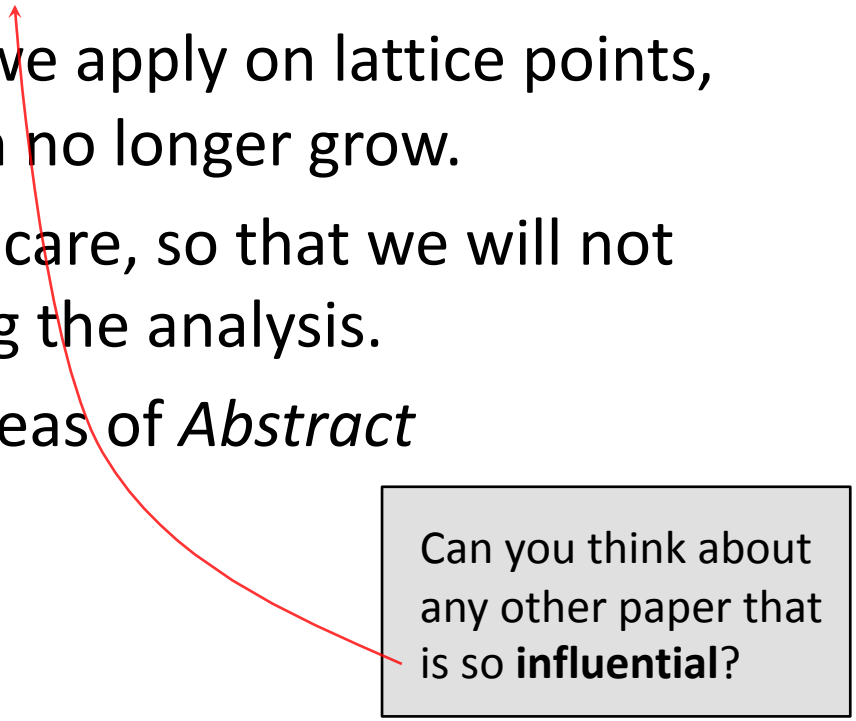
And even if we choose a finite lattice, say, the 32-bit signed integers, the height of this lattice may be too high. The algorithm may take too long to terminate.



How can we deal with this problem?

Widening

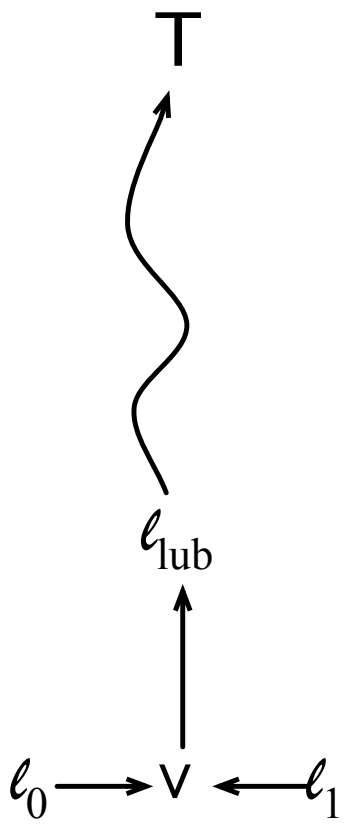
- In the seventies, Cousot and Cousot came up with the idea of widening (and narrowing), to deal with potentially infinite lattices♡.
  - This paper had more than 7,200 citations, in Apr of 2019!
- Widening is an operator that we apply on lattice points, to widen it so much that it can no longer grow.
- We must apply widening with care, so that we will not lose too much precision during the analysis.
- Widening is one of the core ideas of *Abstract Interpretation*.

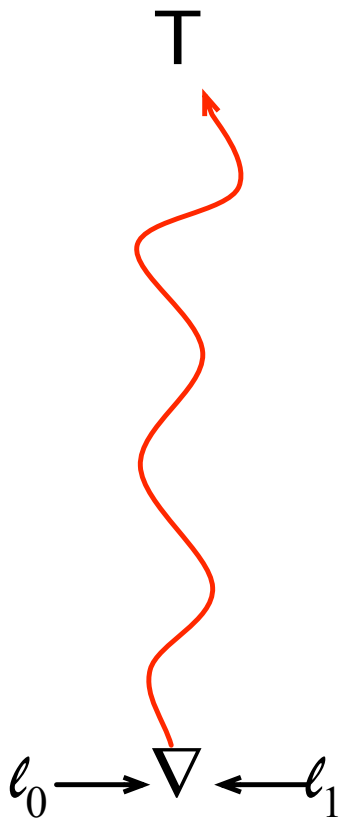


♡: Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, POPL, 1977

More About Widening

So, widening is a bit like a "join" operation (or a meet, depending on how you see the lattice). But, instead of giving the least upper bound, widening gives a value that is so large that you are sure that it cannot grow anymore in the lattice. In other words, widening is a way of "giving up" on precision. You tried a bit to find a fixed point, but you could not, then you decided to make that search as imprecise as possible.





Abstract Interpretation

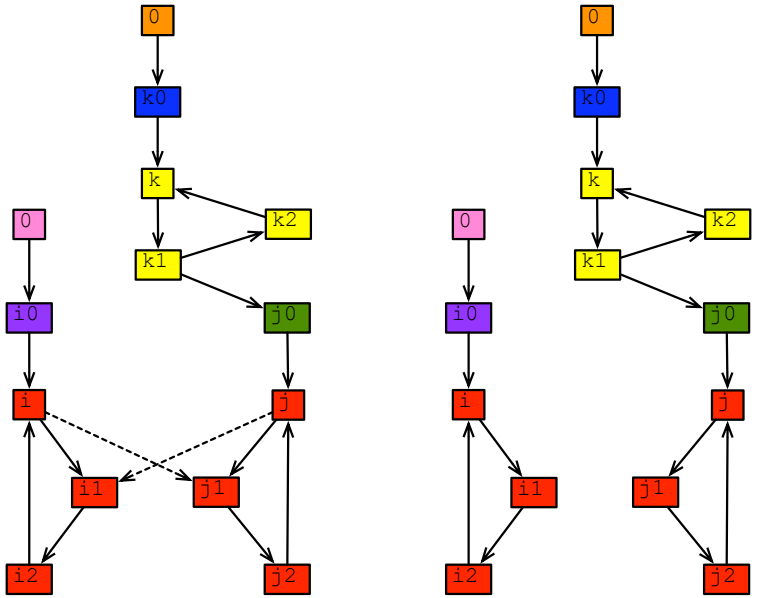
- Again, if we interpret the program, to find ranges for the variables, then our algorithm could not terminate.
- But we can interpret the program abstractly.
- Each variable v has an *abstract state* [v].
- We have an *abstract version* of each program instruction, which reads the abstract states of the variables, and uses this information to update the abstract state of the variable that the instruction defines.
- Cousot and Cousot have defined a formal way to prove that a given abstract semantics is correct, with respect to the concrete semantics.
  - But we will not see such techniques in this course.

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

Solving Range Analysis

- In the rest of this class, we will see an algorithm that solves range analysis.
- This algorithm exercises several techniques used in static analysis:
  - Strongly Connected Components.
  - Widening
  - Narrowing (ok, we have not seen this yet, but wait...)
  - Different program representations
- It is fairly precise, and quite fast.
- But, before we move into the algorithm, let's see, intuitively, how we solve range analysis.

RANGE ANALYSIS IN ONE EXAMPLE



Example

An intuition on how range analysis works

```
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 100 in the loop

Can you guess the range of j?

Example

An intuition on how range analysis works

Can you replace the range of k in j?

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

An intuition on how range analysis works

```
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]

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

Why?

- Because it is upper bounded by k.
- And we know that, inside the loop, k < 100

Can you guess the range of i?

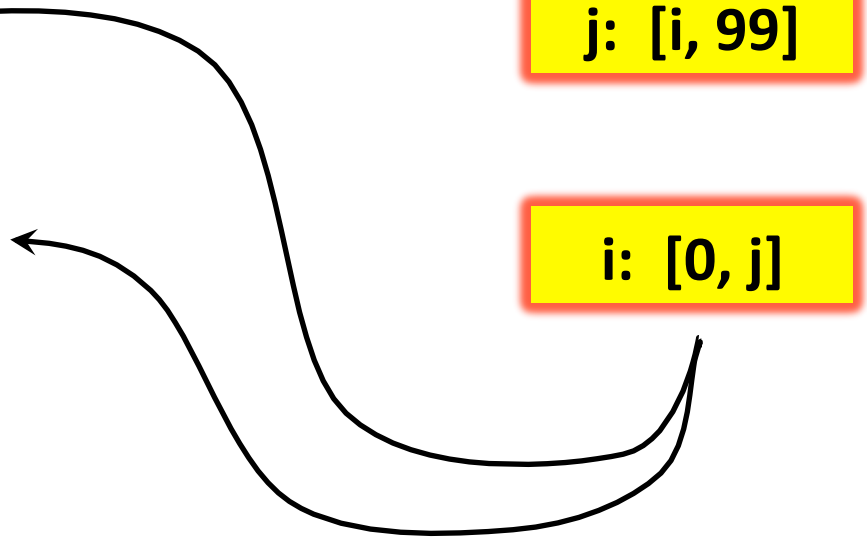
Example

An intuition on how range analysis works

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]

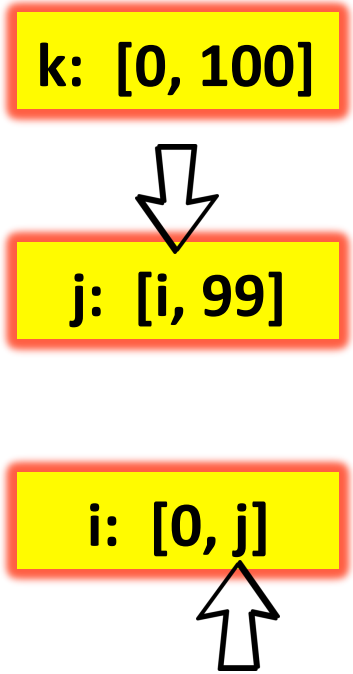
Futures

An intuition on how range analysis works

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
```

Can you estimate the values of the futures of i and j?



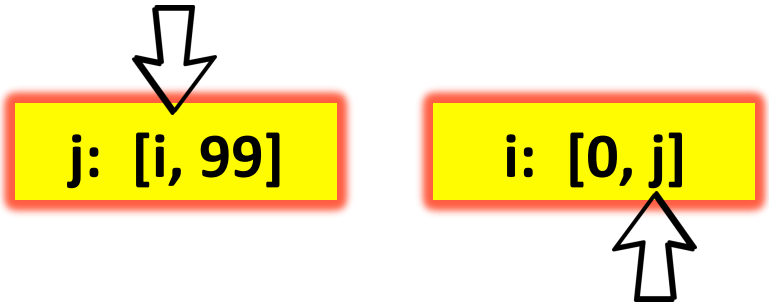
A future is like a *promise*: if we can find a good estimation for its value, than we can find a good fixed point solution to the interval analysis.

Standoff

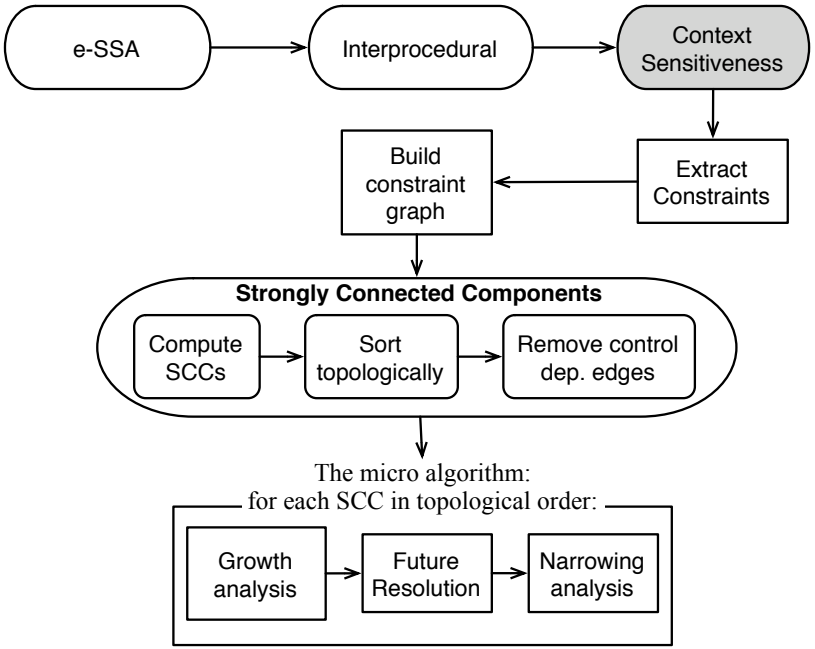
To find the bounds of j, we need the bounds of i. Yet, to find the bounds of i, we need the bounds of j.

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

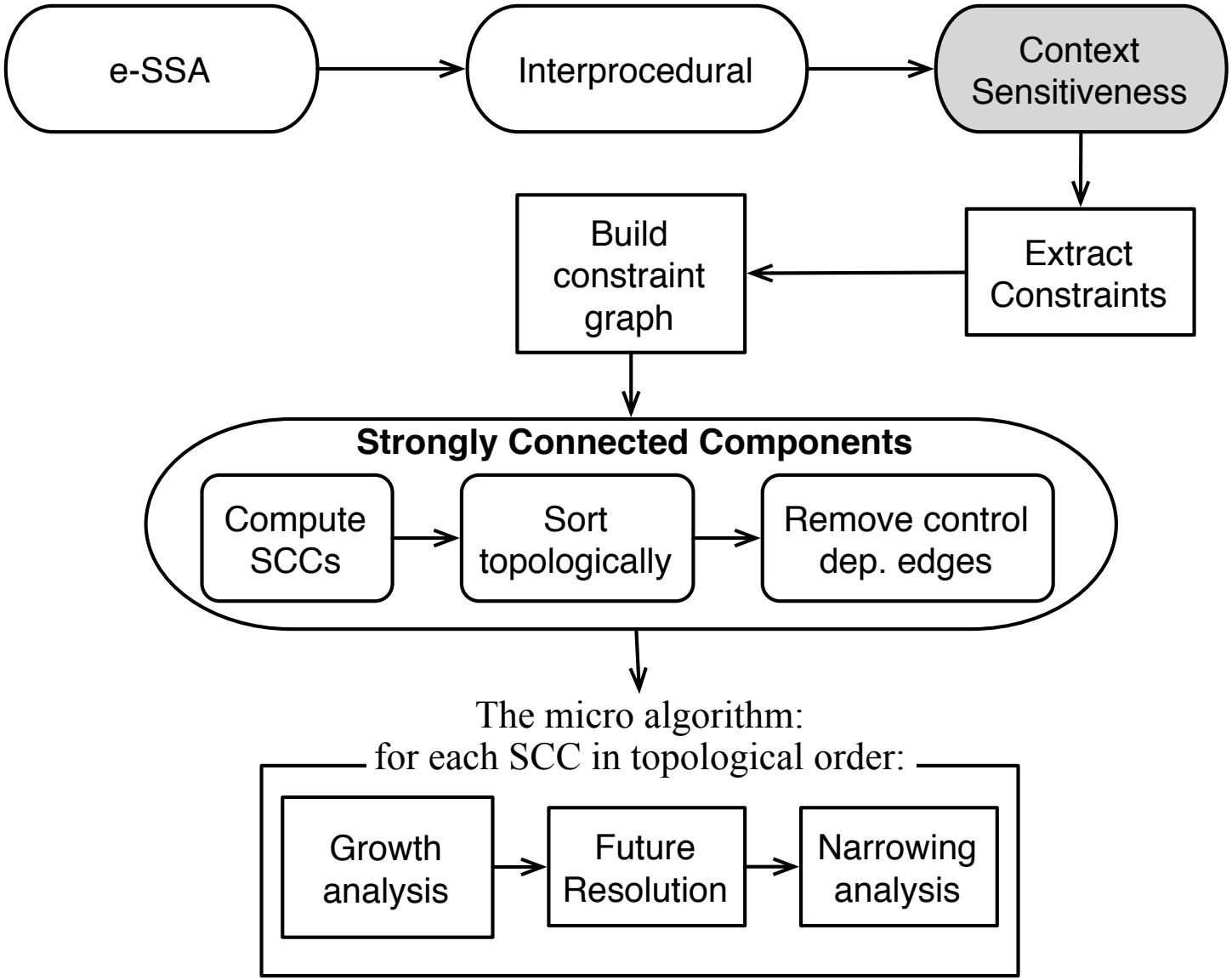
How can we solve this apparent deadlock?



The Range Analysis Algorithm

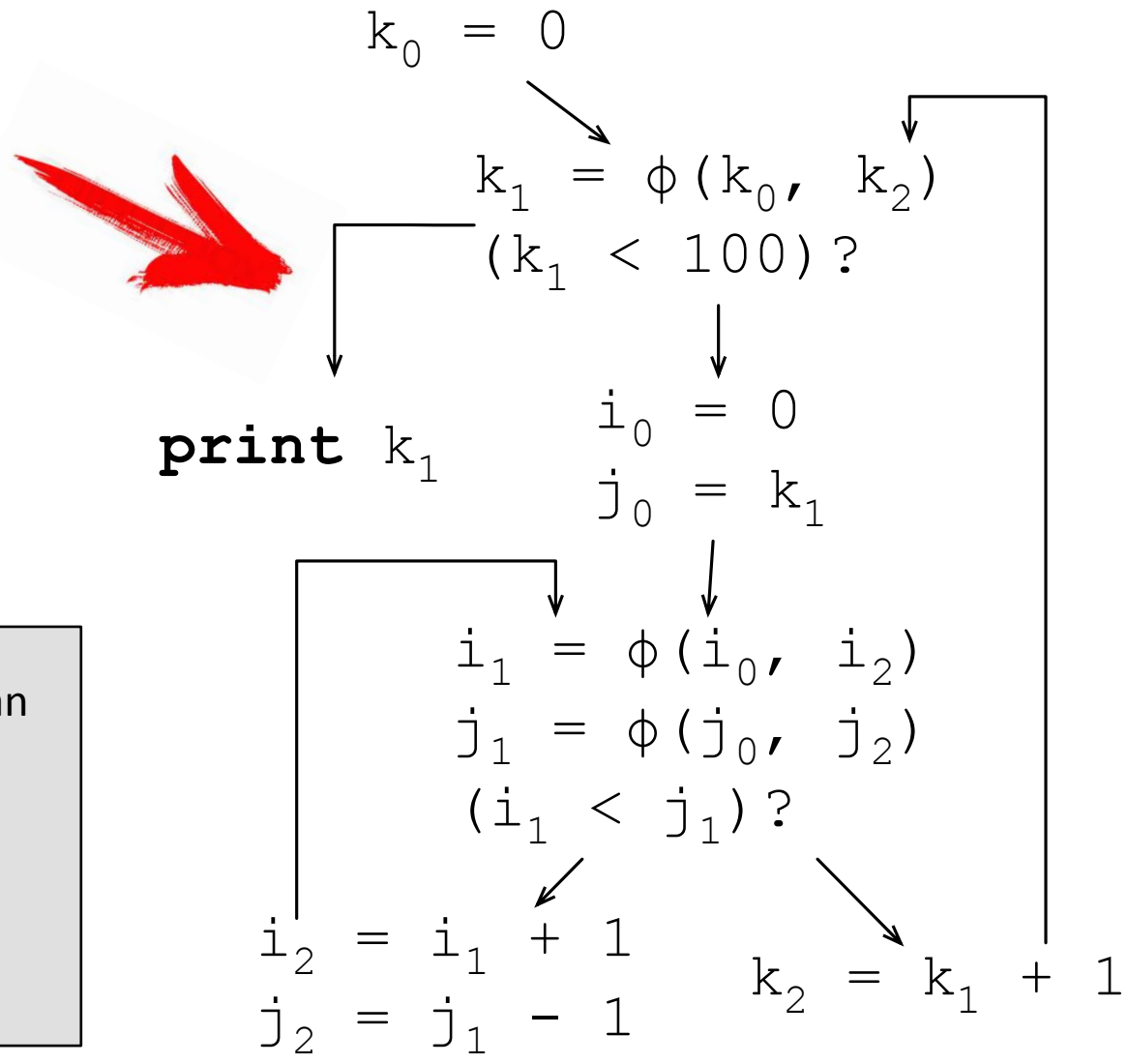


The Overall Structure of the Algorithm



Solving Range Analysis Sparsely

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

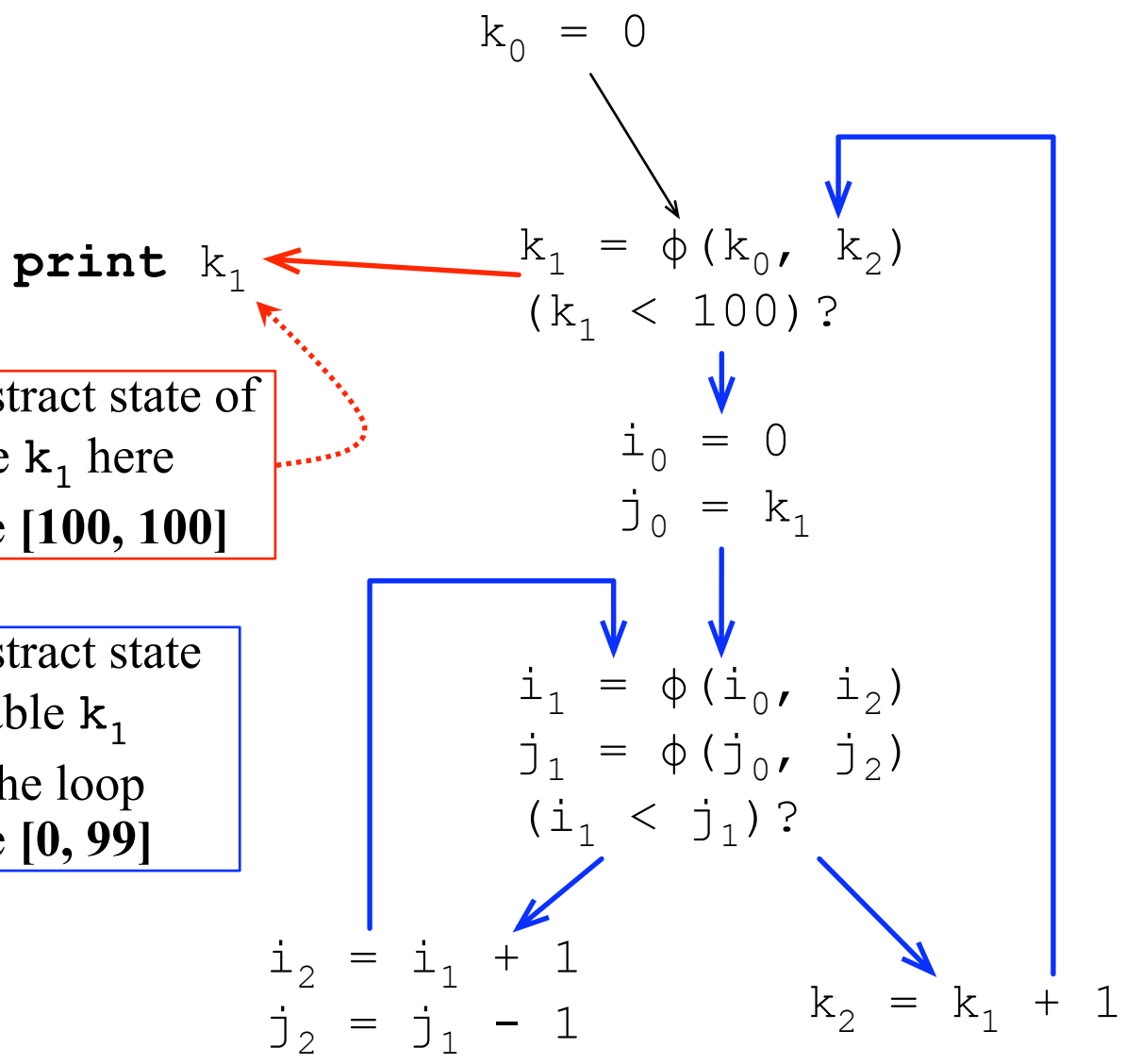


We would like to associate an abstract state, i.e., a range, with each variable. What would be the abstract state associated with variable k_0? What about k_1?

Solving Range Analysis Sparsely

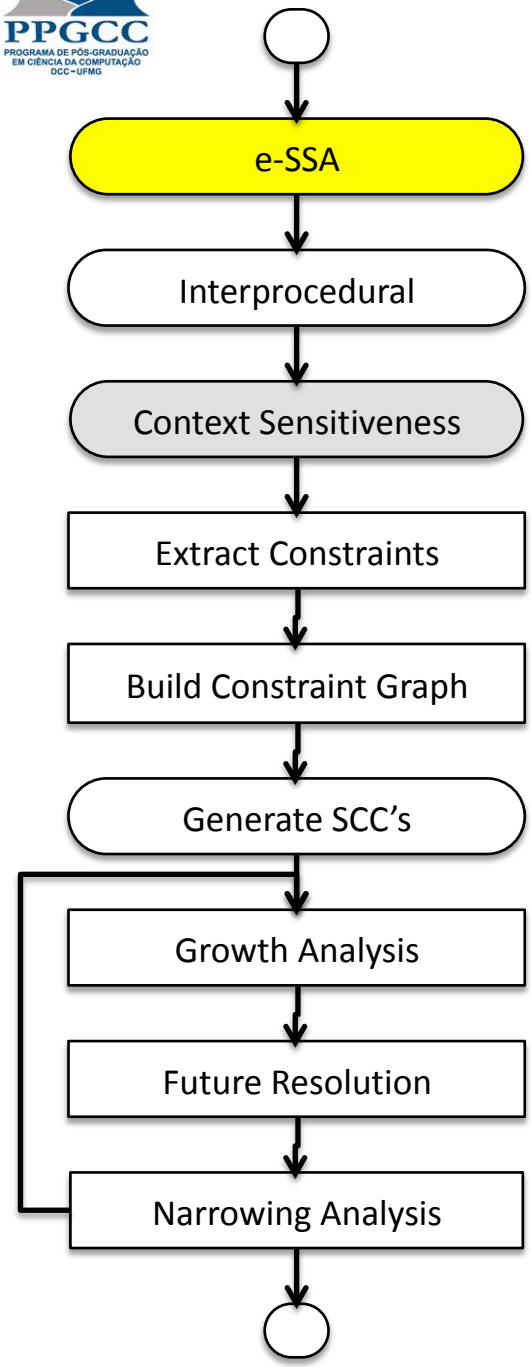
1) So, in the end, what should be the abstract state of k_1?

2) Can we improve these bounds?



The abstract state of variable k_1 here must be **[100, 100]**

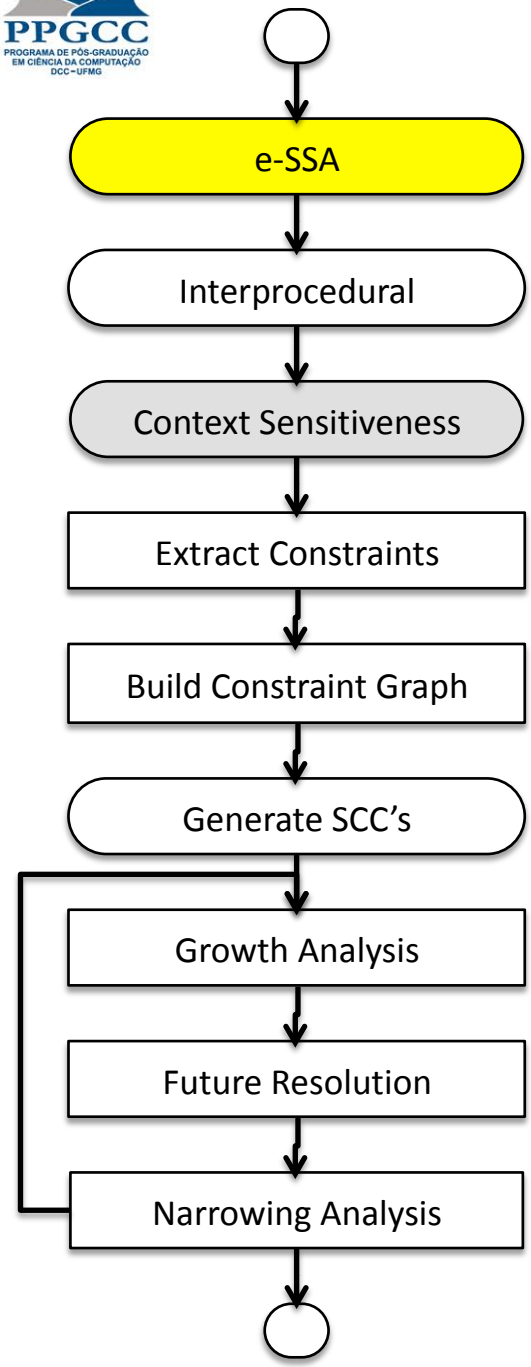
The abstract state of variable k_1 inside the loop must be **[0, 99]**



Extended Static Single Assignment Form

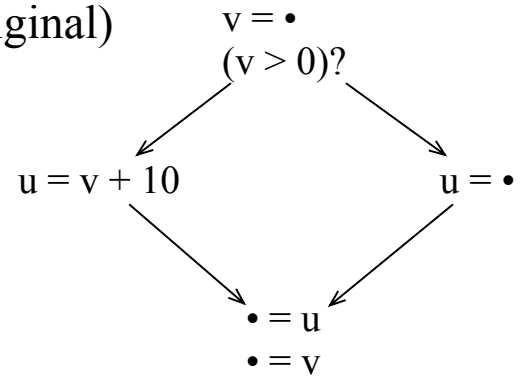
- The first step to solve range analysis is to convert the target program into e-SSA form[◇].
- This intermediate representation let us learn from conditional tests.
  - Hence, it improves precision of the range analysis.
- It increases the program size, but not too much.
  - Less than 10% on the average.

◇: ABCD: eliminating array bounds checks on demand (PLDI'00)

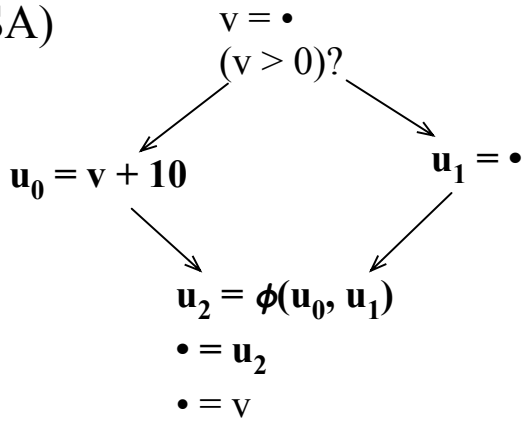


Extended Static Single Assignment Form

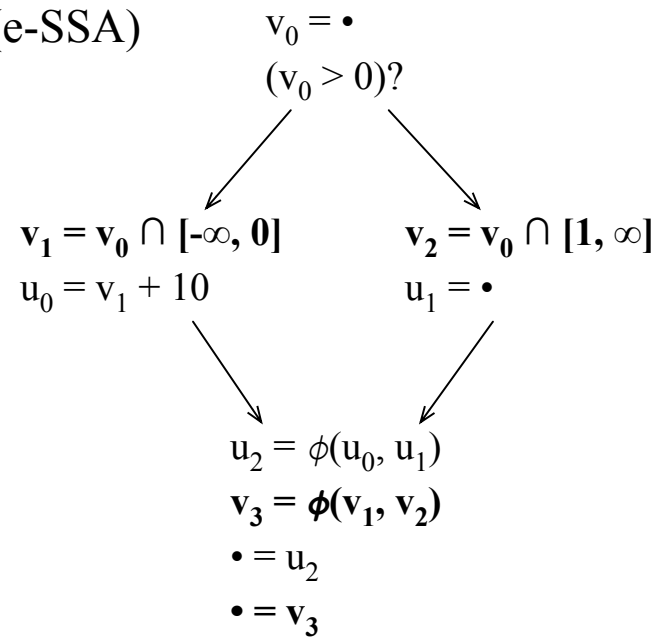
(original)



(SSA)



(e-SSA)



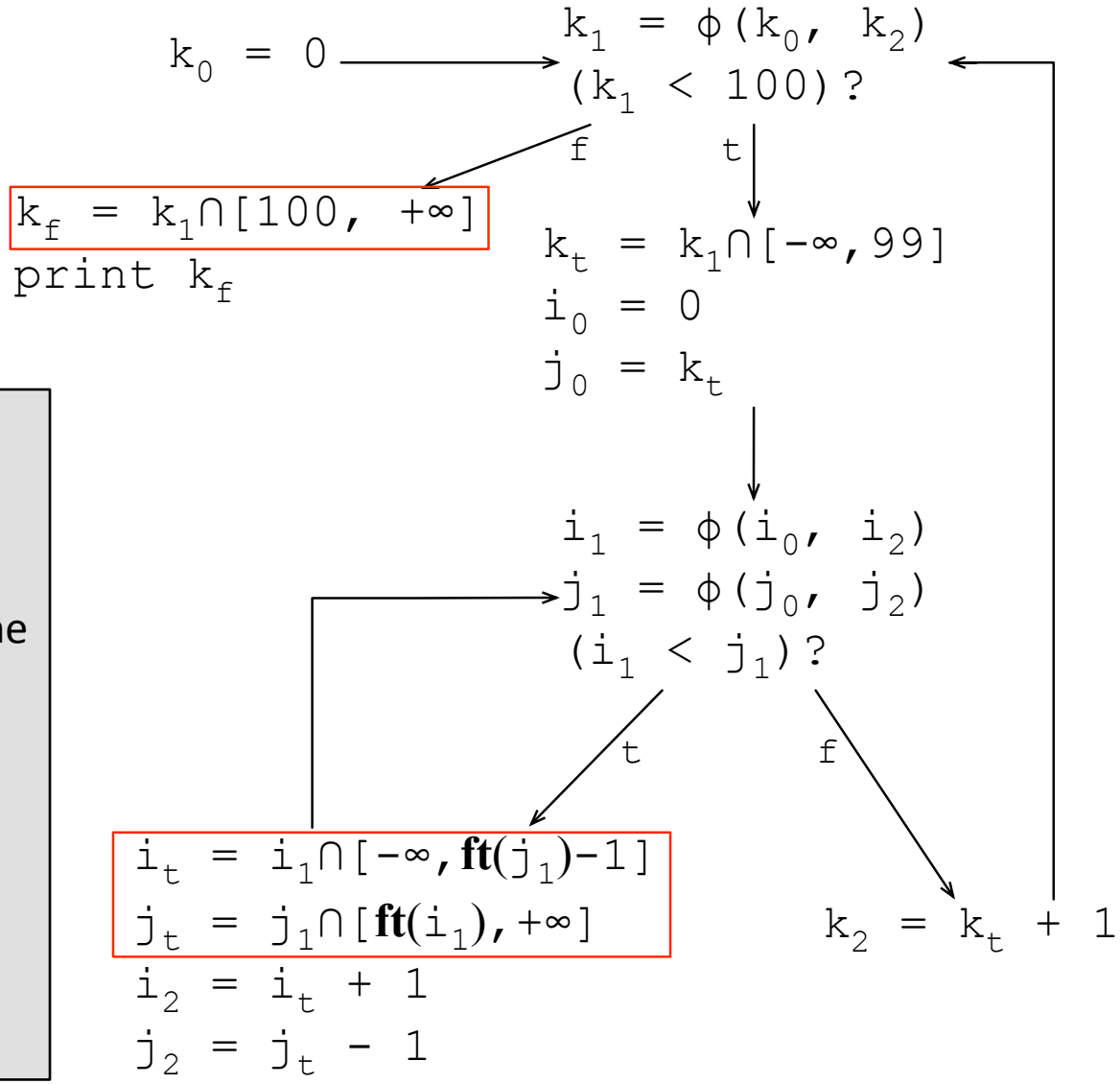
To generate e-SSA form, we split the live ranges of variables that are used right after conditional tests.

Extended Static Single Assignment Form

1) What is now $[k_1]$, $[k_f]$, and $[k_t]$?

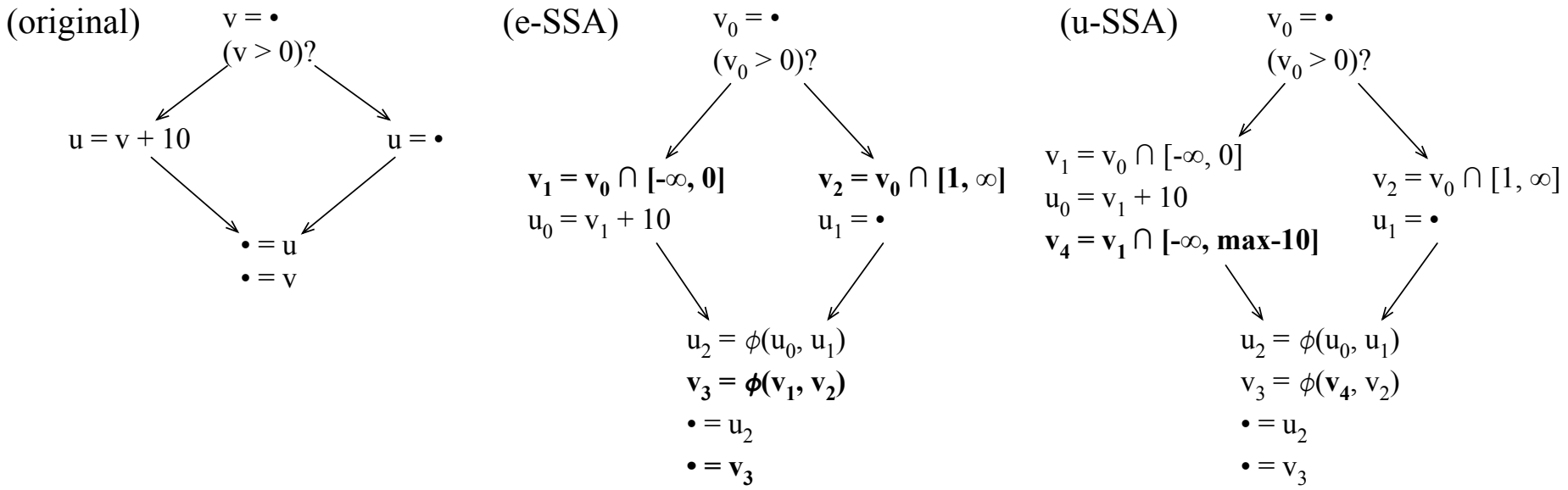
2) How does e-SSA improves the precision of our results?

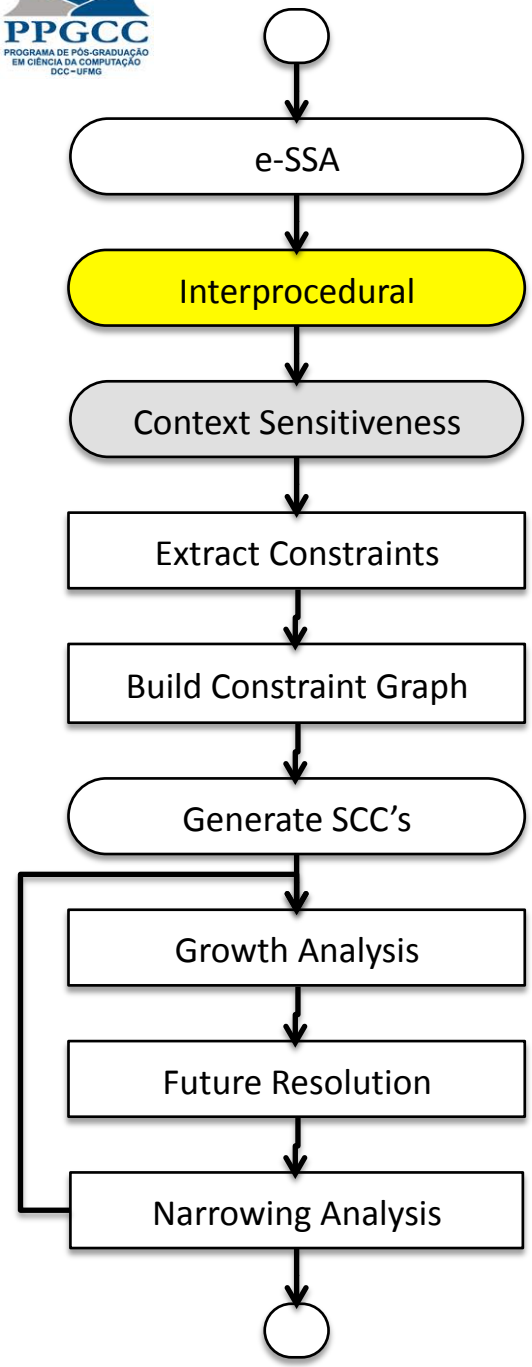
3) Can you think about any other intermediate representation that could improve our results even further?



Splitting After Uses

Let's imagine that we are guarding programs against integer overflows, and that an overflow would abort the program. *In this case*, we can be more aggressive. If we have an operation such as u = v + 10, and an overflow did not happen, then we know that after u is defined, v must be less than MAX_INT – 10.



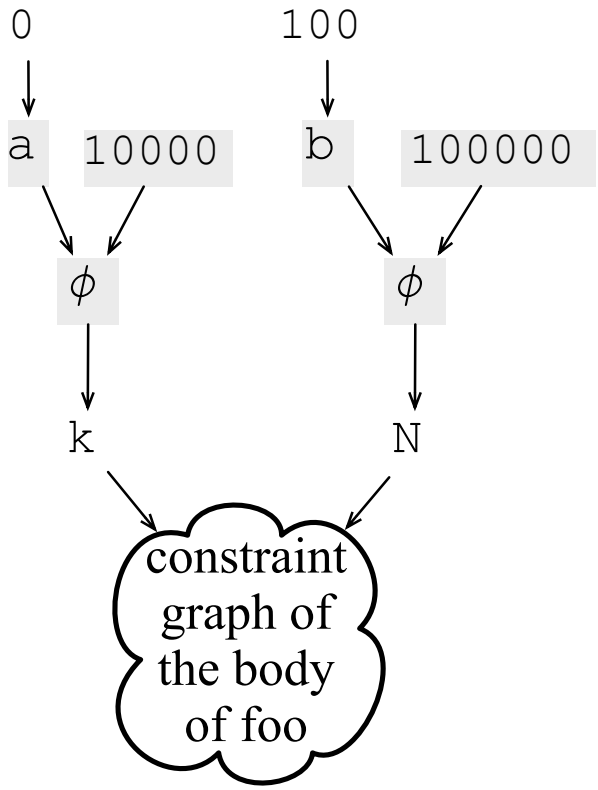


Inter-procedural Analysis

We can make our analysis inter-procedural by adding dependencies between formal and actual parameters.

```
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
```



Inter-procedural Analysis

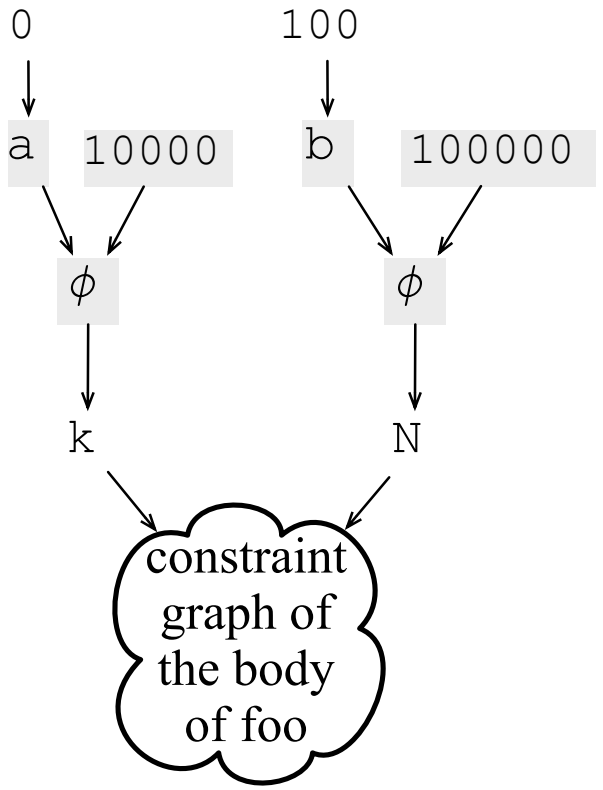
1) What is [k] in this example?

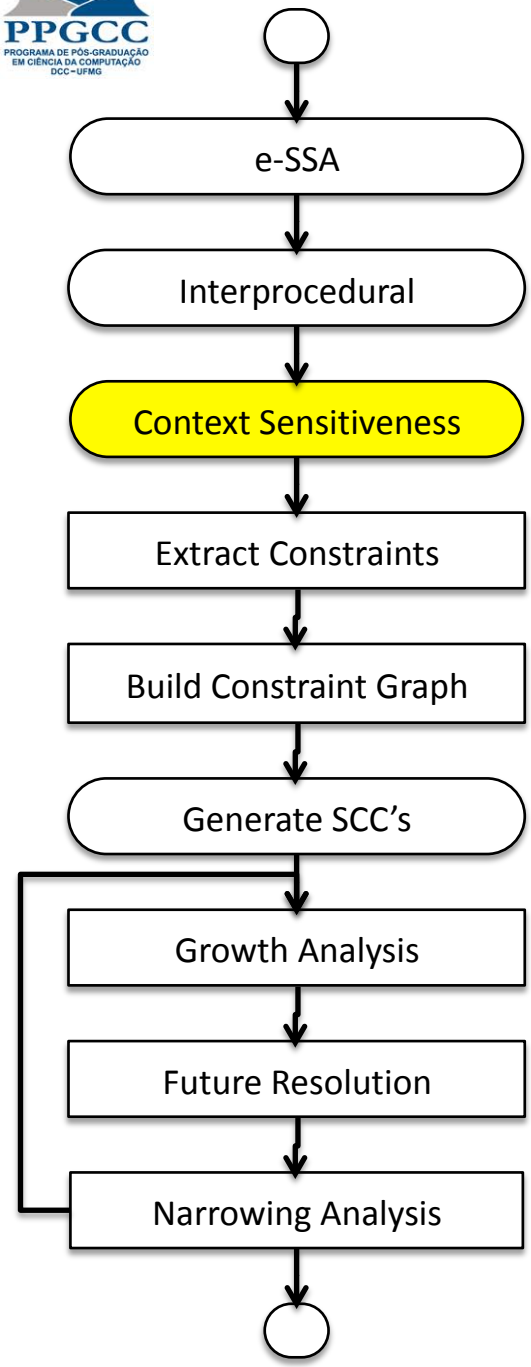
2) How could we make this analysis more precise?

We can make our analysis inter-procedural by adding dependencies between formal and actual parameters.

```
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
```





Context Sensitiveness

We can distinguish different calling sites if we do function inlining.

```
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
```

Context Sensitiveness

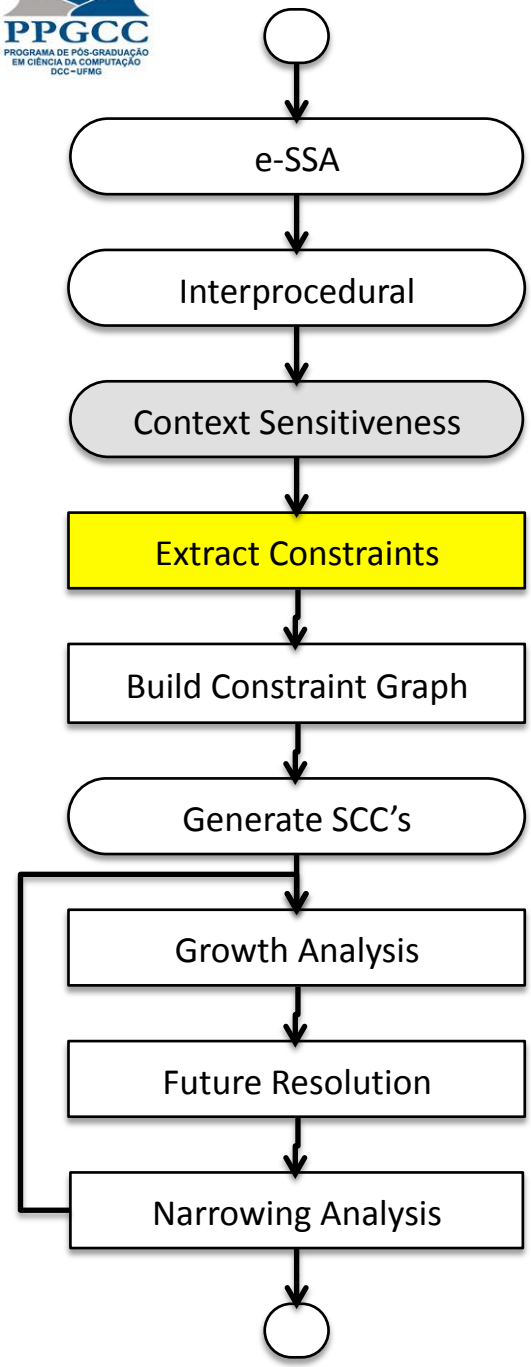
We can distinguish different calling sites if we do function inlining.

1) What is $[k_a]$, and $[k_b]$ in this example?
2) What are the disadvantages of doing function inlining?
3) In which other ways could we make our analysis context sensitive?

```
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

To solve range analysis, we must solve a constraint system. Constraints are built around an evaluation function *e*.

$$Y = [l, u] \qquad e(Y) = [l, u]$$

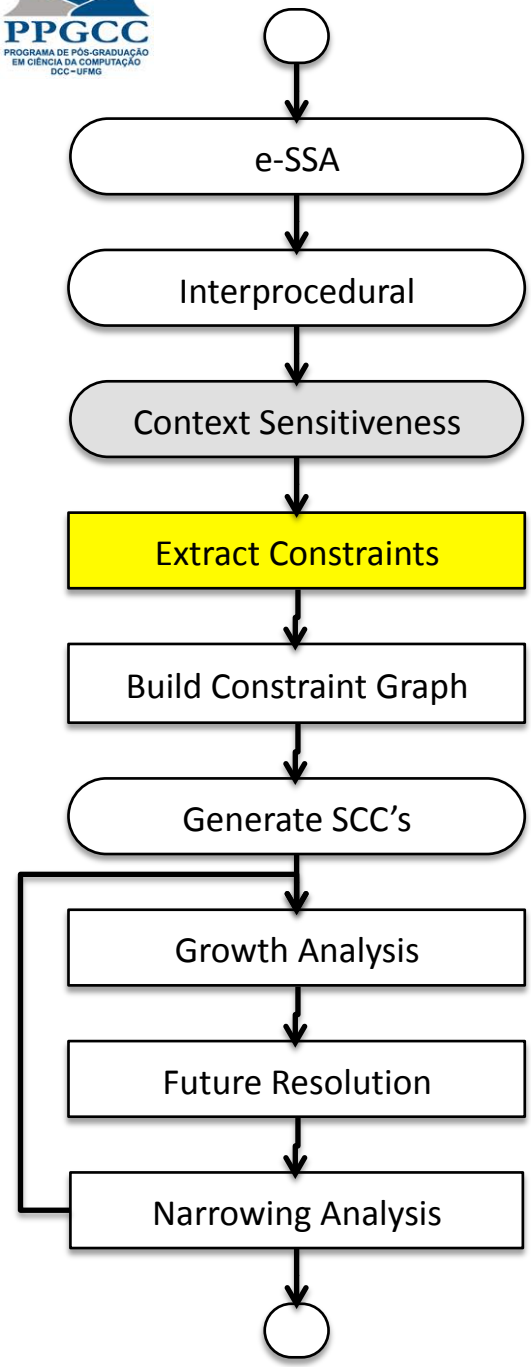
$$Y = \phi(X_1, X_2) \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1, u_1] \sqcup [l_2, u_2]}$$

$$Y = X_1 + X_2 \qquad \frac{I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [l_1 + l_2, u_1 + u_2]}$$

$$Y = X_1 \times X_2 \qquad \frac{L = \{l_1 l_2, l_1 u_2, u_1 l_2, u_1 u_2\} \quad I[X_1] = [l_1, u_1] \qquad I[X_2] = [l_2, u_2]}{e(Y) = [\min(L), \max(L)]}$$

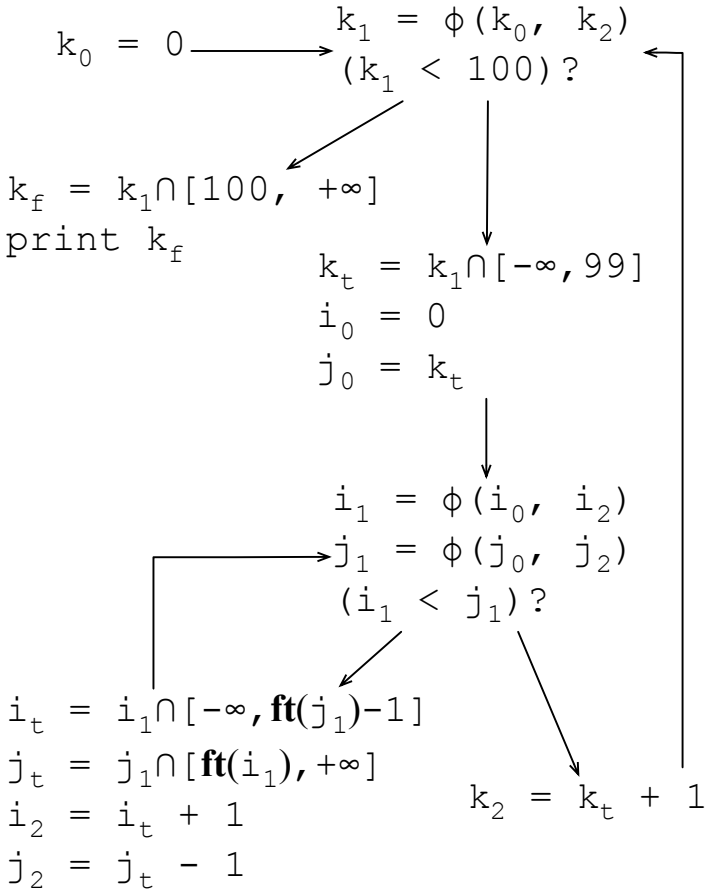
$$Y = aX + b \qquad \frac{I[X] = [l, u] \qquad k_l = al + b \qquad k_u = au + b}{e(Y) = [\min(k_l, k_u), \max(k_l, k_u)]}$$

$$Y = X \sqcap [l', u'] \qquad \frac{I[X] = [l, u]}{e(Y) = [l, u] \sqcap [l', u']}$$

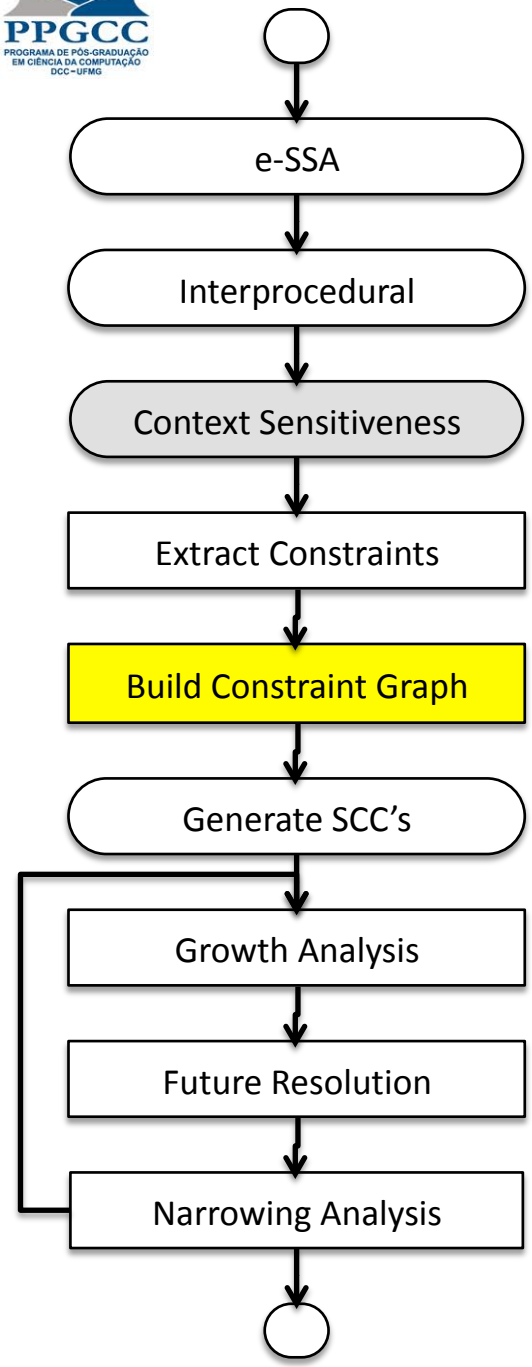


Extraction of Constraints

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.



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



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 → v

The Constraint Graph

$k_0 = 0$

$k_1 = \phi(k_0, k_2)$

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

$i_0 = 0$

$j_0 = k_t$

$i_1 = \phi(i_0, i_2)$

$j_1 = \phi(j_0, j_2)$

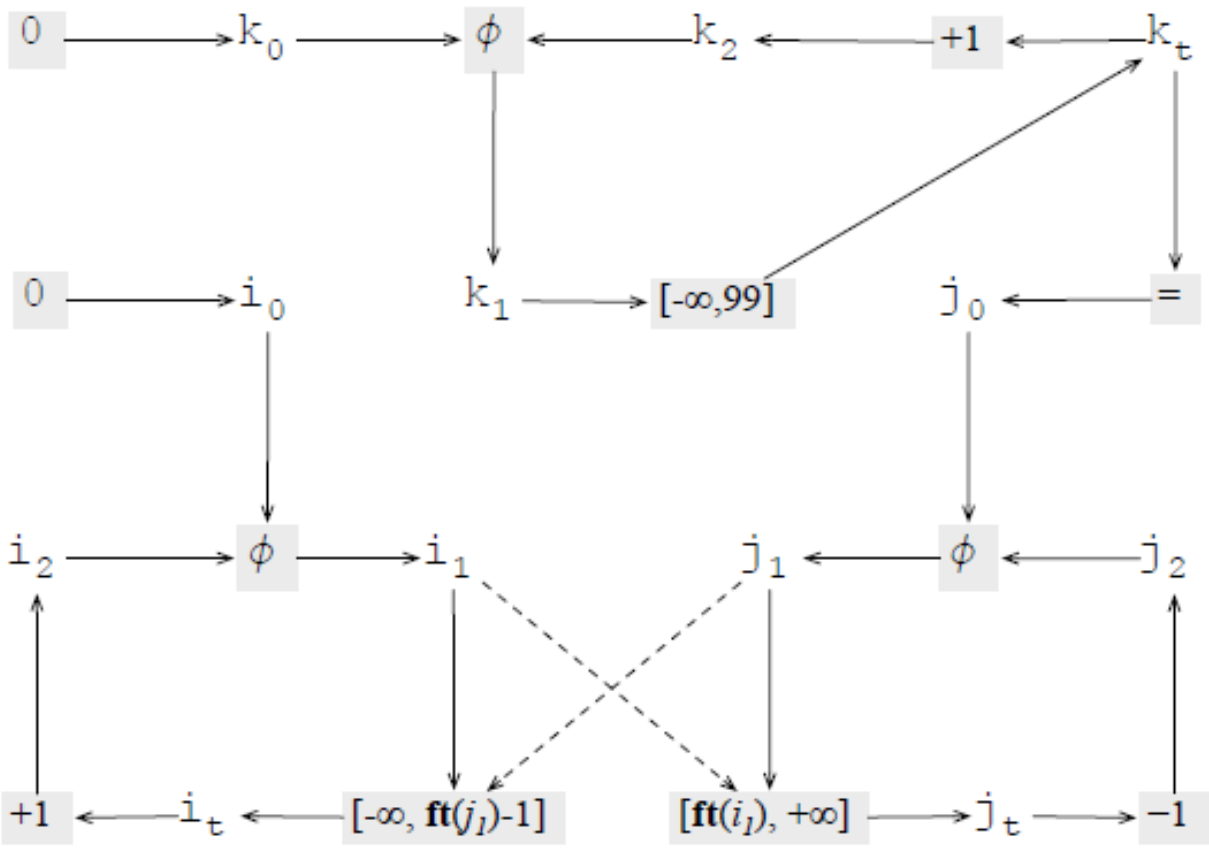
$i_2 = i_t + 1$

$j_2 = j_t - 1$

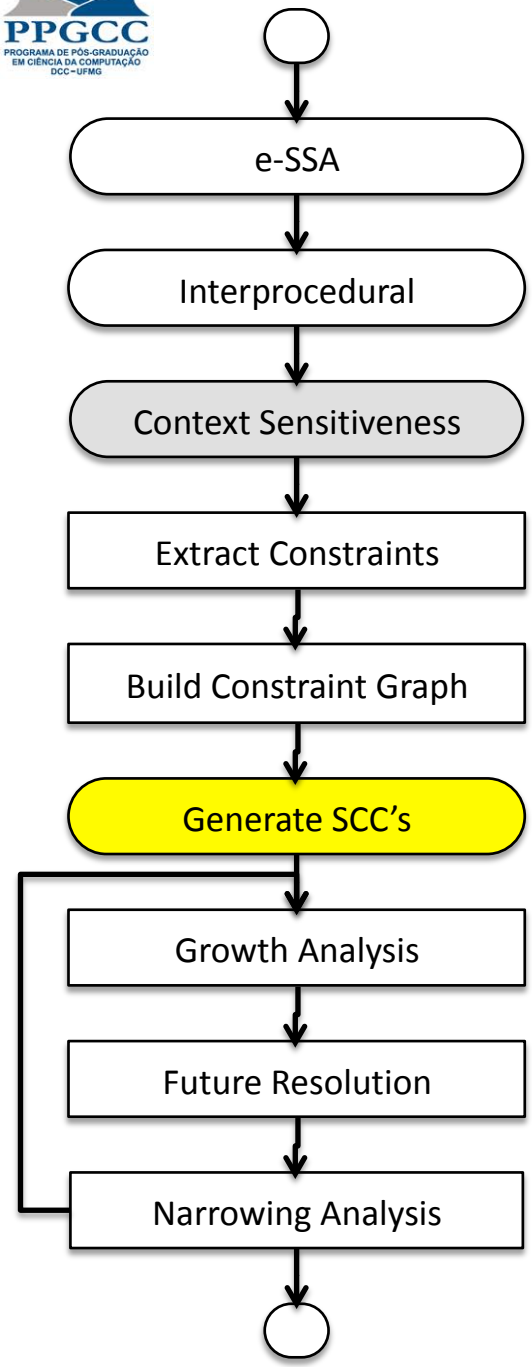
$k_2 = k_t + 1$

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

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



The dashed edges denote dependences between a variable and the owner of its *future* bound.



Strongly Connected Components

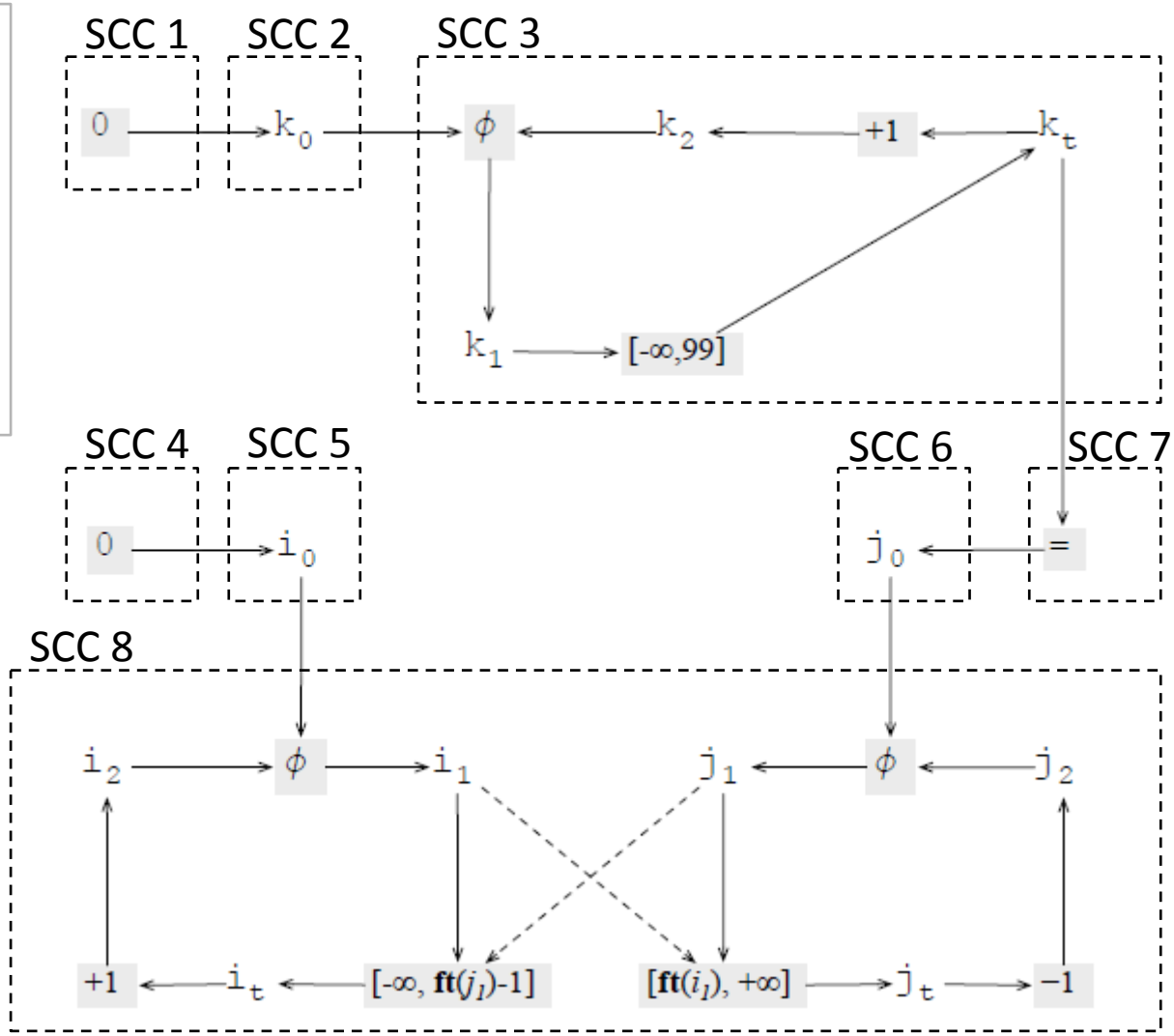
- As we have seen in the beginning of our course, we can improve the speed of the constraint solver if we process strongly connected components of our constraint graph in topological order.
- Furthermore, SCCs also improves the precision of our results, as we will see soon.

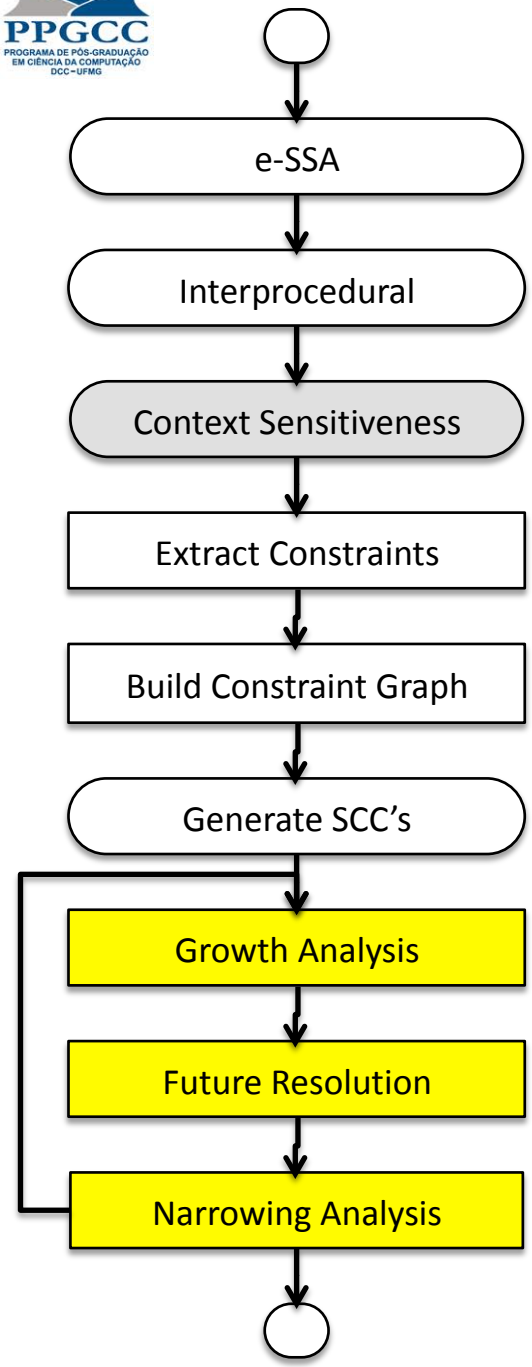
Do you recall the algorithm to find the strong components of a graph?

Strongly Connected Components

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

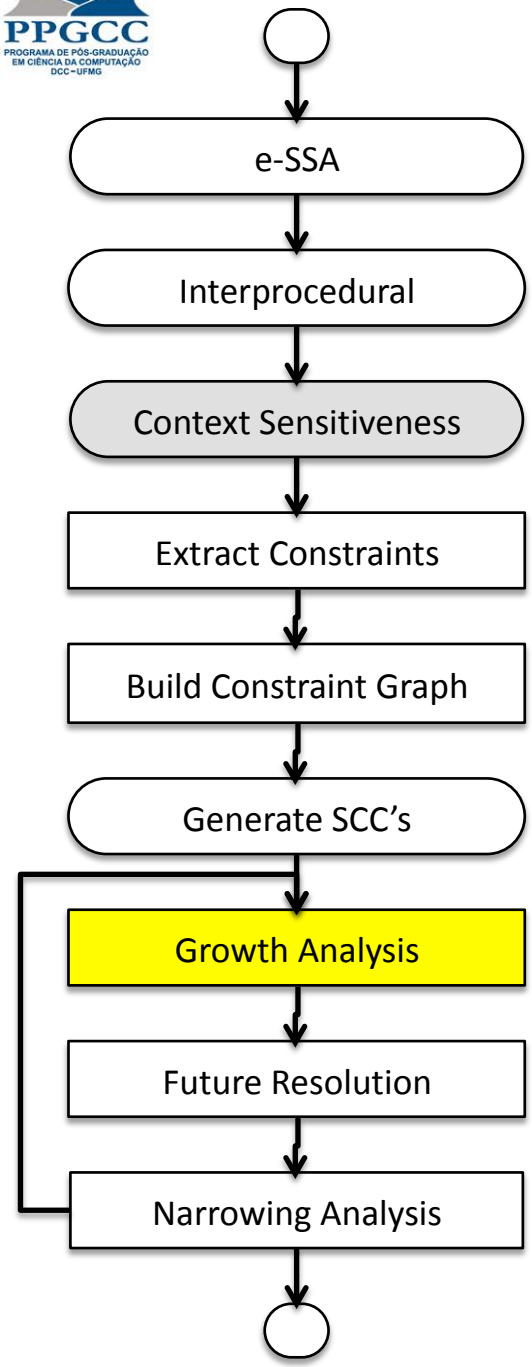
Quick recap: why can we solve range analysis faster if we use SCCs?





The Three-Phases Approach

- In the next phase of this 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 imprecision of the widening phase by considering the bounds in conditional tests.
- Widening ensures termination. Narrowing improves precision.



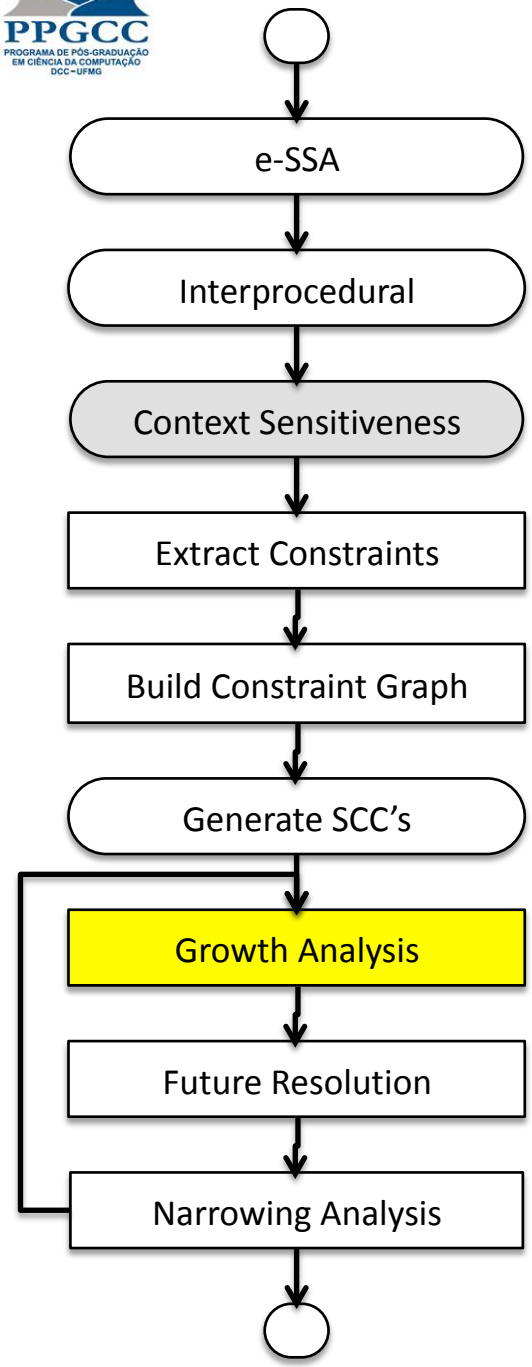
Widening

- 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 $+\infty$.
- We do not consider any bounds imposed by conditionals at this point.

$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)} \qquad \frac{e(Y)_\downarrow < I[Y]_\downarrow \qquad e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [-\infty, +\infty]}$$

$$\frac{e(Y)_\downarrow < I[Y]_\downarrow}{I[Y] \leftarrow [-\infty, I[Y]_\uparrow]} \qquad \frac{e(Y)_\uparrow > I[Y]_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, +\infty]}$$

How do we read this notation?



How to read this Notation

Each constraint variable Y is associated with a current interval I[Y].

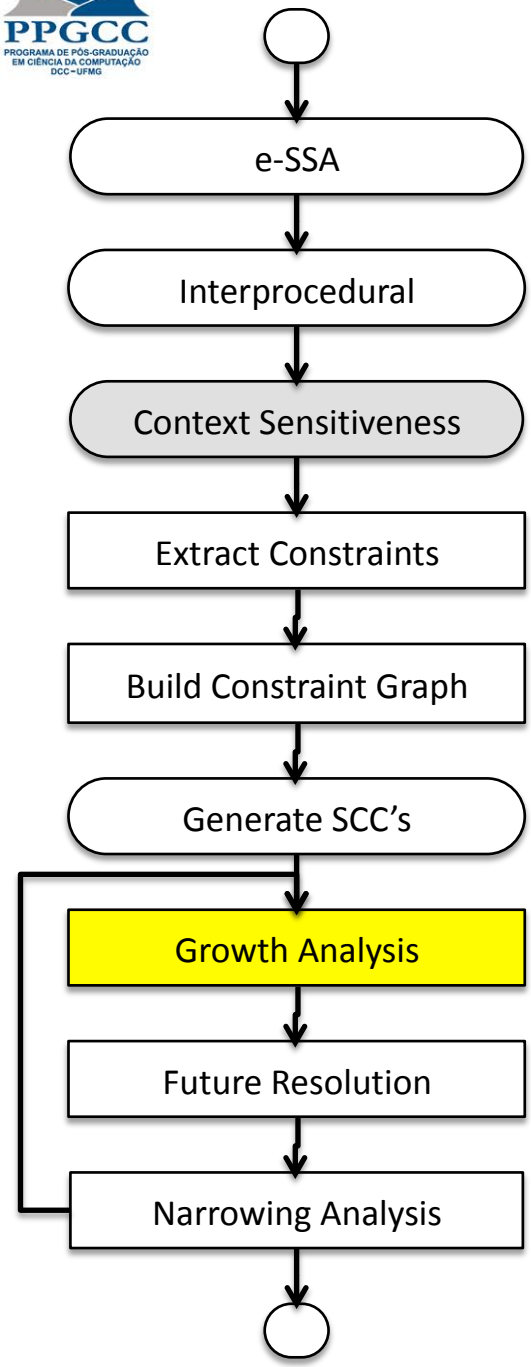
The abstract semantics of this variable is given by an evaluation function e(Y).

If e(Y) will decrease the lower bound of Y, then we make this lower bound the most imprecise, e.g., we assign it minus infinite.

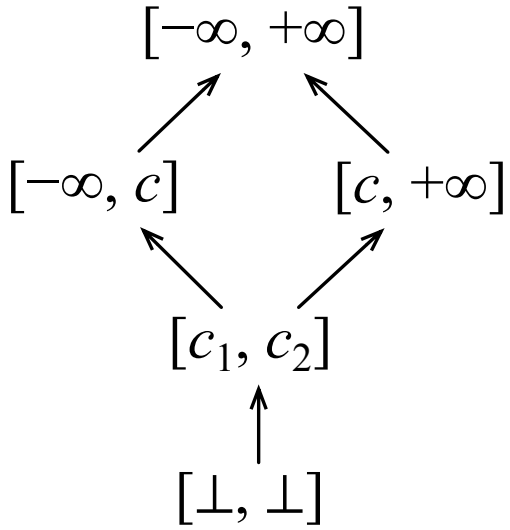
The other constraints have similar meanings.

$$\frac{e(Y)_{\downarrow} < I[Y]_{\downarrow}}{I[Y] \leftarrow [-\infty, I[Y]_{\uparrow}]}$$

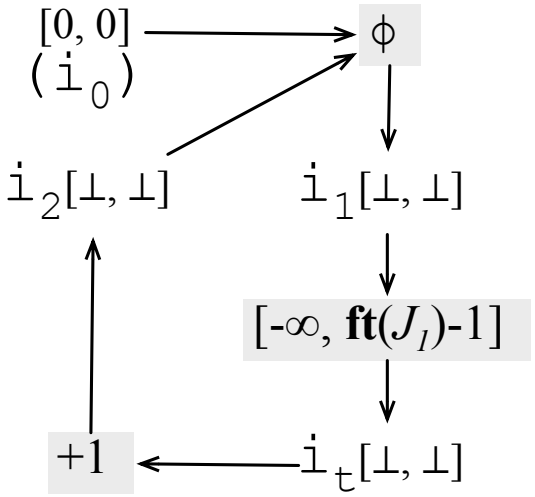
How is the lattice of this growth analysis?



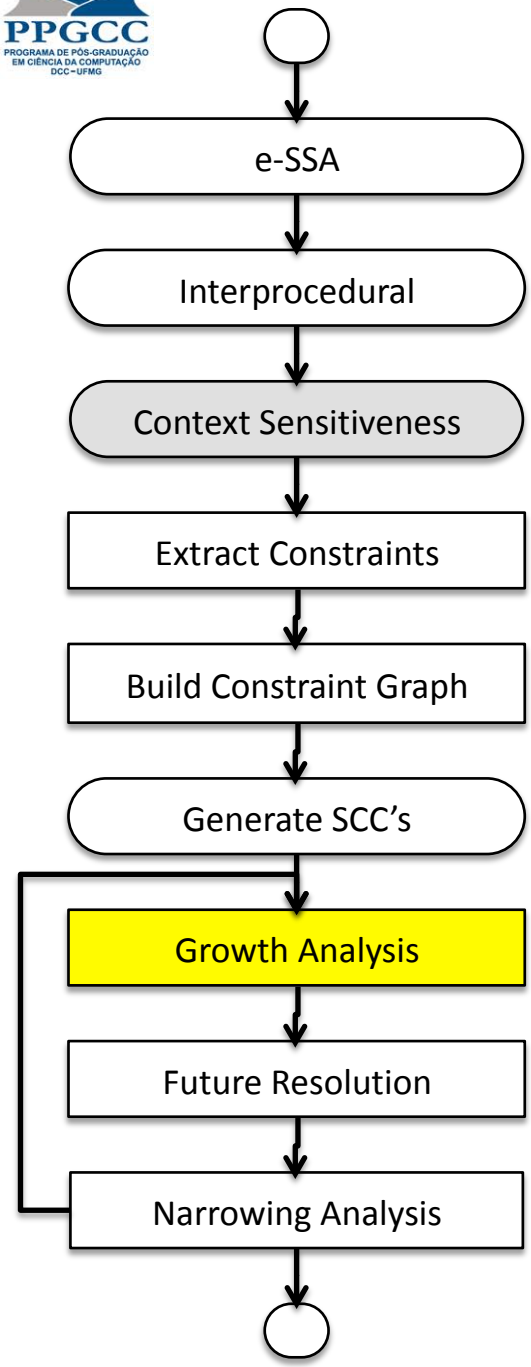
Widening



We can use this very simple widening operator to find out how each variable grows. Notice that our lattice is pretty short now.



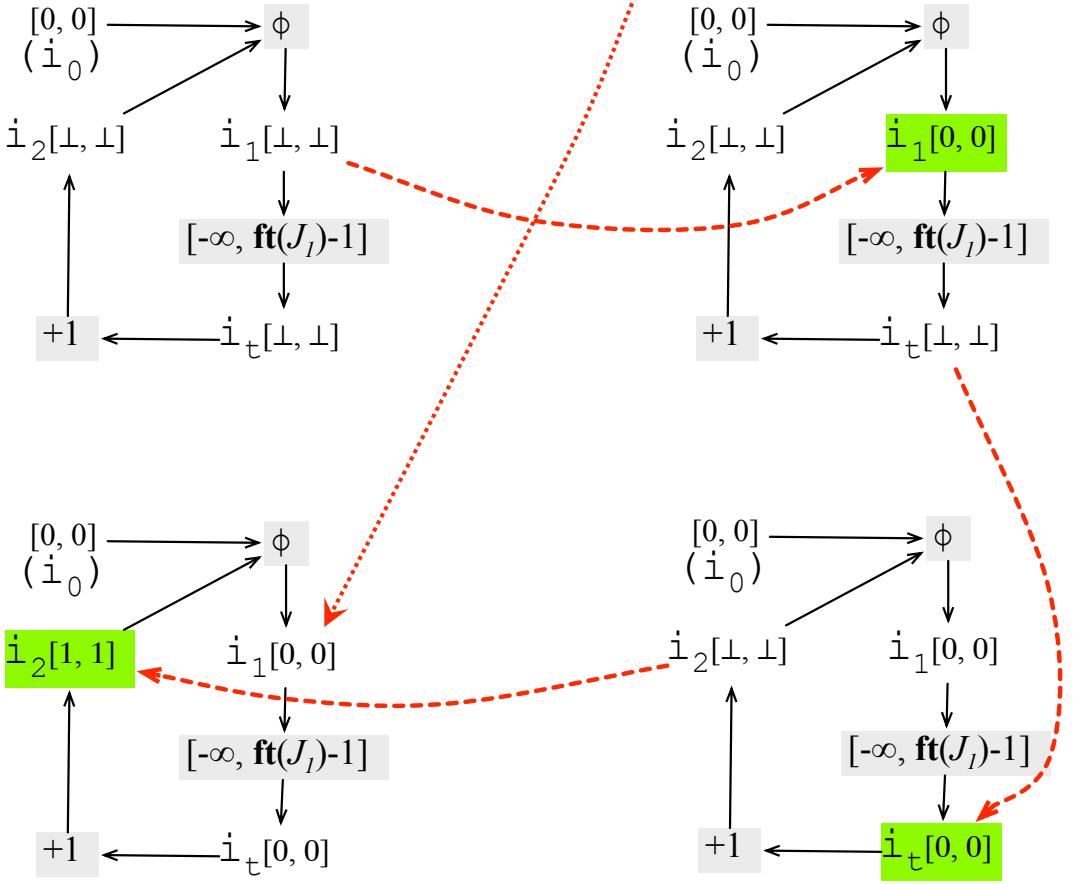
What is the result of our growth analysis in this program on the left?

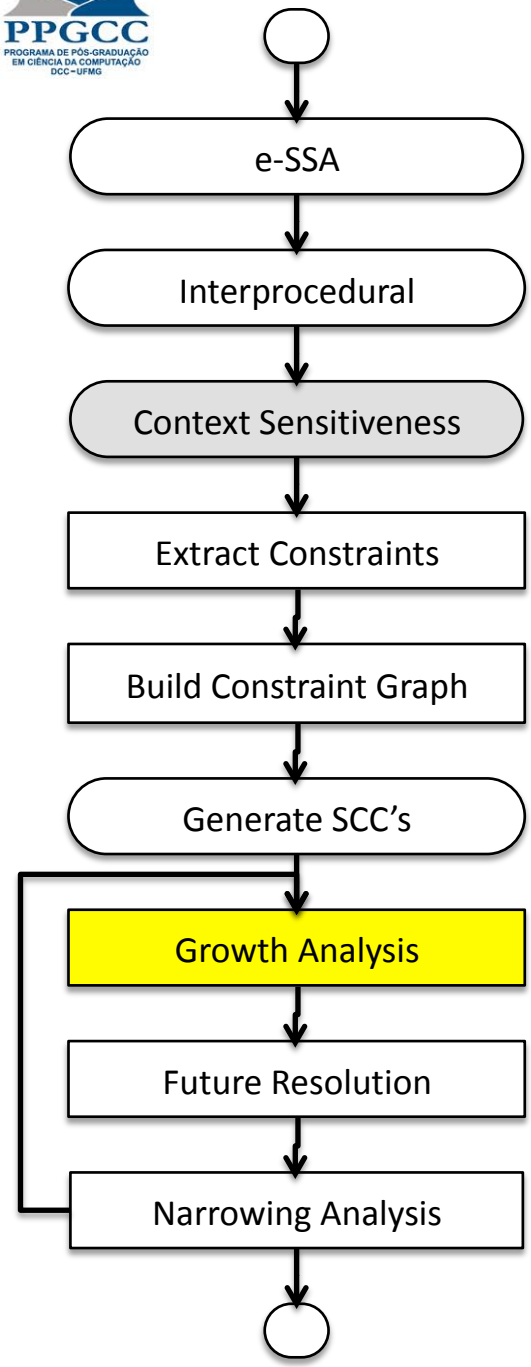


Widening

$$\frac{I[Y] = [\bot, \bot]}{I[Y] \leftarrow e(Y)}$$

What is going to be the next state of i_1?

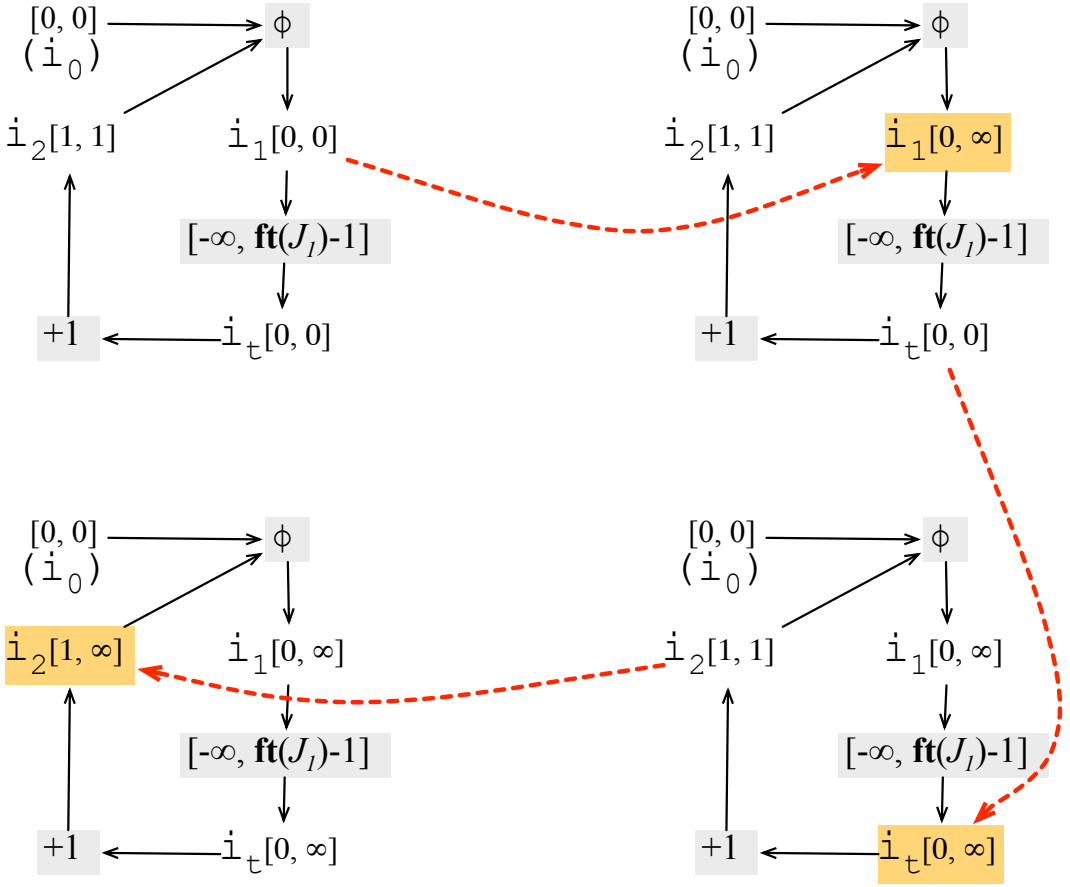


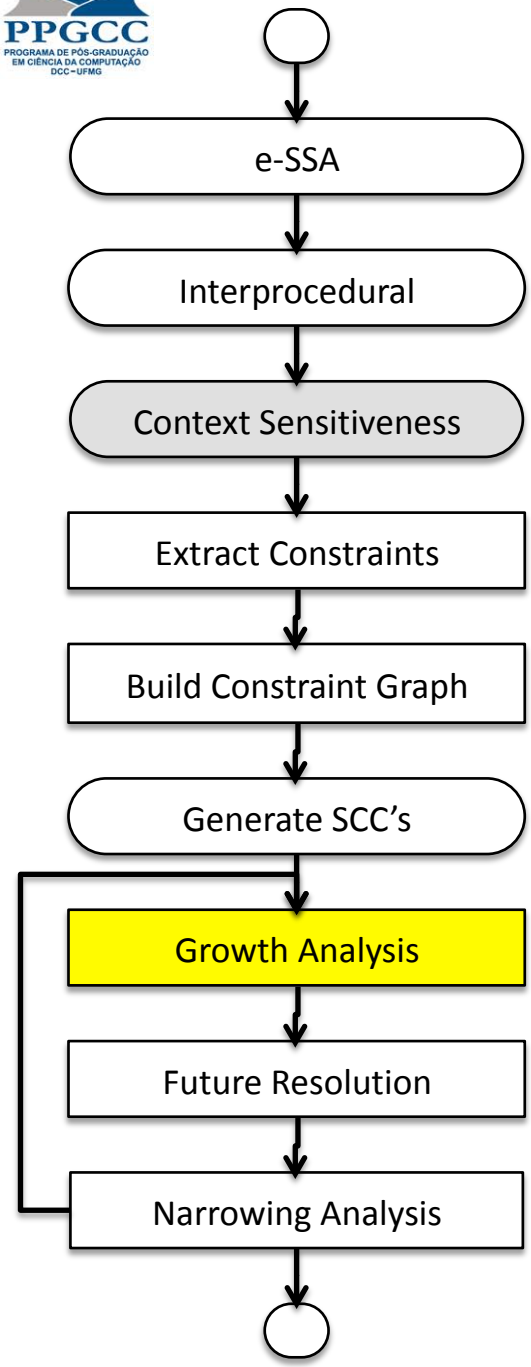


Widening

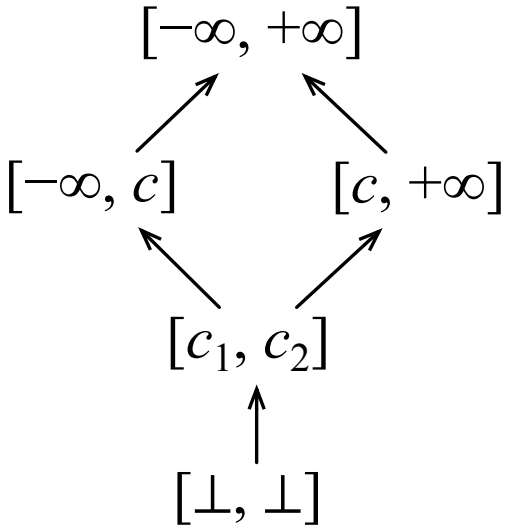
$$\frac{e(Y)_{\uparrow} > I[Y]_{\uparrow}}{I[Y] \leftarrow [I[Y]_{\downarrow}, +\infty]}$$

What is the next change?

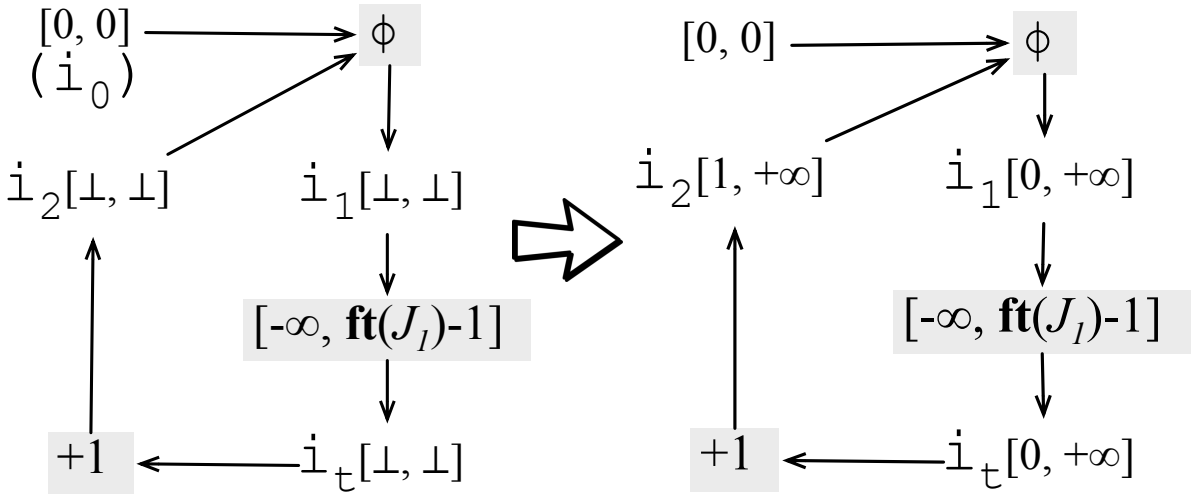


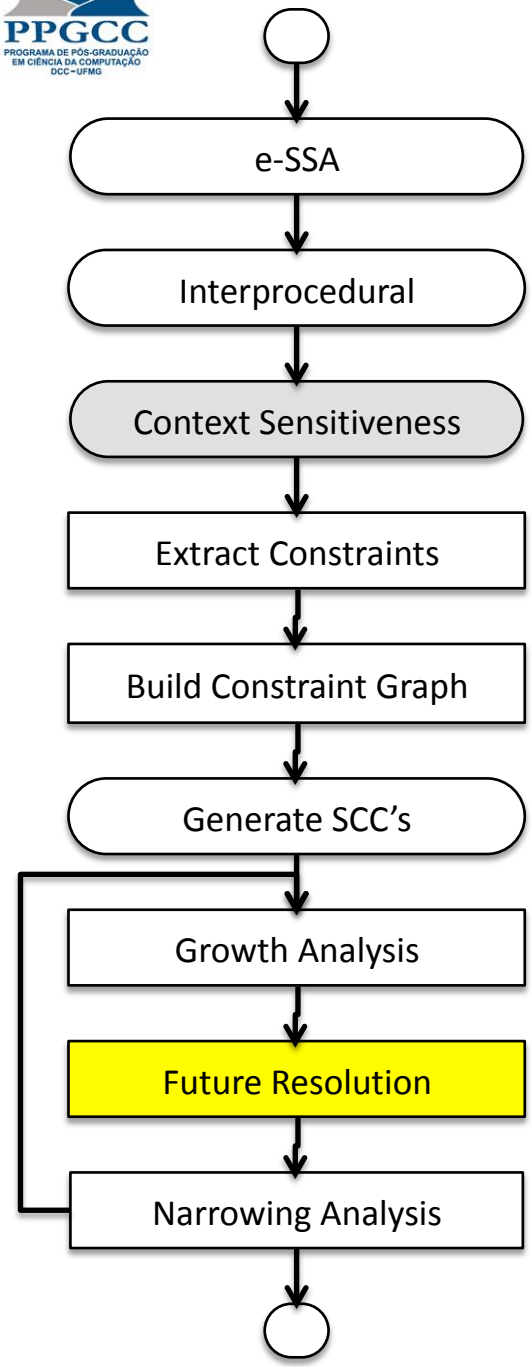


Widening



What is the complexity of the growth analysis?

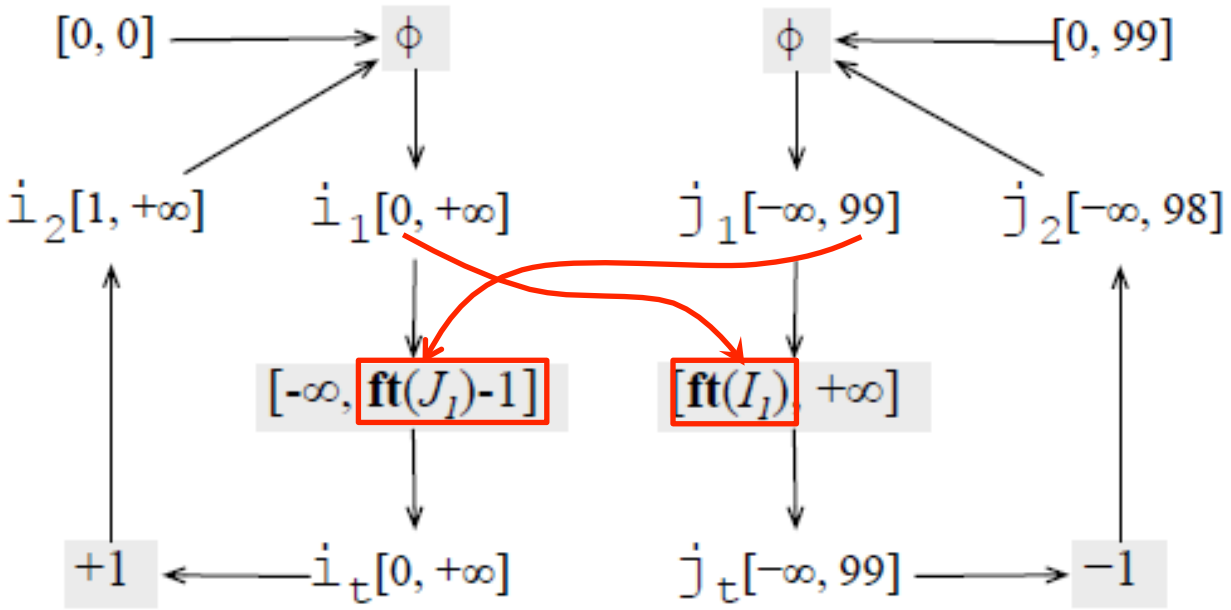


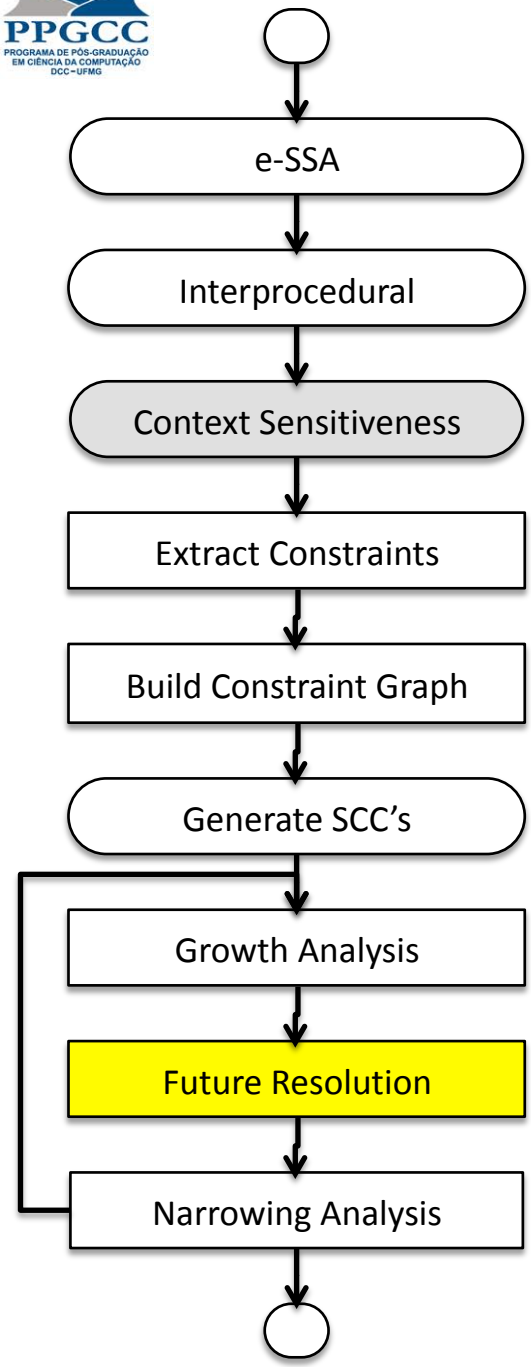


Future Resolution

How is the constraint graph after future resolution?

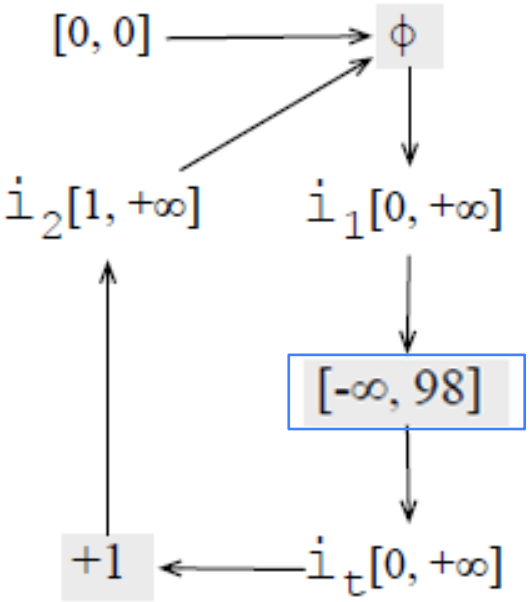
- In the next step, we must replace futures by concrete bounds.
- Futures exist at intersection constraints.
- If a variable's bound remained stable during the widening step, then we know that it can only shrink.

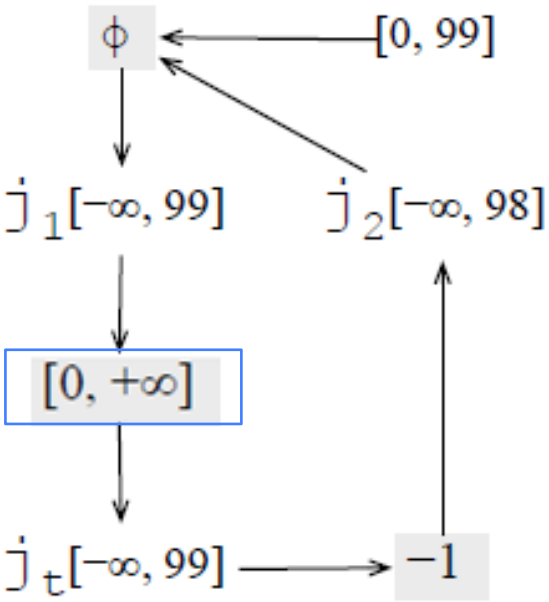


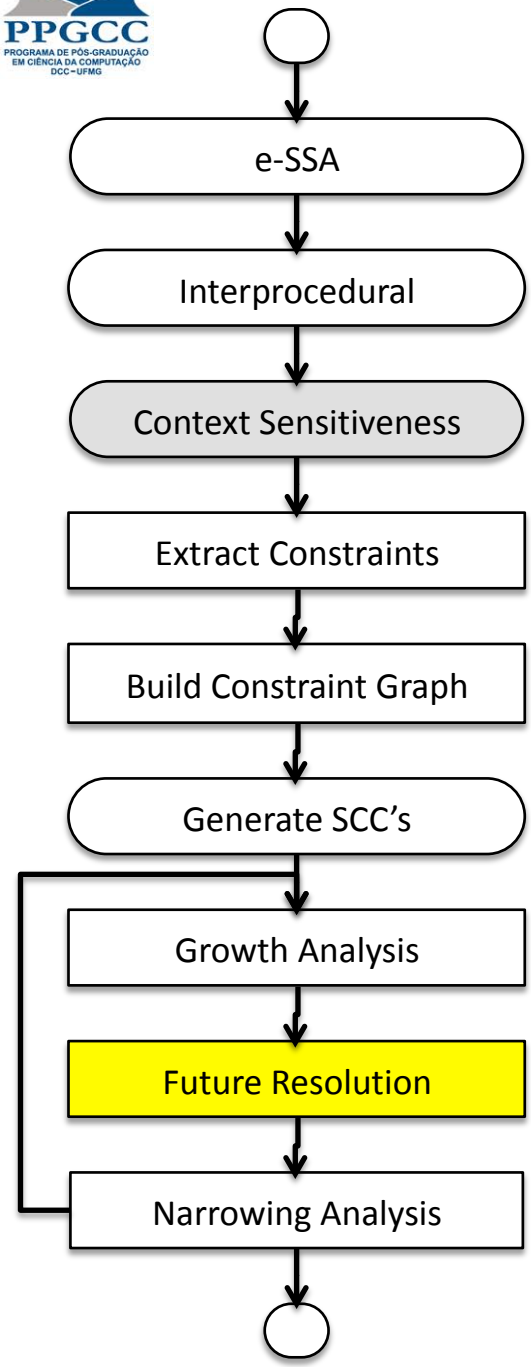


Future Resolution

How can we guarantee that we will have applied the growth analysis on a variable before solving the futures that depend on it?

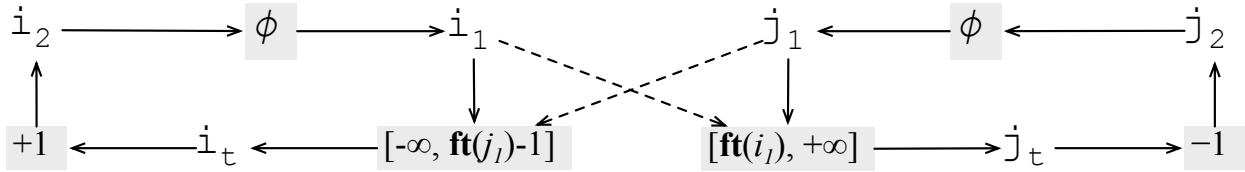


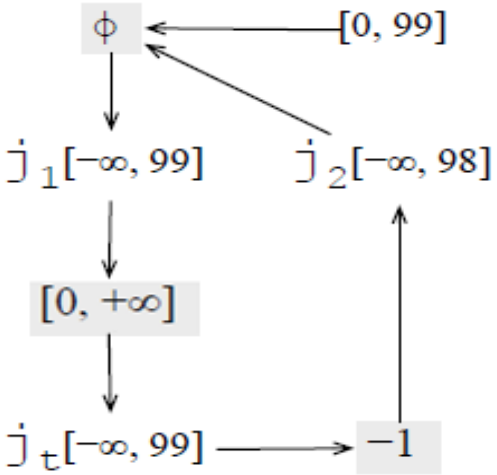




Future Resolution

The control dependency edges that we add to our constraint graph ensure the correct ordering when solving futures. If v is a future of u, either v's component will be solved before u's, or these variables are both in the same strong component.



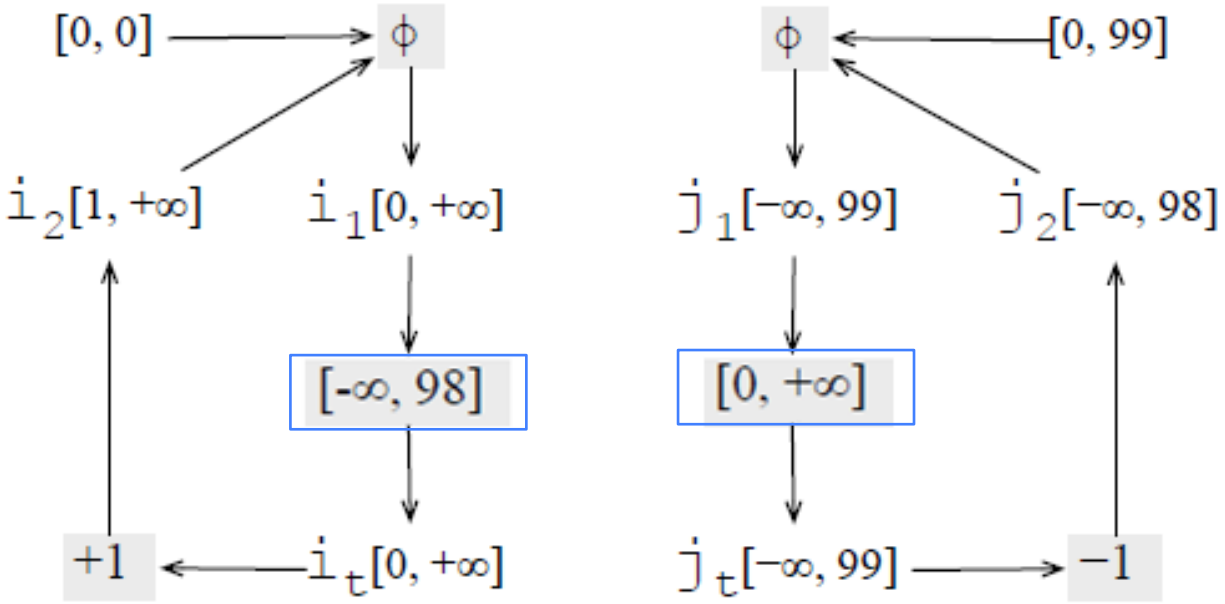


Can you come up with rules to replace futures by concrete bounds?

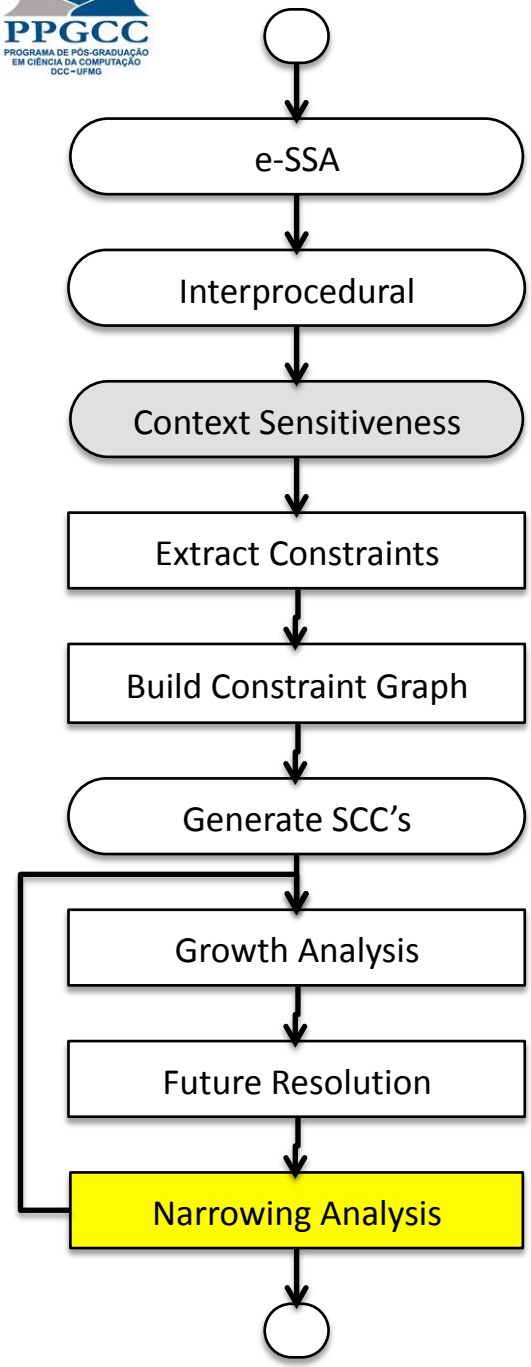
Future Resolution

$$\frac{Y = X \sqcap [l, \mathbf{ft}(V) + c] \qquad I[V]_{\uparrow} = u}{Y = X \sqcap [l, u + c]} \quad u, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

$$\frac{Y = X \sqcap [\mathbf{ft}(V) + c, u] \qquad I[V]_{\downarrow} = l}{Y = X \sqcap [l + c, u]} \quad l, c \in \mathbb{Z} \cup \{-\infty, +\infty\}$$

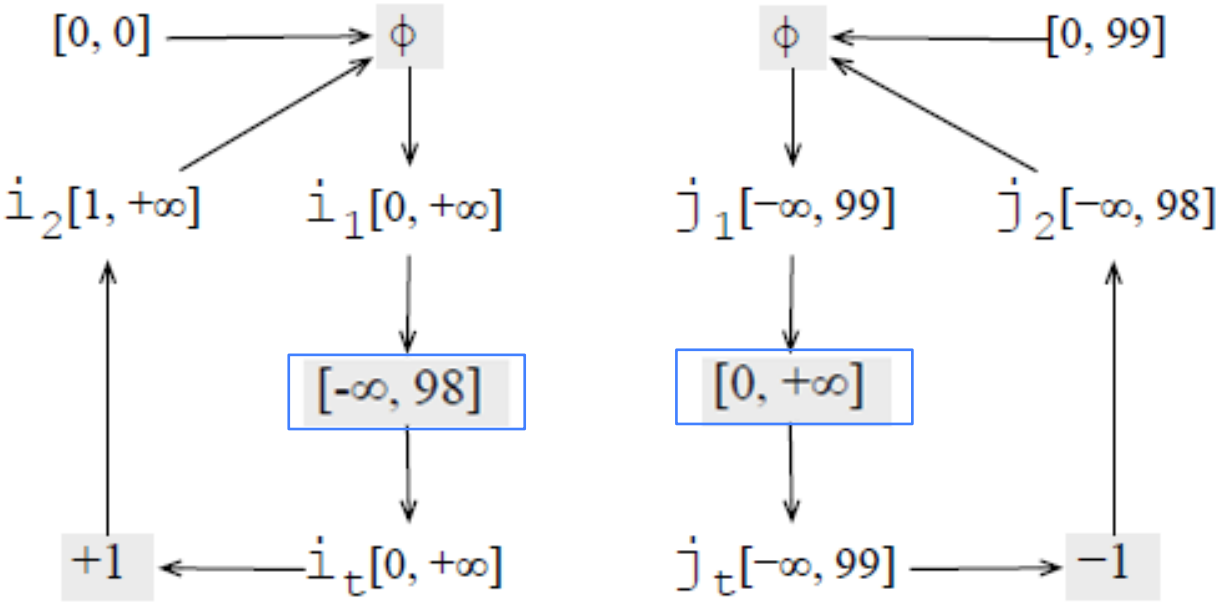


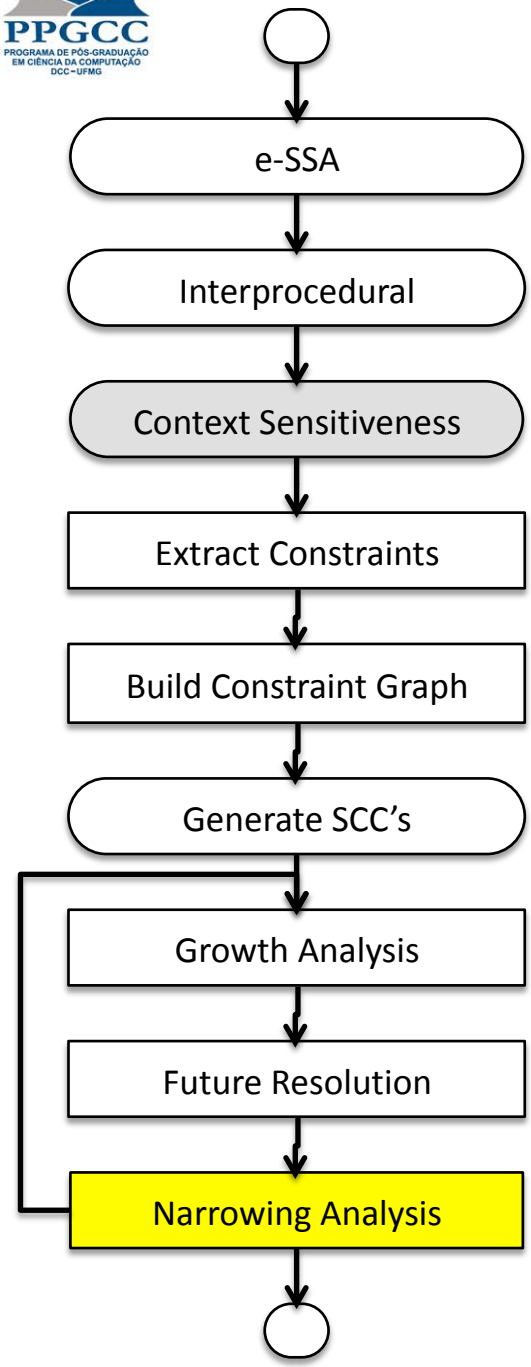
By the way, our results are still very imprecise. How could we improve our results for the program on the left?



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.

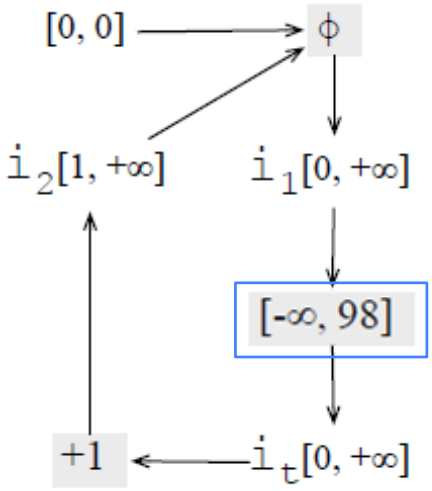


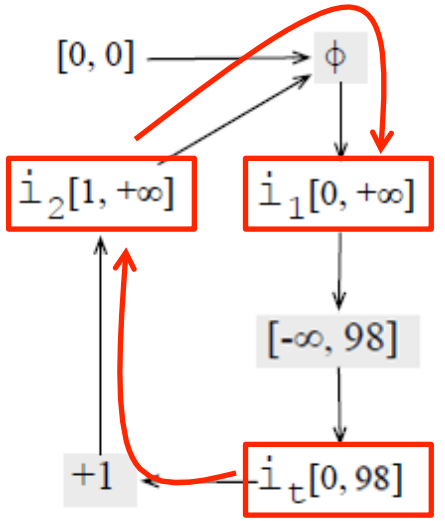


Narrowing Analysis

- 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.

Could you provide an abstract interpretation of the instructions that resulted in our narrowing analysis?





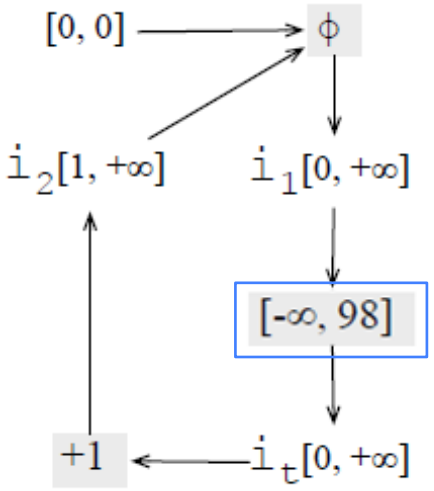
Narrowing Analysis

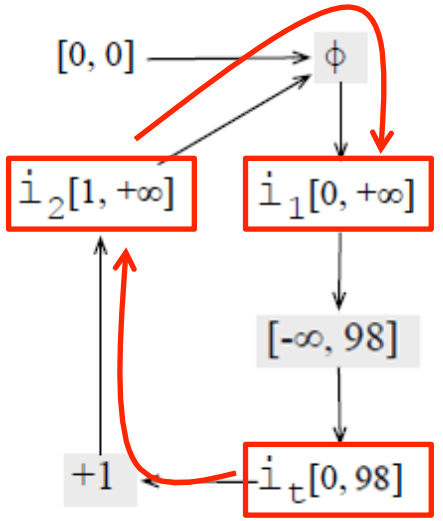
$$\frac{I[Y]_{\downarrow} = -\infty \qquad e(Y)_{\downarrow} > -\infty}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]} \qquad \frac{I[Y]_{\downarrow} > e(Y)_{\downarrow}}{I[Y] \leftarrow [e(Y)_{\downarrow}, I[Y]_{\uparrow}]}$$

$$\frac{I[Y]_{\uparrow} = +\infty \qquad e(Y)_{\uparrow} < +\infty}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]} \qquad \frac{I[Y]_{\uparrow} < e(Y)_{\uparrow}}{I[Y] \leftarrow [I[Y]_{\downarrow}, e(Y)_{\uparrow}]}$$

1) What is the role of the two rules on the left side?

2) What is the role of the two rules on the right side?





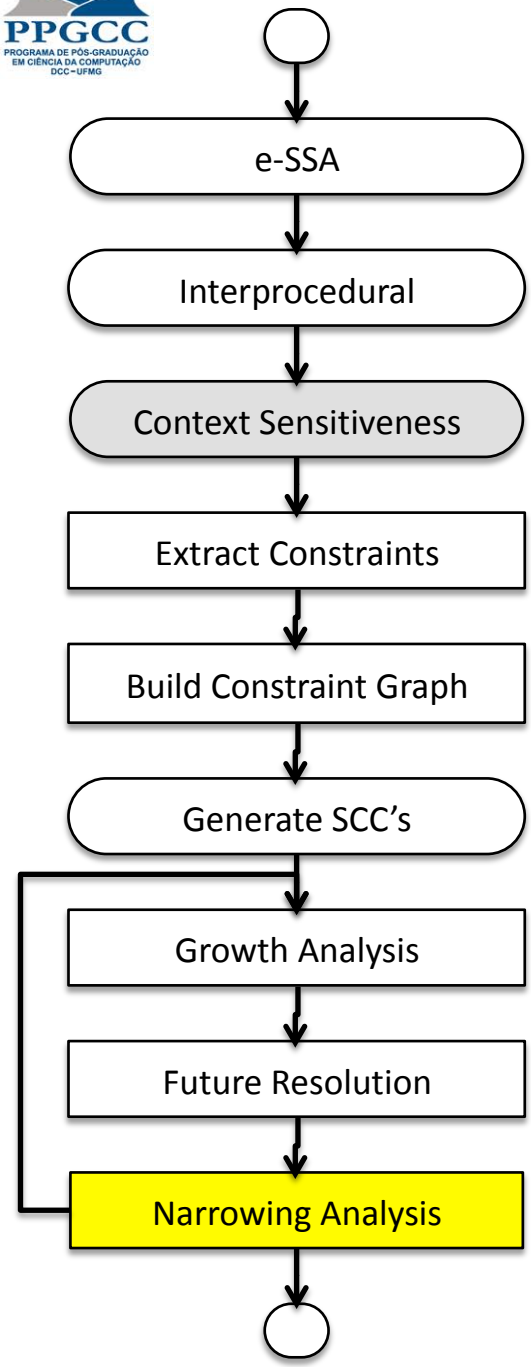
Narrowing Analysis

$$\frac{I[Y]_\downarrow = -\infty \qquad e(Y)_\downarrow > -\infty}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]} \qquad \frac{I[Y]_\downarrow > e(Y)_\downarrow}{I[Y] \leftarrow [e(Y)_\downarrow, I[Y]_\uparrow]}$$

$$\frac{I[Y]_\uparrow = +\infty \qquad e(Y)_\uparrow < +\infty}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]} \qquad \frac{I[Y]_\uparrow < e(Y)_\uparrow}{I[Y] \leftarrow [I[Y]_\downarrow, e(Y)_\uparrow]}$$

The two rules on the left do the actual narrowing. Notice that we are only recovering ranges that were bound to either minus or plus infinite by the growth analysis. The other two rules, on the right, are there to ensure that our analysis terminates.

What is the complexity of the narrowing analysis? How long does it take to converge?



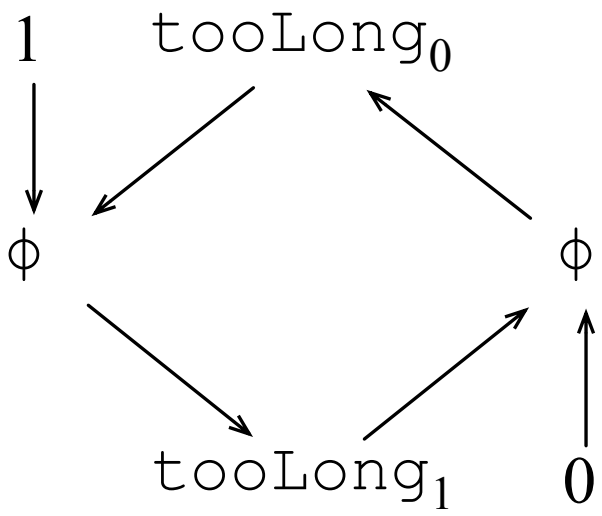
Final Solution

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

$$
\begin{aligned}
I[i_0] &= [0, 0] \\
I[i_1] &= [0, 99] \\
I[i_2] &= [1, 99] \\
I[i_t] &= [0, 98] \\
I[j_0] &= [0, 99] \\
I[j_1] &= [-1, 99] \\
I[j_2] &= [-1, 98] \\
I[j_t] &= [0, 99] \\
I[k_0] &= [0, 0] \\
I[k_1] &= [0, 100] \\
I[k_2] &= [1, 100] \\
I[k_t] &= [0, 99]
\end{aligned}
$$

Improving Precision

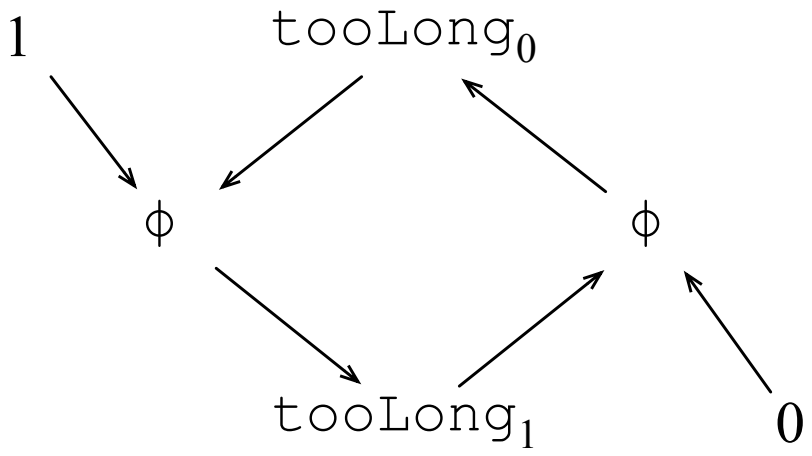
```
int main(int N) {
  int i = 0;
  while (1) {
    int tooLong = 0;
    while (i <= N) {
      if (i == N) {
        tooLong = 1;
      }
    }
    if (tooLong) {
      break;
    }
  }
  return i;
}
```

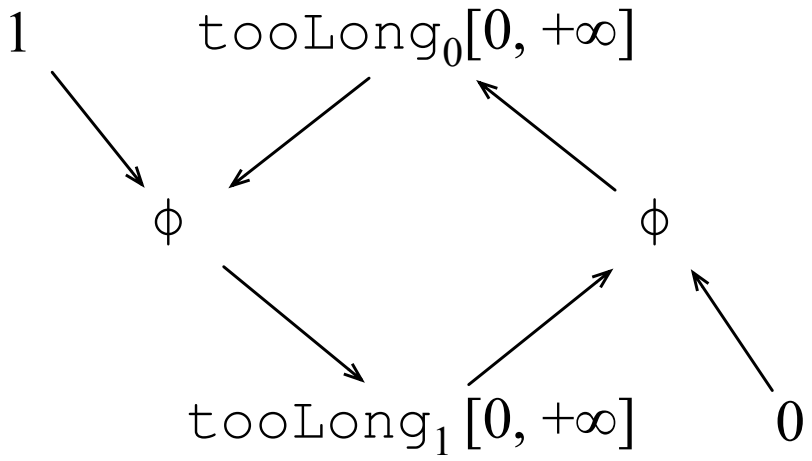


The code on the left was taken from SPEC CPU 2006. It illustrates an imprecision of our widening operator.

What would be the result of the range analysis on the constraint graph above?

Improving Precision





1) clearly, [tooLong$_0$], and [tooLong$_1$] are [0, 1], yet we are super imprecise. What is the source of this imprecision?

2) Would narrowing or future resolution be able to recover this imprecision?

3) How can we solve this imprecision?

Jump-Set Widening

- We could solve the previous example precisely, if we could iterate two rounds of abstract interpretation before applying widening.

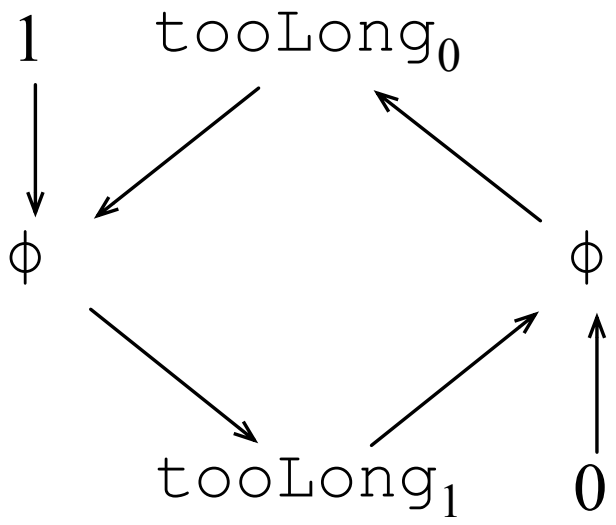
Why is that so?

- But the previous strategy does not work always.
- The problem of our widening operator is that it jumps into either $-\infty$ or $+\infty$ too fast.
- It would be good if we could consider some smaller limits before going to maximum imprecision.

But, which constants should we use?

Jump-Set Widening

- A common heuristics is to collect the constants in the source code of the program representation, and to use these constants as the gradual limits of widening.
- In our example, we would jump to {0, 1, +∞}



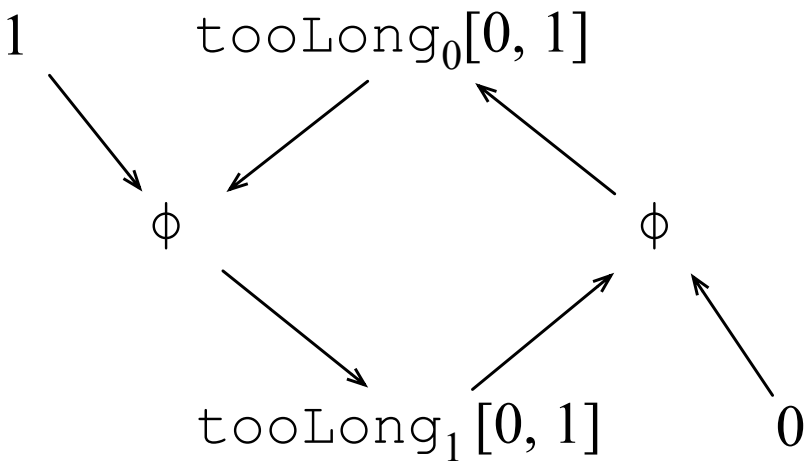
1) How would be the results of jump-set widening for this particular example?

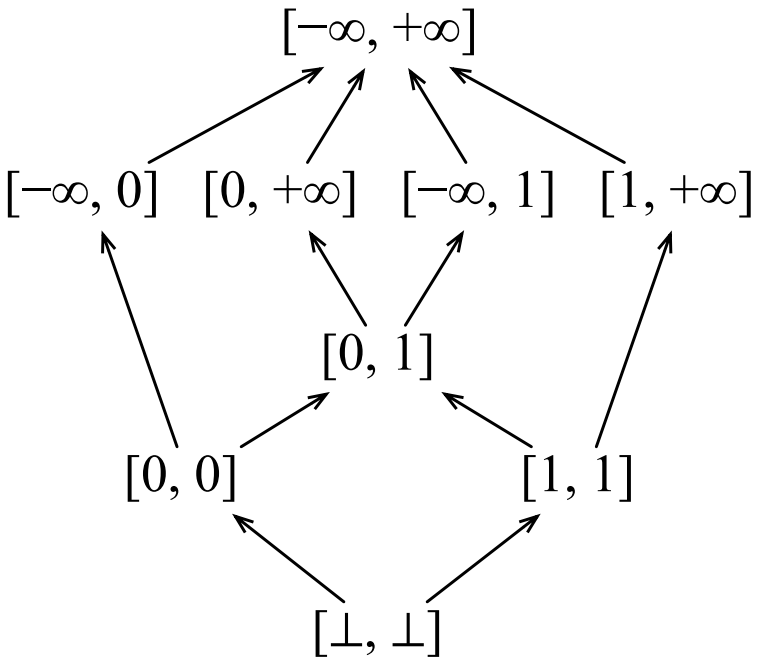
2) How is the lattice used in the growth analysis with jump-set widening?

3) Is this lattice always the same for all the instances of constraint graphs?

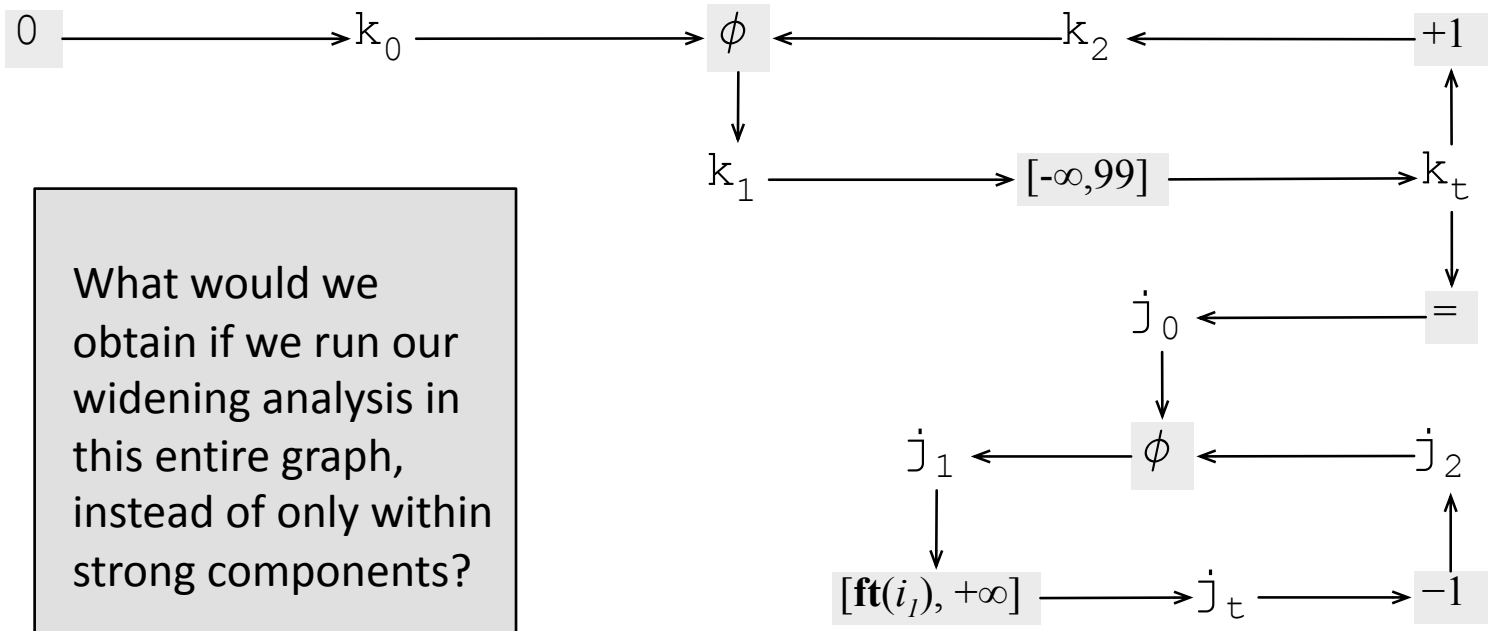
Jump-Set Widening

Because each different constraint graph will have different constants, each of them will be using a different lattice for the widening operator.



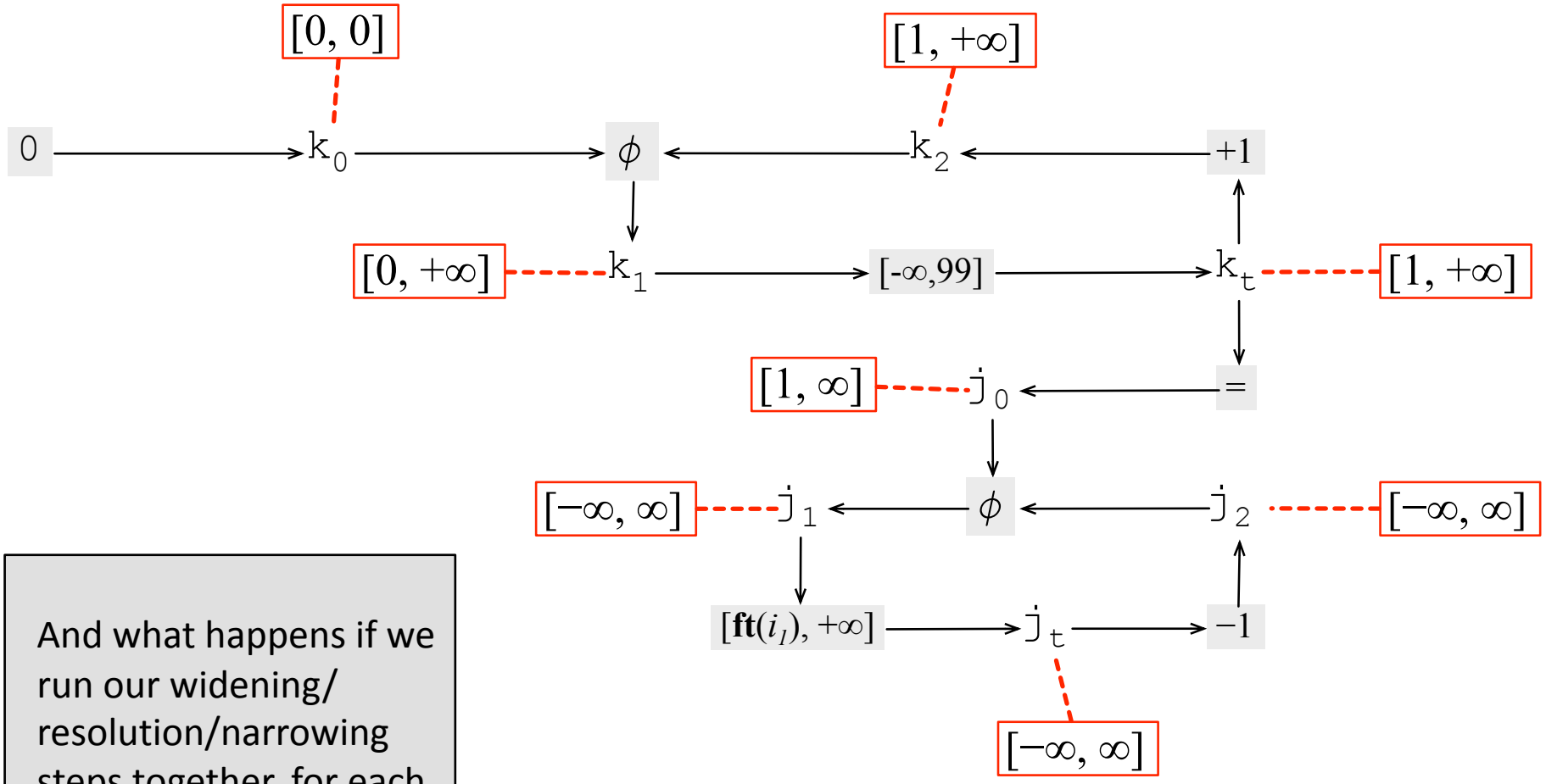


Strong Components and Precision



What would we obtain if we run our widening analysis in this entire graph, instead of only within strong components?

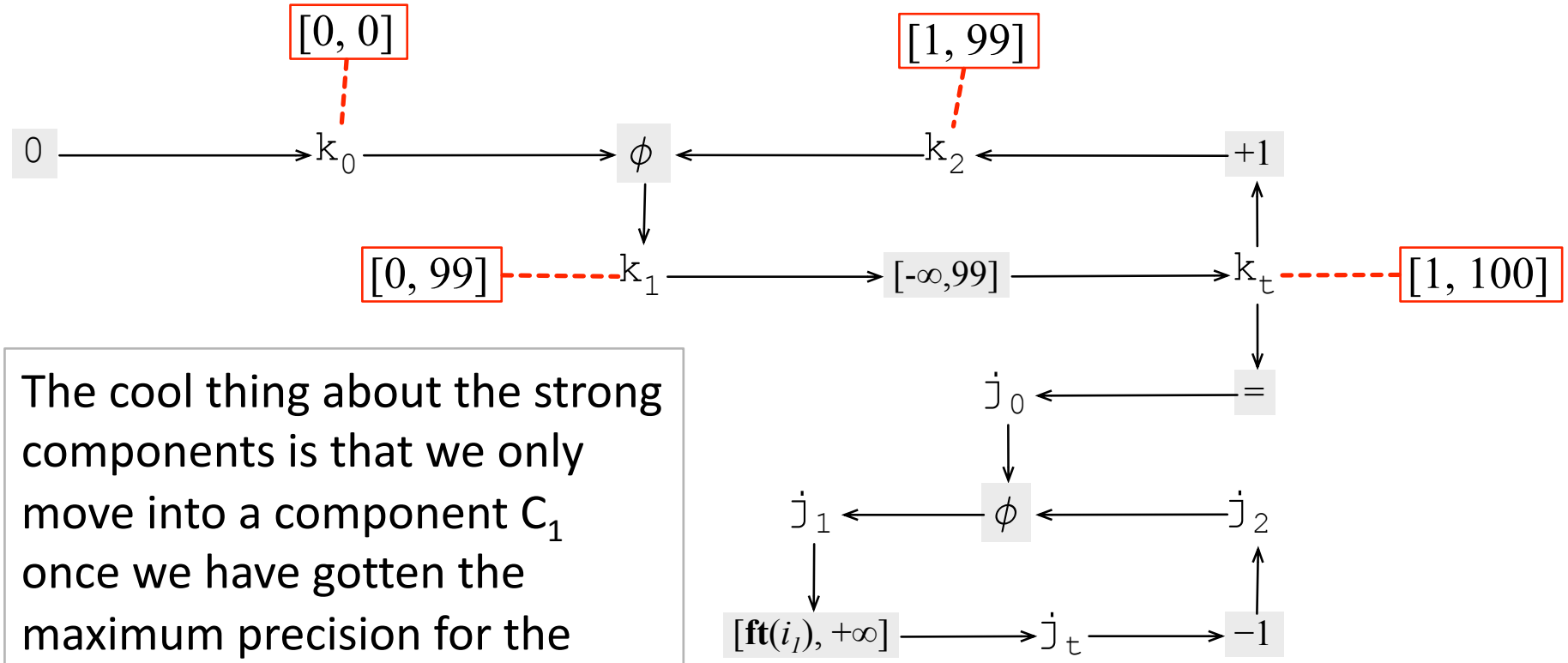
Widening of the Entire Graph



And what happens if we run our widening/ resolution/narrowing steps together, for each component, in topological order?

Argh: the bounds of j are very imprecise…

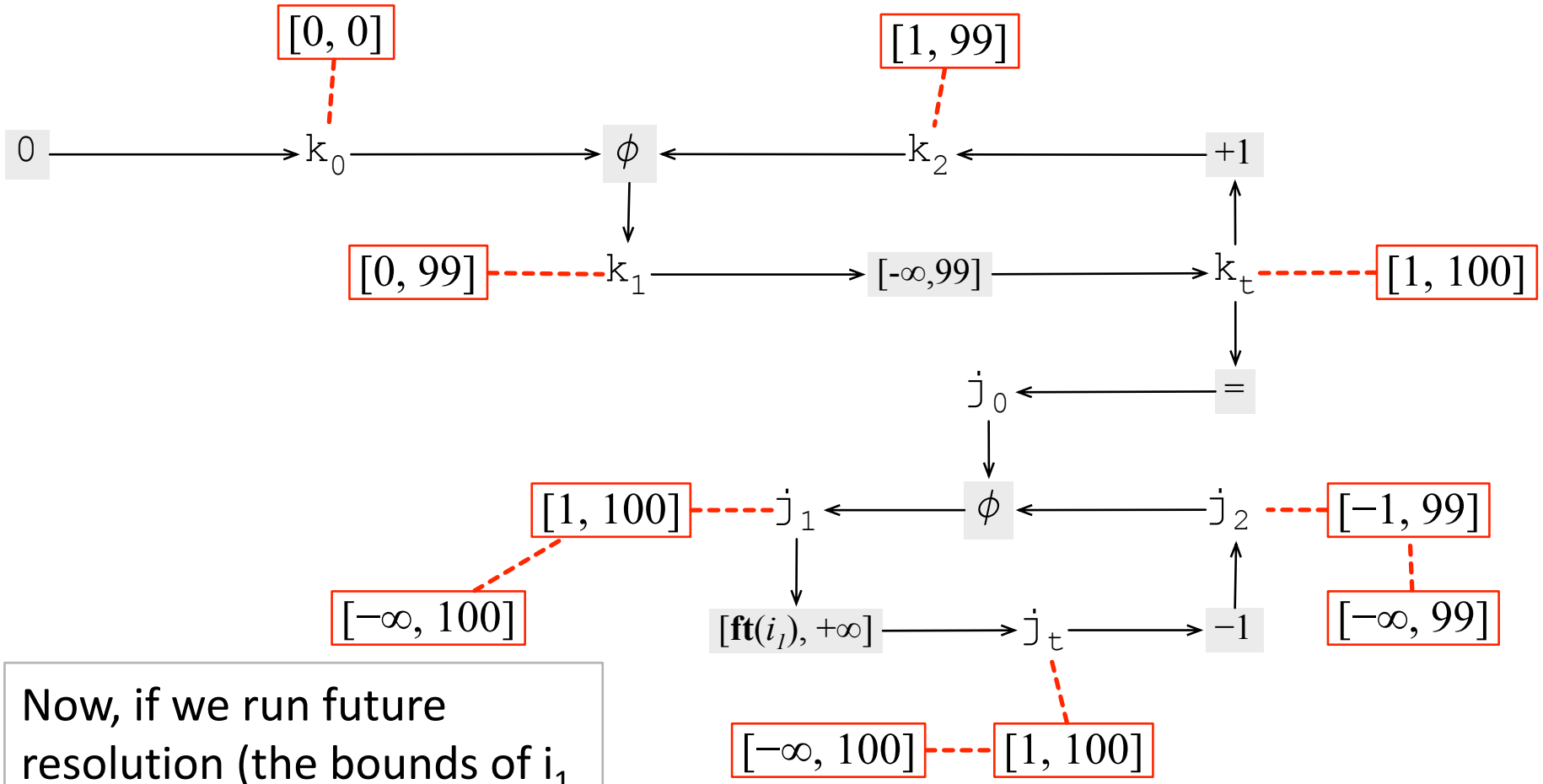
Strong Components and Precision



The cool thing about the strong components is that we only move into a component C_1 once we have gotten the maximum precision for the intervals of all the components that C_1 depends on. Like in this example, j_0 will only get a constant interval, instead of $+\infty$, from k_t.

And how would the abstract state of the j-variables be now, after widening?

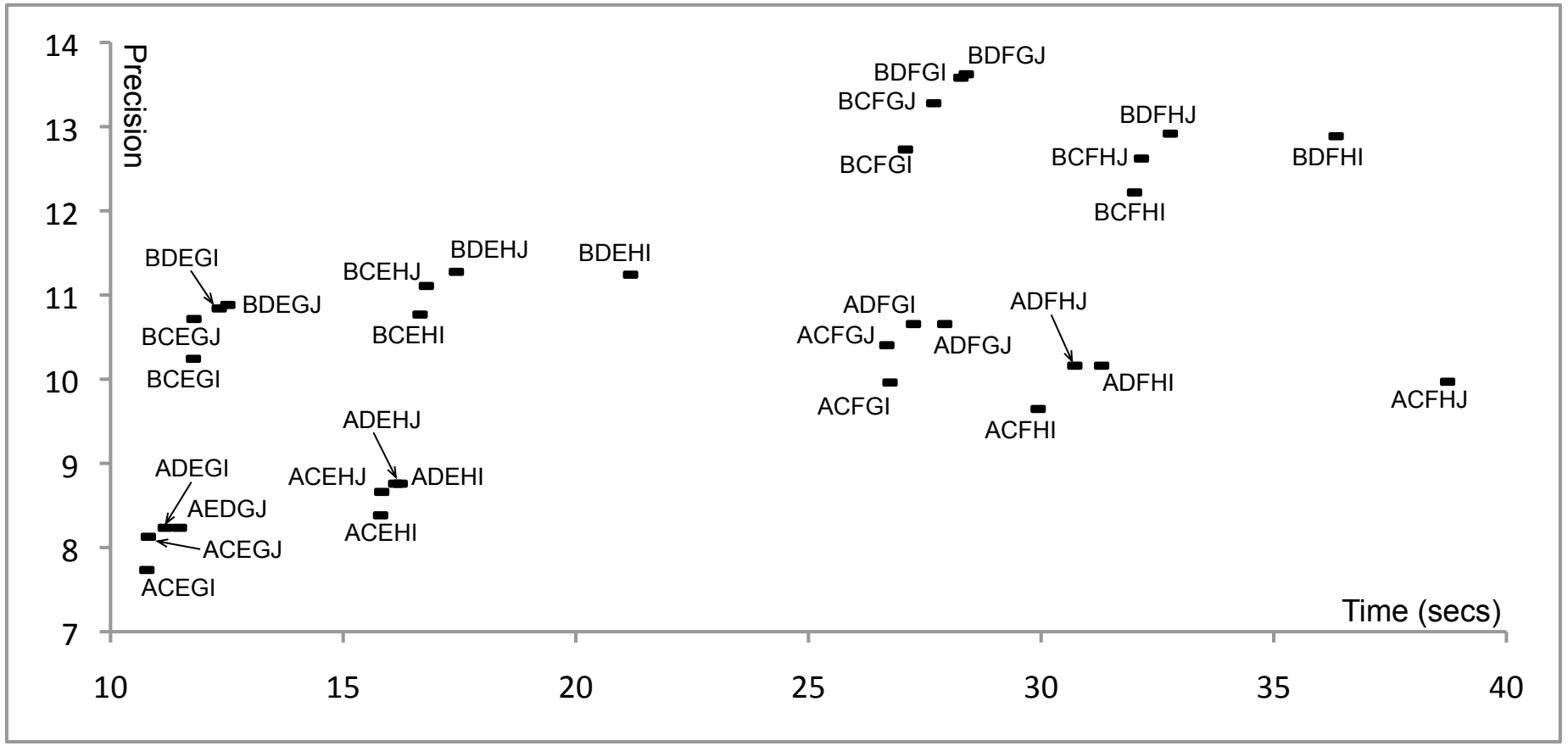
Strong Components and Precision



Now, if we run future resolution (the bounds of i_1 are not shown), we would have a very precise result for the j variables.

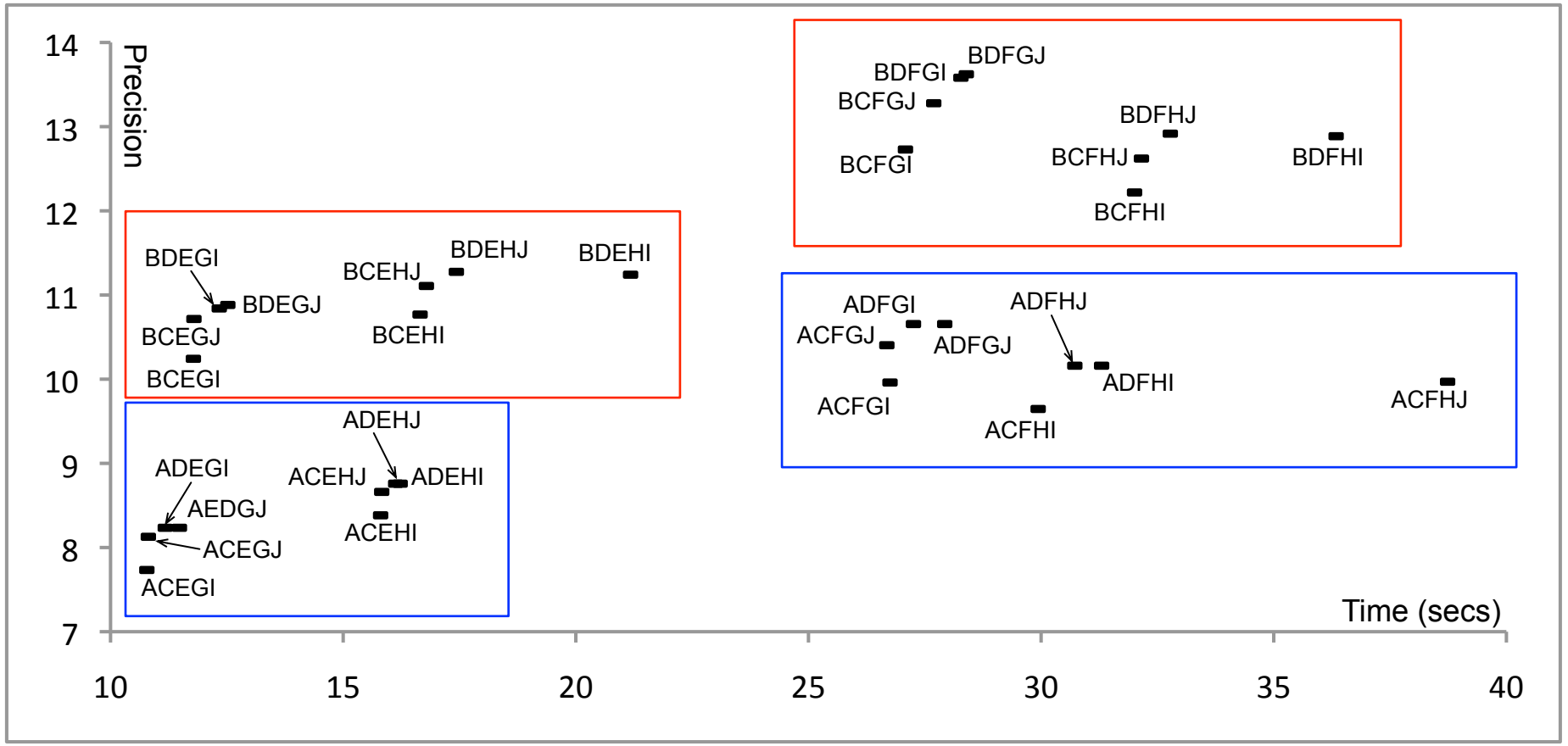
A Vast Space to Explore

Which are the good, the bad and the ugly points in this chart?



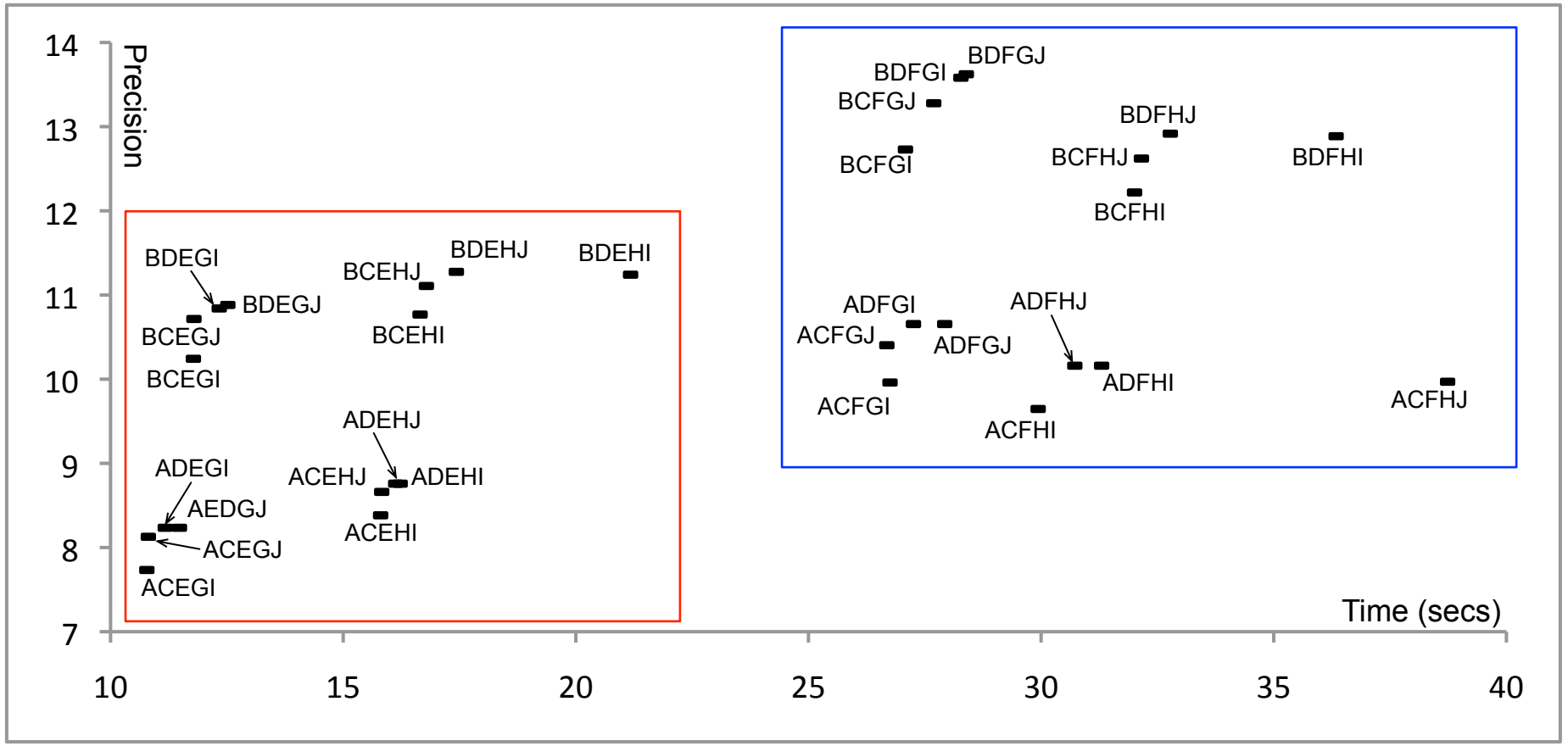
A: SSA	C: 0 iterations	E: intra	G: no inlining	I: simple widening
B: e-SSA	D: 16 iterations	F: inter	H: inlining	J: jump-set widening

The Intermediate Representation



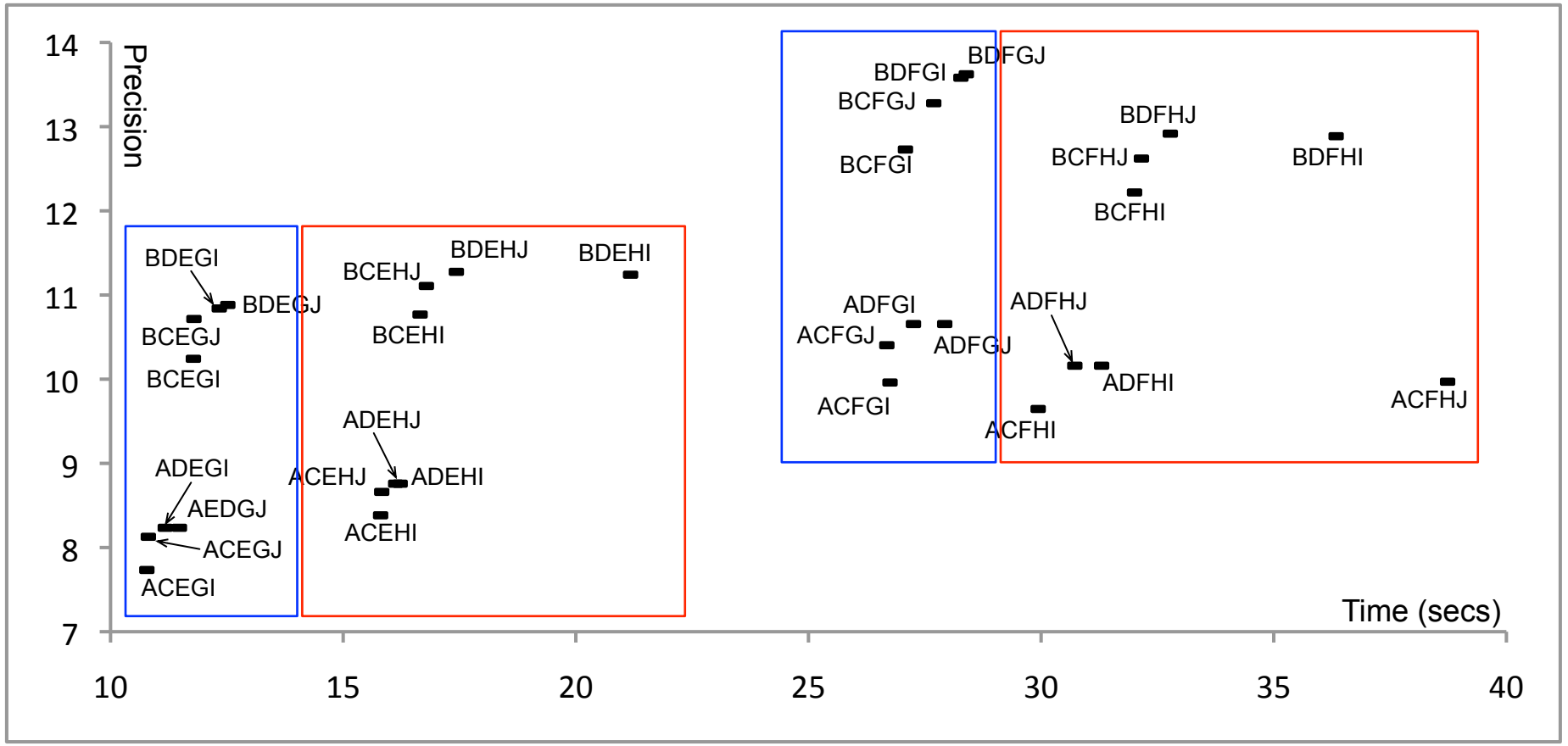
A: SSA	C: 0 iterations	E: intra	G: no inlining	I: simple widening
B: e-SSA	D: 16 iterations	F: inter	H: inlining	J: jump-set widening

Intra vs Inter Procedural Analysis



A: SSA	C: 0 iterations	E: intra	G: no inlining	I: simple widening
B: e-SSA	D: 16 iterations	F: inter	H: inlining	J: jump-set widening

The Impact of Inlining



A: SSA	C: 0 iterations	E: intra	G: no inlining	I: simple widening
B: e-SSA	D: 16 iterations	F: inter	H: inlining	J: jump-set widening

Limits of Range Analysis on the Interval Lattice

```
k = 0
s = 0
while k < 100:
   k = k + 1
   s = s + 1
print k
print s
```

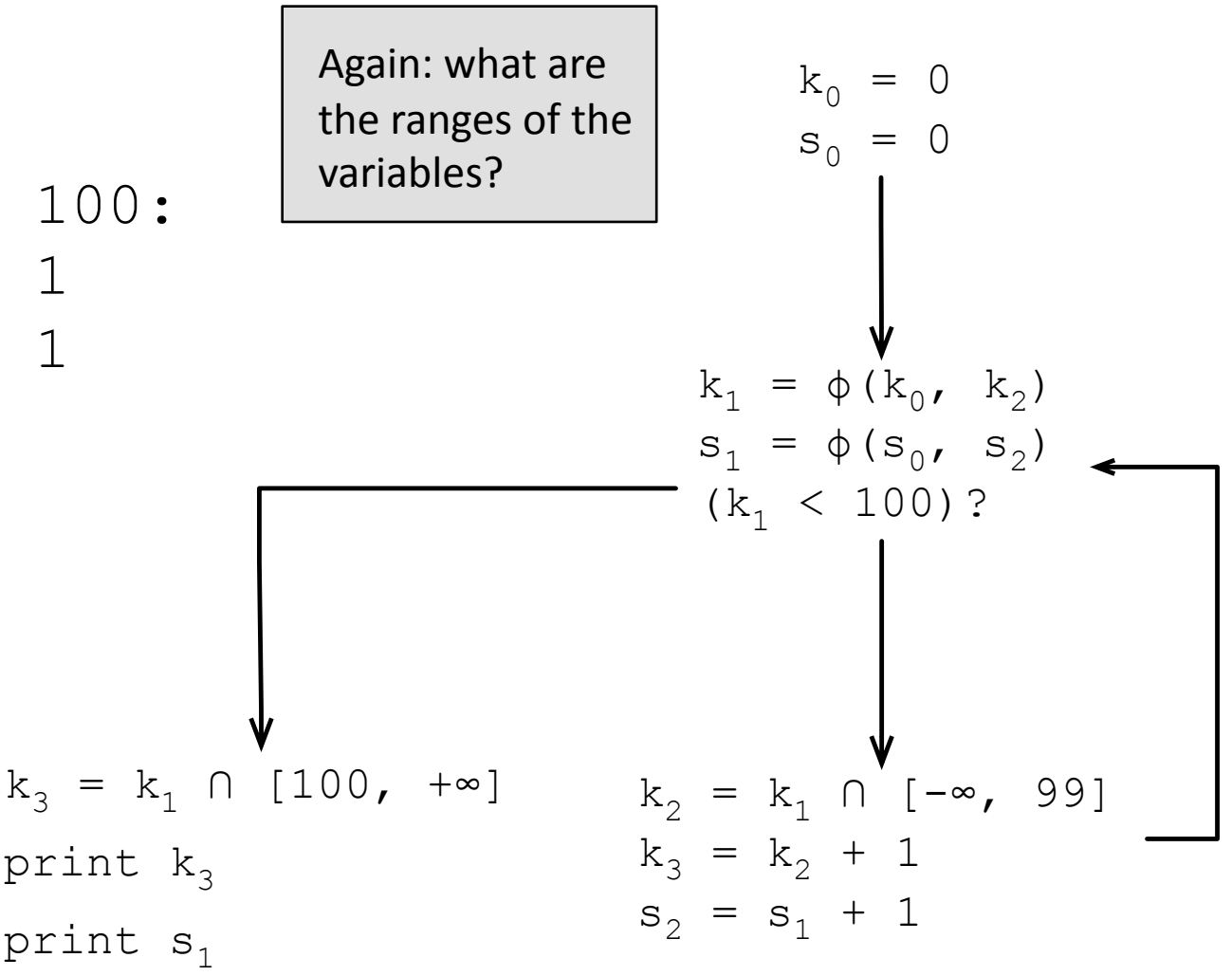
1) Considering the program on the left. What is the range of k?

2) What would be the range of s, using our algorithm?

Limits of Range Analysis on the Interval Lattice

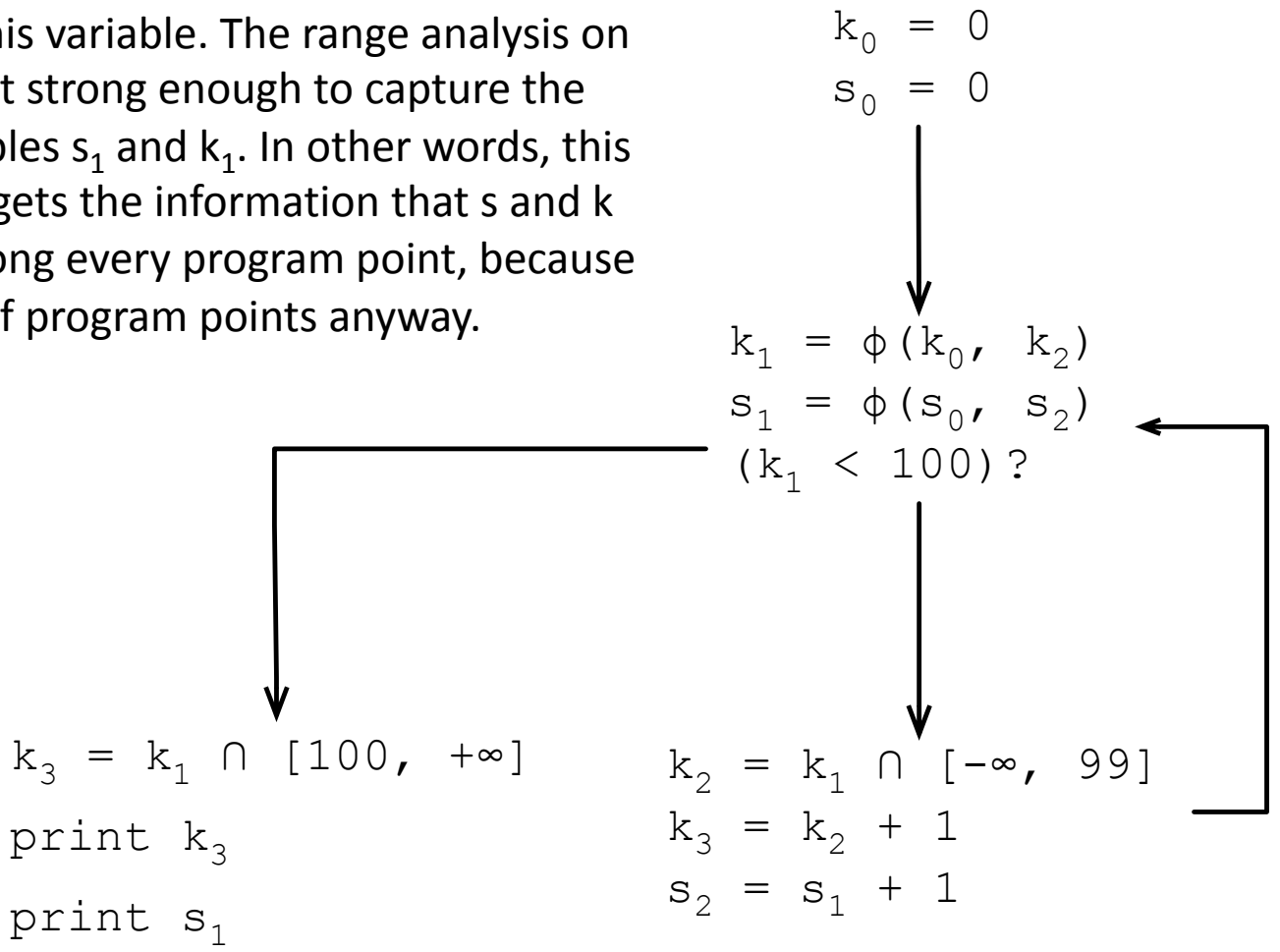
```
k = 0
s = 0
while k < 100:
  k = k + 1
  s = s + 1
print k
print s
```

Again: what are the ranges of the variables?



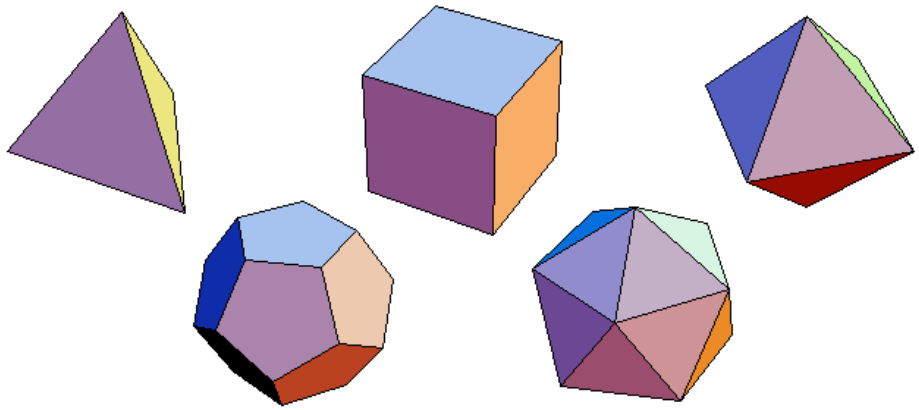
Limits of Range Analysis on the Interval Lattice

We end up bounding s to [0, +∞], because there are no constraints bounding this variable. The range analysis on the interval lattice is not strong enough to capture the relation between variables s_1 and k_1. In other words, this analysis is sparse. It forgets the information that s and k are varying together along every program point, because it does not keep track of program points anyway.



Relational Analyses

- Analyses that can capture the relation between s and k, in the previous example, are called relational analyses.
- They are more precise than the range analysis on the interval lattice.
- But they are more expensive also.
  - Polyhedrons♤
  - Octagons♧



- Usually these analyses are very geometrical.
- They also use the ideas of widening and narrowing.

♤: Automatic Discovery of Linear Restraints Among Variables of a Program

♧: The octagon abstract domain, 2006

A Bit of History

- The ideas of widening and narrowing were introduced by Cousot and Cousot, together with all the framework of abstract interpretation.
- The algorithm that we have presented was invented by Campos *et al*., and subsequently improved by Rodrigues *et al*. An initial version of it appeared in a paper by Douglas and Pereira.

- Cousot, P. and Cousot, R. "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints", POPL, p 238-252 (1977)
- Couto, D. and Pereira, F. "The Design and Implementation of a Non-Iterative Range Analysis Algorithm on a Production Compiler", SBLP, p 45-59 (2011)
- Campos, V., Rodrigues, R., and Pereira, F. "Speed and Precision in Range Analysis", SBLP, p 42-56 (2012)
- Rodrigues, R., Campos, V. and Pereira, F. "A Fast and Low Overhead Technique to Secure Programs Against Integer Overflows", CGO, (2013)