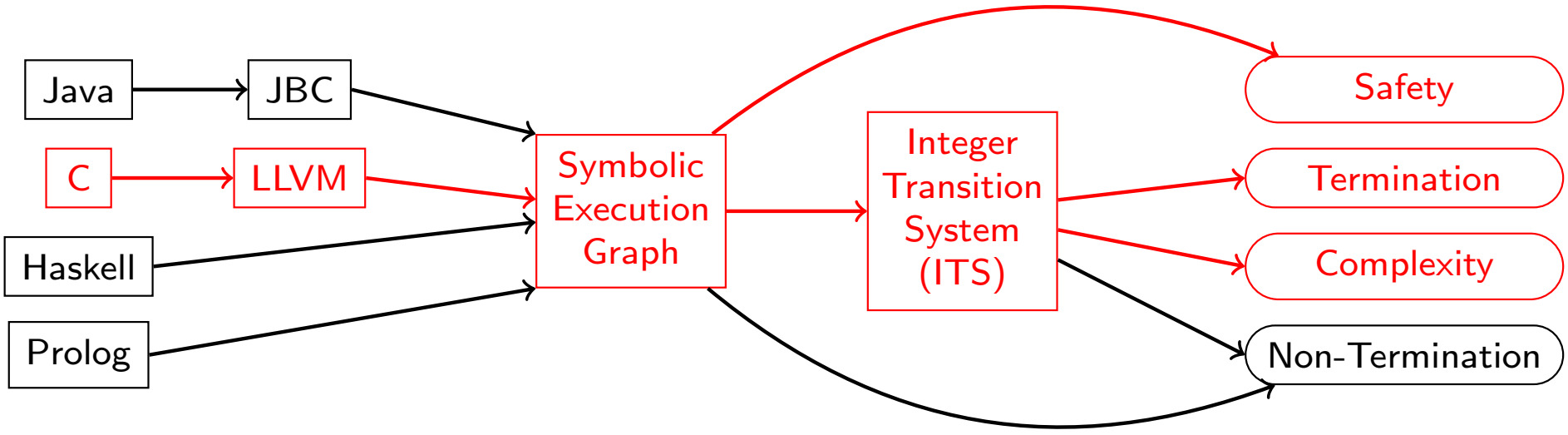
Complexity Analysis for Bitvector Programs

Jürgen Giesl

LuFG Informatik 2, RWTH Aachen University, Germany

joint work with Jera Hensel and Florian Frohn

Termination and Complexity Analysis in AProVE



- Handling of C programs with explicit pointer arithmetic
- **Drawback:**
  many tools assume mathematical integers $\mathbb{Z}$ instead of bitvectors

Mathematical Integers $\mathbb{Z}$ vs. Bitvectors

```
void f(unsigned int x)  {
    unsigned int j = 0;
    while (j <= x) j++;  }
```

for $\mathbb{Z}$: termination
for bitvectors: non-termination

```
void g(unsigned int j)  {
    while (j > 0) j++;  }
```

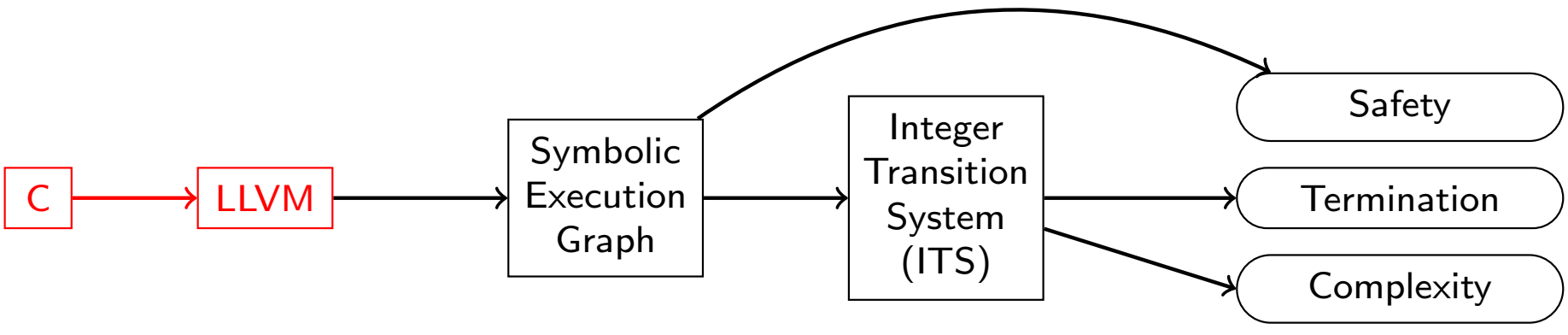
for $\mathbb{Z}$: non-termination
for bitvectors: termination

- **Goal:** adapt byte-accurate symbolic execution to bitvector arithmetic
- **Solution:** express bitvector relations by relations on $\mathbb{Z}$
  - standard SMT solving over $\mathbb{Z}$ for symbolic execution
  - standard ITSs over $\mathbb{Z}$ for termination and complexity analysis

From C to LLVM

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

```
void g(unsigned int j)  {
    while (j > 0) j++;  }
```



Abstract States

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

a

$(\texttt{entry}, 2)$	$\Longleftarrow$ *pos*
$\{\texttt{j} = v_{\texttt{j}}, \texttt{ad} = v_{\texttt{ad}}\}$	$\Longleftarrow$ *PV*
$\{[\![v_{\texttt{ad}}, v_{end}]\!]\}$	$\Longleftarrow$ *AL*
$\{v_{end} = v_{\texttt{ad}} + 3\}$	$\Longleftarrow$ *KB*
$\{v_{\texttt{ad}} \hookrightarrow_{\texttt{i32}} v_{\texttt{j}}\}$	$\Longleftarrow$ *PT*

Abstract state *a*: *ERR* or

- *pos*: program position (block, next instruction)
- *PV*: program variables $\rightarrow$ symbolic variables
- *AL*: allocation list $[\![v_1, v_2]\!]$
- *KB*: knowledge base (FO-(in)equalities over symbolic variables)
- *PT*: points-to atoms $v_1 \hookrightarrow_{\texttt{type}} v_2$

Abstract States

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

a

$$
\begin{array}{|l|l}
\hline
(\mathtt{entry}, 2) & \Longleftarrow \mathit{pos} \\
\{\mathtt{j} = v_{\mathtt{j}}, \mathtt{ad} = v_{\mathtt{ad}}\} & \Longleftarrow \mathit{PV} \\
\{[\![v_{\mathtt{ad}}, v_{\mathit{end}}]\!]\} & \Longleftarrow \mathit{AL} \\
\{v_{\mathit{end}} = v_{\mathtt{ad}} + 3\} & \Longleftarrow \mathit{KB} \\
\{v_{\mathtt{ad}} \hookrightarrow_{\mathtt{i32}} v_{\mathtt{j}}\} & \Longleftarrow \mathit{PT} \\
\hline
\end{array}
$$

- $\langle a \rangle$: FO formula containing
  - KB and consequences of AL and PT
  - information on ranges of integers:

    $\mathtt{j}$ has type $\mathtt{i32} \quad \Rightarrow \quad 0 \leq \underbrace{PV(\mathtt{j})}_{v_{\mathtt{j}}} \leq \underbrace{\mathrm{umax}_{32}}_{2^{32}-1}$

Symbolic Execution

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

A

$(\texttt{entry}, 0)$ $\Longleftarrow$ *pos*

$\{\texttt{j} = v_{\texttt{j}}, \ldots\}$ $\Longleftarrow$ *PV*

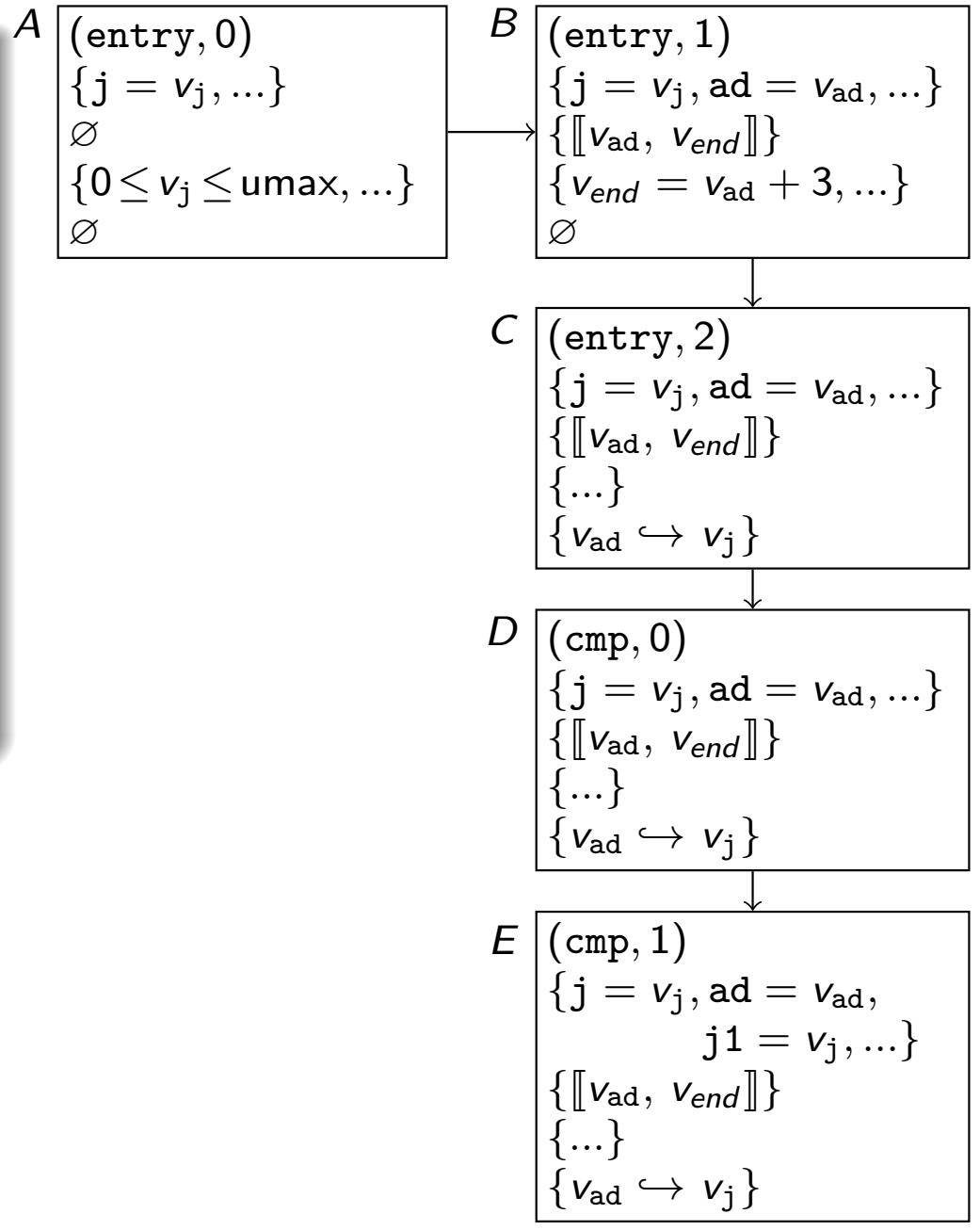
$\varnothing$ $\Longleftarrow$ *AL*

$\{0 \leq v_{\texttt{j}} \leq \texttt{umax}, \ldots\}$ $\Longleftarrow$ *KB*

$\varnothing$ $\Longleftarrow$ *PT*

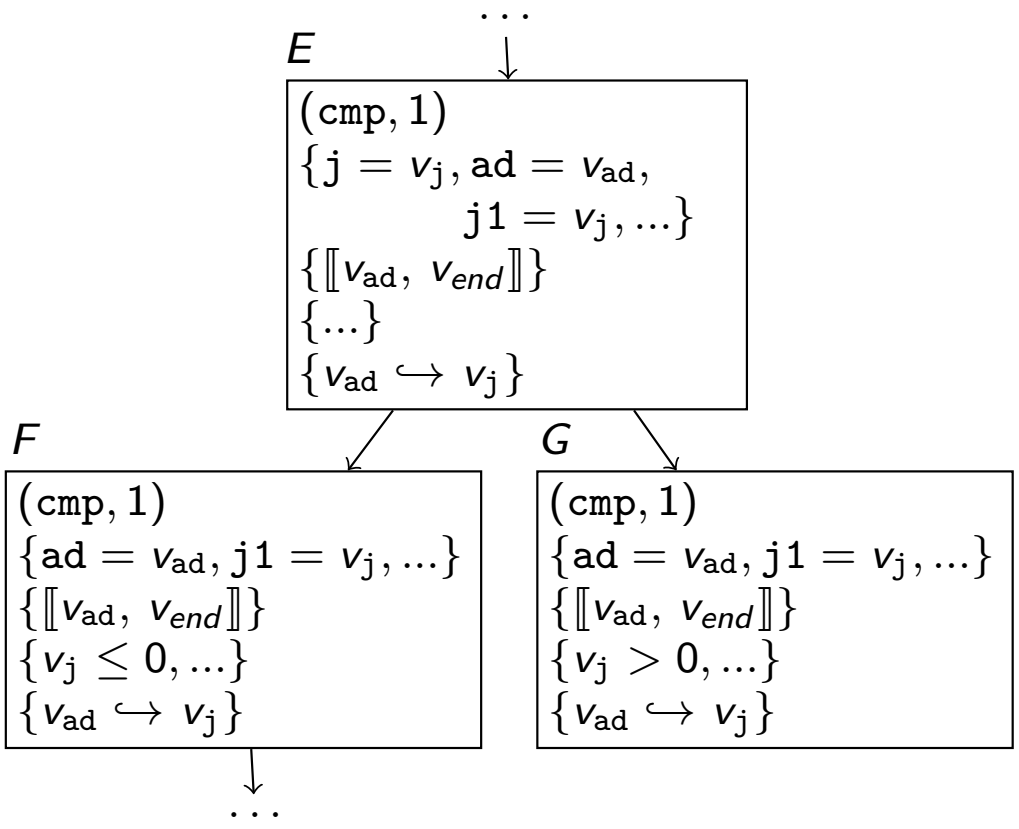
Symbolic Execution

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```



Integer Comparison

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

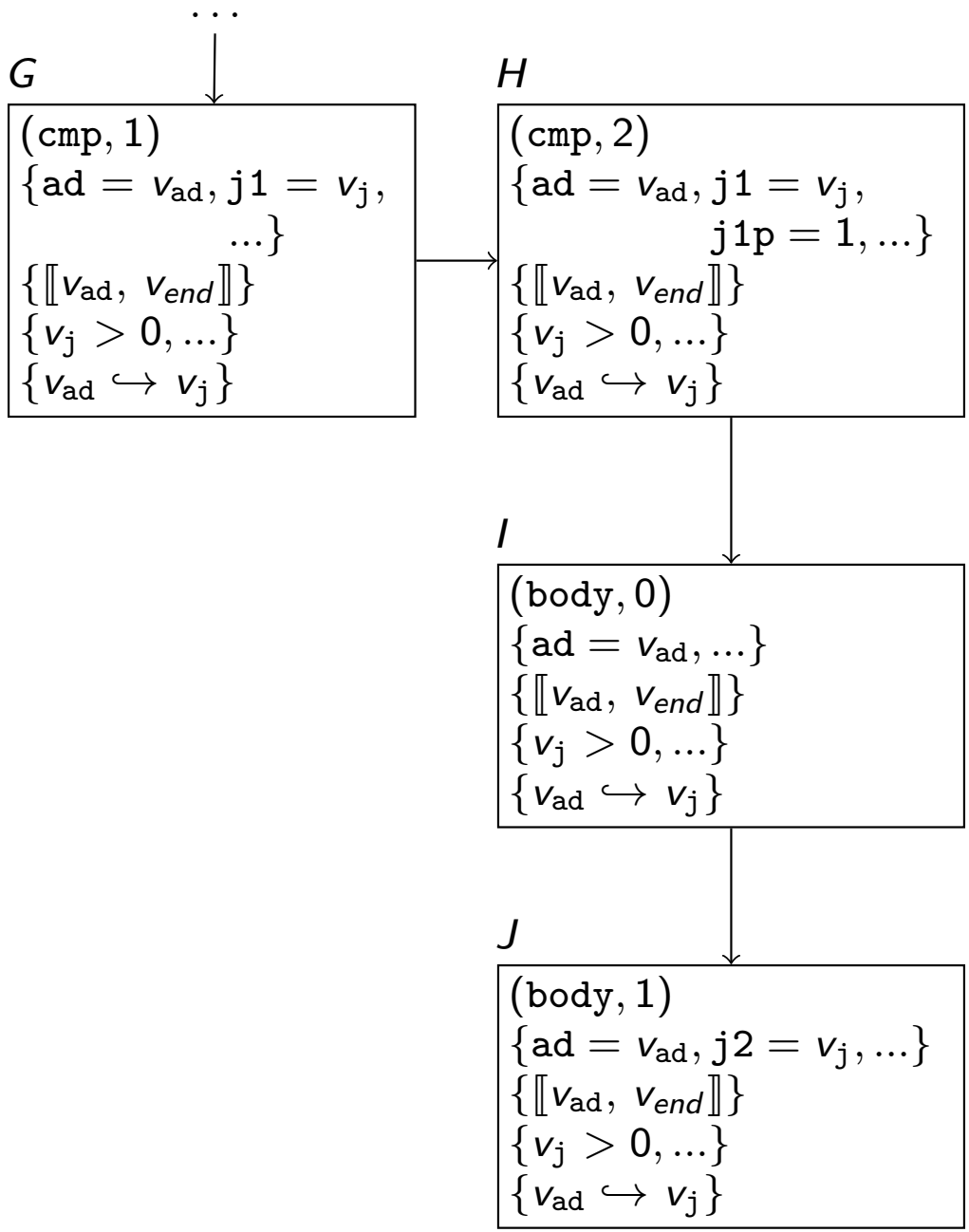


Symbolic execution rule for `x = icmp ugt i32` t_1, t_2

- set `x` to 1 if $\models \langle a \rangle \implies (PV(t_1) > PV(t_2))$
- set `x` to 0 if $\models \langle a \rangle \implies (PV(t_1) \leq PV(t_2))$
- otherwise: case analysis

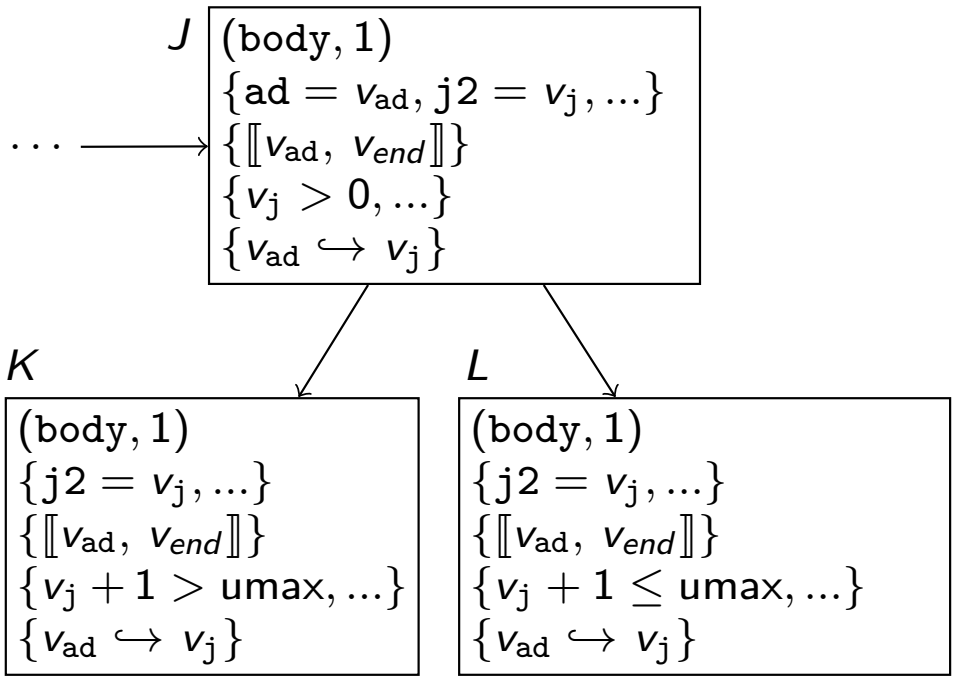
Symbolic Execution

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```



Addition

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

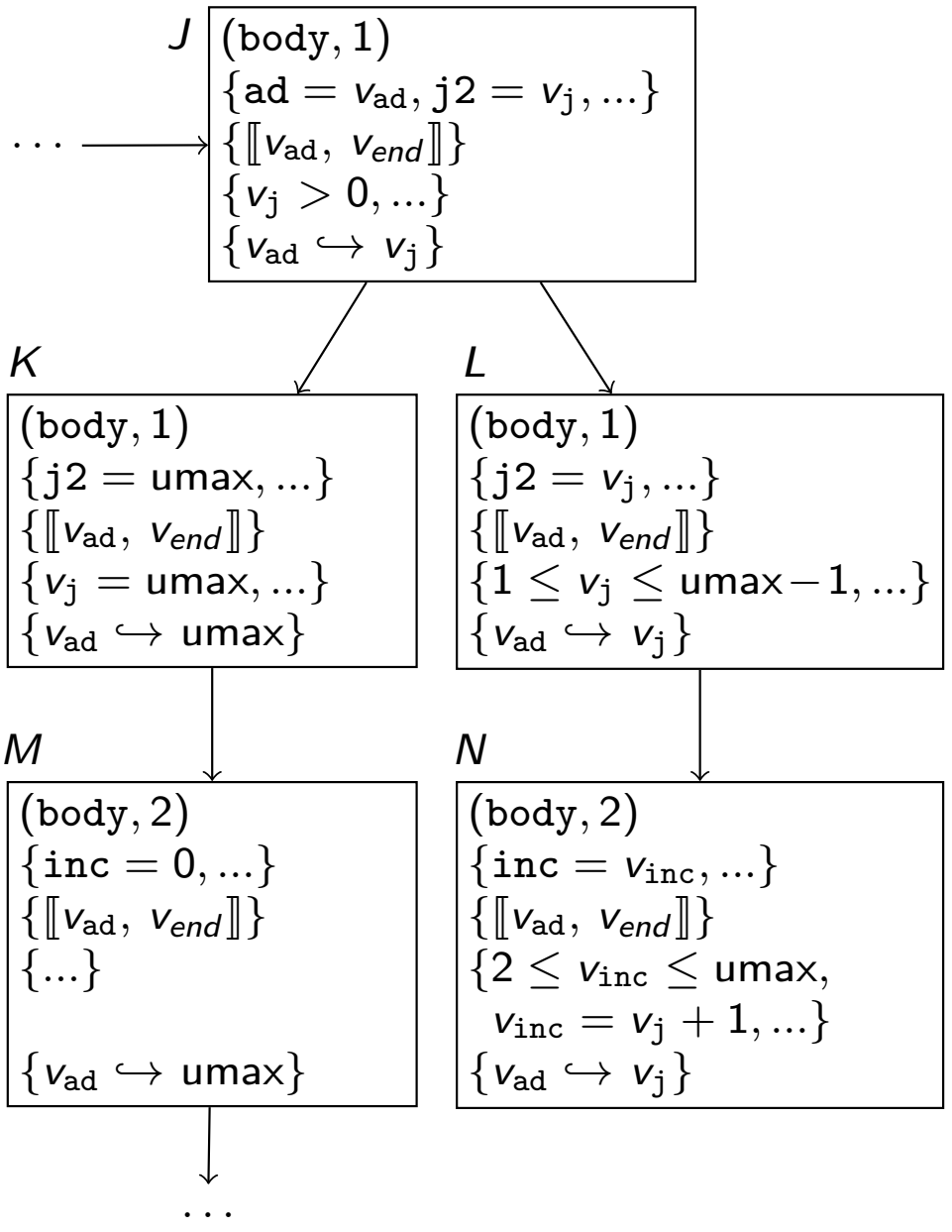


Symbolic execution rule for `x = add i32` t_1, t_2

- set `x` to $PV(t_1) + PV(t_2)$ if $\models \langle a \rangle \implies PV(t_1) + PV(t_2) \leq \text{umax}_{32}$
- set `x` to $PV(t_1) + PV(t_2) - 2^{32}$ if $\models \langle a \rangle \implies PV(t_1) + PV(t_2) > \text{umax}_{32}$
- otherwise: case analysis

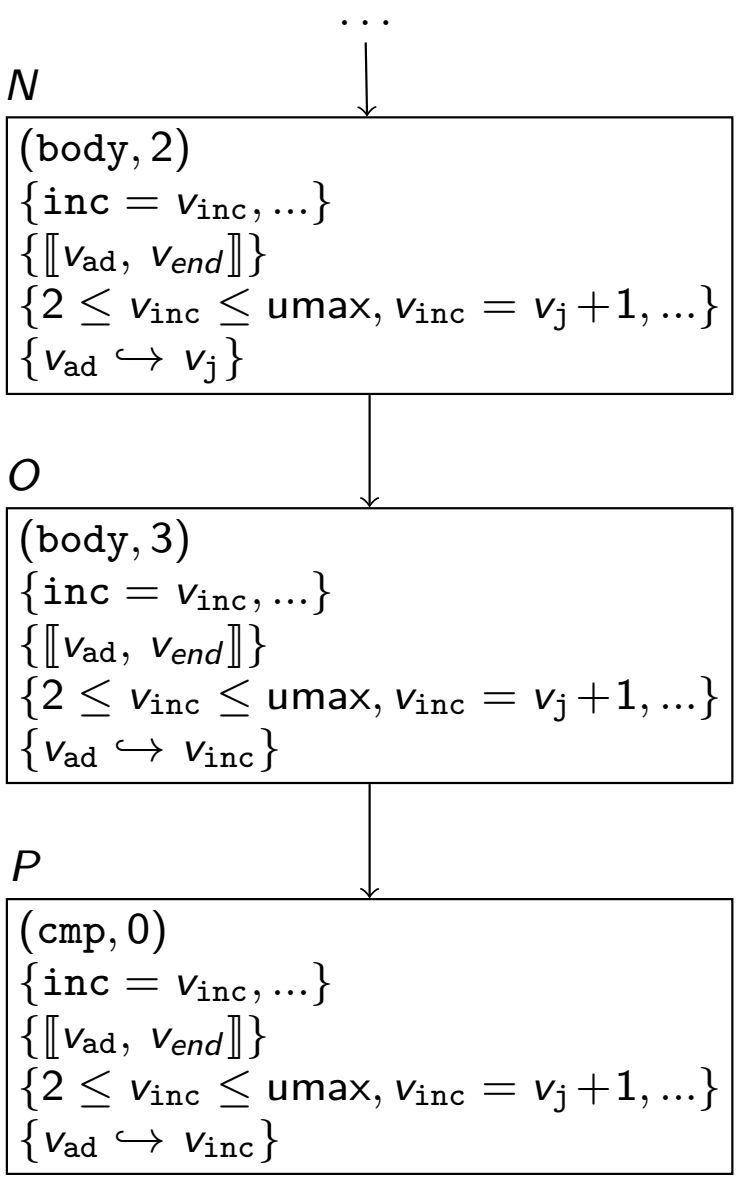
Addition

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```



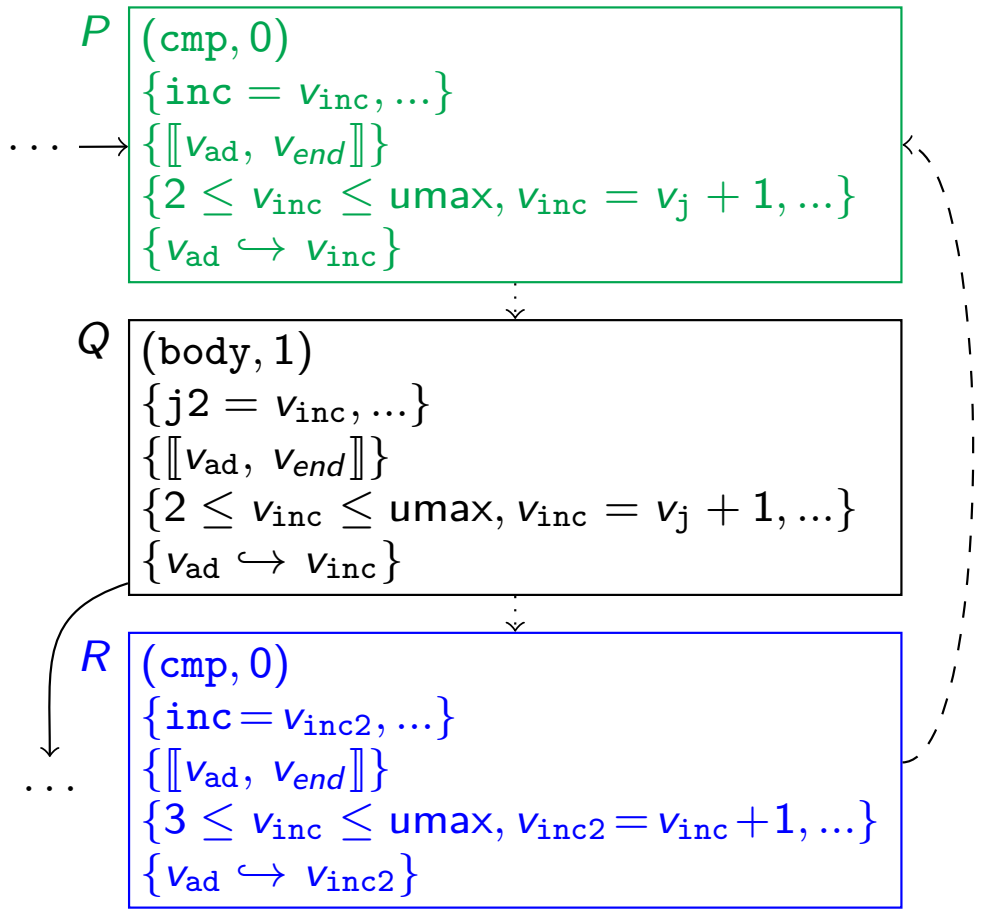
Symbolic Execution

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```



Generalization

```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```

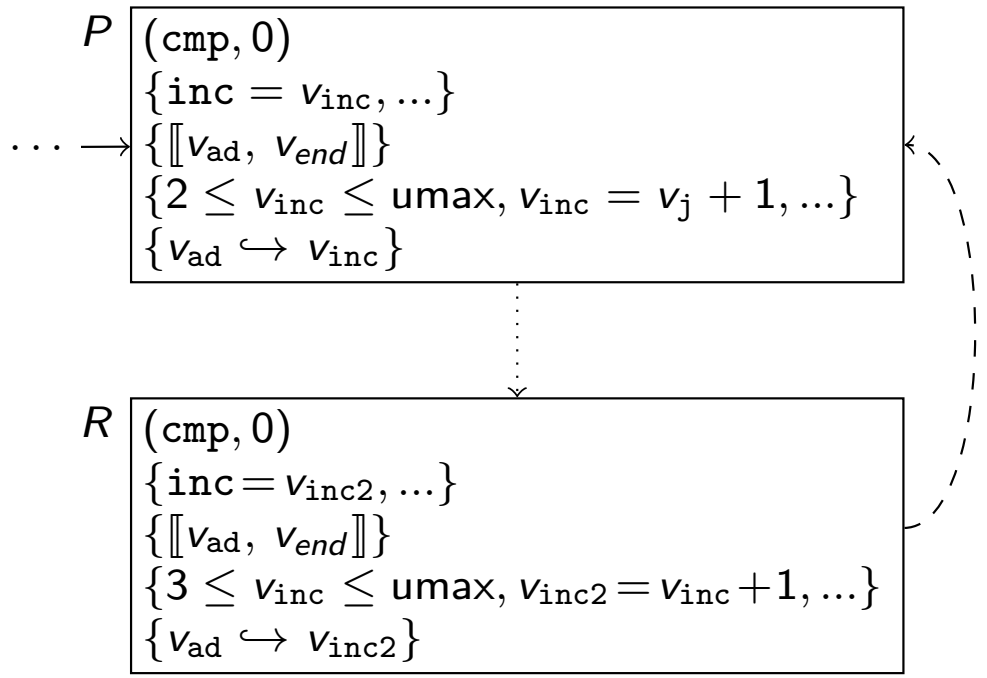


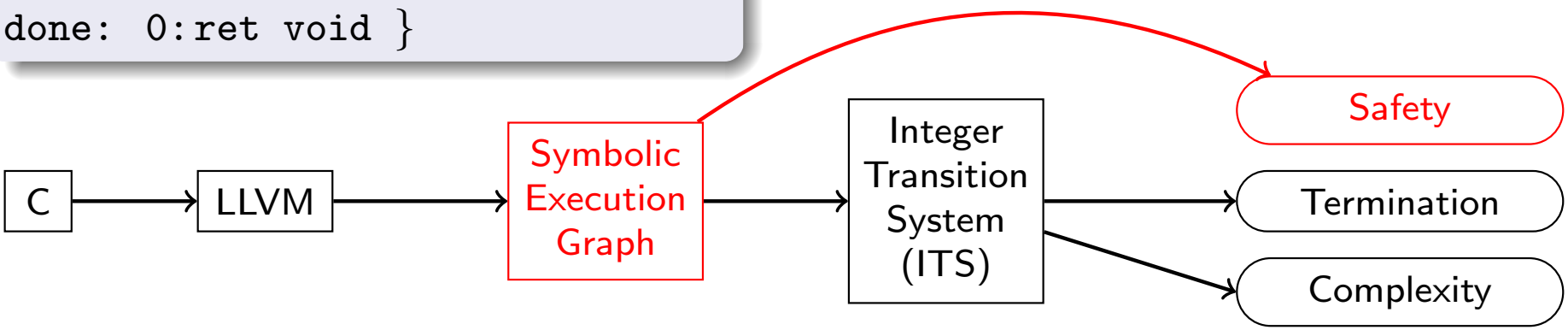
P is *generalization* of R with $\mu(v_j) = v_{inc}$, $\mu(v_{inc}) = v_{inc2}$

- $\mu(PV_P(x)) = PV_R(x)$ for all program variables x
- $[\![v_1, v_2]\!] \in AL_P$ implies $[\![\mu(v_1), \mu(v_2)]\!] \in AL_R$
- $\models \langle R \rangle \implies \mu(KB_P)$
- $v_1 \hookrightarrow v_2 \in PT_P$ implies $\mu(v_1) \hookrightarrow \mu(v_2) \in PT_R$

Safety

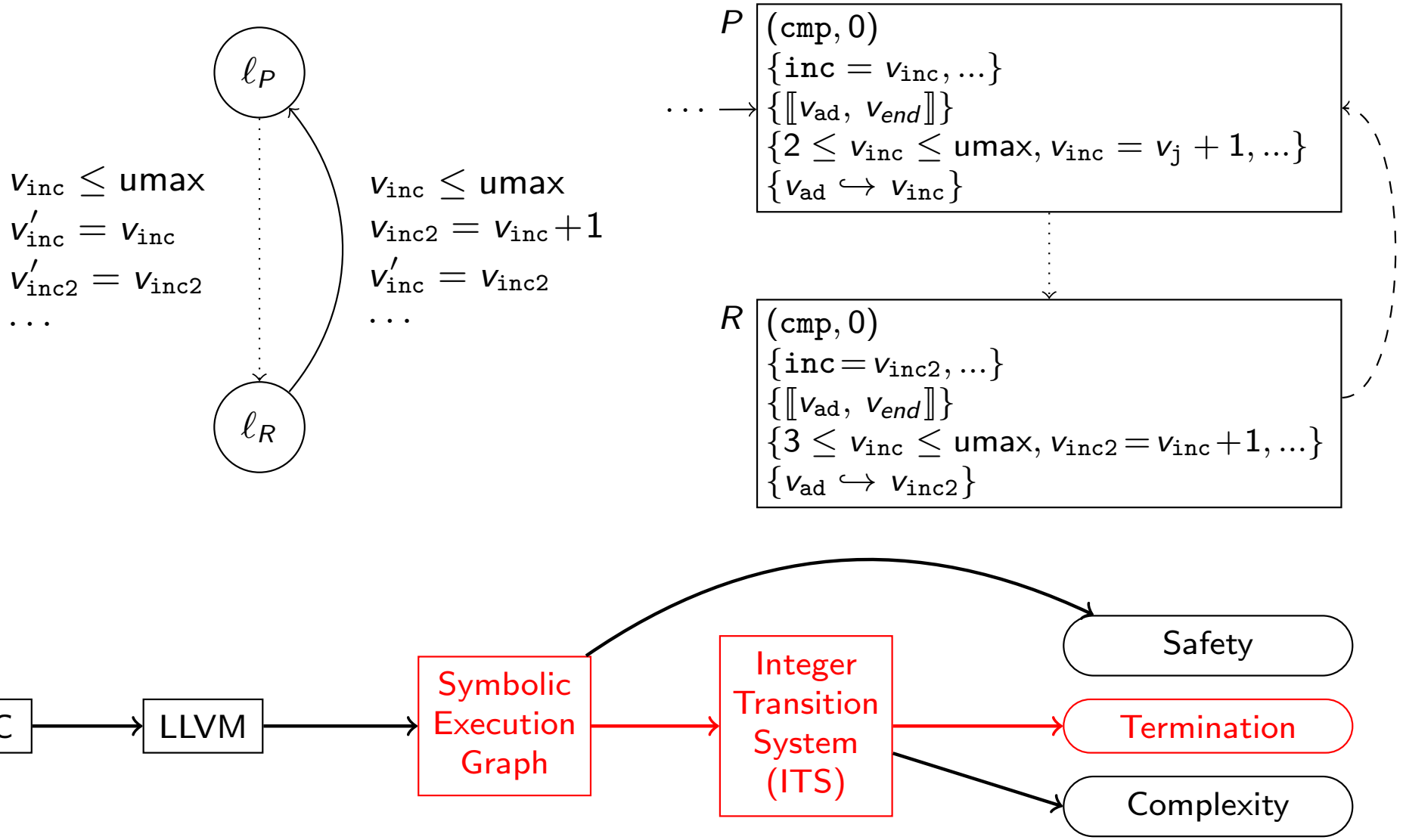
```
define void @g(i32 j) {
entry: 0:ad = alloca i32
       1:store i32 j, i32* ad
       2:br label cmp
cmp:   0:j1 = load i32* ad
       1:j1p = icmp ugt i32 j1, 0
       2:br i1 j1p, label body,
                    label done
body:  0:j2 = load i32* ad
       1:inc = add i32 j2, 1
       2:store i32 inc, i32* ad
       3:br label cmp
done:  0:ret void }
```





- Symbolic execution graph complete if leaves correspond to `return`
- Complete symbolic execution graph without *ERR* $\Longrightarrow$ Safety

Termination



- ITS from cycles of symbolic execution graph
- ITS termination by existing tools $\Longrightarrow$ LLVM program terminates

Complexity

- Adapt transformation from symbolic execution graph to ITSs
- Use existing complexity tools for ITSs (over $\mathbb{Z}$)
- Arithmetic programs $\in \mathcal{O}(1)$ $\Rightarrow$ asymptotic complexity meaningless
  - m instructions
  - k variables $x_1 : \mathtt{i}n_1, \ldots, x_k : \mathtt{i}n_k$
  - runtime bounded by $m \cdot 2^{n_1} \cdot \ldots \cdot 2^{n_k}$
- **Goal:** infer concrete bounds $\leq m \cdot 2^{n_1} \cdot \ldots \cdot 2^{n_k}$

  bounds depending on program's input parameters are better than bounds depending on sizes of types $\mathtt{i}n$

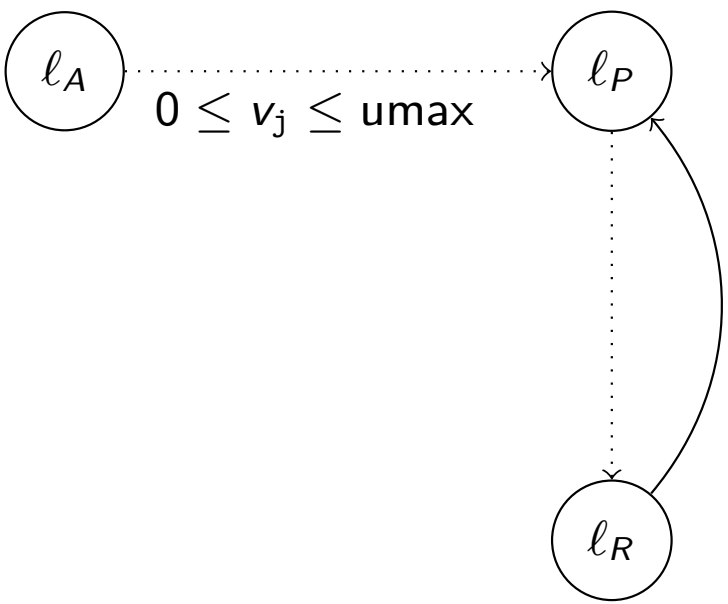
Complexity: Adapt Approach for Termination

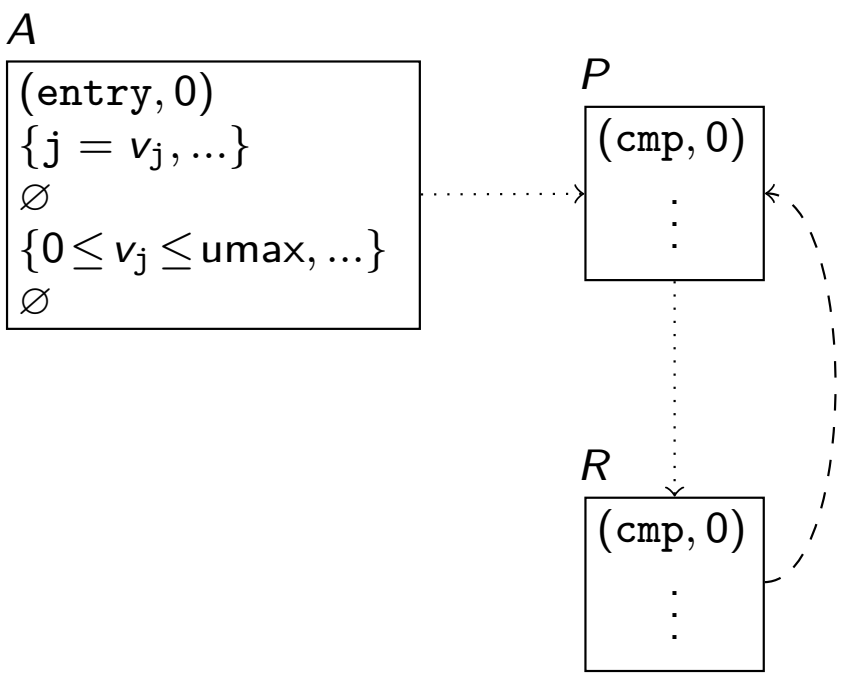
1. Every execution step counts for complexity

   $\Rightarrow$ generate ITS from whole graph, not just from cycles

2. Find bound on length of ITS evaluations

   depending on values in initial state

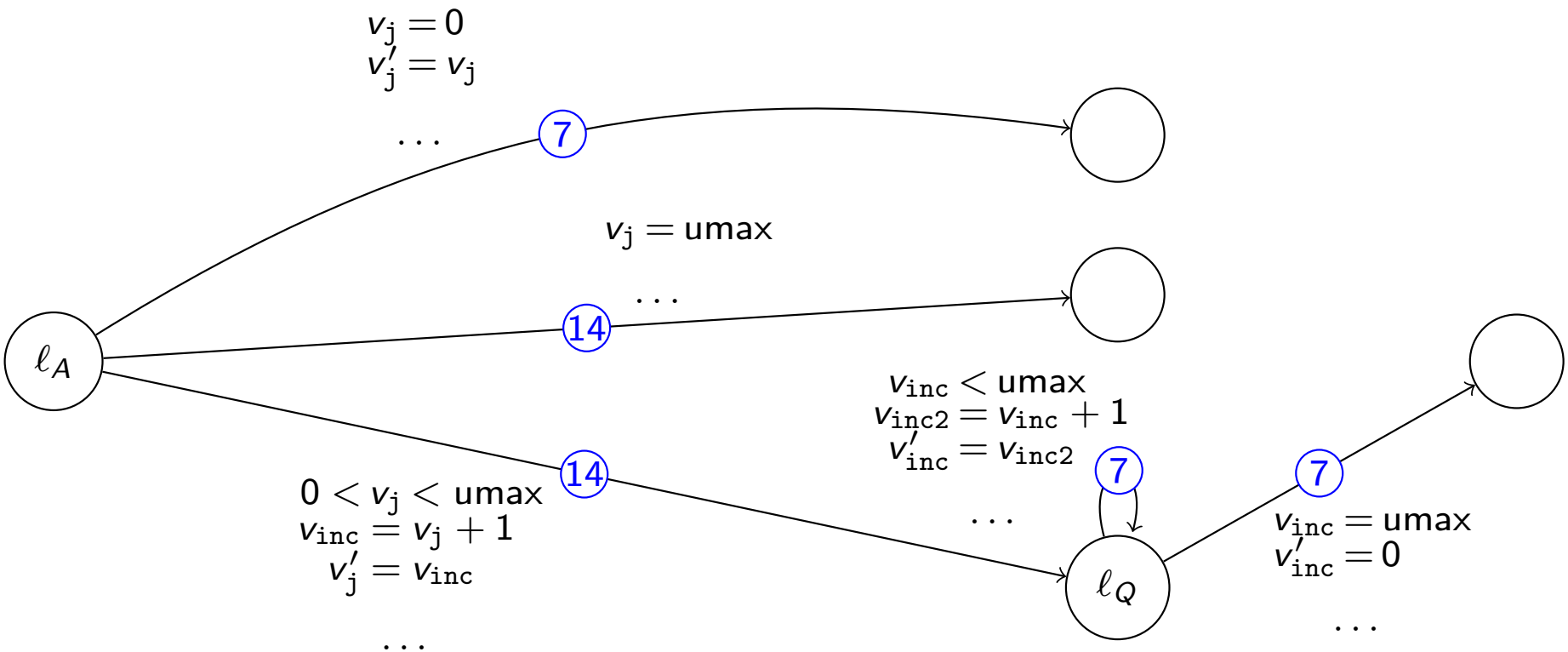




Complexity: Adapt Approach for Termination

3. Simplify ITSs by filtering away variables and compressing transitions

   $\Rightarrow$ use weighted transitions

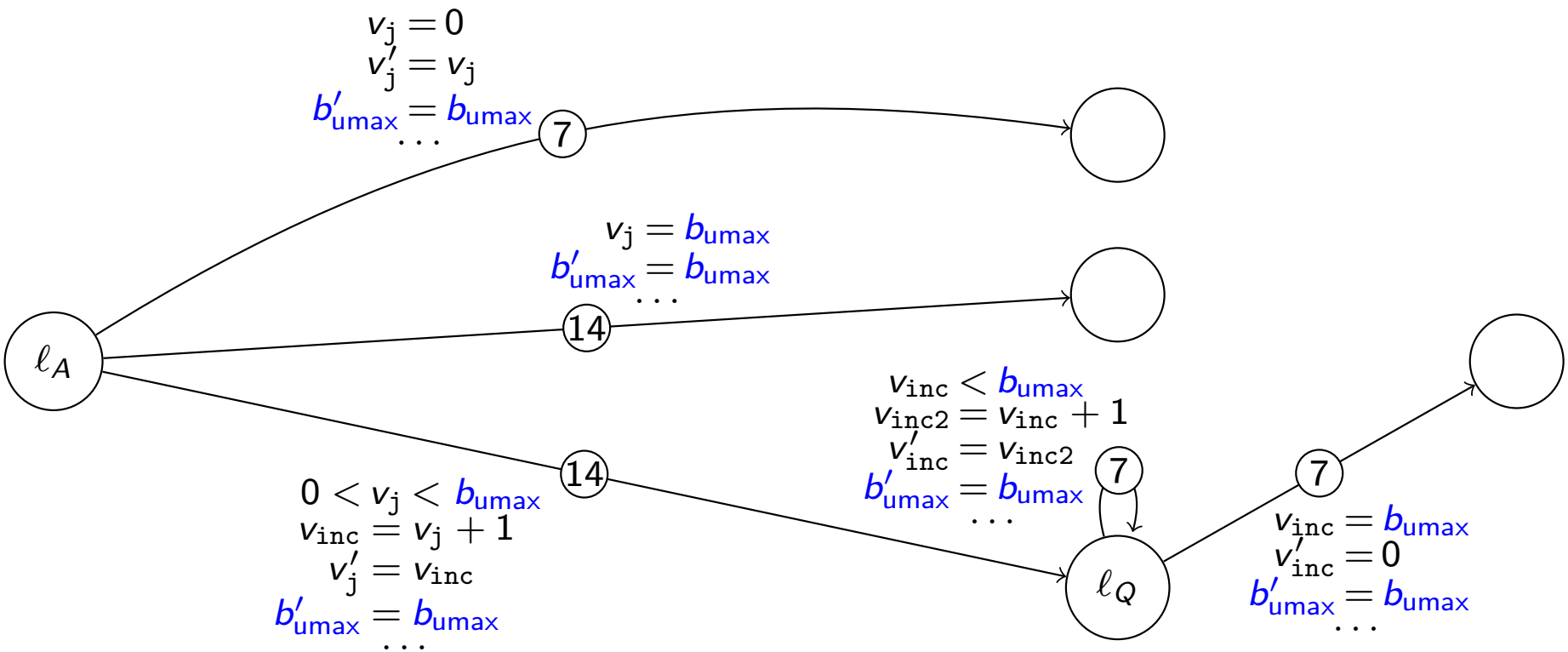


Complexity: Adapt Approach for Termination

4. ITS tools prefer small asymptotic bounds

   (huge constant umax preferred to bound depending on input parameters v_j)

   $\Rightarrow$ replace size constant umax by variable b_{umax}

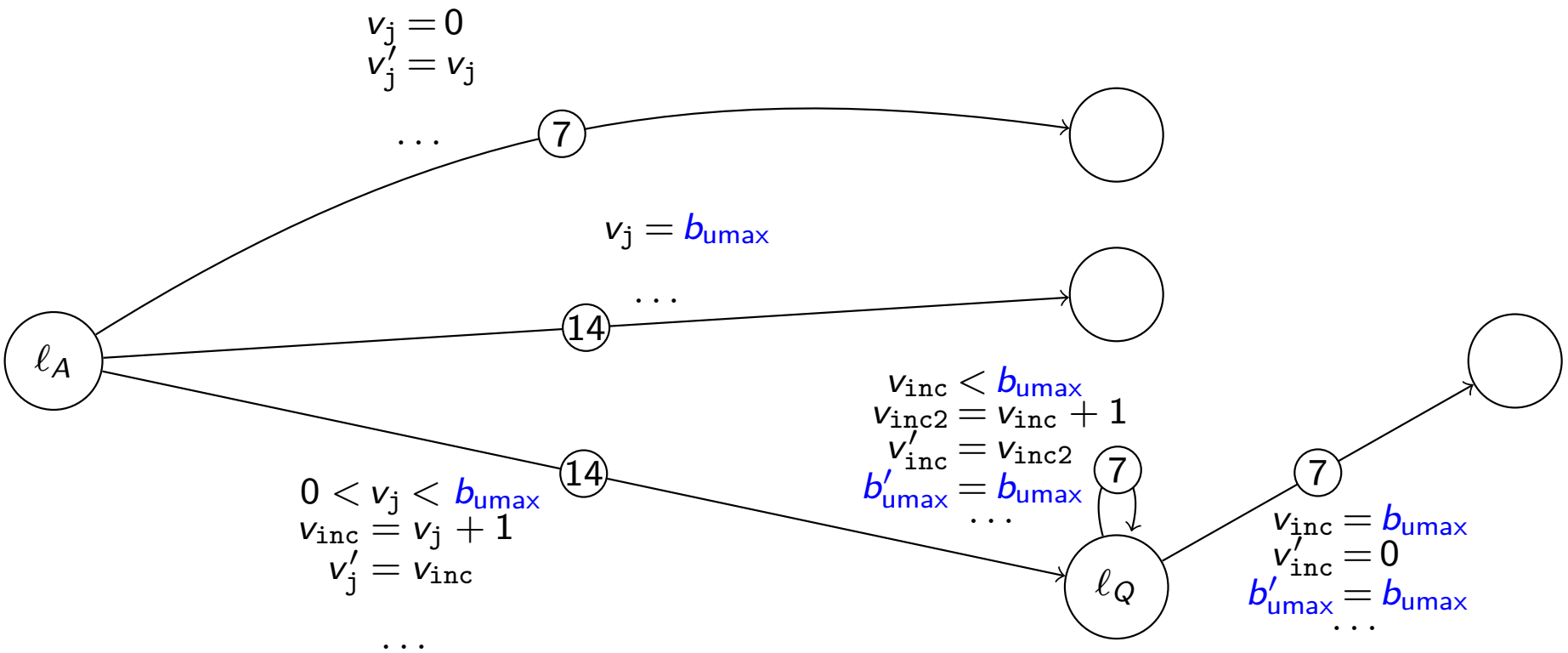


Complexity: Adapt Approach for Termination

5. Bounds depending on program's input parameters v_j are better than bounds depending on sizes of types b_{umax}

   $\Rightarrow$ assign b_{umax} non-deterministically in initial transitions

   $\Rightarrow$ if this fails, add $b'_{umax} = b_{umax}$ in initial transitions

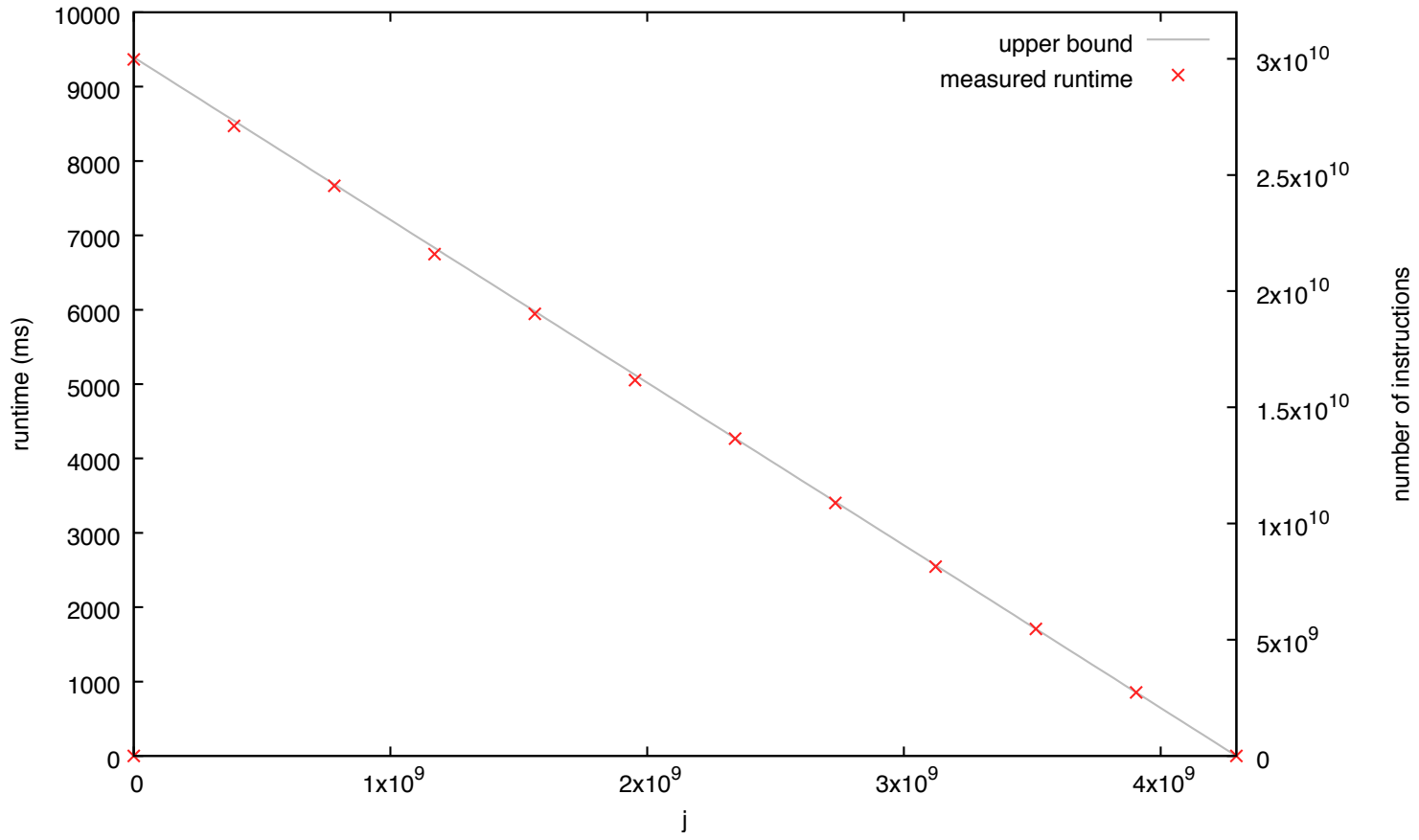


Complexity: Adapt Approach for Termination

KoAT and CoFloCo yield:

$$\max(21, 7 \cdot \underbrace{\text{umax}}_{2^{32}-1} - 7 \cdot \text{j} + 14)$$

```
void g(unsigned int j)  {
    while (j > 0) j++;  }
```



Complexity Analysis for Bitvector Programs

- Handling of bitvectors during symbolic execution

- Representation of bitvectors by relations on $\mathbb{Z}$

  $\Rightarrow$ standard SMT solving over $\mathbb{Z}$

  $\Rightarrow$ standard back-end tools for termination and complexity analysis of ITSs

- Generation of ITSs slightly different for termination or complexity

- Implementation in AProVE (using KoAT and CoFloCo in the back-end)
  - 118 bitvector C programs from evaluations of other termination tools
  - 95 programs: AProVE proves termination
  - 60 programs: AProVE infers upper bound
    - 7 programs: small constant bound
    - 41 programs: linear or quadratic in input variables
    - 12 programs: bound also depends on size of types