Handling pointer arithmetic.

Original C code

```
char s [10];
char *p;
p = s + 7;
p[5] = 'a';
```

Handling pointer arithmetic.

Instrumented C code

Original C code

```
char s [10];
char *p;
p = s + 7;
p [5]  = ’a’;
```

```
char s [10];  int sAlloc = 10;
char *p;      int pAlloc = 0;
              assert (7 <= sAlloc);
p = s + 7;    pAlloc = sAlloc - 7;
              assert (5 < pAlloc);
p [5]  = ’a’;
```

The second assert condition does not hold, as desired.

Complex control flow constructs are automatically handled.

```
char s [10];
int i;
for (i=0; i<=15; i++) {
  s[i] = ’a’;
}
```

```
char s [10];  int sAlloc = 10;
int i;
for (i=0; i <=15; i++) {
          assert (i < sAlloc);
  s[i] = ’a’;
}
```

The asserted condition will be violated at some point during the execution of the program, as desired.

String manipulation functions like strcpy, strlen, strcat should be treated directly, without analyzing their code.

```
char s [10];
char t [10];
strcpy (s,t);
```

This code is vulnerable.

Cannot be detected from information about sAlloc and tAlloc.

Need further variables:

sIsNull	s is a null terminated string (boolean)
sLen	length of s

Instrumented code

```
char s [10];     int sAlloc=10, sIsNull=false, sLen;
char t [10];     int tAlloc=10, tIsNull=false, tLen;
                 assert (tIsNull && tLen < sAlloc)
strcpy (s,t);
                 sIsNull=true; sLen=tLen;
```

The asserted condition is violated, as desired.

```
char *p;              int pAlloc=0, pIsNull=false, pLen;
char s [20];          int sAlloc=20, sIsNull=false, sLen;
p=”Hello World!”; pAlloc=13; pIsNull=true; pLen=12;
                      assert(pIsNull && pLen < sAlloc)
strcpy(s,p);
                      sIsNull=true; sLen=pLen;
```

The asserted condition holds, as desired.

Dealing with string overlaps.

```
char *p, *q, s[20], t[20];      ... instrumentation code ...
p="Hello World!";               ...
q=s+6;                          ...
                    /* here qIsNull == sIsNull == false */
strcpy(s,p);                    sIsNull=true; sLen=pLen;
                  /* here sIsNull == true, qIsNull == false */
                                assert (qIsNull && qLen < tAlloc)
strcpy(t,q);                    ...
```

The asserted condition for second strcpy fails. ⇒ Bad analysis.

After the first strcpy, the variables qIsNull and qLen are not updated.

Dealing with string overlaps.

```
char *p, *q, s[20], t[20];        ...  instrumentation code ...
p=”Hello World!”;                 ...
q=s+6;                            ...
                      /* here qIsNull == sIsNull == false */
strcpy(s,p);                     sIsNull=true; sLen=pLen;
                    /* here sIsNull == true, qIsNull == false */
                                 assert (qIsNull && qLen < tAlloc)
strcpy(t,q);                      ...
```

The asserted condition for second strcpy fails. ⇒ Bad analysis.

After the first strcpy, the variables qIsNull and qLen are not updated.

⟹ need further variables for keeping track of overlaps between strings.

Putting together

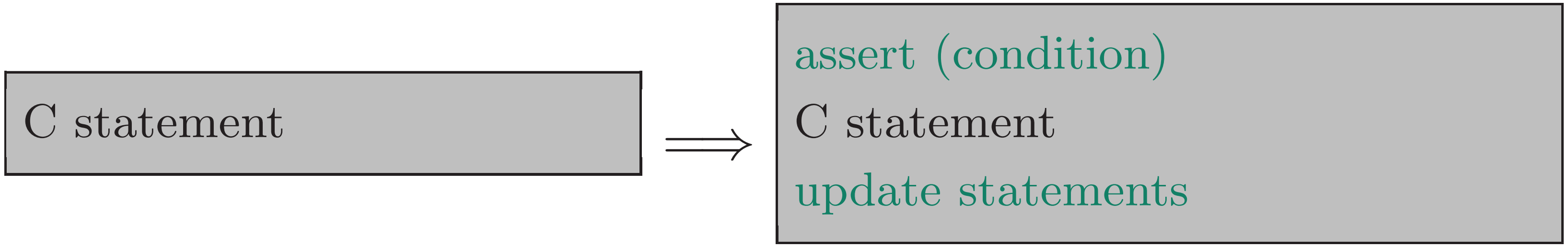
The required list of variables:

sAlloc	space allocated for string ccodes
sIsNull	whether string s is null terminated
sLen	length of string s
s_overlaps_t	whether strings s and t point inside the same allocated buffer
s_diff_t	amount of overlap between strings s and t

s_overlaps_t is same as t_overlaps_s.

s_diff_t = - t_diff_s.

Schema for instrumenting the C code.



C statement

$\Longrightarrow$

assert (condition)
C statement
update statements

Clean program: all the string operations have a well defined output (according to standard specifications.)

The instrumentation preserves the bahaviour of clean C programs.

In a program is unclean, the condition for the corresponding statement is violated at some time during execution.

Allocation

C statement	condition	update

char s [20];	true	sAlloc = 20;
		sIsNull = false;
		FOREACH a
		a_overlaps_s = false

No safety conditions required.

The string is not null-terminated and has no overlap with any other string.

Allocation

p = malloc(exp)	true

```
if (p)
   pAlloc = exp;
   else pAlloc = 0;
pIsNull = false;
FOREACH a
   a_overlaps_p = false;
```

If allocation fails then no space is allocated for the string.

Constant string assignment

s = ”some string”;

true

```
sAlloc = 12;
sIsNull = true;
sLen = 11;
FOREACH a
   s_overlaps_a = false;
```

No assertion conditions.

The string is null terminated and has no overlap with other strings.

Safe even with other pointers to the same string constant, as no updates are allowed in memory-region where constant strings are stored.

Pointer arithmetic For simplicity consider only exp ≥ 0

C statement

```
p = q + exp;
```

condition

```
exp <= qAlloc
```

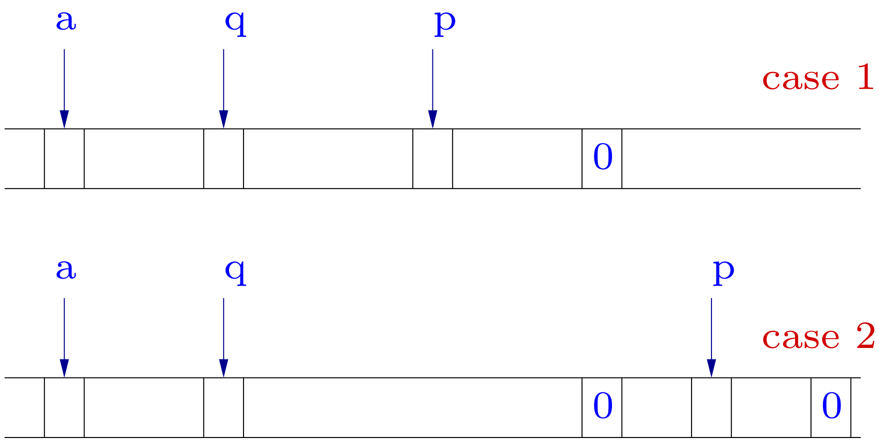
update

```
pAlloc = qAlloc - exp;

p_overlaps_q = true; p_diff_q = exp;

FOREACH a
  p_overlaps_a = q_overlaps_a;
  p_diff_a = q_diff_a + exp;
...
```

```
...
if (qIsNull && qLen >= exp) {
  pIsNull = true; pLen = qLen - exp;
} else RECOMPUTE (p);
```



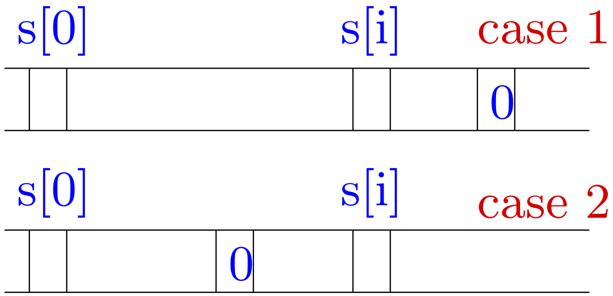
```
#define RECOMPUTE (s)
  sLen = strlen(s);
  sIsNull = (sLen < sAlloc ? true : false)
/* however strlen cannot be analyzed precisely! */
```

String update We consider only $i \geq 0$

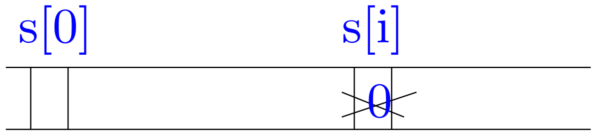
C statement	condition
`s[i] = exp;`	i < sAlloc

Update

```
if (exp == 0) {
  if (!sIsNull || sLen > i) {
    sIsNull = true;
    sLen = i;
  }
  FOREACH a
    DESTRUCTIVE_UPDATE (a,s)
}
```



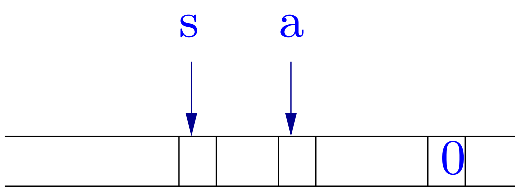
```
else {
    if (sIsNull && i == sLen)
        RECOMPUTE (s);
    FOREACH a
        DESTRUCTIVE_UPDATE (a,s);
}
```

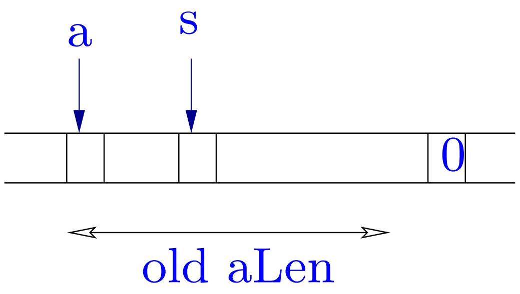


DESTRUCTIVE_UPDATE

The string `s` has been modified and variables `sIsNull` and `sLen` have been updated. The corresponding variables for overlapping strings need to be updated.

```
#define DESTRUCTIVE_UPDATE (a,s)
  if (a_overlaps_s)
    if (sIsNull && a_diff_s <= sLen &&
        (!aIsNull || a_diff_s >= -aLen)) {
      aIsNull = true;
      aLen = sLen - a_diff_s;
    } else RECOMPUTE (a);
```





Library functions: strcpy

C statement

```
strcpy (s,t);
```

condition

```
tIsNull & tLen < sAlloc
```

update

```
sIsNull = true;
sLen = tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

The copied string should be null terminated and the destination should have enough space.

Library functions: strcat

C statement

```
strcat (s,t);
```

condition

```
sIsNull && tIsNull
   && tLen + sLen < sAlloc
```

update

```
sLen = sLen + tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

Both the source and destination strings should be null terminated before concatenation.

Library functions: strcat

C statement

```
strcat (s,t);
```

condition

```
sIsNull && tIsNull
    && tLen + sLen < sAlloc
```

update

```
sLen = sLen + tLen;
FOREACH a
  DESTRUCTIVE_UPDATE (a,s);
```

Both the source and destination strings should be null terminated before concatenation.

Normal functions: not yet considered.

Given a C program, we have shown how to compute an instrumented C program which preserves the semantics.

If the original C program is clean then the instrumented C program has the same behaviour and all assertions always hold.

If the original C program has an unclean expression then the corresponding assertion will be false at some time.

Next, we use integer analysis algorithms to check whether any of the assertions are violated.

A program state at a certain point of time during the program execution tells us the value of each program variable at that time.

Execution of an instruction leads to a modification in the program state.

Each program point can be reached several times during execution (loops).

Hence several program states are possible at each program point.

Goal: for each program point, compute an upper approximation of the set of possible program states.

Upper approximation of the set of possible states is a safe approximation.

Scenario 1:

```
char s [20];
for (i=0; i<10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(9,18)$$

Suppose our analysis tells us that at this program point:

$0 \leq i \leq 9 \wedge 0 \leq j \leq 18$ upper approximation

We conclude that the program is clean safe

Upper approximation of the set of possible states is a safe approximation.

Scenario 2:

```
char s [20];
for (i=0; i<10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(9,18)$$

Suppose our analysis tells us that at this program point:

$0 \leq i < \infty \wedge 0 \leq j < \infty$ upper approximation

We conclude that the program is not clean safe

Upper approximation of the set of possible states is a safe approximation.

Scenario 3:

```
char s [20];
for (i=0; i<=10; i++) {
  j = 2 * i;
  /* j is hopefully < 20 */
  s[j] = 'a';
}
```

The possible values of (i, j) before the string update operation are

$$(0,0),(1,2),(2,4)...(10,20)$$

We compute upper approximation of the set of possible states.

Hence our analysis should always tell us that j can become 20.

We conclude that the program is not clean safe

We transform the instrumented program to a program with only integer variables $\Longrightarrow$ further safe approximation.

e1 is non-integer variable:

$$\text{e1} = \text{e2}; \Longrightarrow ;$$

e contains non-integer variables and constants:

$$\text{x} = \text{e}; \Longrightarrow \text{x} = ?;$$

$$\text{if (e) s1 else s2} \Longrightarrow \text{if (?) s1 else s2}$$

Similarly for loops

The expression ? can take all possible values non-deterministically.
(In practice, use a special uninitialized variable in its place.)

Safe approximation: all executions of the original program are still allowed after approximation.

Instrumented program

```
char s [20];    int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;   assert (sAlloc > j)
  s[j] = ’a’; if (97 == 0)  ...
}
```

Instrumented program

```
char s [20];     int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;     assert (sAlloc > j)
  s[j] = 'a';    if (97 == 0)  ...
}
```

Corresponding integer program

```
                 int sAlloc=20, sIsNull=false, sLen;
for (i=0; i<=10; i++) {
  j = 2 * i;     assert (sAlloc > j)
                 if (97 == 0)  ...
}
```

This may involve some safe approximation

Instrumeted program:

```
char s[10], *t;     ...
t = ”Hello!”;       tAlloc = 7; tIsNull = 0; tLen=6; ...
strcpy (s,t);       ...sLen=tLen
 if (s[0]==72) i = 5; else i = 6;
s[i] = 0;           if (0==0) { if (!sIsNull || sLen > i) {
                        sIsNull=true; sLen=i;}...
```

This may involve some safe approximation

Instrumeted program:

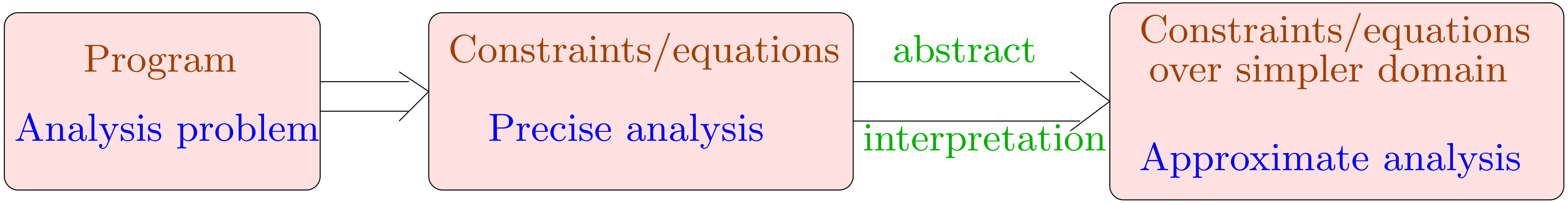
```
char s[10], *t;  ...
t = ”Hello!”;    tAlloc = 7; tIsNull = 0; tLen=6; ...
strcpy (s,t);    ...sLen=tLen
 if (s[0]==72) i = 5; else i = 6;
s[i] = 0;        if (0==0) { if (!sIsNull || sLen > i) {
                     sIsNull=true; sLen=i;}...
```

Integer program:

```
                 ... int any;
                 tAlloc = 7; tIsNull = 0; tLen=6; ...
                 ...sLen=tLen
 if (any       ) i = 5; else i = 6;
                 if (0==0) { if (!sIsNull || sLen > i) {
                     sIsNull=true; sLen=i;}...
```

Program analysis for integers relations

Our methodology:



Program

Analysis problem

Constraints/equations

Precise analysis

abstract

interpretation

Constraints/equations over simpler domain

Approximate analysis

Precise analysis:	e.g.: what values are taken by variable x at a certain program point?	infinite domain: $\mathbb{Z}$
Approximate analysis:	e.g.: does variable x ever take a negative value at a certain program point?	finite domain: $\{+, -, 0\}$

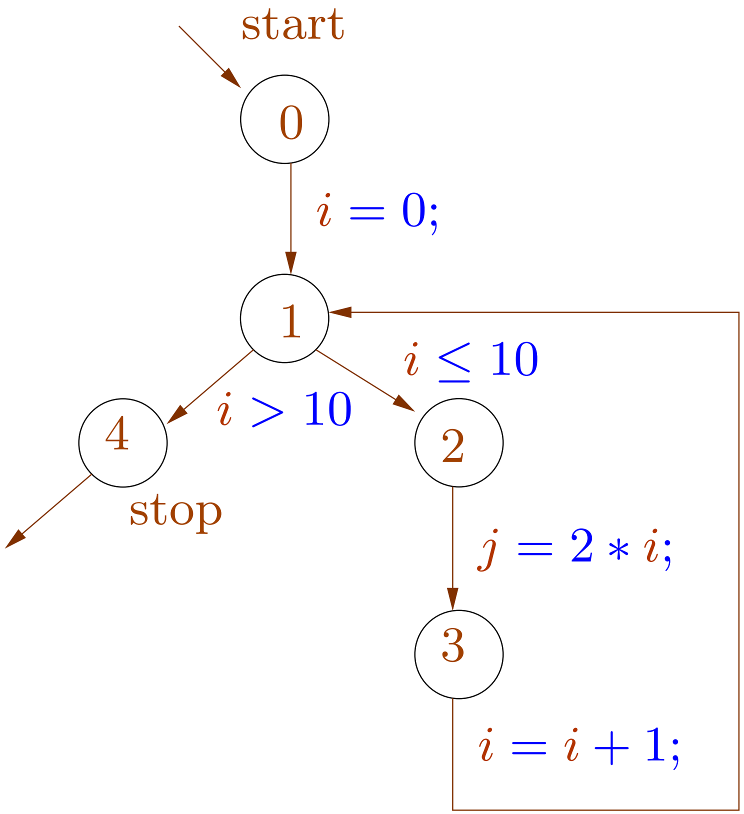
We consider a set Vars of variables ranging over integers.

Program consists of statements of the form

NOP	;
Assignments	x = e;
Conditions	if (e) s1 else s2
Jumps	goto L

While and for loops: translated using conditions and goto statements.

We represent programs using control flow graphs (CFGs).



Distinguished *start* and *stop* nodes.

Edges k are of the form (u, l, v) where u and v are nodes and label l is an assignment or a condition.

The set of possible states state of the program is

$$\mathcal{S} = \mathsf{Vars} \to \mathbb{Z}$$

The evaluation of an arithmetic expression e under state $\rho \in \mathcal{S}$ is denoted

$$[\![e]\!]\ \rho : \mathbb{Z}$$

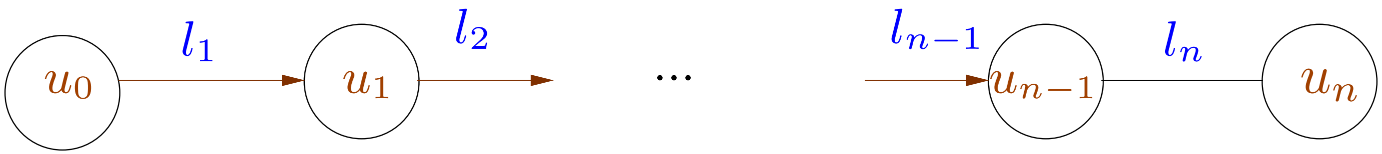
An edge $k = (u, l, v)$ induces a partial transformation on program states. The transformation depends only on the label l.

$$[\![k]\!]\ \rho = [\![l]\!]\ \rho$$

$$\text{where } [\![l]\!]\ : \mathcal{S} \to \mathcal{S}$$

$$\begin{array}{lll}
[\![;]\!]\ \rho & = \rho; & \\
[\![x = e;]\!]\ \rho & = \rho \oplus \{x \mapsto [\![e]\!]\ \rho\} & //\text{i.e. } \rho \text{ modified at point } x \\
[\![e_1 \geq e_2]\!]\ \rho & = \rho & \text{if } [\![e_1]\!]\ \rho \geq [\![e_2]\!]\ \rho
\end{array}$$

A path π is a sequence of consequetive edges in the CFG.



$$\pi = k_1, \ldots, k_n \text{ where each } k_i \text{ is of the form } (u_{i-1}, l_i, u_i).$$

$$\text{We write } \pi : u_0 \to^* u_n$$

The transformation induced by a path is the composition of the transformations induced by the edges.

$$[\![\pi]\!] = [\![k_n]\!] \circ \ldots \circ [\![k_1]\!]$$

Each node can be reached through possibly infinitely many paths, leading to infinitely many different states at each program point.

We are interested in the set of all such states at each program point.

Suppose we know that a set V of states is possible at a node u.

By following an edge $k = (u, l, v)$, a new set of states becomes possible at node v. This set is denoted $[\![k]\!]^\sharp\ V = [\![l]\!]^\sharp\ V : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$.

We define abstract transformation

$$[\![l]\!]^\sharp\ V = \{[\![l]\!]\ \rho \mid \rho \in V \text{ and } [\![l]\!] \text{ is defined for } \rho\}.$$

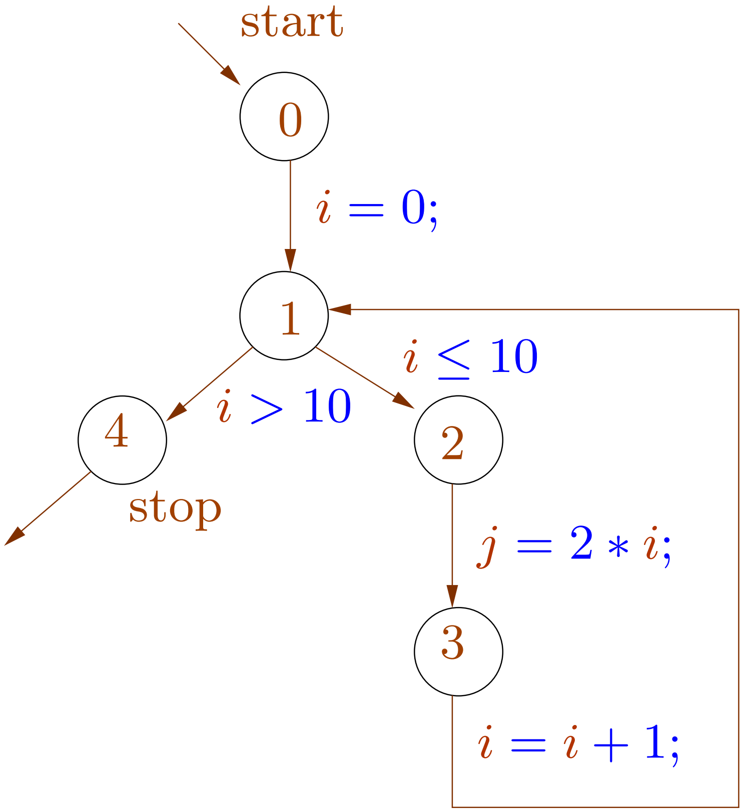
As before, $[\![k_1, \ldots, k_n]\!]^\sharp\ V = ([\![k_n]\!]^\sharp \circ \ldots \circ [\![k_1]\!]^\sharp)V$.

At the *start* node, all states are possible.

For each node v we want to compute the set

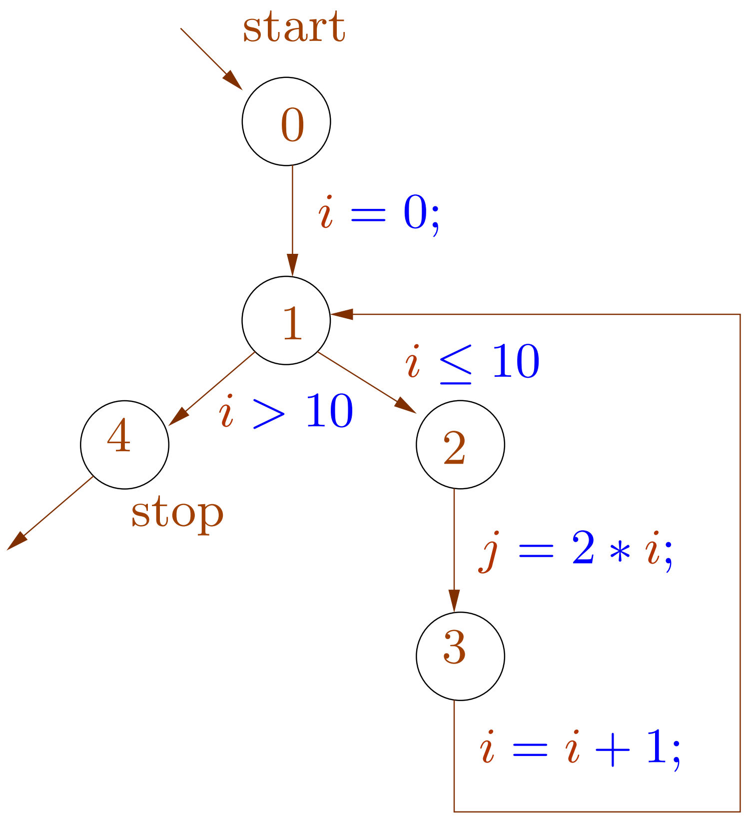
$$\mathcal{V}^*[v] = \bigcup\{[\![\pi]\!]^\sharp\ \mathcal{S} \mid \pi : \mathit{start} \to^* v\}$$

Example



u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 11 \wedge j = 2i{-}2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i{-}2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

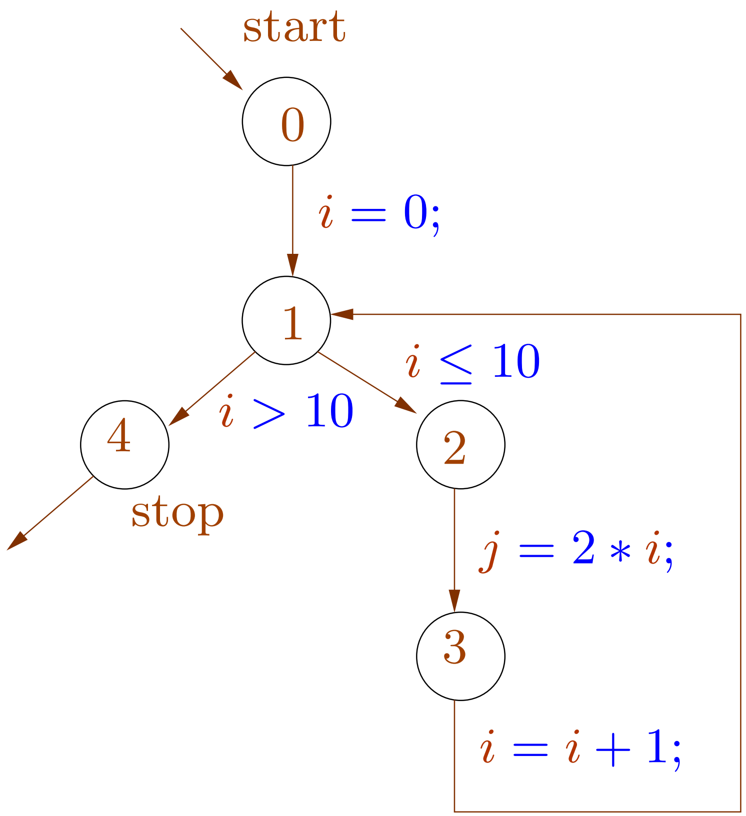
Example



u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 11 \wedge j = 2i - 2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i - 2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

How to compute the sets $\mathcal{V}^*[v]$ in general?

Example

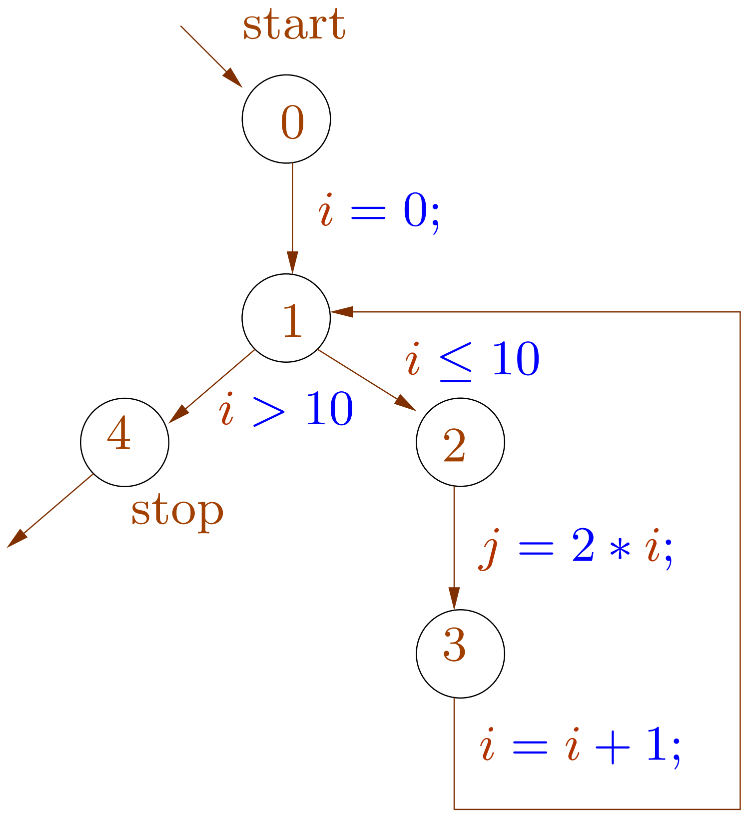


u	$\mathcal{V}^*[u]$
0	$-\infty < i, j < \infty$
1	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 11 \wedge j = 2i-2$
2	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i-2$
3	$i = 0 \wedge -\infty < j < \infty$ $\vee 1 \leq i \leq 10 \wedge j = 2i$
4	$i = 11 \wedge j = 20$

How to compute the sets $\mathcal{V}^*[v]$ in general?

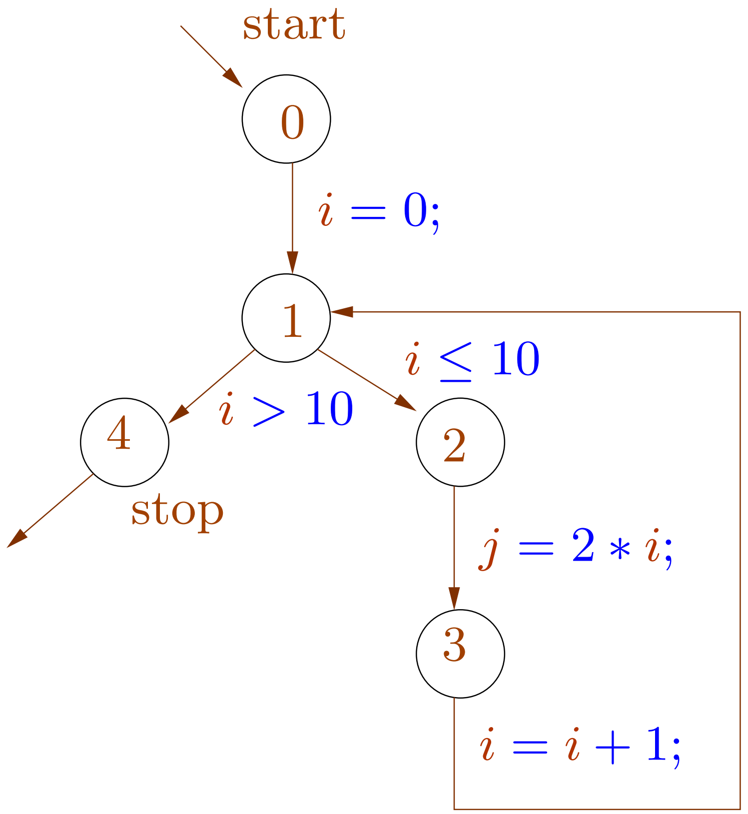
In general they are not computable!

We set up a constraint system.



$$\begin{array}{lll}
\mathcal{V}[0] & \supseteq & \mathcal{S} \\
\mathcal{V}[1] & \supseteq & [\![i = 0;]\!]\ \mathcal{V}[0] \\
\mathcal{V}[1] & \supseteq & [\![i = i{+}1;]\!]\ \mathcal{V}[3] \\
\mathcal{V}[2] & \supseteq & [\![i \leq 10]\!]\ \mathcal{V}[1] \\
\mathcal{V}[3] & \supseteq & [\![j = 2{*}i;]\!]\ \mathcal{V}[2] \\
\mathcal{V}[4] & \supseteq & [\![i > 10]\!]\ \mathcal{V}[1]
\end{array}$$

We set up a constraint system.



$$\begin{aligned}
\mathcal{V}[0] &\supseteq \mathcal{S} \\
\mathcal{V}[1] &\supseteq [\![i = 0;]\!]\ \mathcal{V}[0] \\
\mathcal{V}[1] &\supseteq [\![i = i+1;]\!]\ \mathcal{V}[3] \\
\mathcal{V}[2] &\supseteq [\![i \leq 10]\!]\ \mathcal{V}[1] \\
\mathcal{V}[3] &\supseteq [\![j = 2*i;]\!]\ \mathcal{V}[2] \\
\mathcal{V}[4] &\supseteq [\![i > 10]\!]\ \mathcal{V}[1]
\end{aligned}$$

The least solution (wrt $\subseteq$) of the constraints is exactly $\mathcal{V}^*$.

The least solution (wrt $\subseteq$) of the constraints is exactly $\mathcal{V}^*$.

Is this always true?

Does such a constraint system always have a least solution?

Is it computable? Efficiently?