An Introduction to BLAST

John Wickerson

What is BLAST?

- An automatic verification tool for checking properties of C programs
- **B**erkeley **L**azy **A**bstraction **S**oftwareverification **T**ool

A correct program

```
int main() {
  int x,y;
  if (x > y) {
    x = x-y;
    if (x <= 0) {
      ERROR: goto ERROR;
    }
  }
}
```

```
$ blast prog/prog.c
BLAST 2.5, Copyright (c) 2002-2008, The
BLAST Team.
...
prog.c:3: Warning: Body of function main
falls-through. Adding a return statement
...
Starting phase 4
[BAT] Calling refiner
addPred: 0: (gui) adding predicate
x@main*-2+y@main*2<=-2 to the system
addPred: 0: (gui) adding predicate
x@main*-2+y@main*2<=-2 to the system
addPred: 1: (gui) adding predicate
x@main*-2<=-2 to the system
addPred: 1: (gui) adding predicate
x@main*-2<=-2 to the system
Adding all preds now...
[BAT] Done refiner
...
No error found.  The system is safe :-)

$
```


3

An incorrect program

```
int main() {
  int x,y;
  if (x > y) {
    x = x-y;
    if (x <= 1)
      ERROR: gotc
    }
  }
}
```

```
$ blast prog/prog.c
BLAST 2.5, Copyright (c) 2002-2008, The BLAST Team.
...
prog.c:3: Warning: Body of function main falls-through.
Adding a return statement
...
0 :: 0:   FunctionCall(__BLAST_initialize_prog/prog.c()) :: -1
0 :: 0:   Block(Return(0);) :: -1
-1 :: -1:     Skip :: 3
3 :: 3:   Pred(x@main  >  y@main) :: -1
4 :: 4:   Block(x@main = x@main  -  y@main;) :: 5
5 :: 5:   Pred(x@main  <=  1) :: -1
...
Error found! The system is unsafe :-(

$
```

Assertion checking

```
#include <assert.h>

int foo(int x) {
 if (x > 0) {
   x++;
   assert(x > 0);
 }
}
```


5

Assertion checking

```c
#include <assert.h>

int foo(int x) {
  if (x > 0) {
    x++;
    assert(x > 0);
  }
}
```

```c
if (!(x > 0)) {
  ERROR: goto ERROR
}
```

```
$ blastpp prog2
Preprocessing prog2.c, please wait...
Done.

$ blast prog2.i -main foo
No error found.  The system is
safe :-)

$
```


5

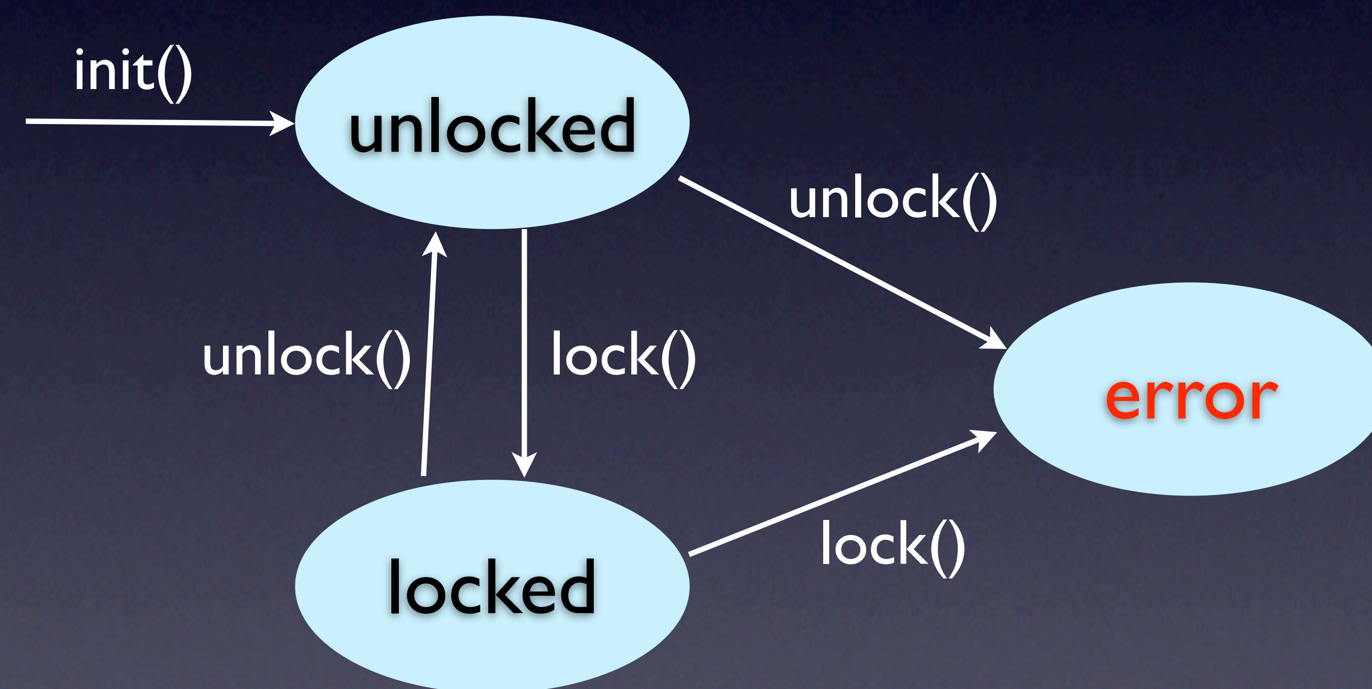
Aside: assertions

```c
#include <assert.h>

int main () {
  int x = 0;
  int y = 0;
  while (x==y) {
    x++;y++;
  }
  assert(0);
}
```

```
$ blastpp prog3
Preprocessing prog3.c, please wait...
Done.

$ blast prog3.i -main foo
No error found.  The system is
safe :-)

$
```

Temporal safety specifications

```
int main () {
  int x,y;
  init();
  do {
    lock();
    y = x;
    if (x < 100) {
      unlock();
      x++;
    }
  } while (x != y);
  unlock();
}
```



Temporal safety specificat

```
int main () {
  int x,y;
  init();
  do {
    lock();
    y = x;
    if (x < 100) {
      unlock();
      x++;
    }
  } while (x != y);
  unlock();
}
```

```
int main () {
  int x,y;
  init();
  int locked = 0;
  do {
    assert(locked == 0);
    lock();
    locked = 1;
    y = x;
    if (x < 100) {
      assert(locked == 1);
      unlock();
      locked = 0;
      x++;
    }
  } while (x != y);
  assert(locked == 1);
  unlock();
  locked = 0;
}
```

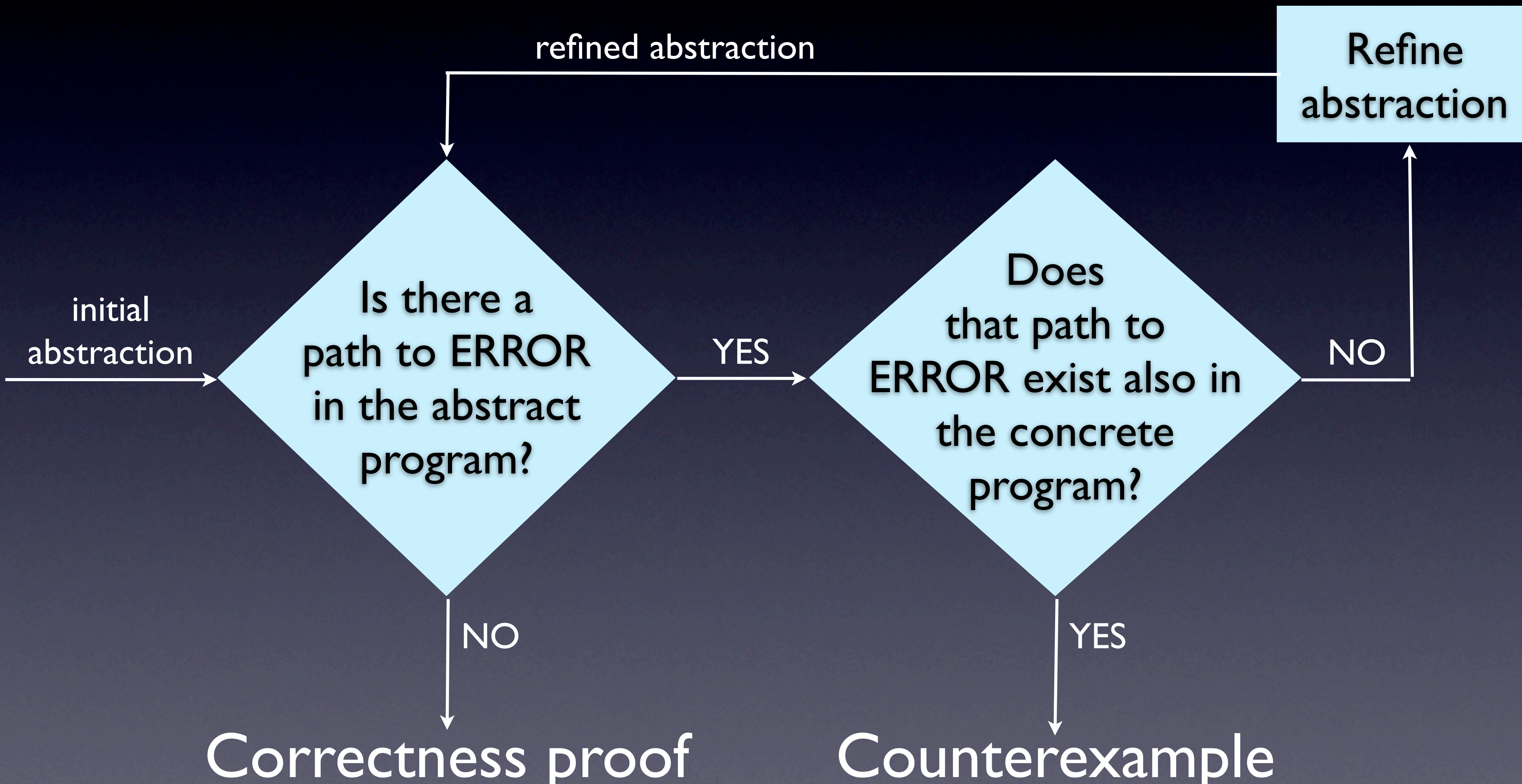
Temporal safety specificat

```
global int locked = 0;

event {
  pattern { $? = init(); }
  action { locked = 0; }
}
event {
  pattern { $? = lock(); }
  guard { locked == 0 }
  action { locked = 1; }
}
event {
  pattern { $? = unlock(); }
  guard { locked == 1 }
  action { locked = 0; }
}
```

```
int main () {
  int x,y;
  init();
  int locked = 0;
  do {
    assert(locked == 0);
    lock();
    locked = 1;
    y = x;
    if (x < 100) {
      assert(locked == 1);
      unlock();
      locked = 0;
      x++;
    }
  } while (x != y);
  assert(locked == 1);
  unlock();
  locked = 0;
}
```


9

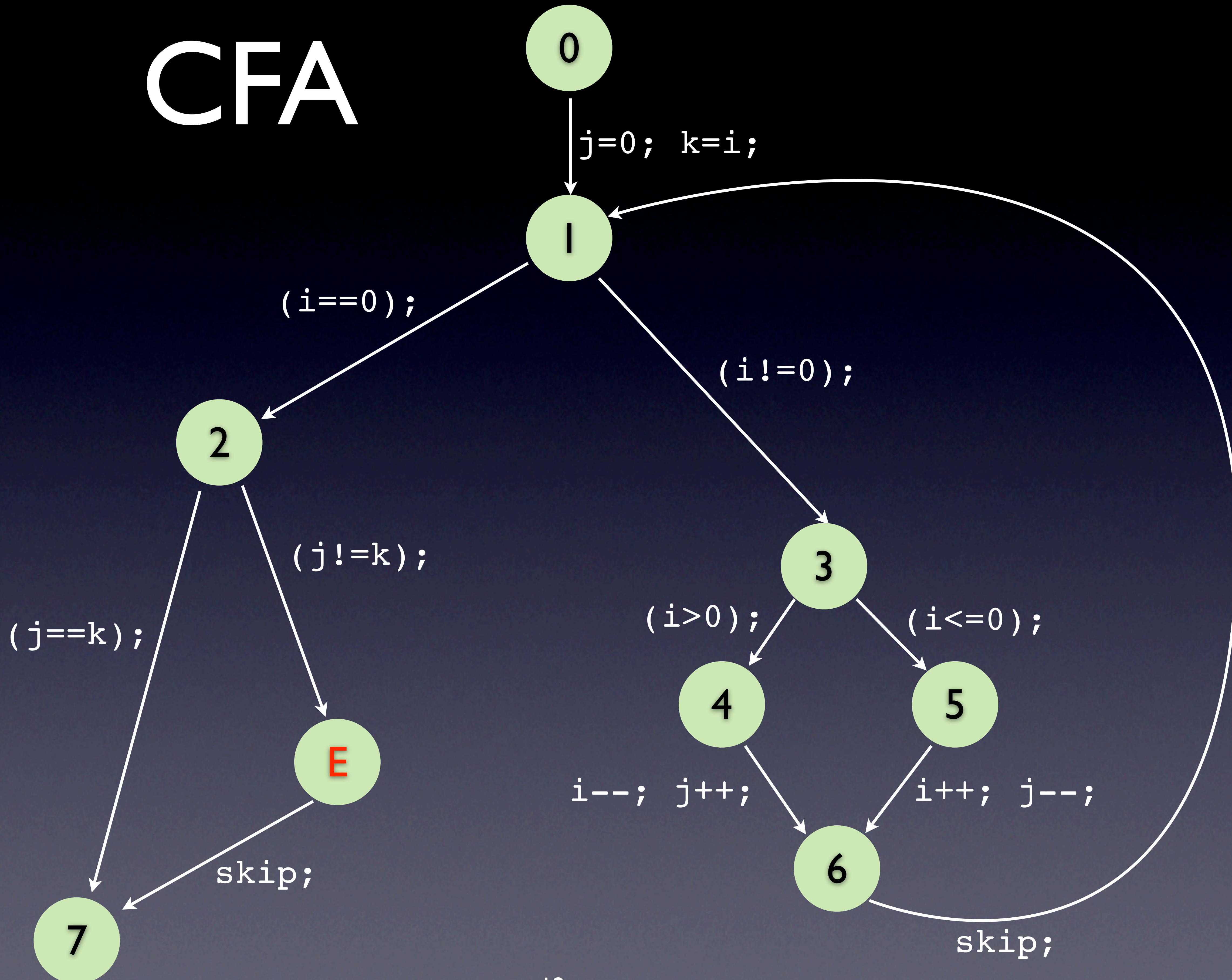
Operational overview



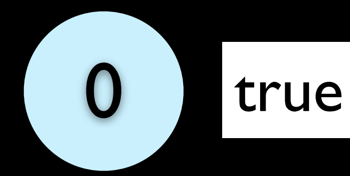
A program

```
int main() {
  int i;
  int j = 0;
  int k = i;
  while(i != 0) {
    if(i > 0){
      i--; j++;
    } else {
      i++; j--;
    }
  }
  if (j != k)
    ERROR: goto ERROR;
}
```

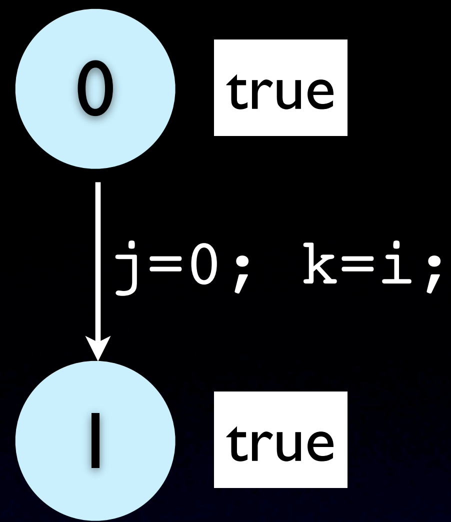
CFA



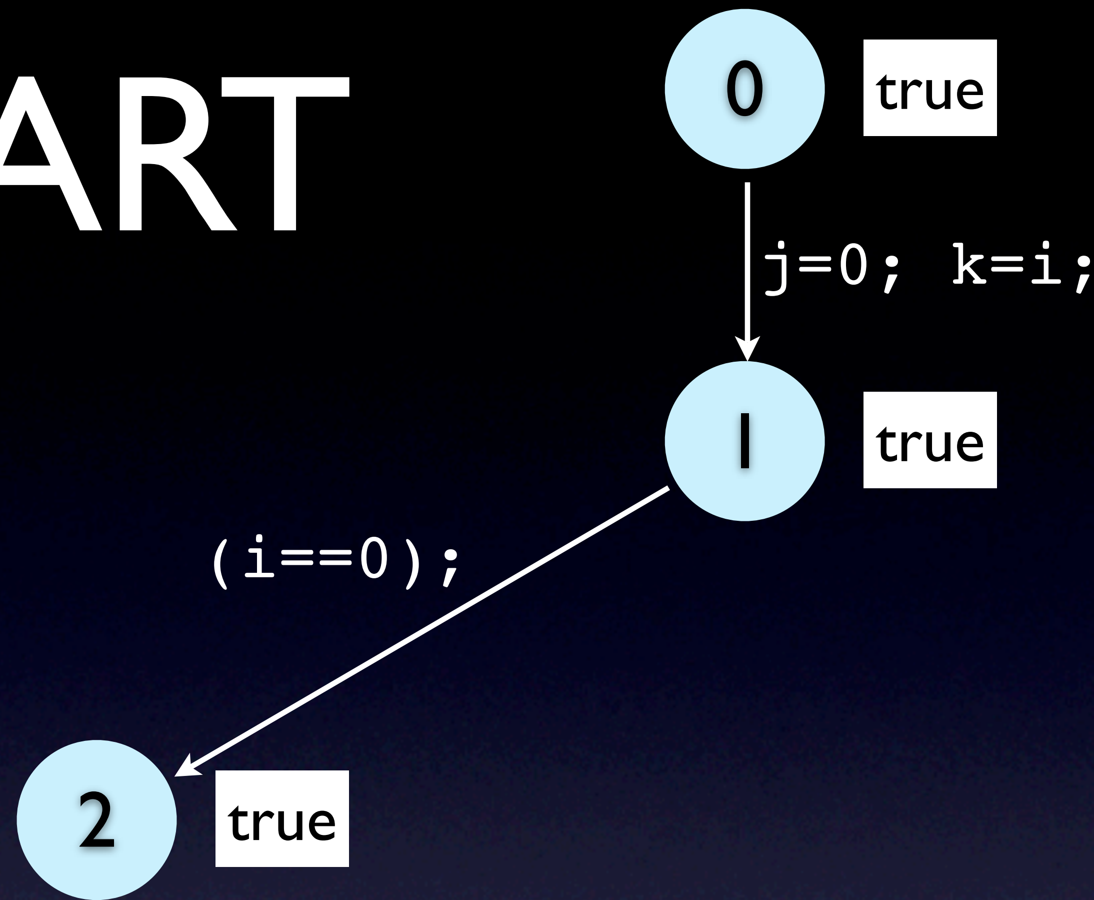
ART



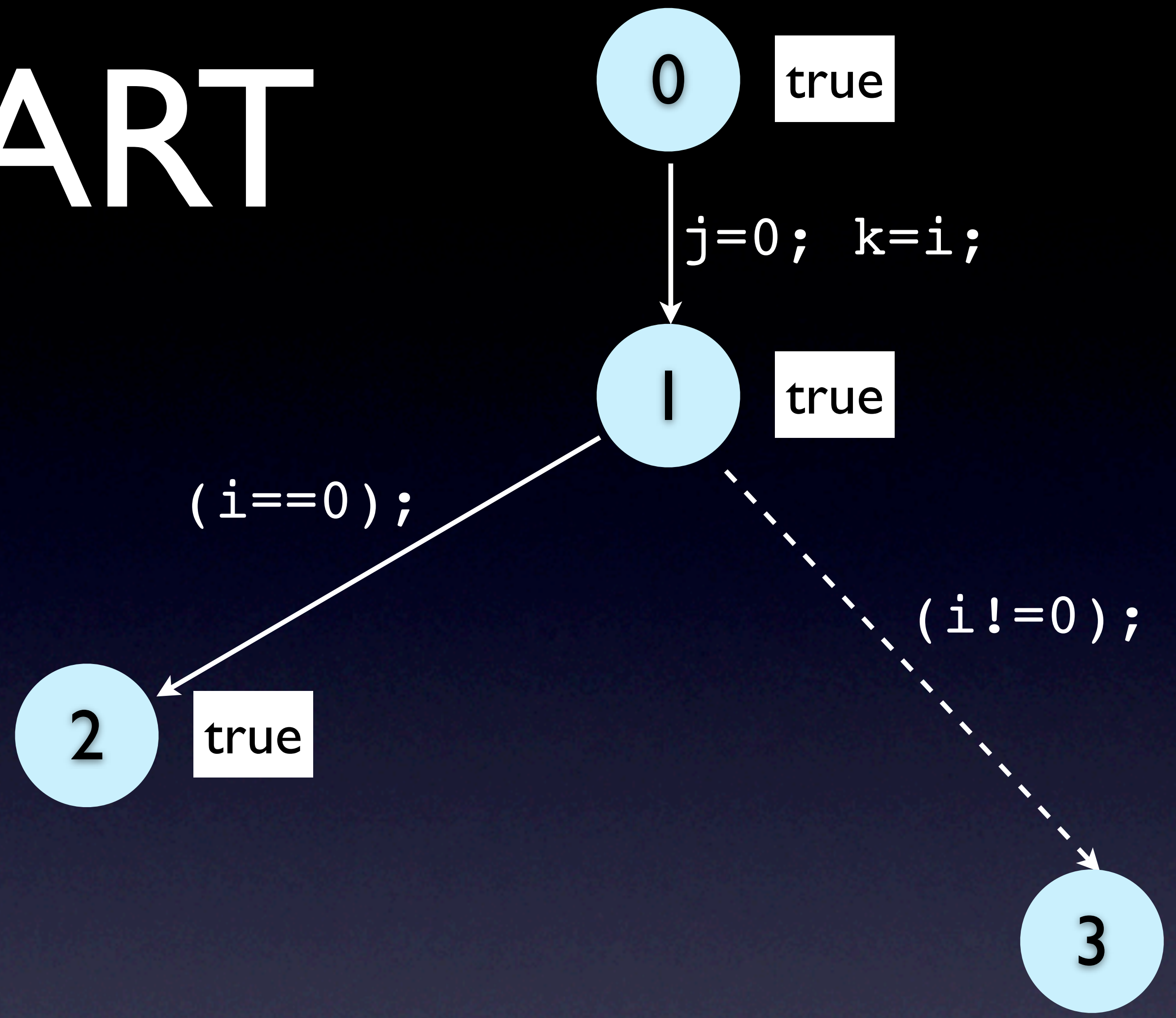
ART



ART

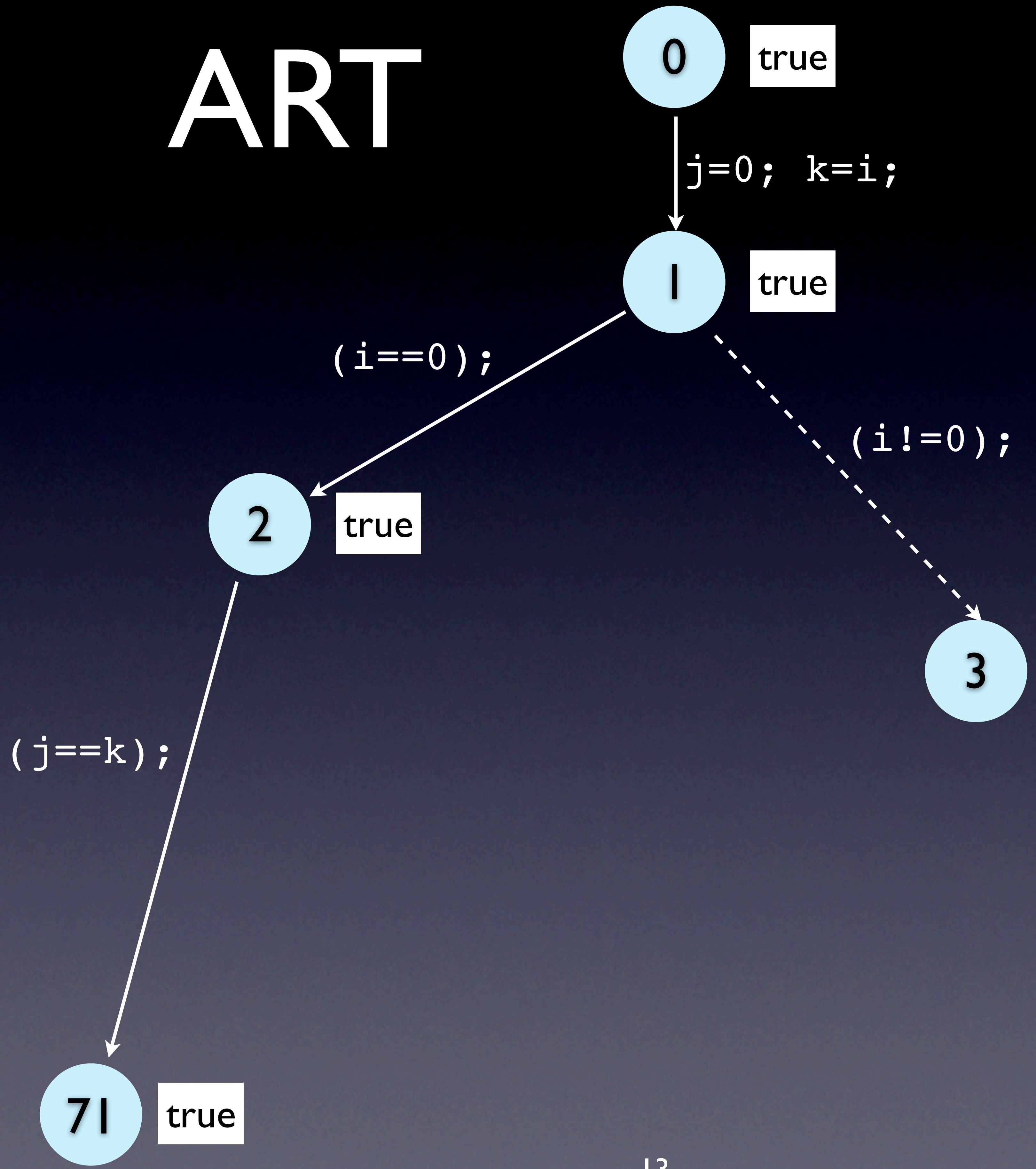


ART

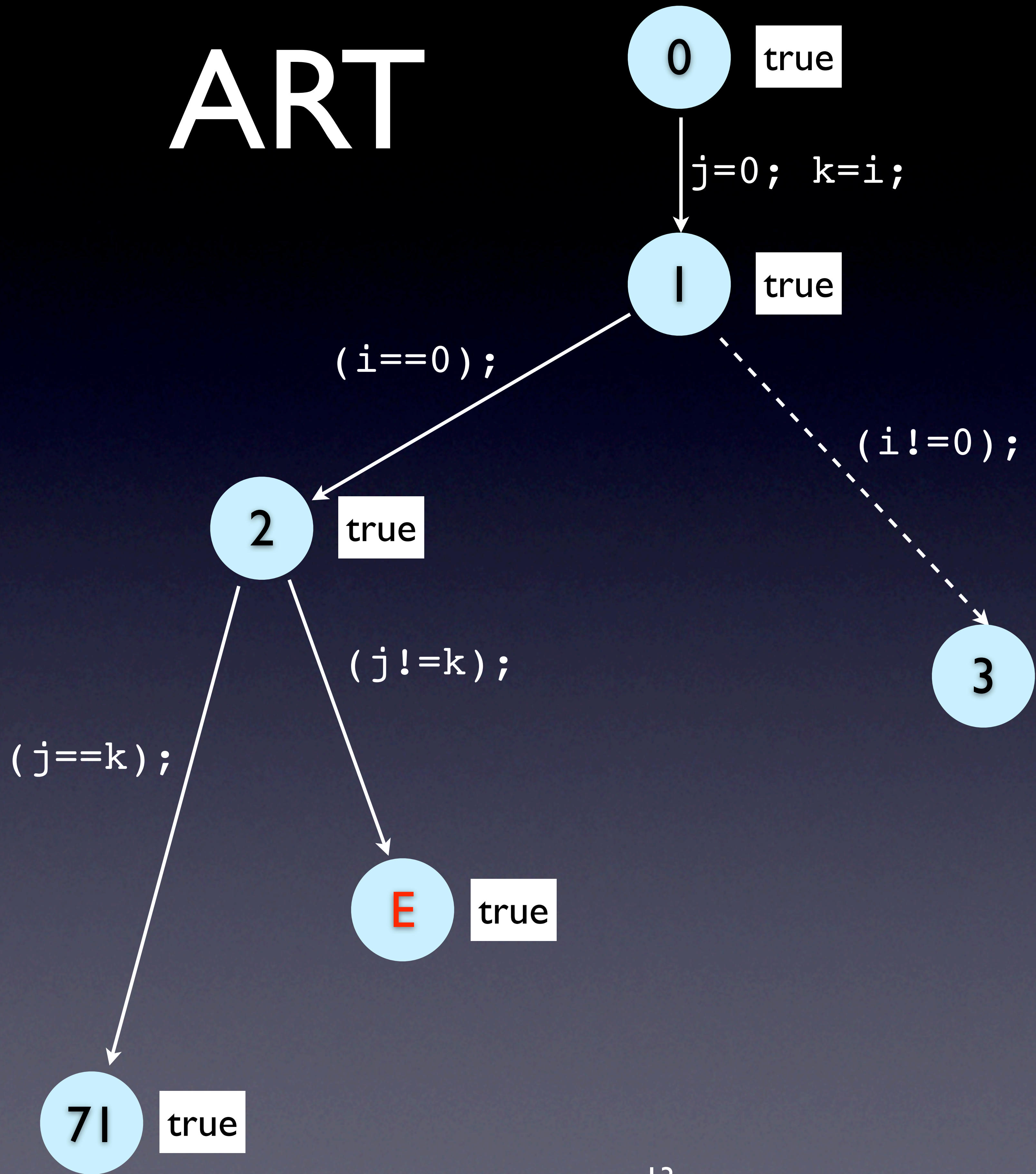


13

ART



ART

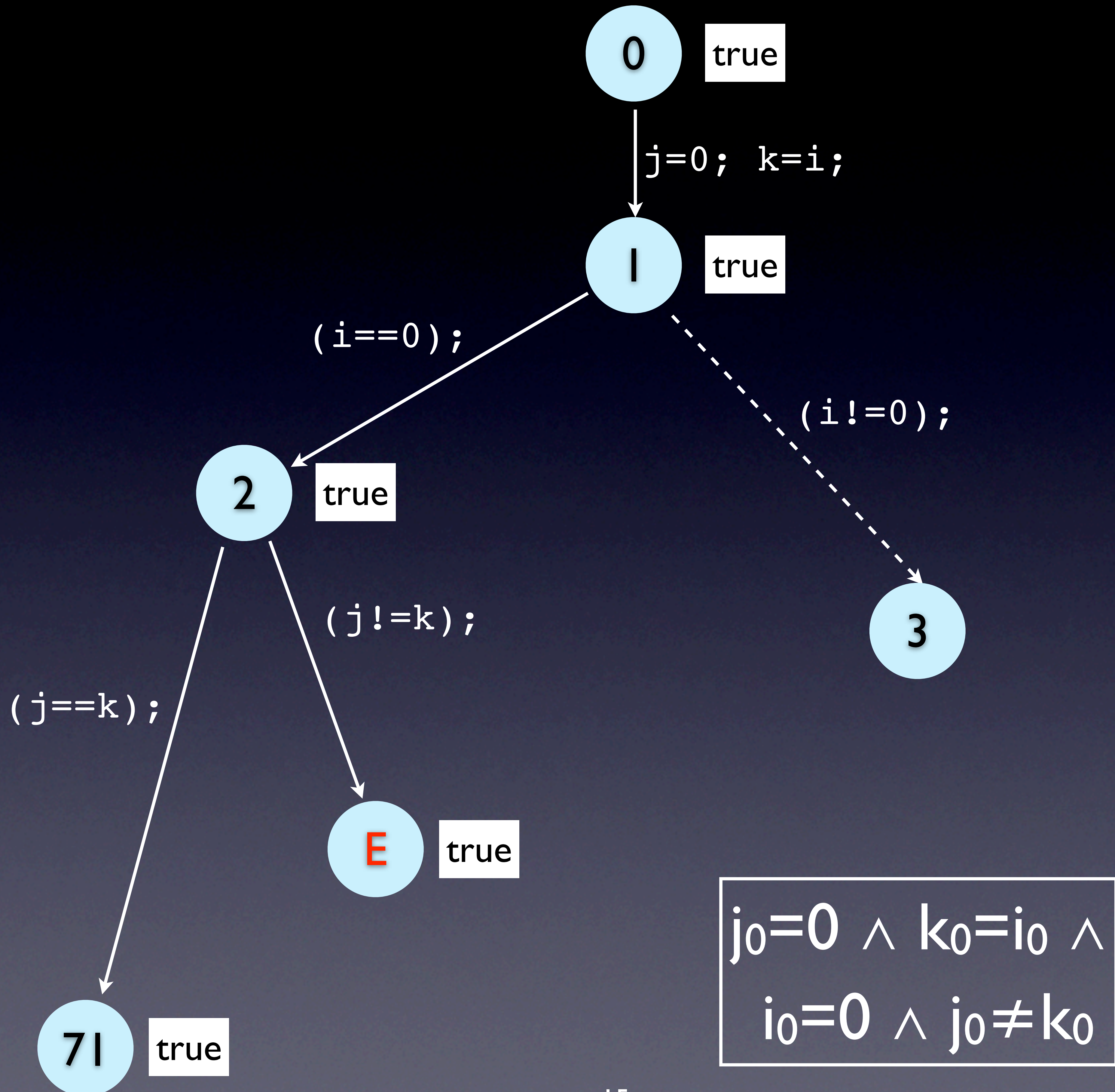


13

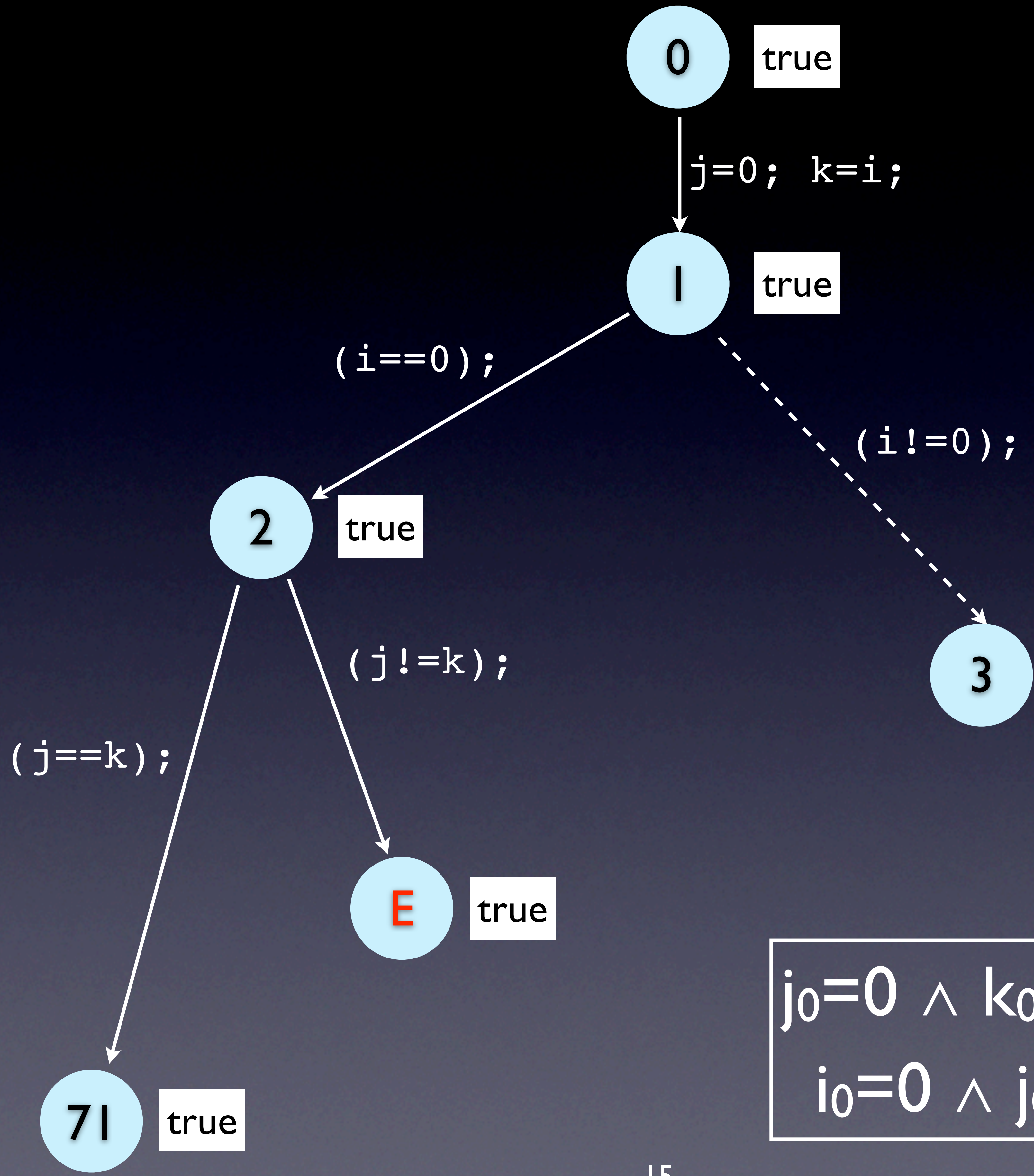
Path formula

- Transform path into SSA form
- Generate constraints for each operation along the path

14



$j_0=0 \wedge k_0=i_0 \wedge i_0=0 \wedge j_0 \neq k_0$



$$j_0=0 \wedge k_0=i_0 \wedge i_0=0 \wedge j_0 \neq k_0 = \text{false}$$

Craig Interpolants

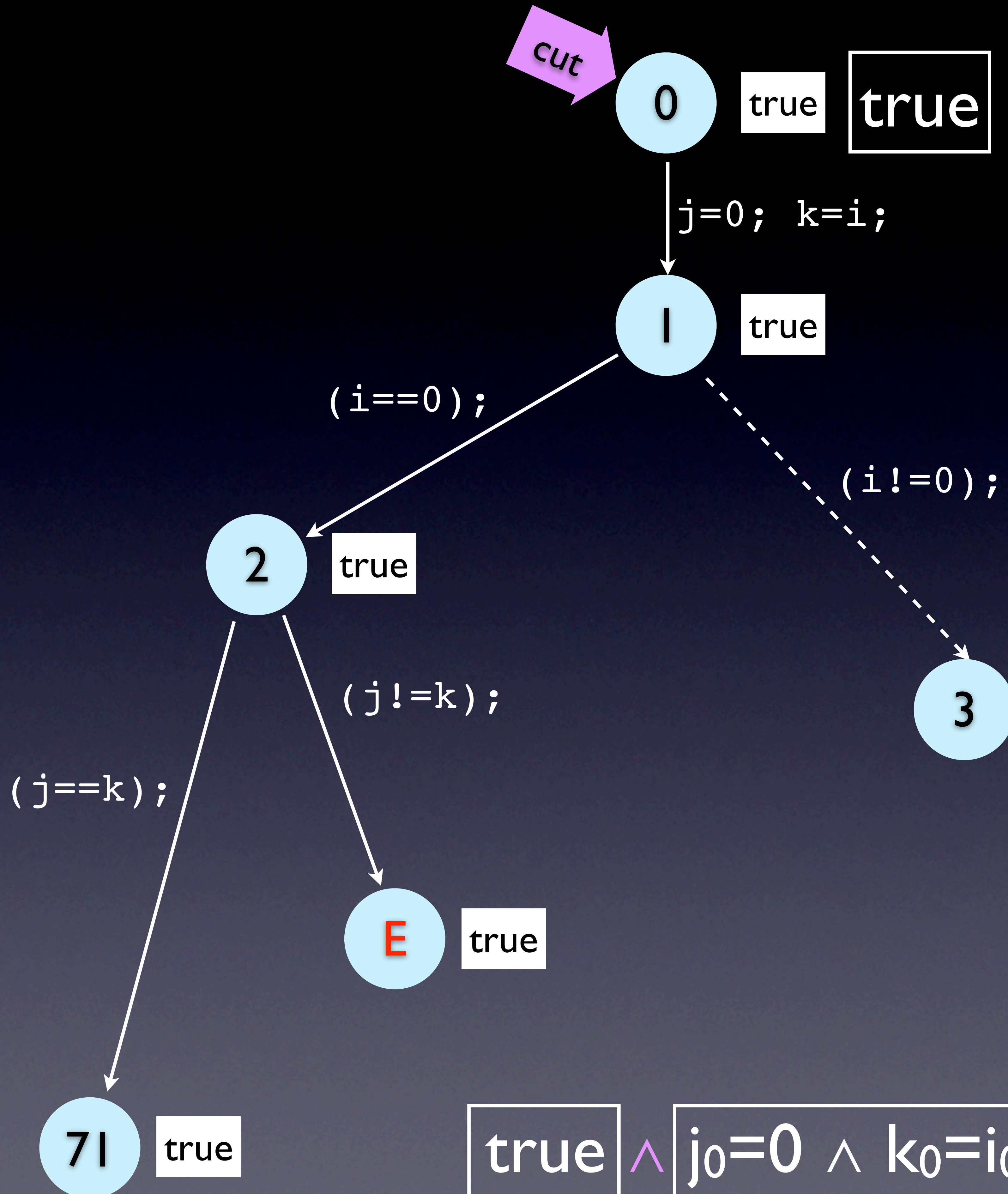
- For a formula $p_1 \wedge p_2$ that is unsatisfiable, a Craig interpolant q is a formula such that
  - $p_1 \Rightarrow q$ is valid
  - $q \wedge p_2$ is unsatisfiable
  - q contains only symbols common to both p_1 and p_2

[William Craig 1957]

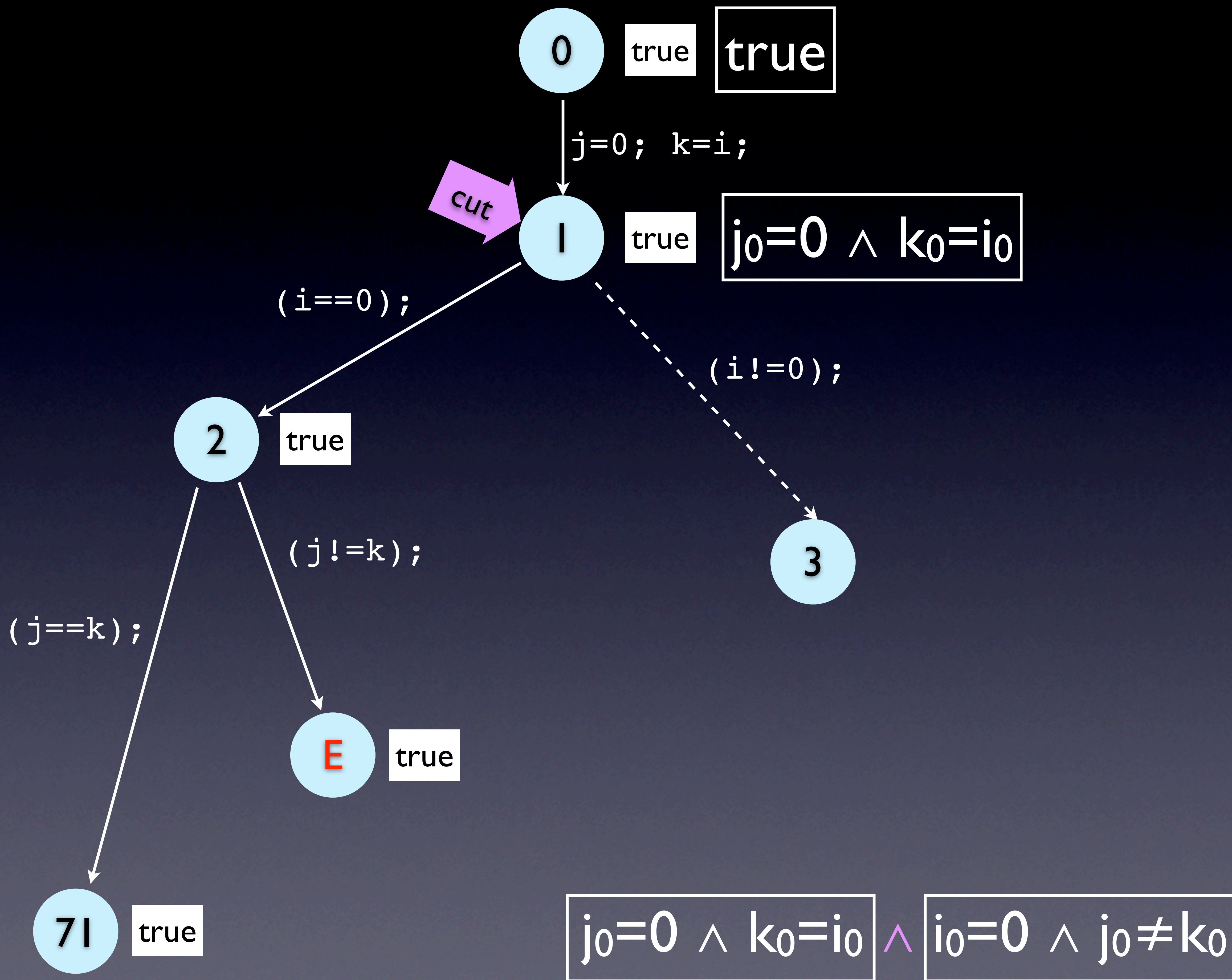
Craig Interpolants

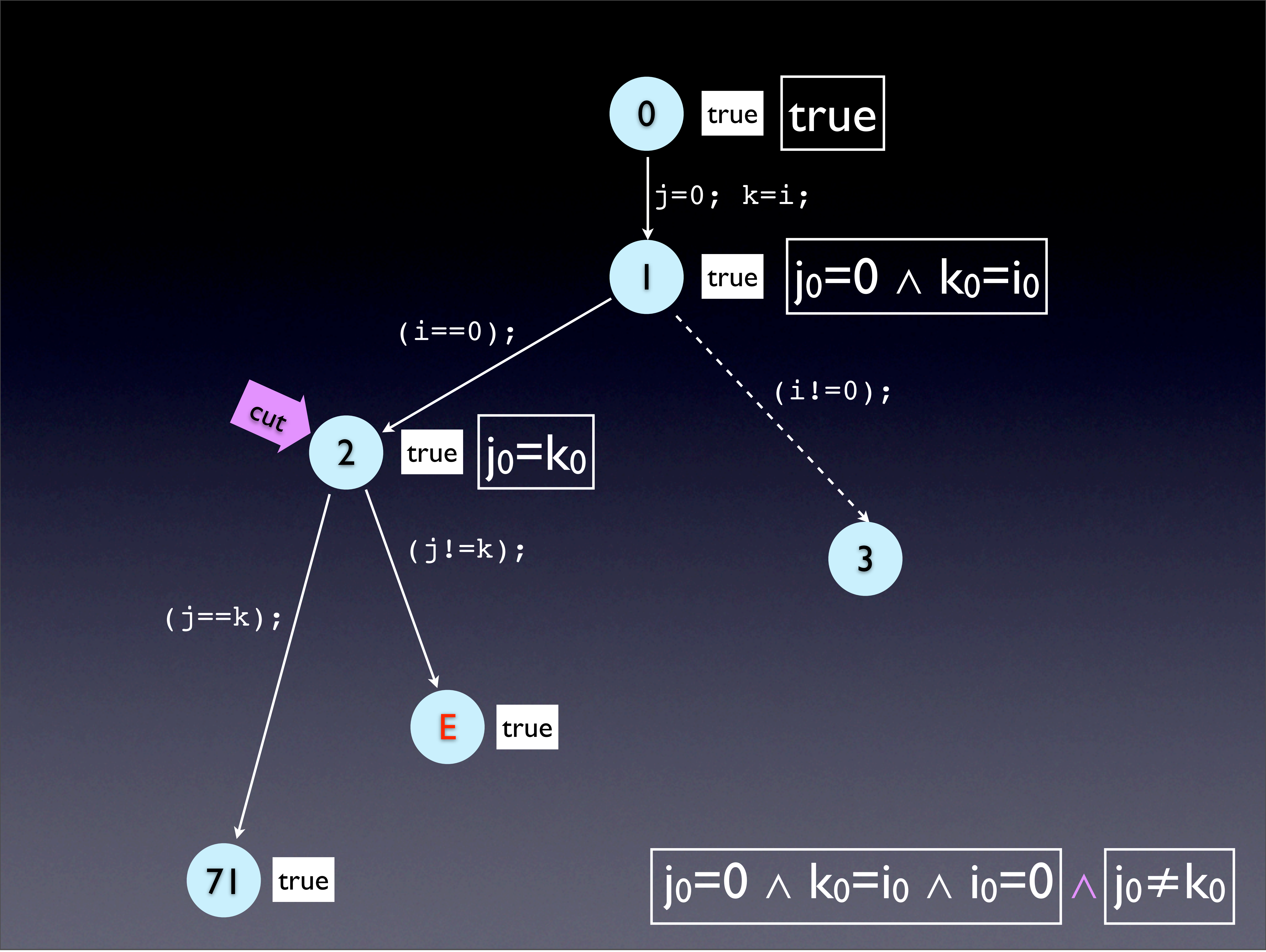
- For the theory of linear arithmetic with uninterpreted functions that is implemented in BLAST, these interpolants are guaranteed to exist

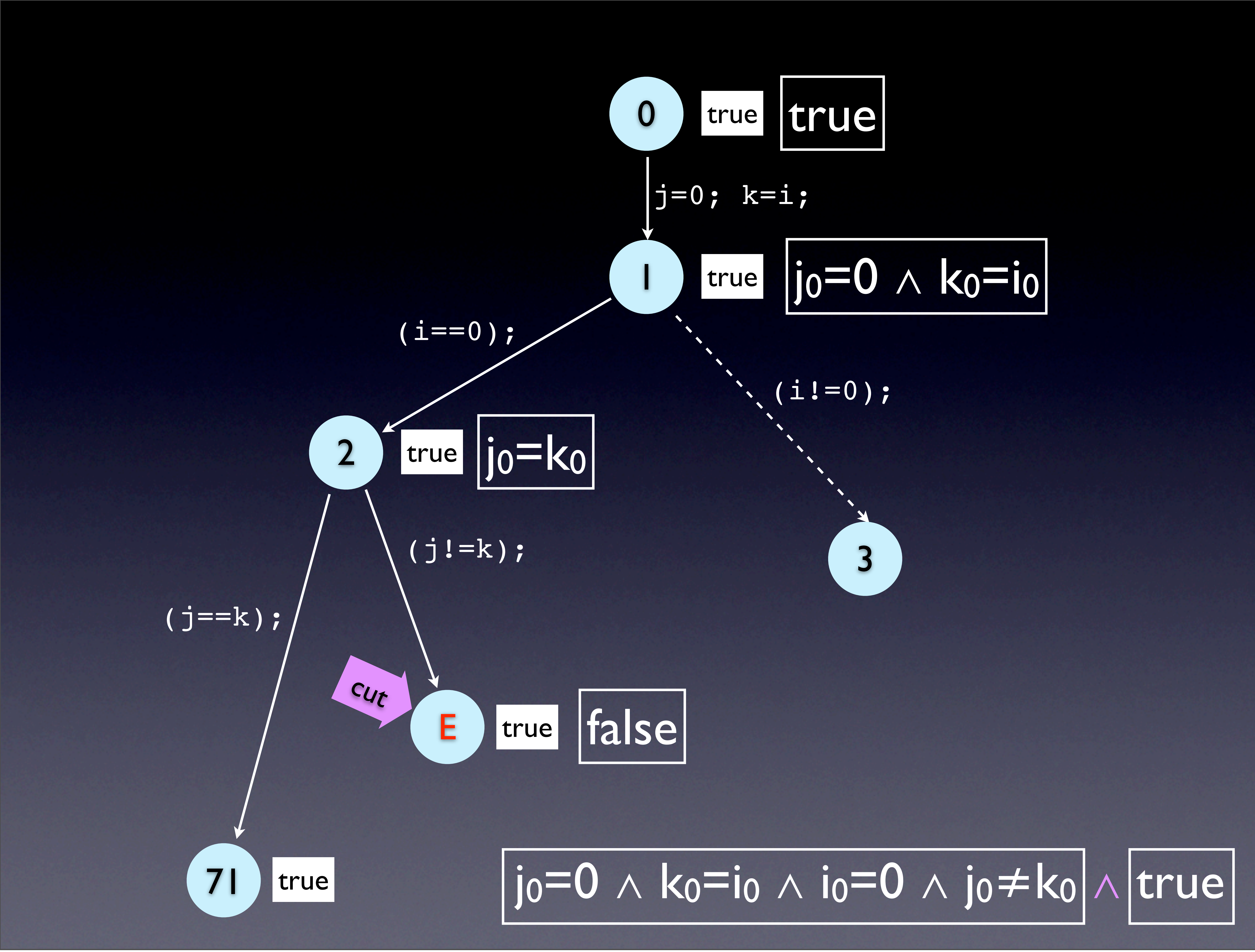
[McMillan 2005]

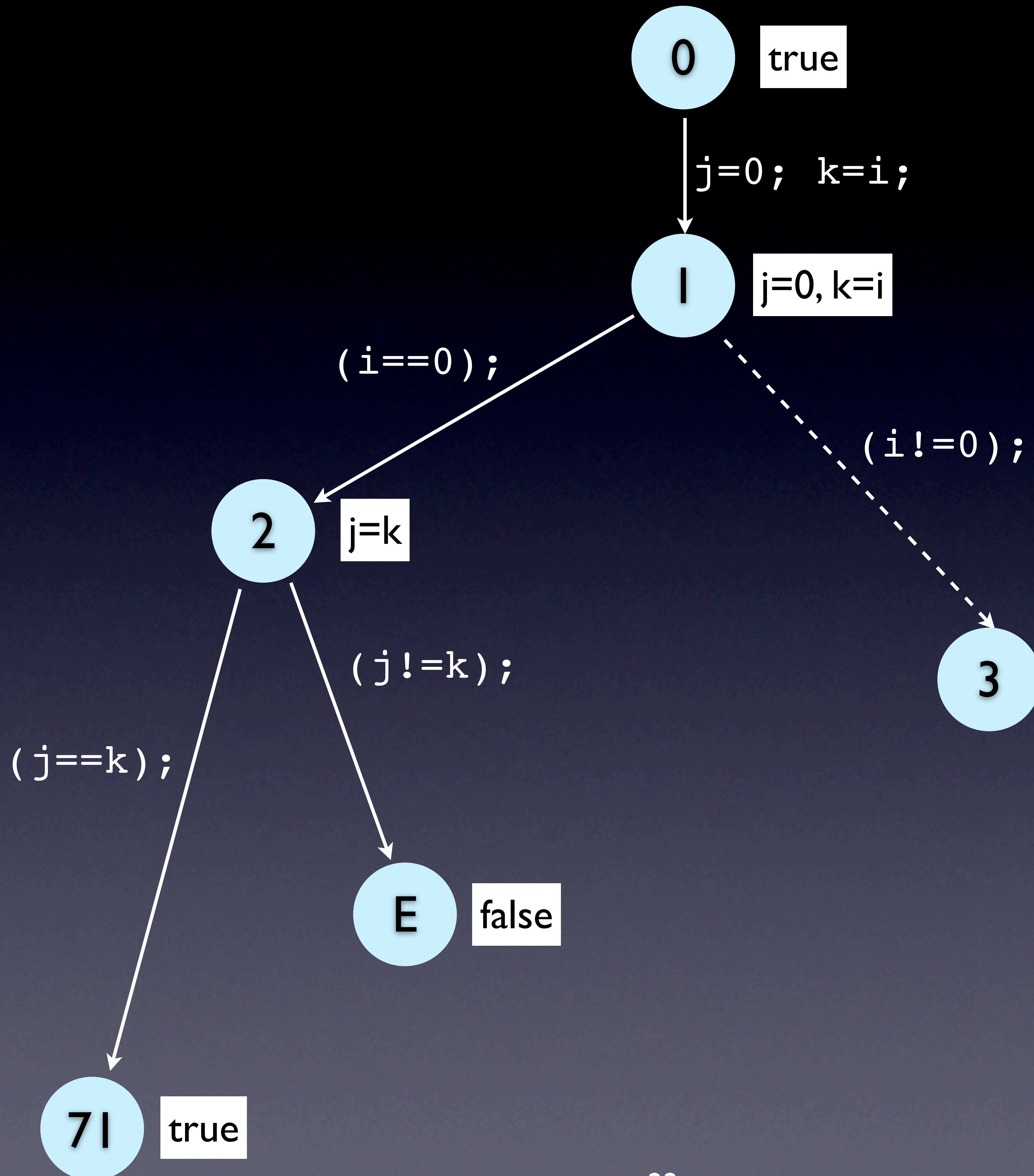


$$\text{true} \wedge j_0=0 \wedge k_0=i_0 \wedge i_0=0 \wedge j_0 \neq k_0$$


0
true
true
j=0; k=i;
cut
1
true
$j_0=0 \wedge k_0=i_0$
(i==0);
(i!=0);
2
true
3
(j!=k);
(j==k);
E
true
71
true
$j_0=0 \wedge k_0=i_0$
$\wedge$
$i_0=0 \wedge j_0 \neq k_0$


0
true
true
j=0; k=i;
1
true
$j_0=0 \wedge k_0=i_0$
(i==0);
(i!=0);
cut
2
true
$j_0=k_0$
3
(j!=k);
(j==k);
E
true
71
true
$j_0=0 \wedge k_0=i_0 \wedge i_0=0$
$\wedge$
$j_0 \neq k_0$


0
true
true
j=0; k=i;
1
true
$j_0=0 \wedge k_0=i_0$
(i==0);
(i!=0);
2
true
$j_0=k_0$
3
(j!=k);
(j==k);
cut
E
true
false
71
true
$j_0=0 \wedge k_0=i_0 \wedge i_0=0 \wedge j_0 \neq k_0 \wedge$ true



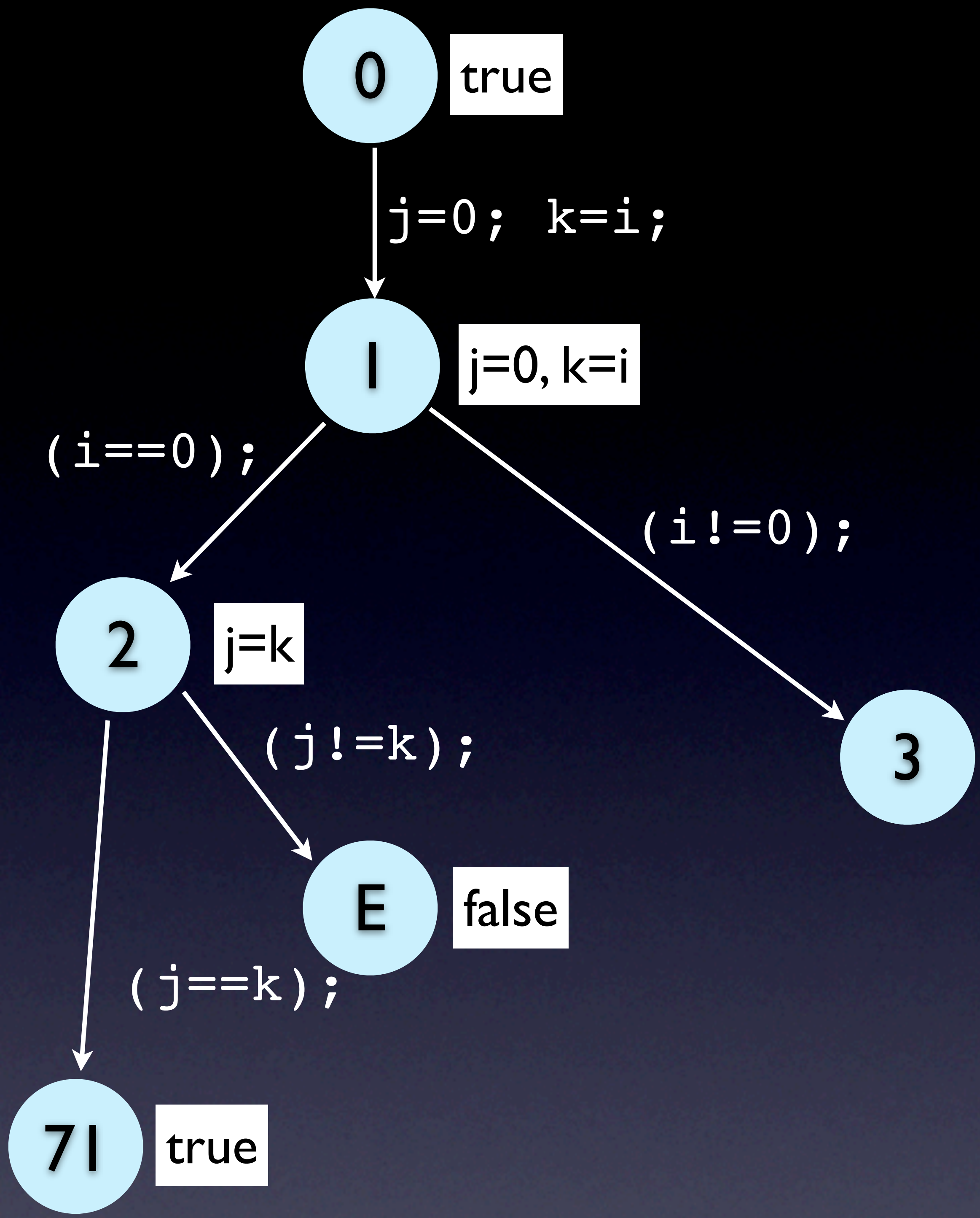
22

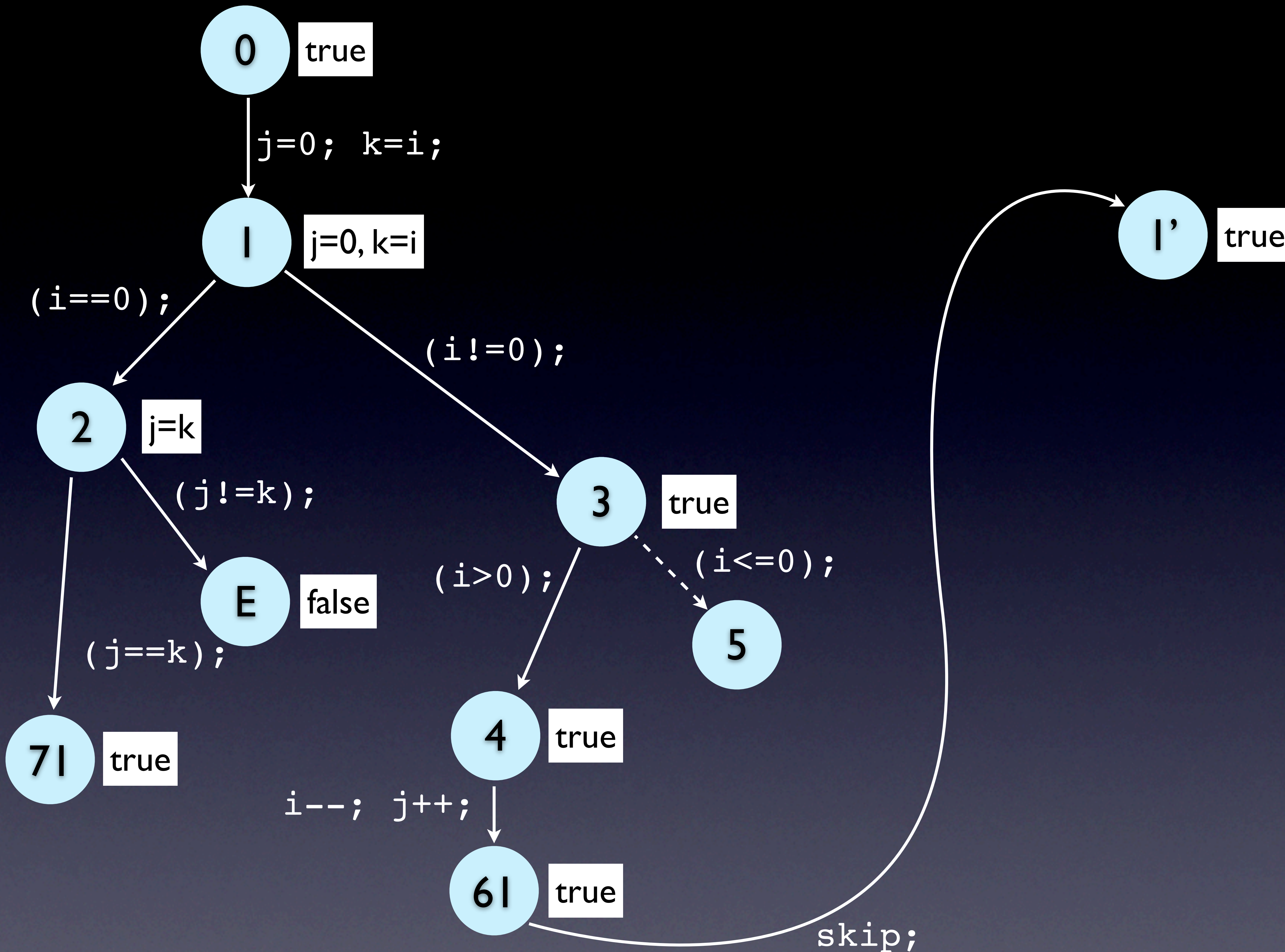
0 true

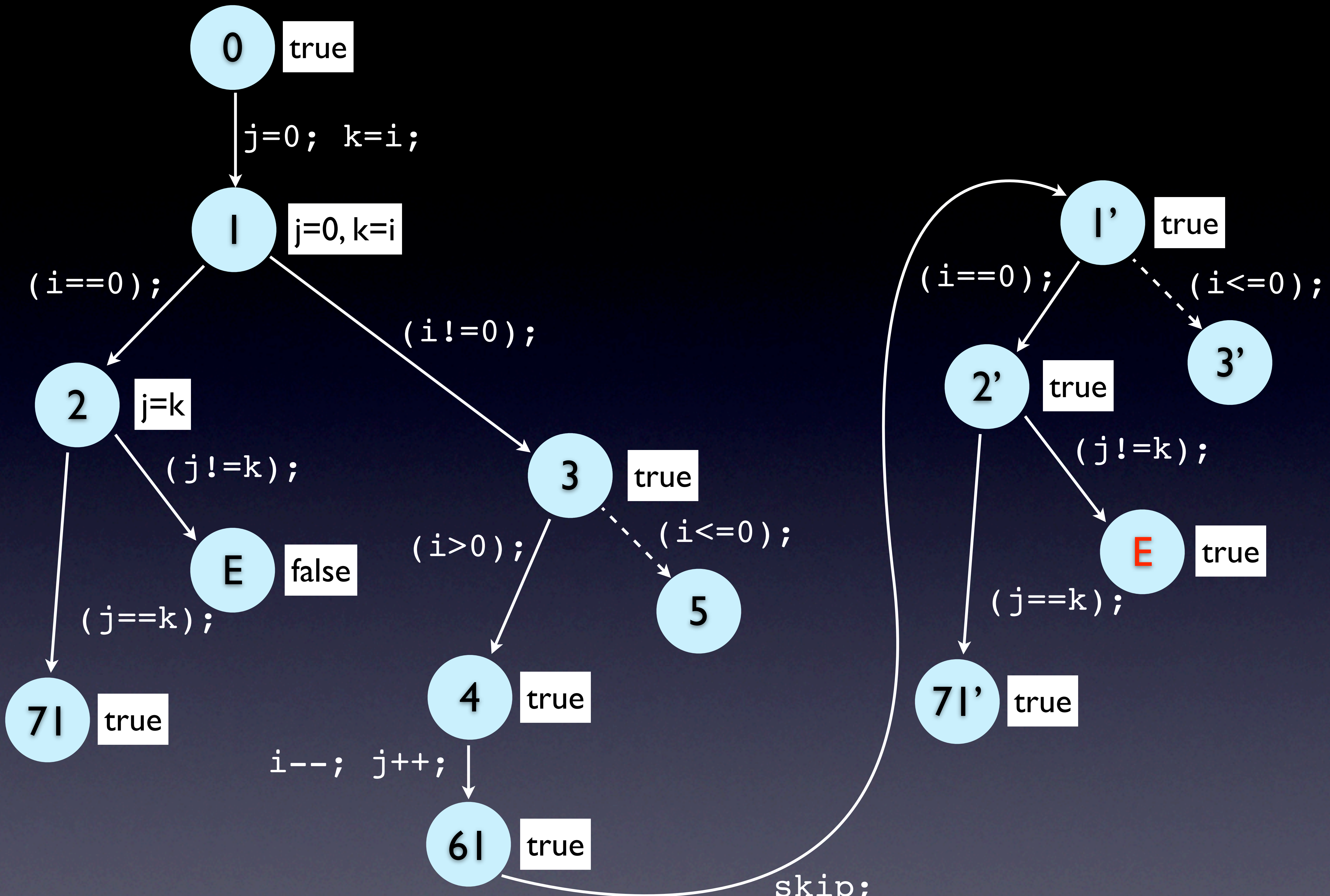
```
$ blast assigner.c
...
Conflicting Blocks
[INFO] 3 : 3:     Block(j@main = 0;k@main = i@main;)
[INFO] 4 : 5:     Pred(i@main  ==  0)
[INFO] 5 : 13:    Pred(j@main  !=  k@main)
...
[BAT] Calling refiner
addPred: 0: (gui) adding predicate i@main==k@main to the system
addPred: 0: (gui) adding predicate i@main==k@main to the system
addPred: 1: (gui) adding predicate j@main==0 to the system
addPred: 1: (gui) adding predicate j@main==0 to the system
addPred: 2: (gui) adding predicate j@main==k@main to the system
addPred: 2: (gui) adding predicate j@main==k@main to the system
Adding all preds now...
[BAT] Done refiner
...

$
```

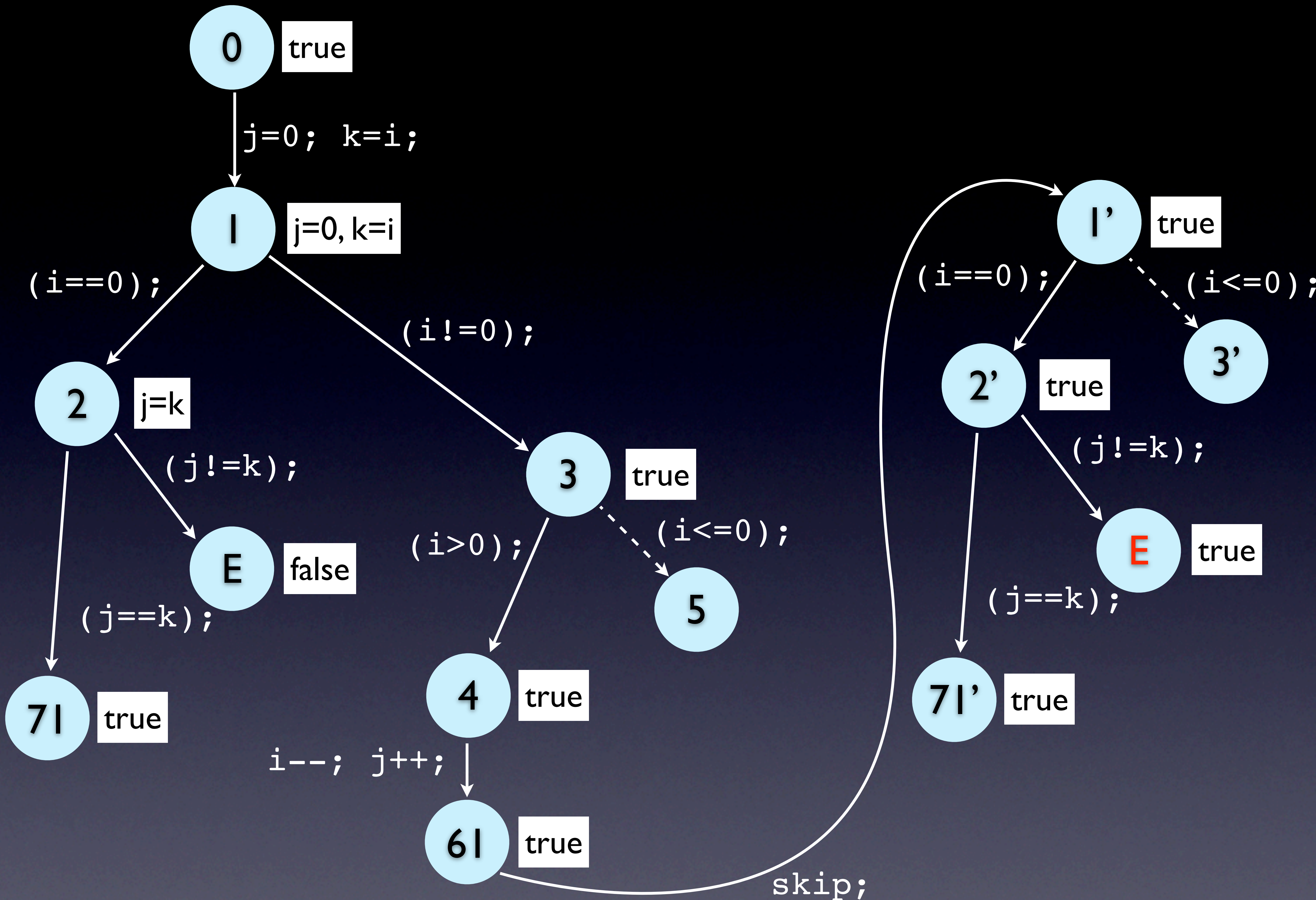
(j

71 true

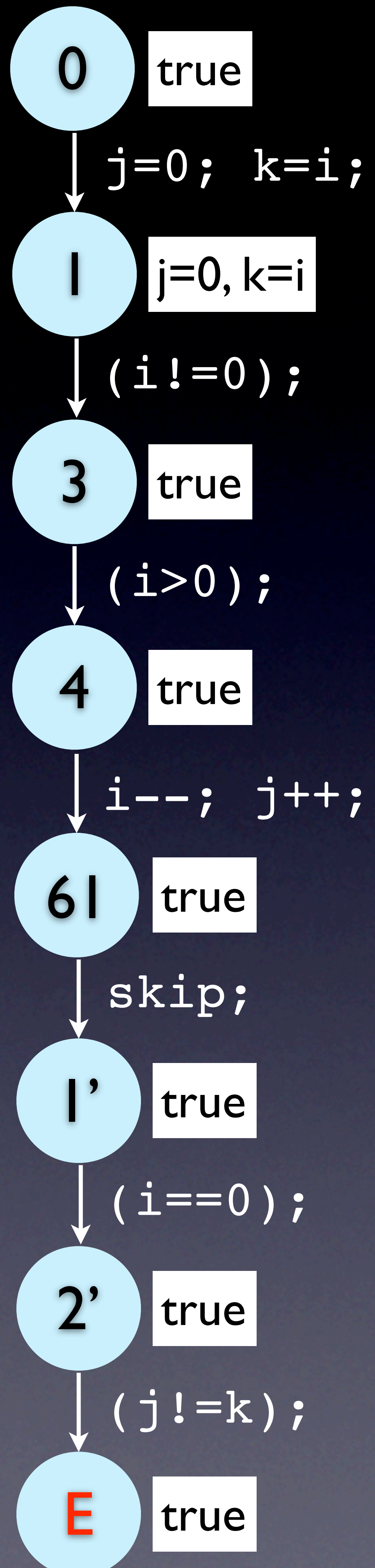

0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
3
E
false
(j==k);
71
true


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
true
(i>0);
(i<=0);
5
4
true
i--; j++;
61
true
skip;
1'
true

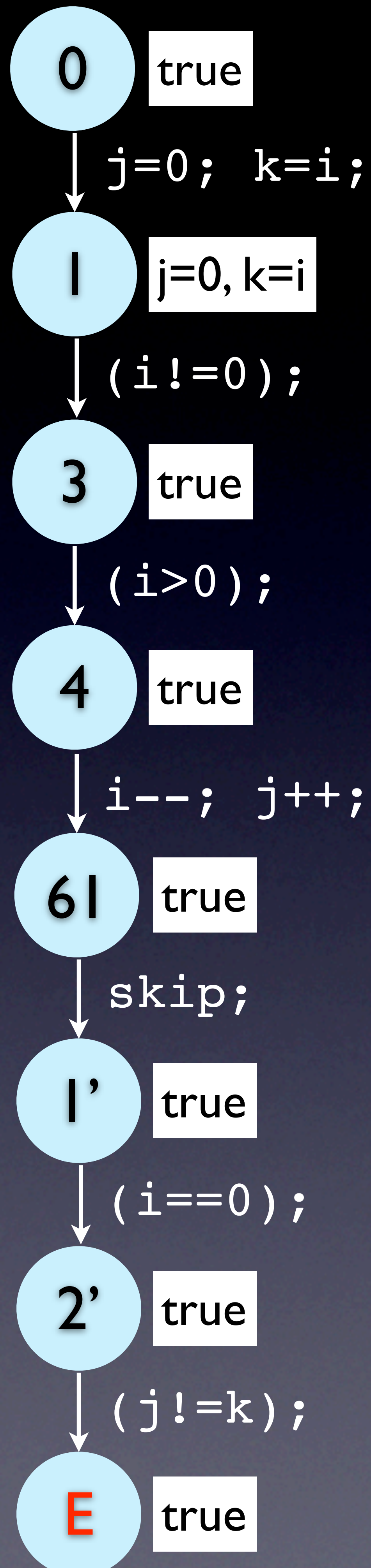

0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
true
(i>0);
(i<=0);
5
4
true
i--; j++;
61
true
skip;
1'
true
(i==0);
(i<=0);
3'
2'
true
(j!=k);
E
true
(j==k);
71'
true



$$j_0=0 \wedge k_0=i_0 \wedge i_0\neq 0 \wedge i_0>0 \wedge i_1=i_0-1 \wedge j_1=j_0+1 \wedge i_1=0 \wedge j_1\neq k_0$$

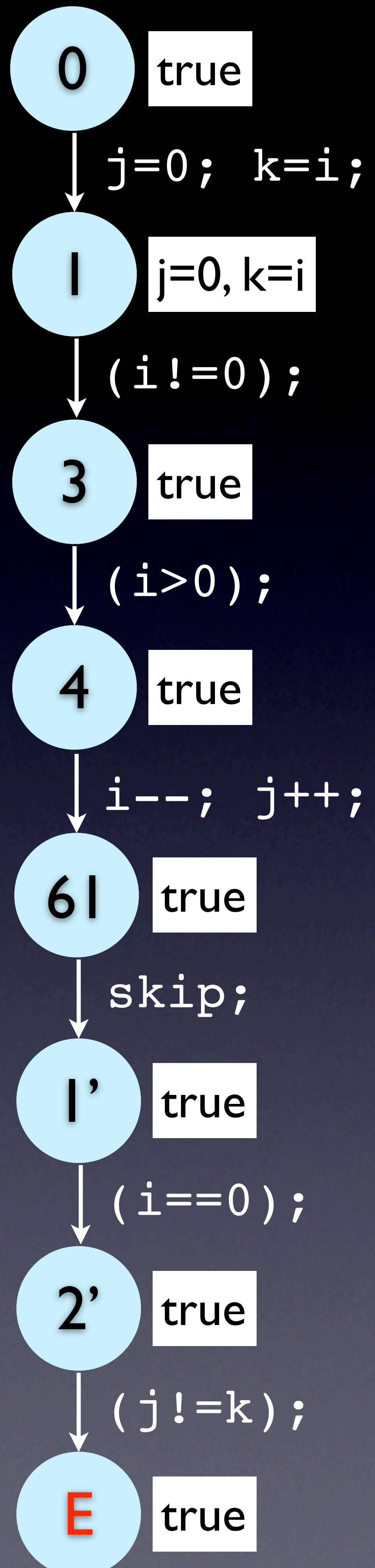


$$j_0=0 \wedge k_0=i_0 \wedge$$
$$i_0 \neq 0 \wedge i_0>0 \wedge$$
$$i_1=i_0-1 \wedge j_1=j_0+1 \wedge$$
$$i_1=0 \wedge j_1 \neq k_0$$



true

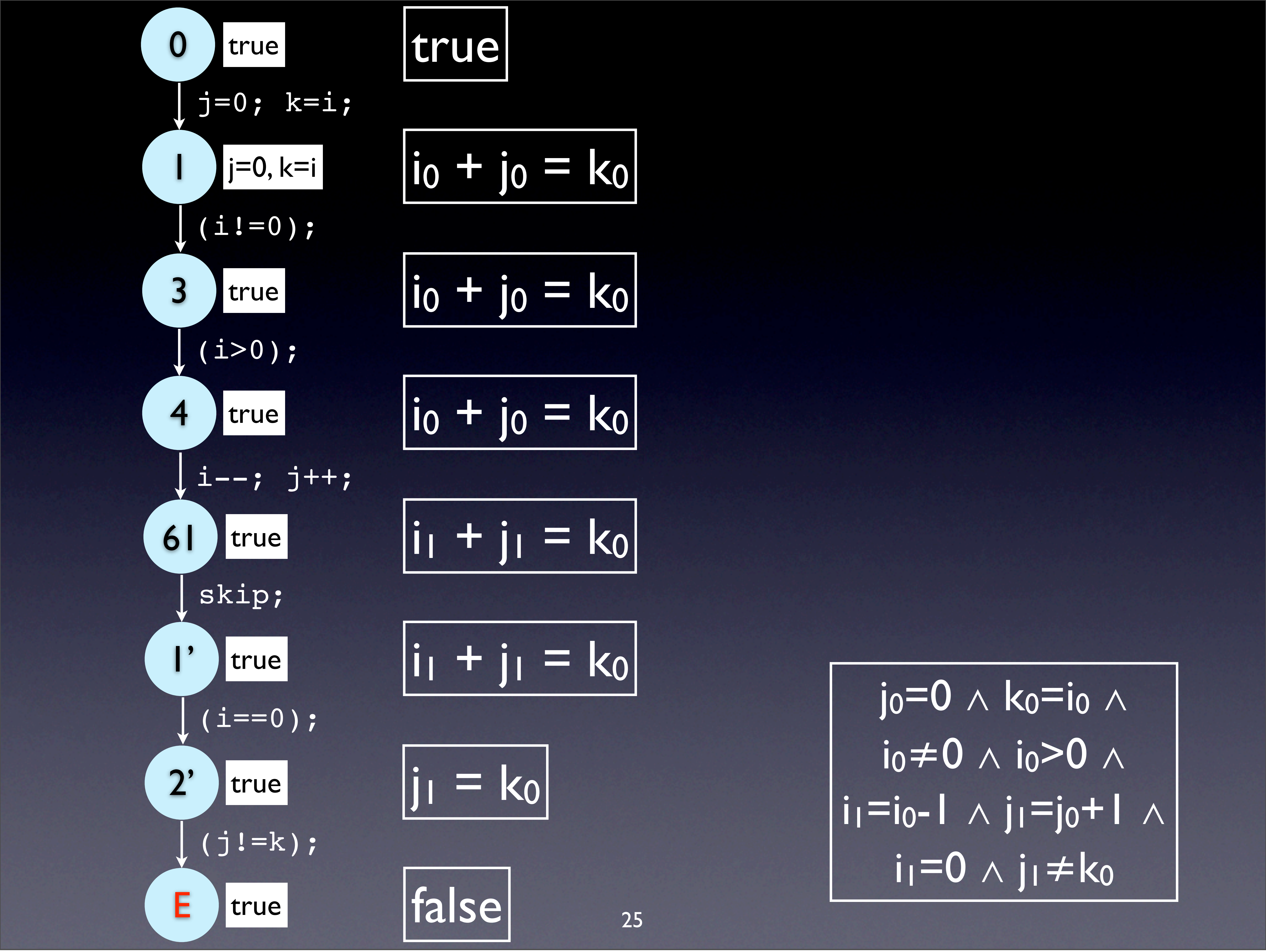
$$j_0=0 \wedge k_0=i_0 \wedge$$
$$i_0\neq 0 \wedge i_0>0 \wedge$$
$$i_1=i_0-1 \wedge j_1=j_0+1 \wedge$$
$$i_1=0 \wedge j_1\neq k_0$$

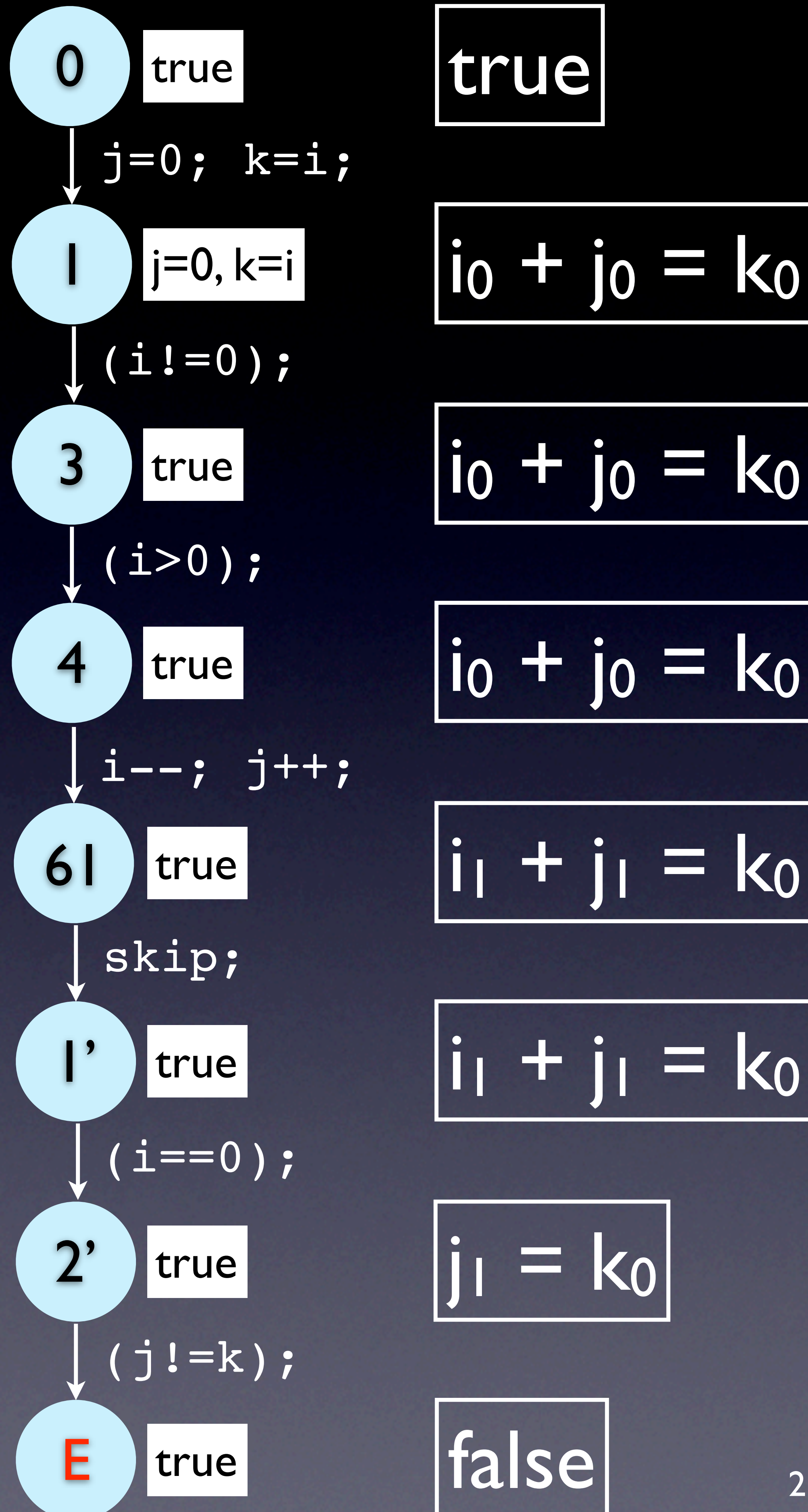


true

$i_0 + j_0 = k_0$

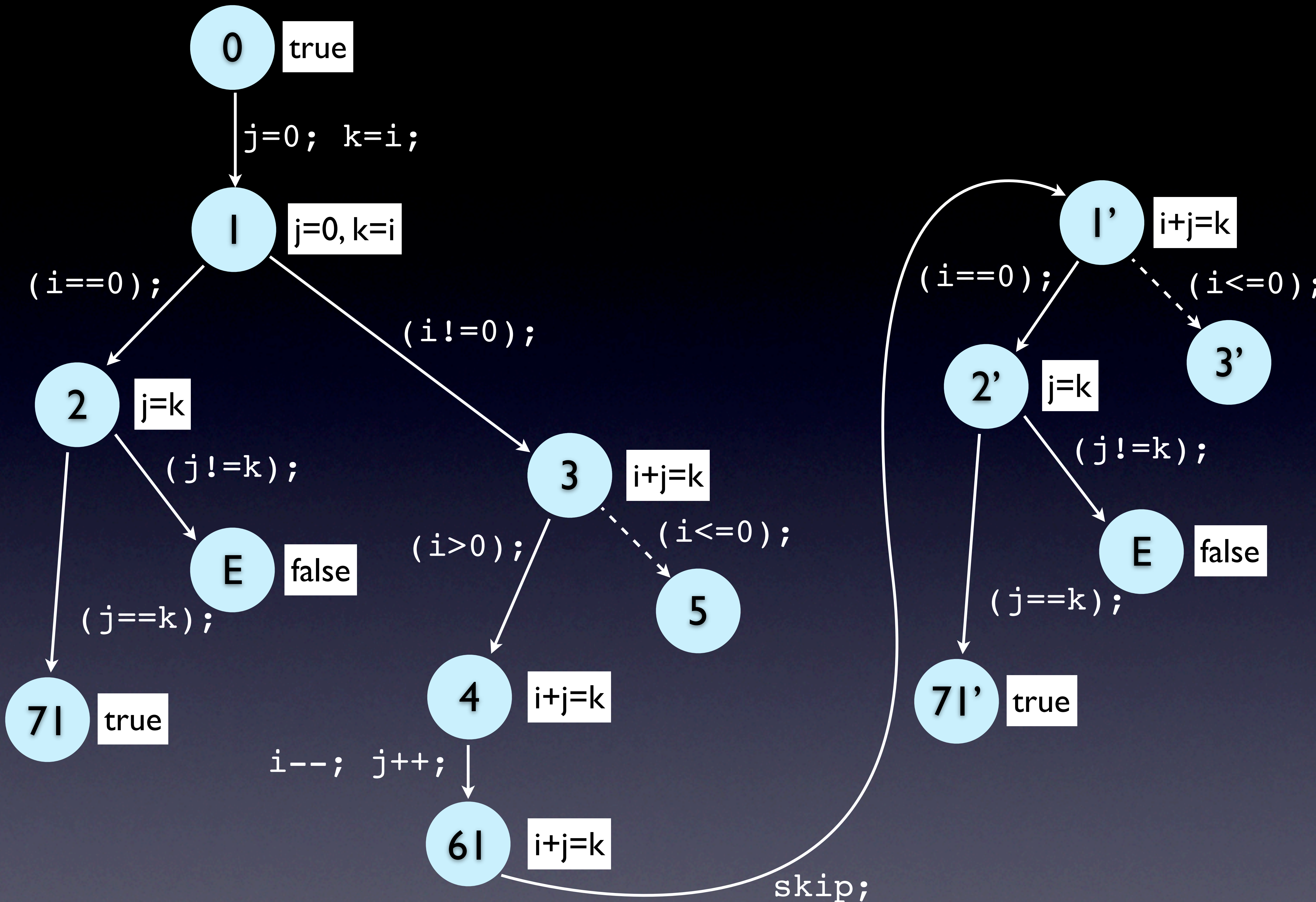
$$j_0=0 \wedge k_0=i_0 \wedge$$
$$i_0 \neq 0 \wedge i_0>0 \wedge$$
$$i_1=i_0-1 \wedge j_1=j_0+1 \wedge$$
$$i_1=0 \wedge j_1 \neq k_0$$

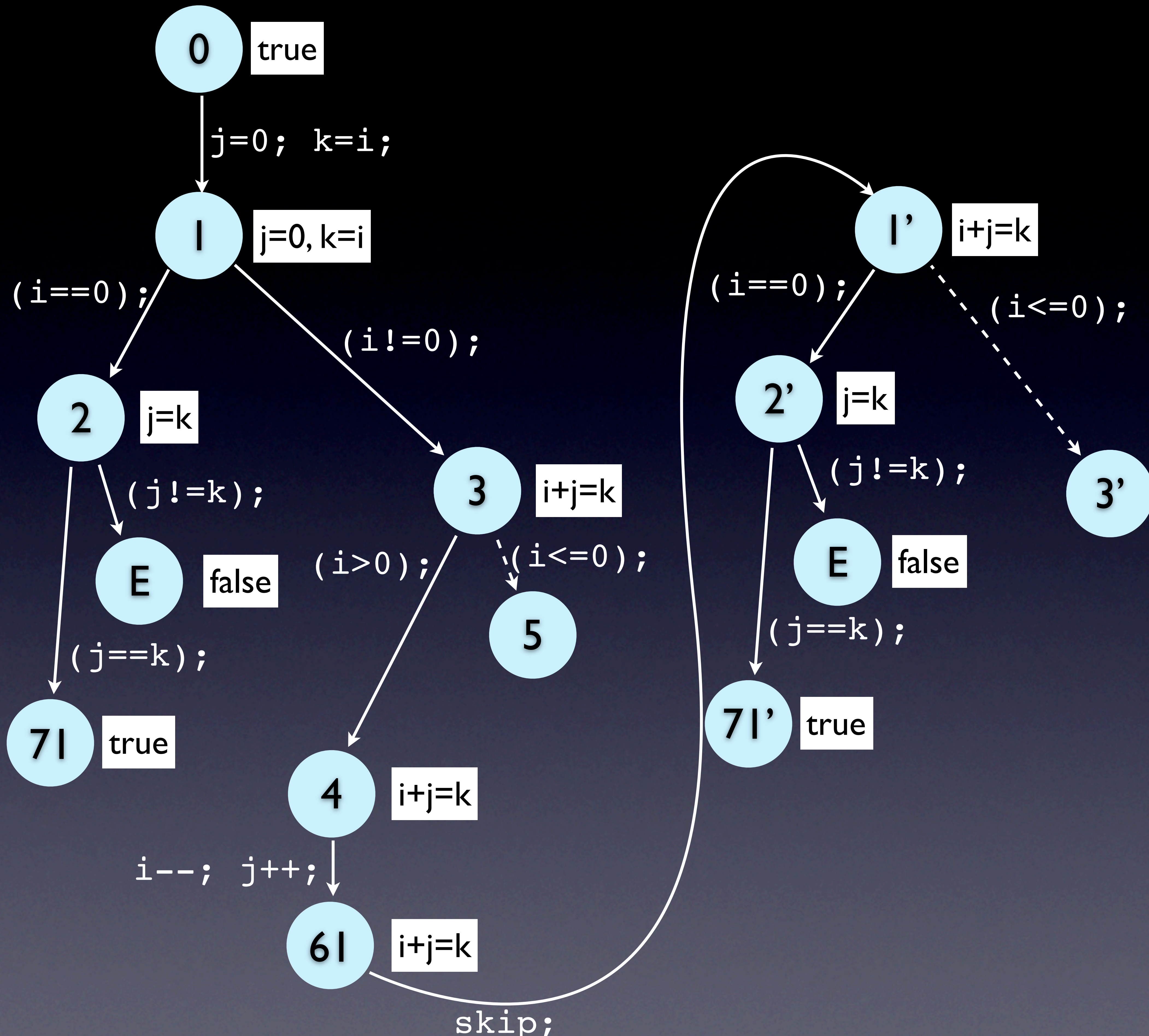

0
true
true
j=0; k=i;
1
j=0, k=i
$i_0 + j_0 = k_0$
(i!=0);
3
true
$i_0 + j_0 = k_0$
(i>0);
4
true
$i_0 + j_0 = k_0$
i--; j++;
61
true
$i_1 + j_1 = k_0$
skip;
1'
true
$i_1 + j_1 = k_0$
(i==0);
2'
true
$j_1 = k_0$
(j!=k);
E
true
false
25
$j_0=0 \wedge k_0=i_0 \wedge$
$i_0 \neq 0 \wedge i_0>0 \wedge$
$i_1=i_0-1 \wedge j_1=j_0+1 \wedge$
$i_1=0 \wedge j_1 \neq k_0$

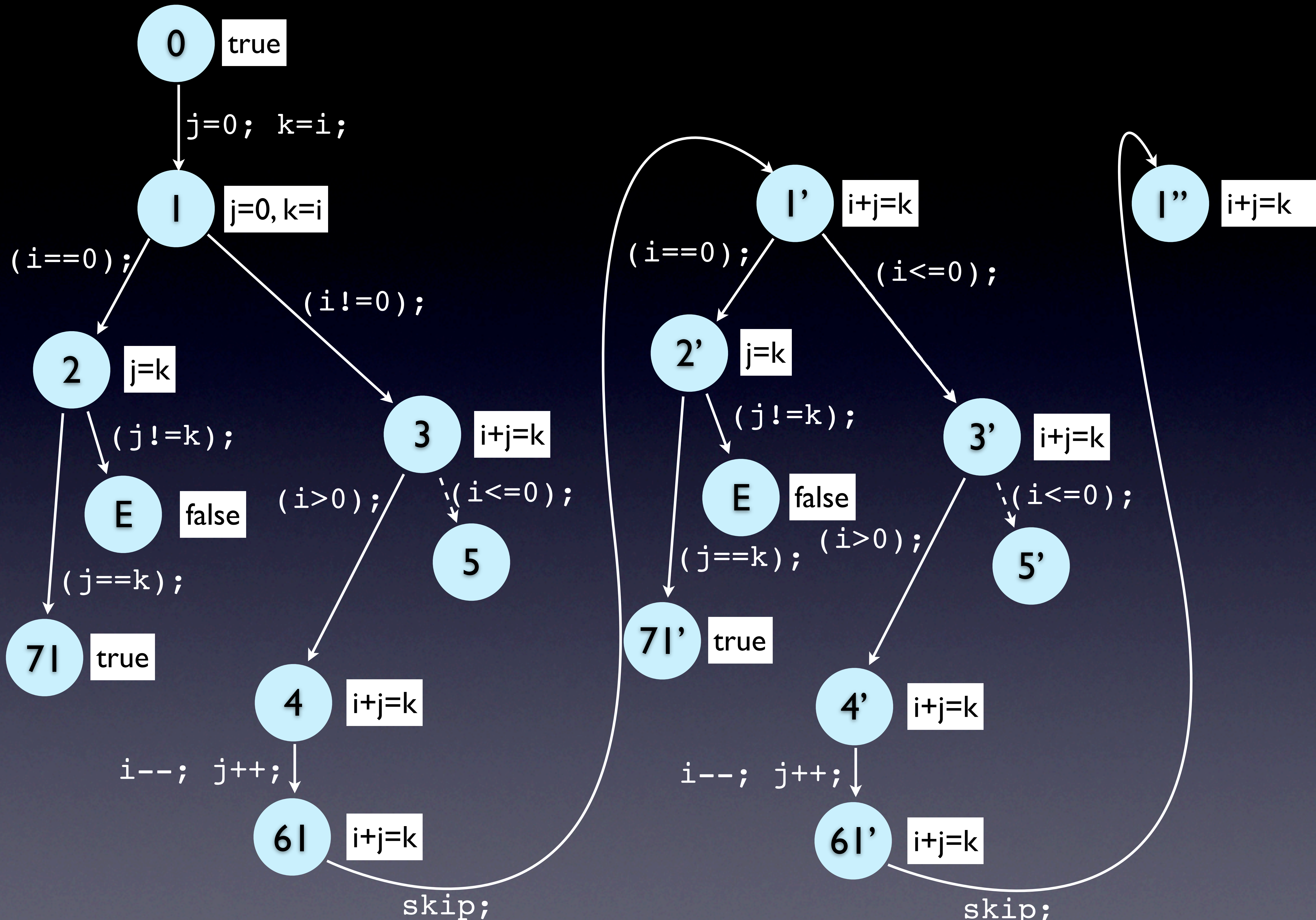


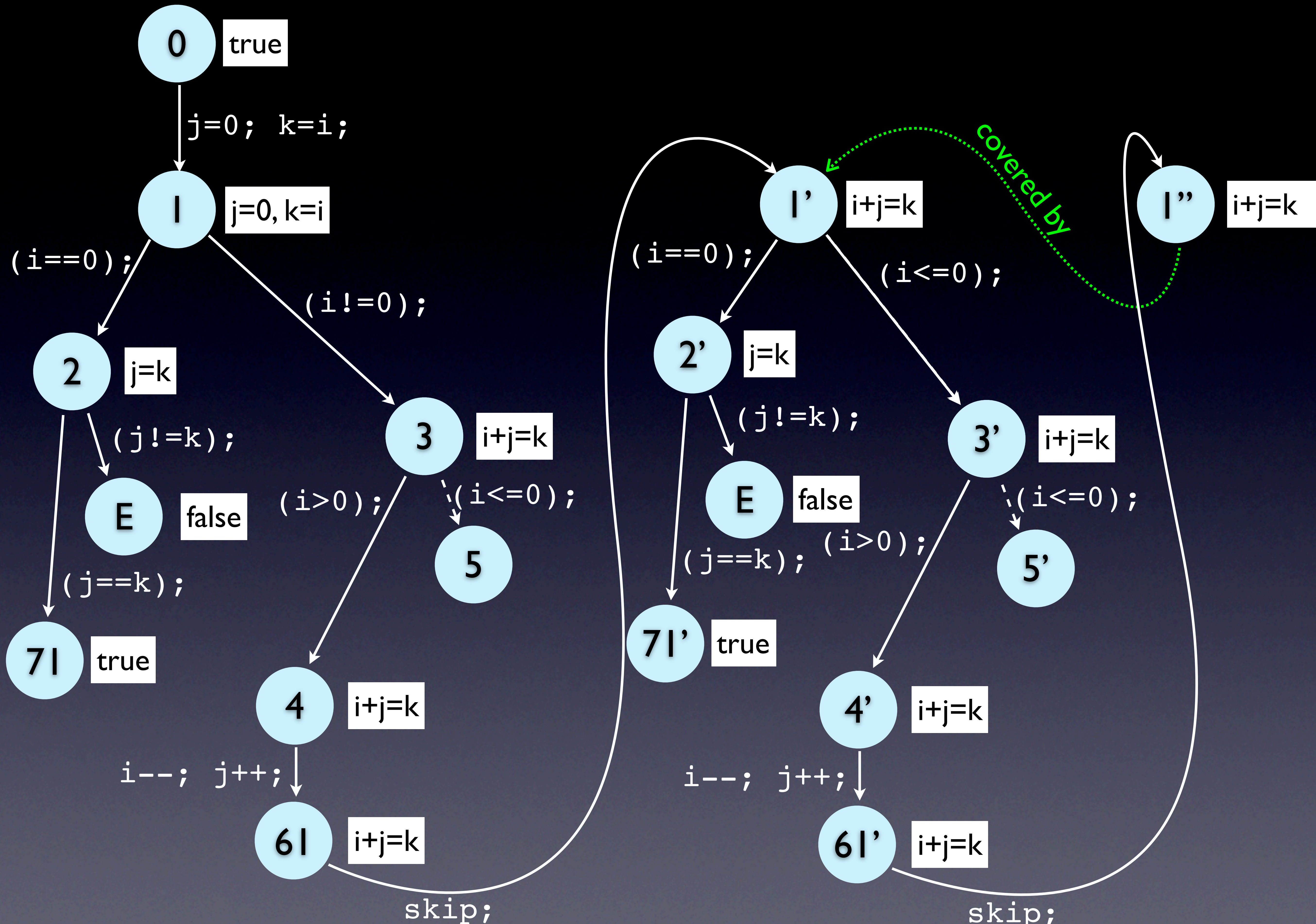

```
$ blast assigner.c
...
addPred: 5: (gui) adding predicate
k@main+-(j@main)+-(i@main)<=0 to the
system
addPred: 6: (gui) adding predicate
i@main+j@main+-(k@main)<=0 to the
system
...

$
```

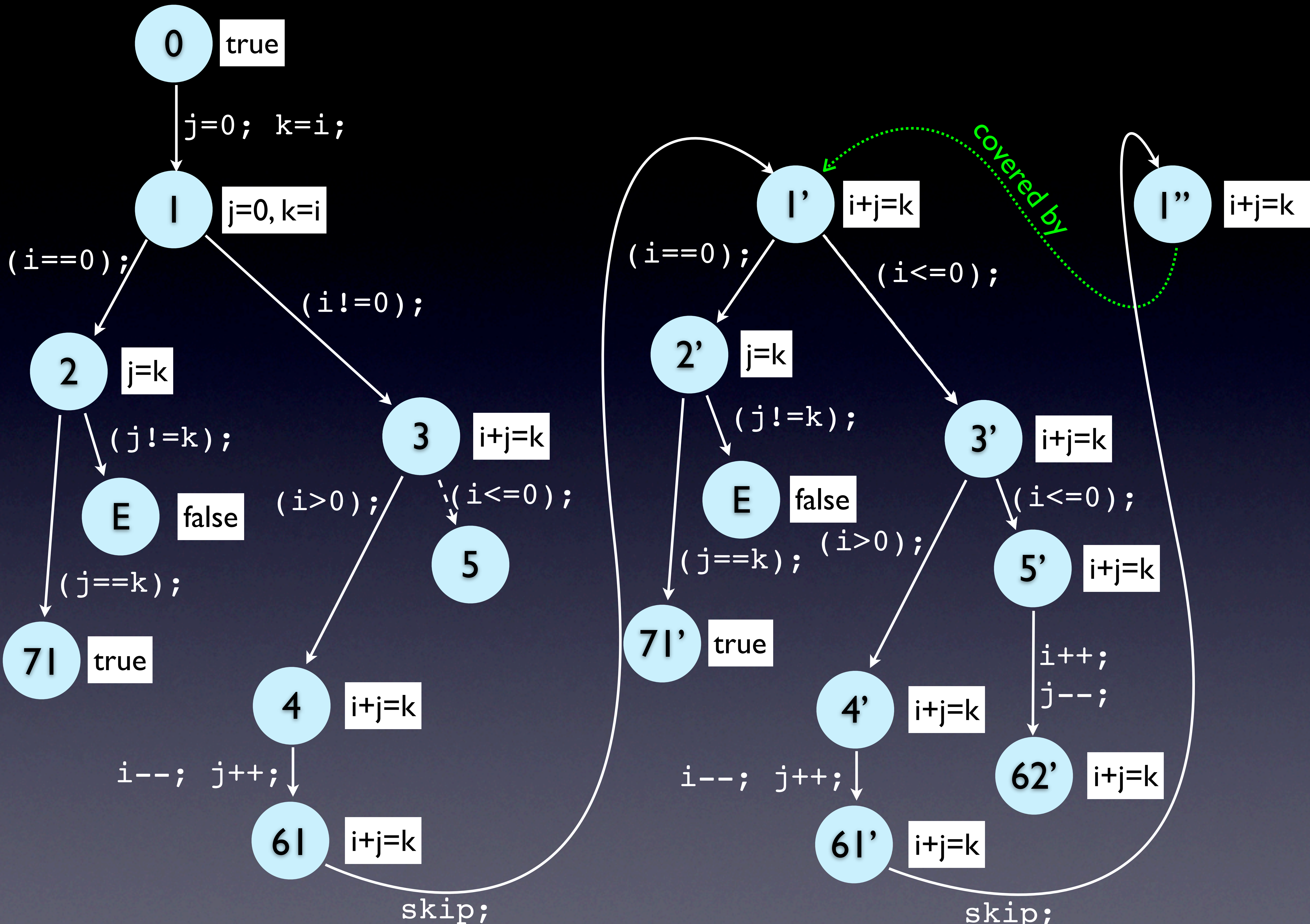
$$j_0=0 \wedge k_0=i_0 \wedge$$
$$i_0\neq 0 \wedge i_0>0 \wedge$$
$$i_1=i_0-1 \wedge j_1=j_0+1 \wedge$$
$$i_1=0 \wedge j_1\neq k_0$$

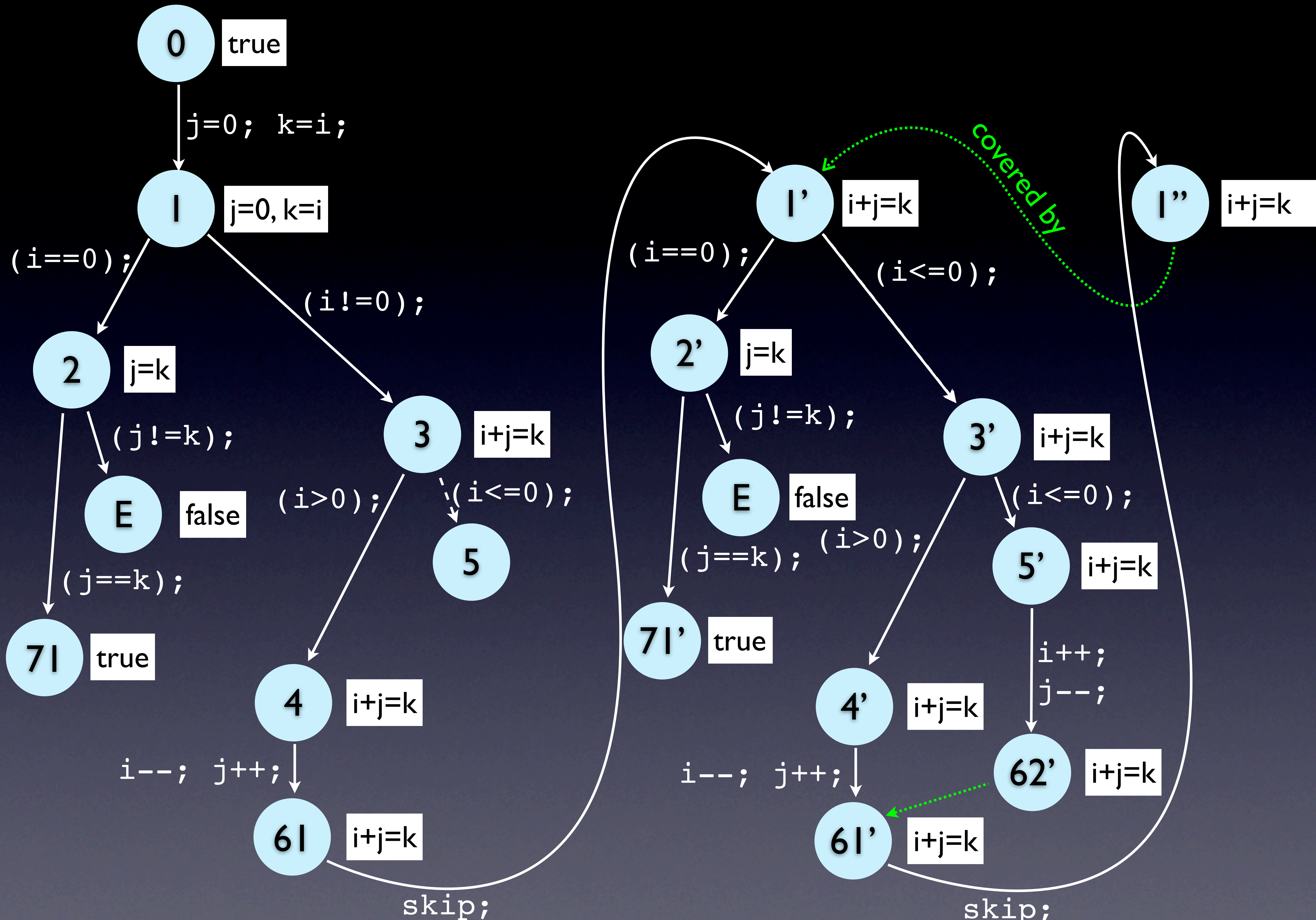

0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
3'
2'
j=k
(j!=k);
E
false
(j==k);
71'
true

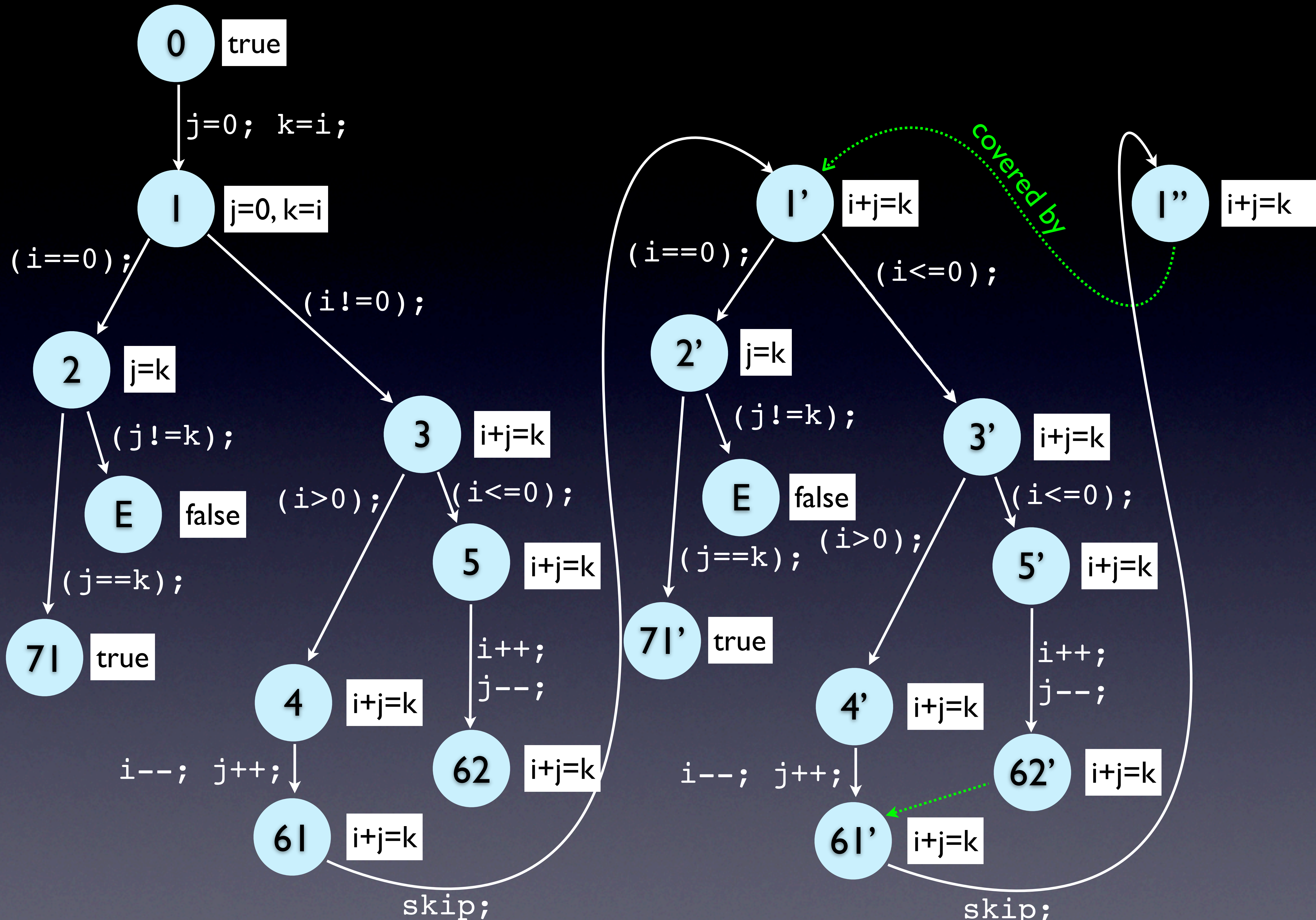

0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
3'

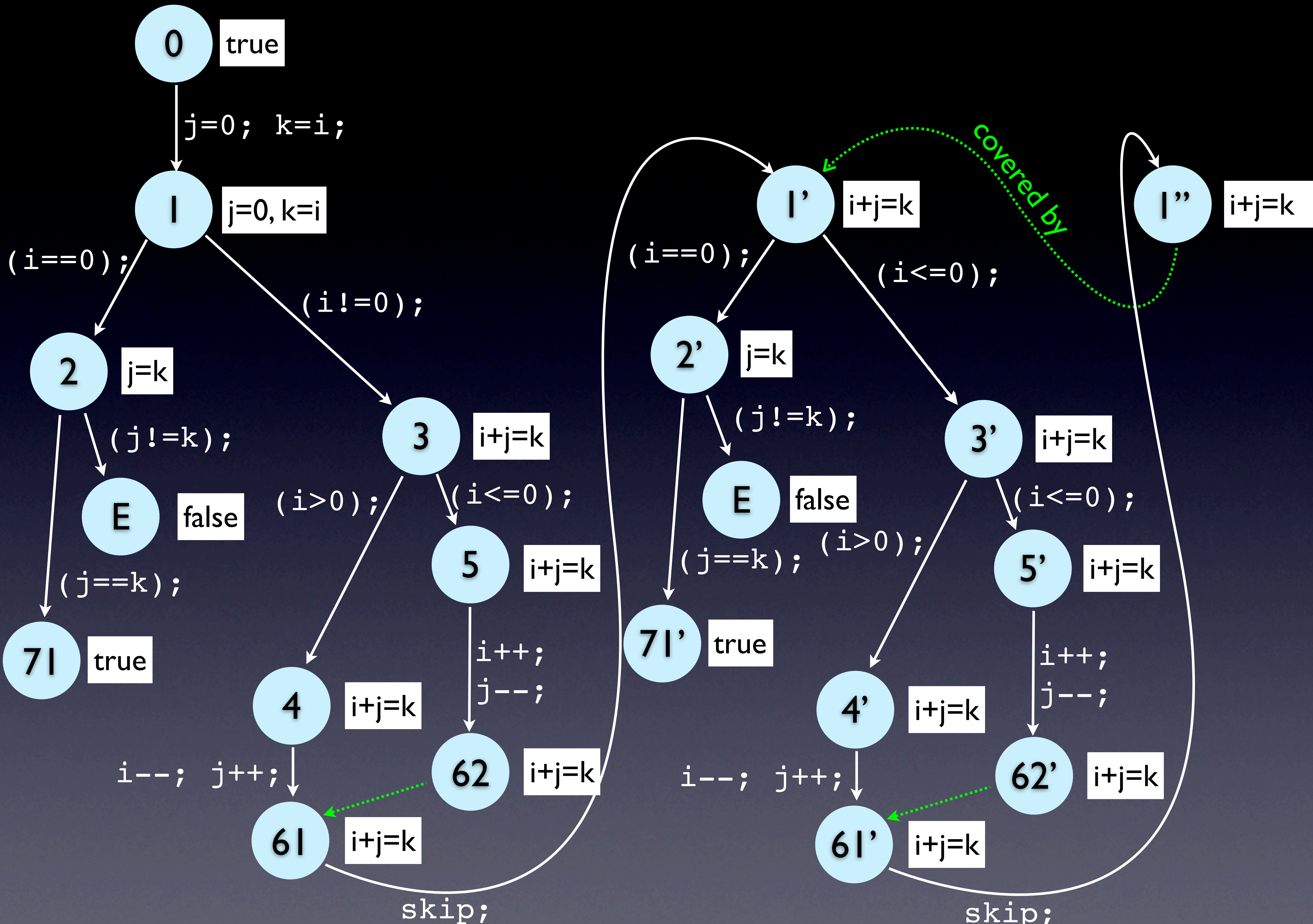

0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
3'
i+j=k
(i>0);
(i<=0);
5'
4'
i+j=k
i--; j++;
61'
i+j=k
skip;
1"
i+j=k


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
covered by
1''
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
3'
i+j=k
(i>0);
(i<=0);
5'
4'
i+j=k
i--; j++;
61'
i+j=k
skip;


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
covered by
1"
i+j=k
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
(i>0);
3'
i+j=k
(i<=0);
5'
i+j=k
i++;
j--;
62'
i+j=k
4'
i+j=k
i--; j++;
61'
i+j=k
skip;


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
4
i+j=k
i--; j++;
61
i+j=k
skip;
1'
i+j=k
covered by
1''
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
(i>0);
3'
i+j=k
(i<=0);
5'
i+j=k
i++;
j--;
4'
i+j=k
i--; j++;
62'
i+j=k
61'
i+j=k
skip;


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
i+j=k
i++;
j--;
4
i+j=k
i--; j++;
61
i+j=k
62
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
(i>0);
71'
true
3'
i+j=k
(i<=0);
5'
i+j=k
i++;
j--;
4'
i+j=k
i--; j++;
61'
i+j=k
62'
i+j=k
skip;
covered by
1''
i+j=k


0
true
j=0; k=i;
1
j=0, k=i
(i==0);
(i!=0);
2
j=k
(j!=k);
E
false
(j==k);
71
true
3
i+j=k
(i>0);
(i<=0);
5
i+j=k
i++;
j--;
4
i+j=k
i--; j++;
62
i+j=k
61
i+j=k
skip;
1'
i+j=k
(i==0);
(i<=0);
2'
j=k
(j!=k);
E
false
(j==k);
71'
true
3'
i+j=k
(i>0);
(i<=0);
5'
i+j=k
i++;
j--;
4'
i+j=k
i--; j++;
62'
i+j=k
61'
i+j=k
skip;
covered by
1''
i+j=k

Completeness

- An ART is <u>complete</u> for a CFA iff
  - The root is labelled with the initial states of the CFA
  - For each internal node (n,r) ∈ ART with r satisfiable, if $n \rightarrow^{op} m$ ∈ CFA then $(n,r) \rightarrow^{op} (n',r')$ ∈ ART and {r}op{r'}
  - For each leaf node (n,r) ∈ ART, either n has no outgoing edges ∈ CFA, or r is unsatisfiable, or the node is covered.

Multi-procedural programs

- Essentially, non-recursive procedures can be handled by inlining calls
- BLAST can handle recursive procedures too

Pointer aliasing

- Andersen's analysis
- For each pointer, store the set of locations to which it may point
- Flow-insensitive

Andersen's Analysis

Statement	Constraints
p = &x	$x \in \uparrow p$
p = q	$\uparrow p \supseteq \uparrow q$
p = *q	$\forall q' \in \uparrow q.\ \uparrow p \supseteq \uparrow q'$
*p = q	$\forall p' \in \uparrow p.\ \uparrow p' \supseteq \uparrow q$

[Andersen 1994]

Pointer aliasing - an example

```
int main() {
  int *a, *b;
  int i = 0;
  a = &i;
  b = &i;
  *b = 1;
  assert(*a == 1);
}
```

```
$ blast -cref pointerprog.i
...
addPred: 0: (gui) adding predicate
* (a@main)==1 to the system
addPred: 0: (gui) adding predicate
* (a@main)==1 to the system
...
No error found.  The system is
safe :-)
...

$
```

BLAST innovations

- BLAST's use of counter-example guided abstraction refinement is inspired by SLAM
- The two main innovations in BLAST are:
  - the use of interpolants between the past and future fragments of an infeasible path to discover new predicates
  - the use of lazy predicate abstraction whereby predicates are tracked locally

Limitations

- Uses linear arithmetic, so it struggles with multiplication and bit-level manipulations
- Assumes set of integers is infinite, so doesn't account for integer overflows
- Assumes preservation of types and safety of pointer arithmetic
- No quantifiers in the predicates

34