CONCOLIC TESTING

Pınar Sağlam

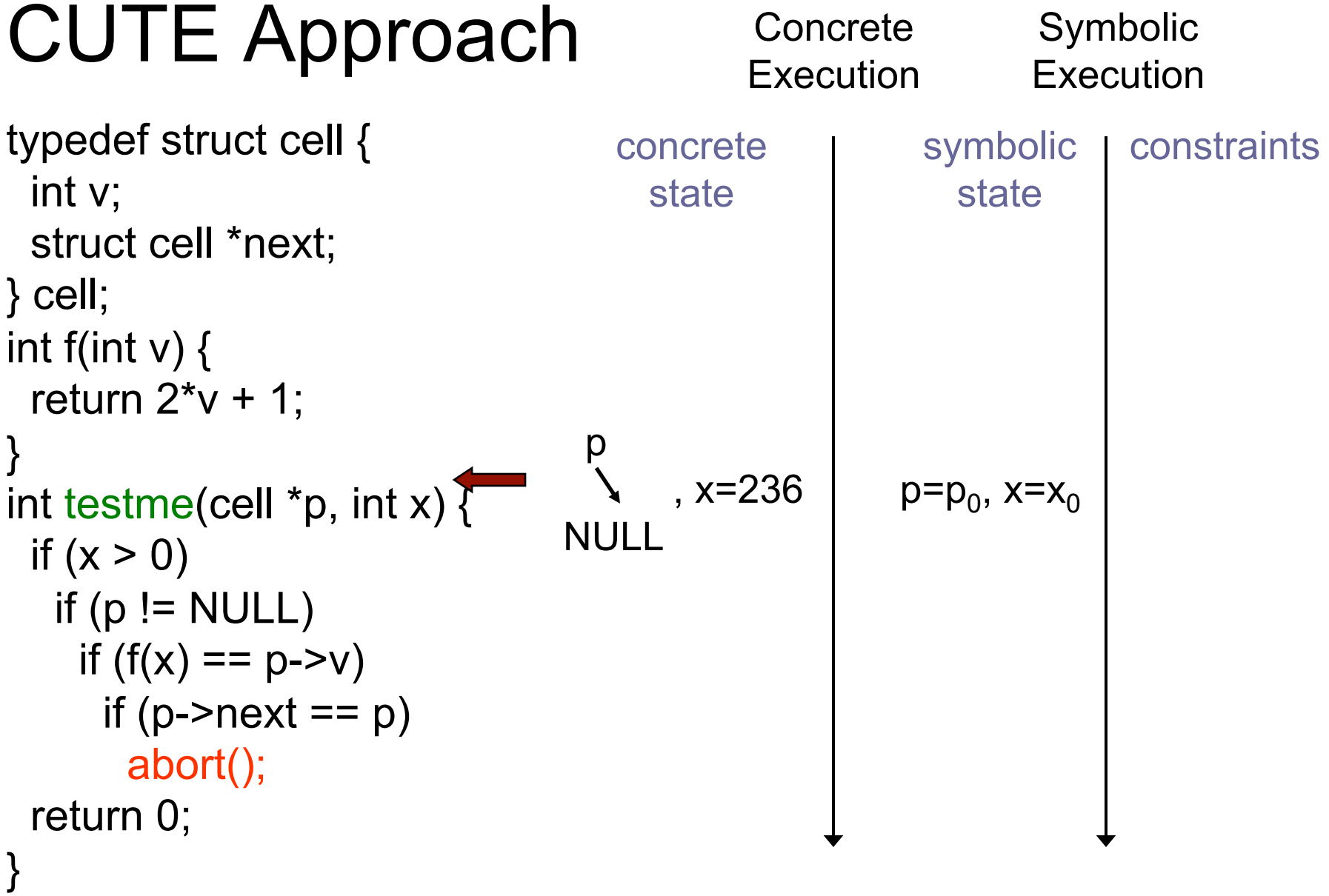
Elided to examples only

Example

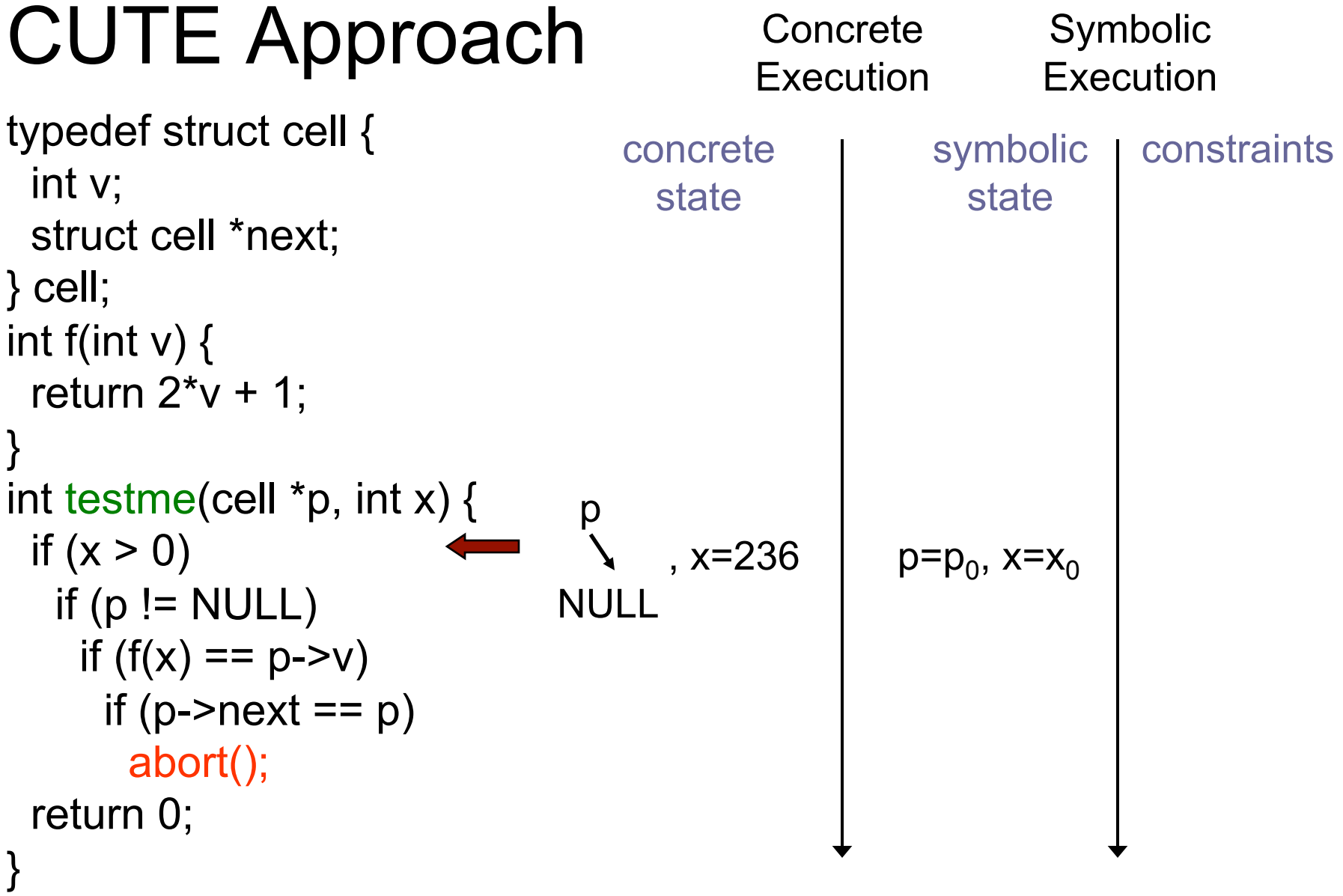
```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

- Random Test Driver:
  - random memory graph reachable from p
  - random value for x
- Probability of reaching abort( ) is extremely low

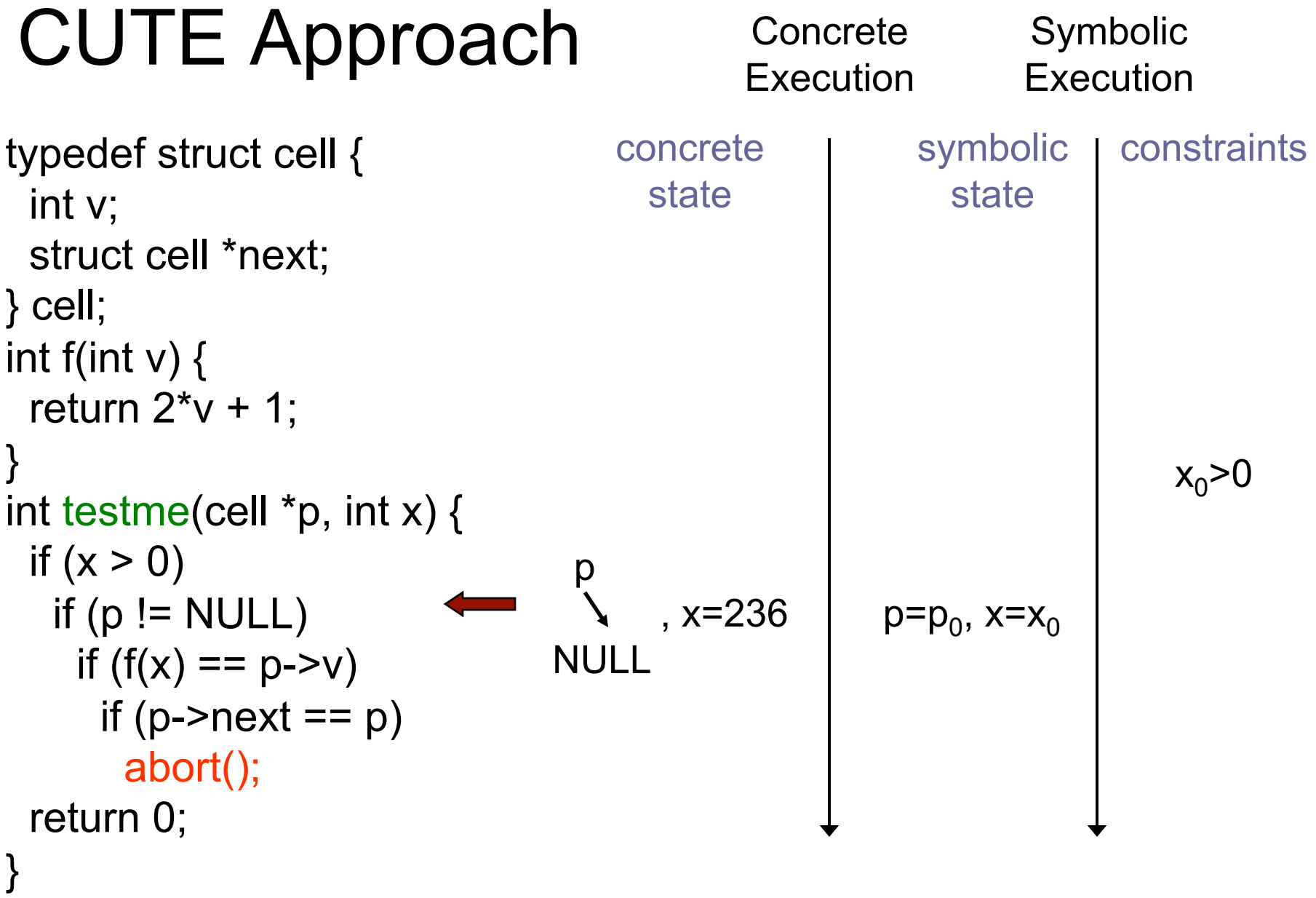
CUTE Approach



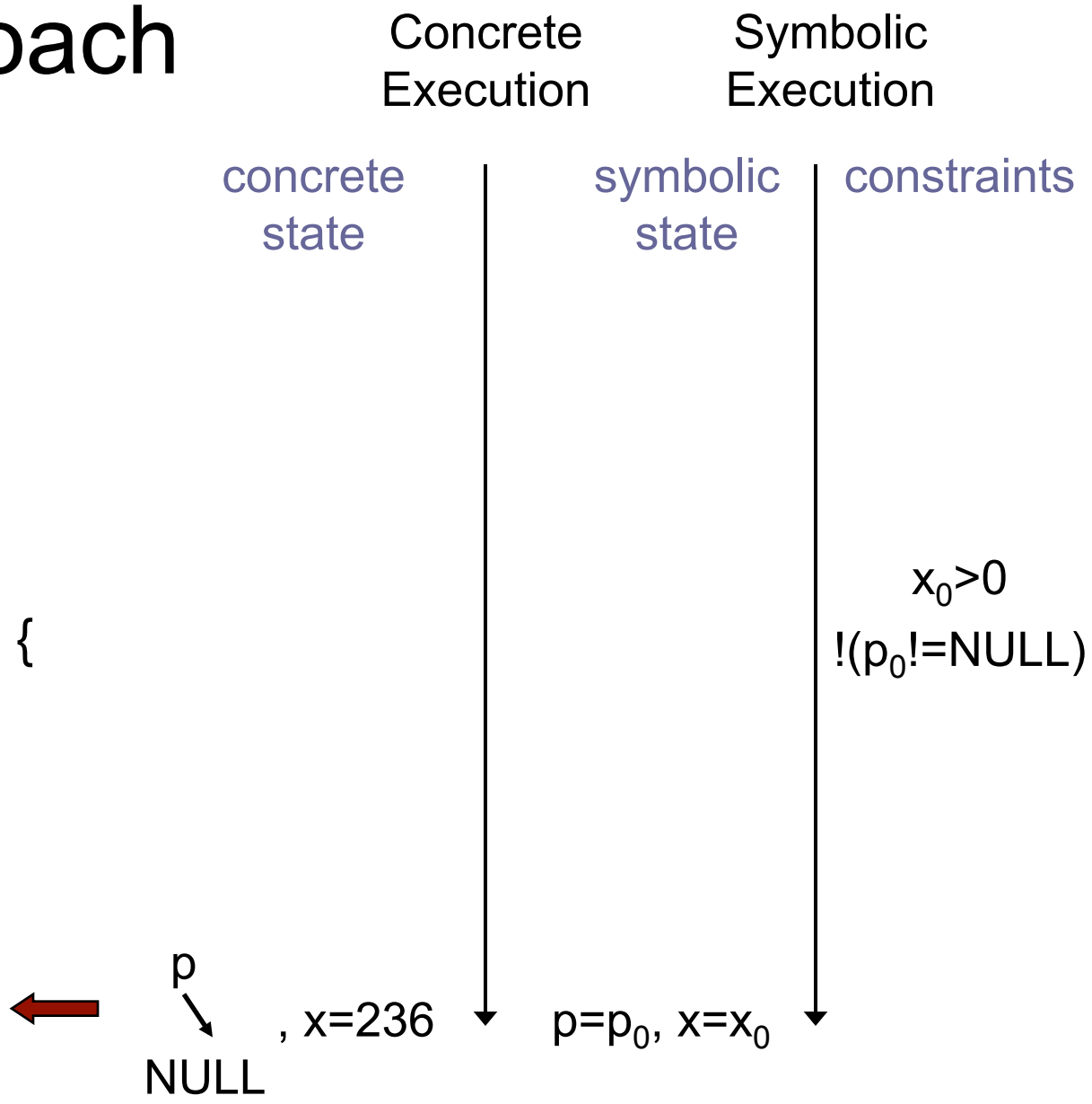
CUTE Approach



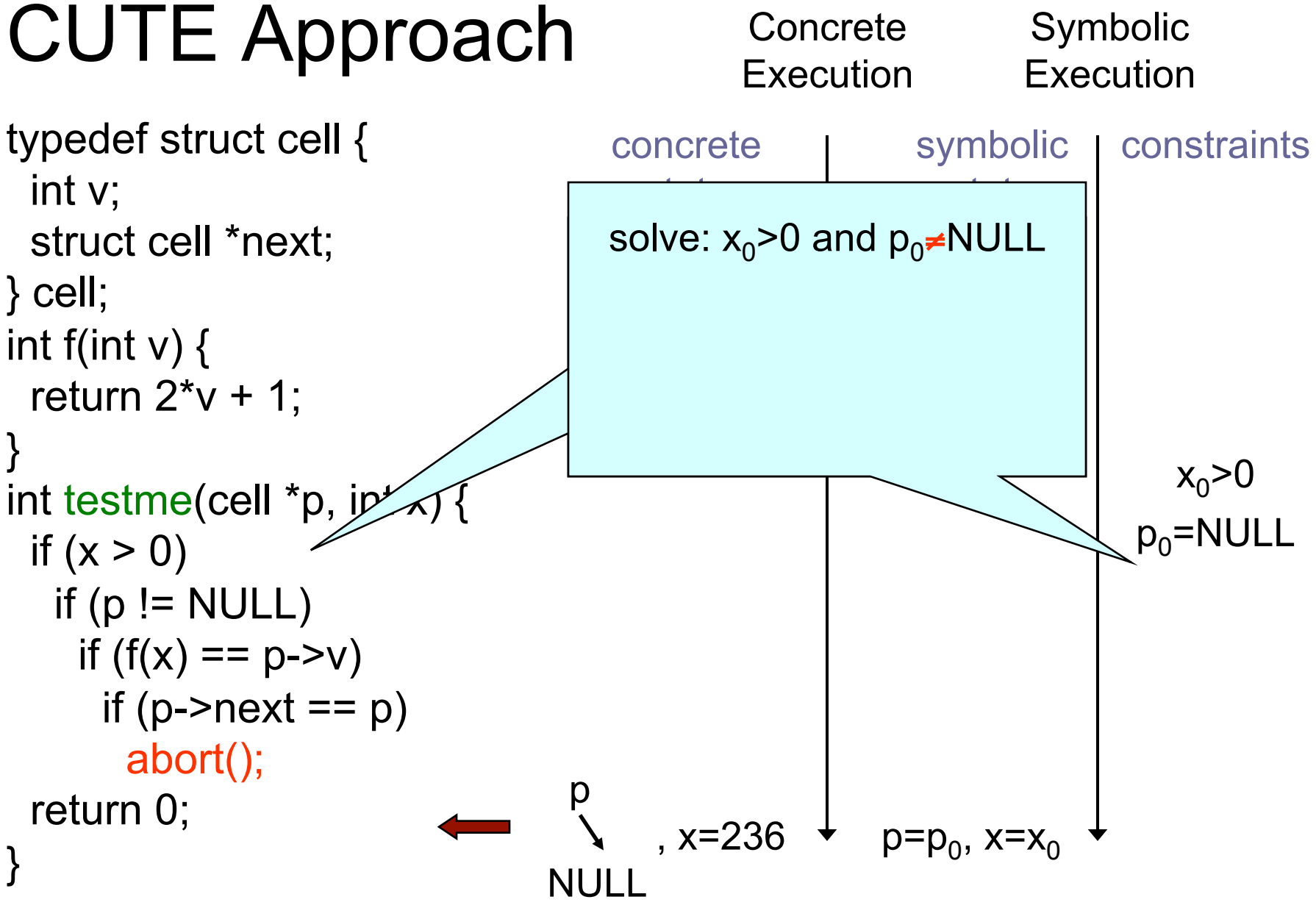
CUTE Approach



CUTE Approach



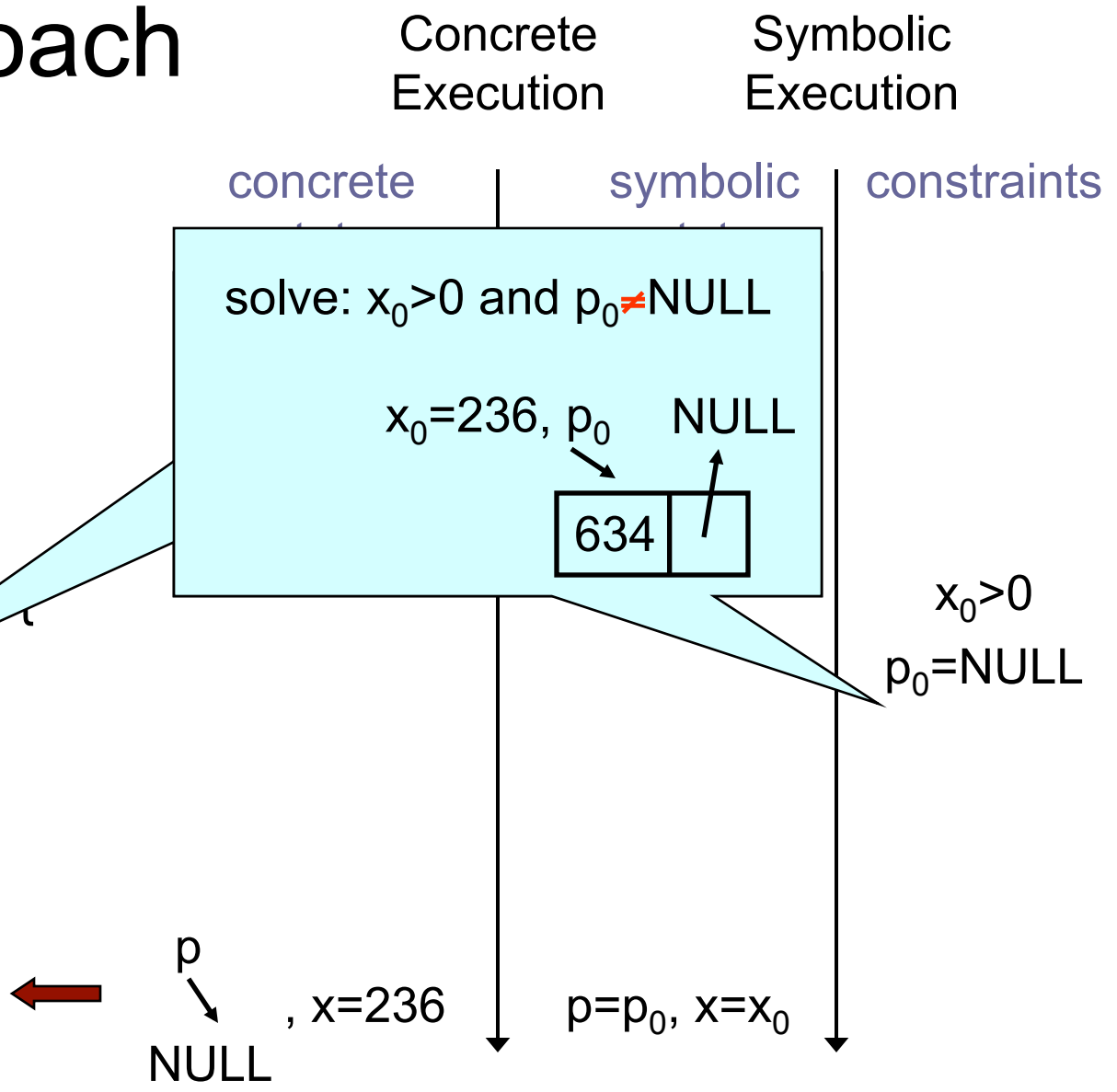
CUTE Approach




```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

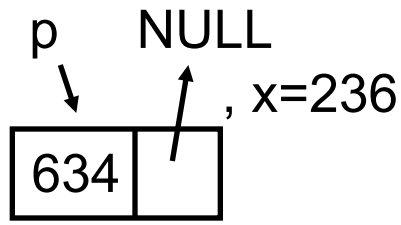
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

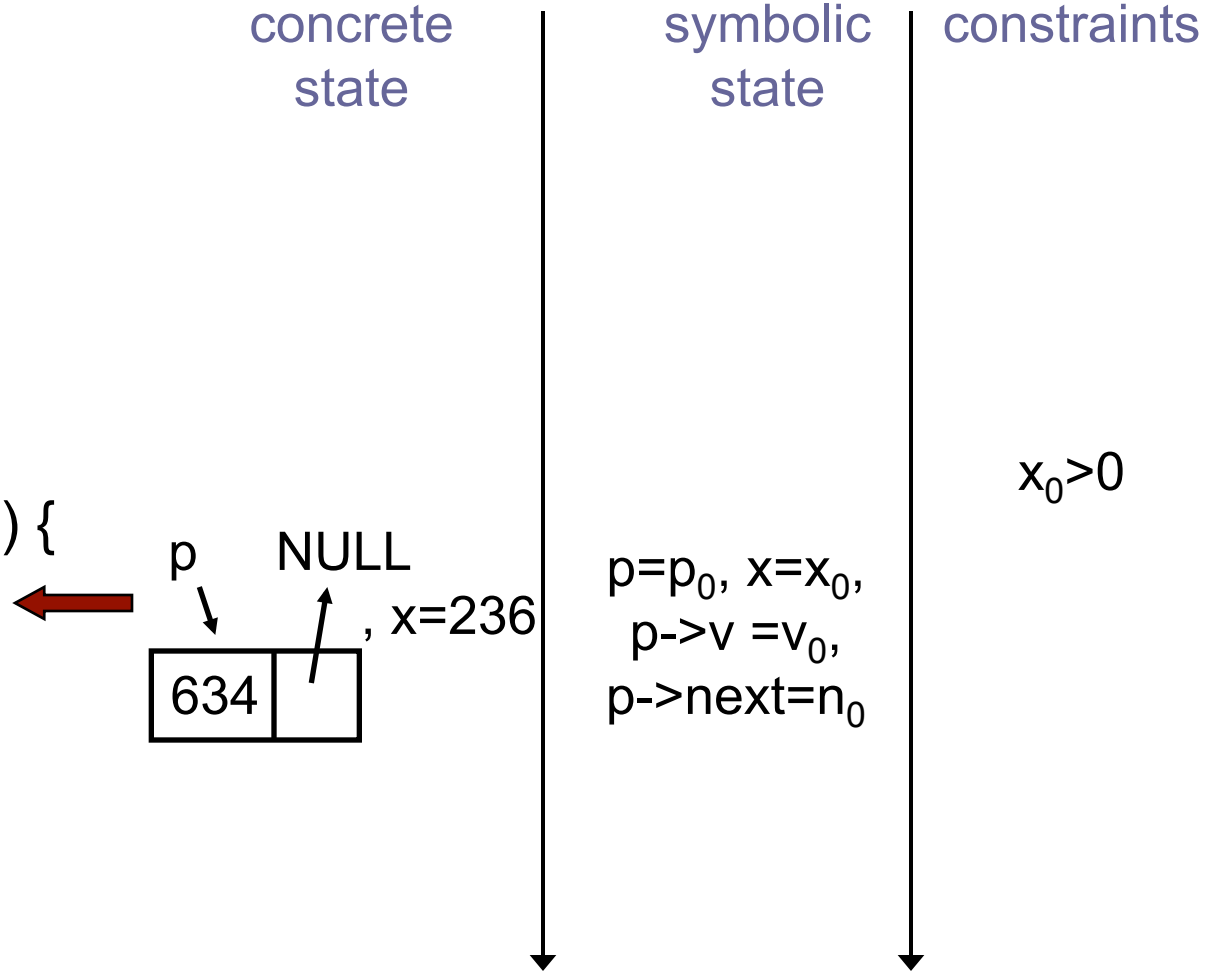
symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

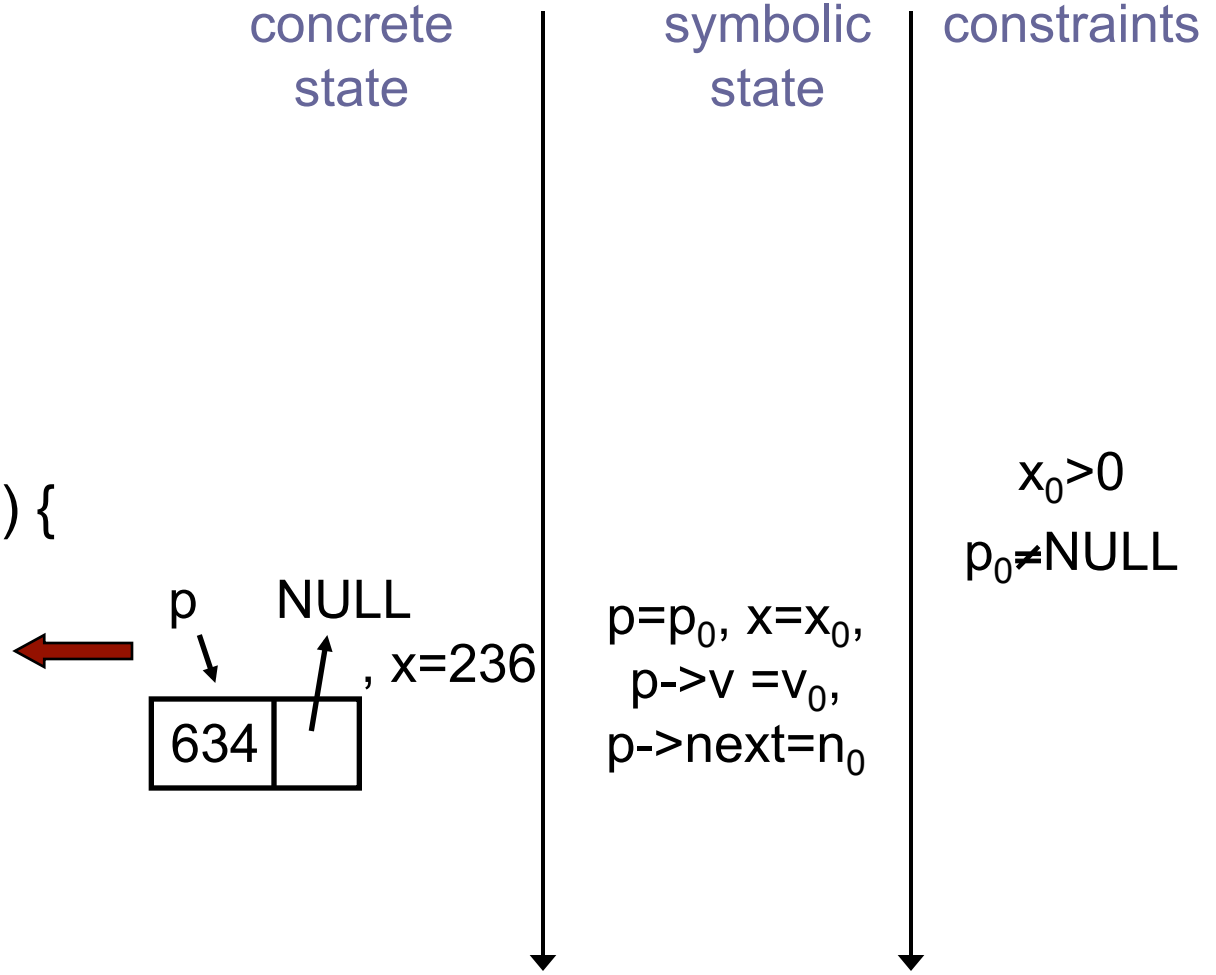
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



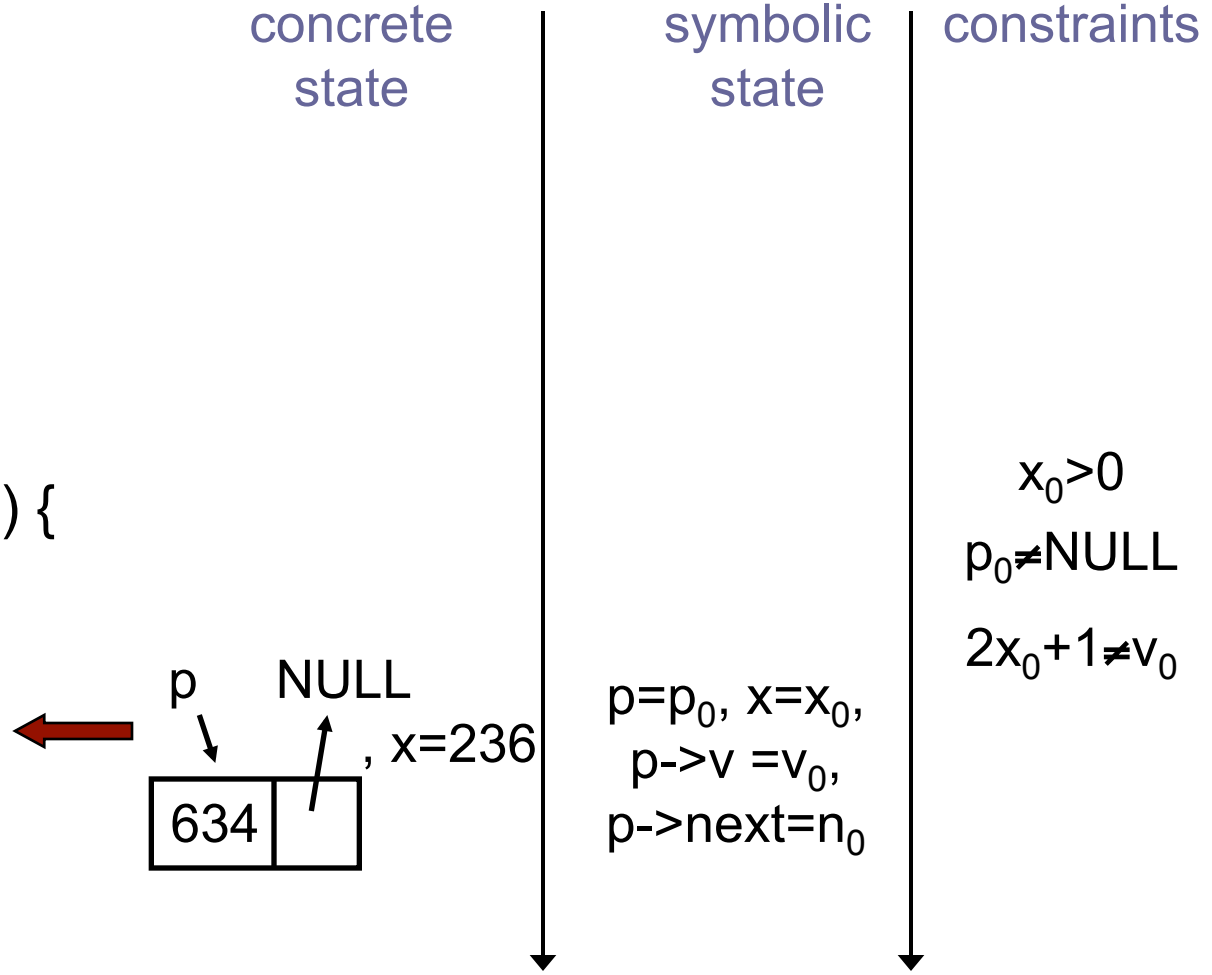
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



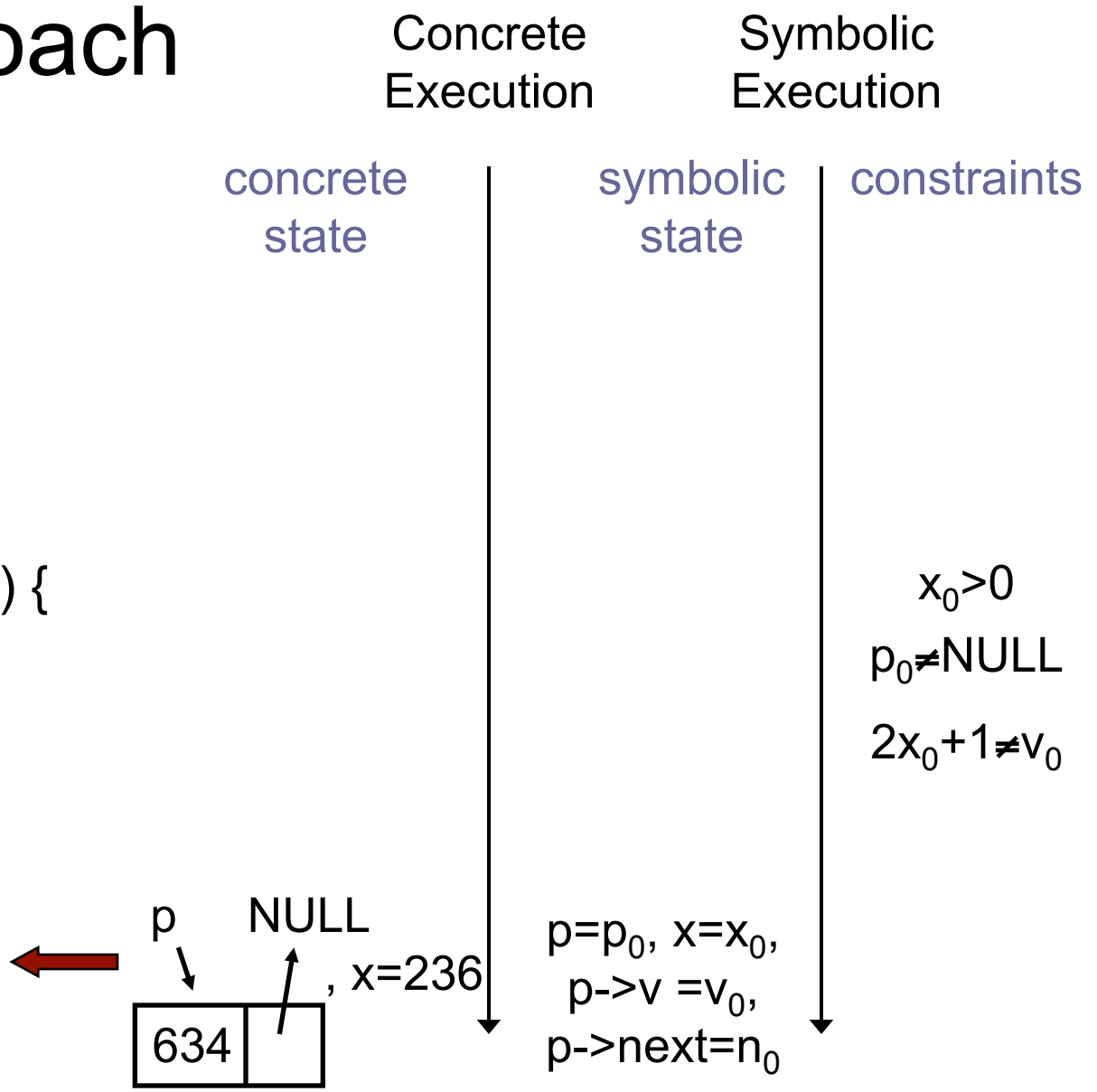
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

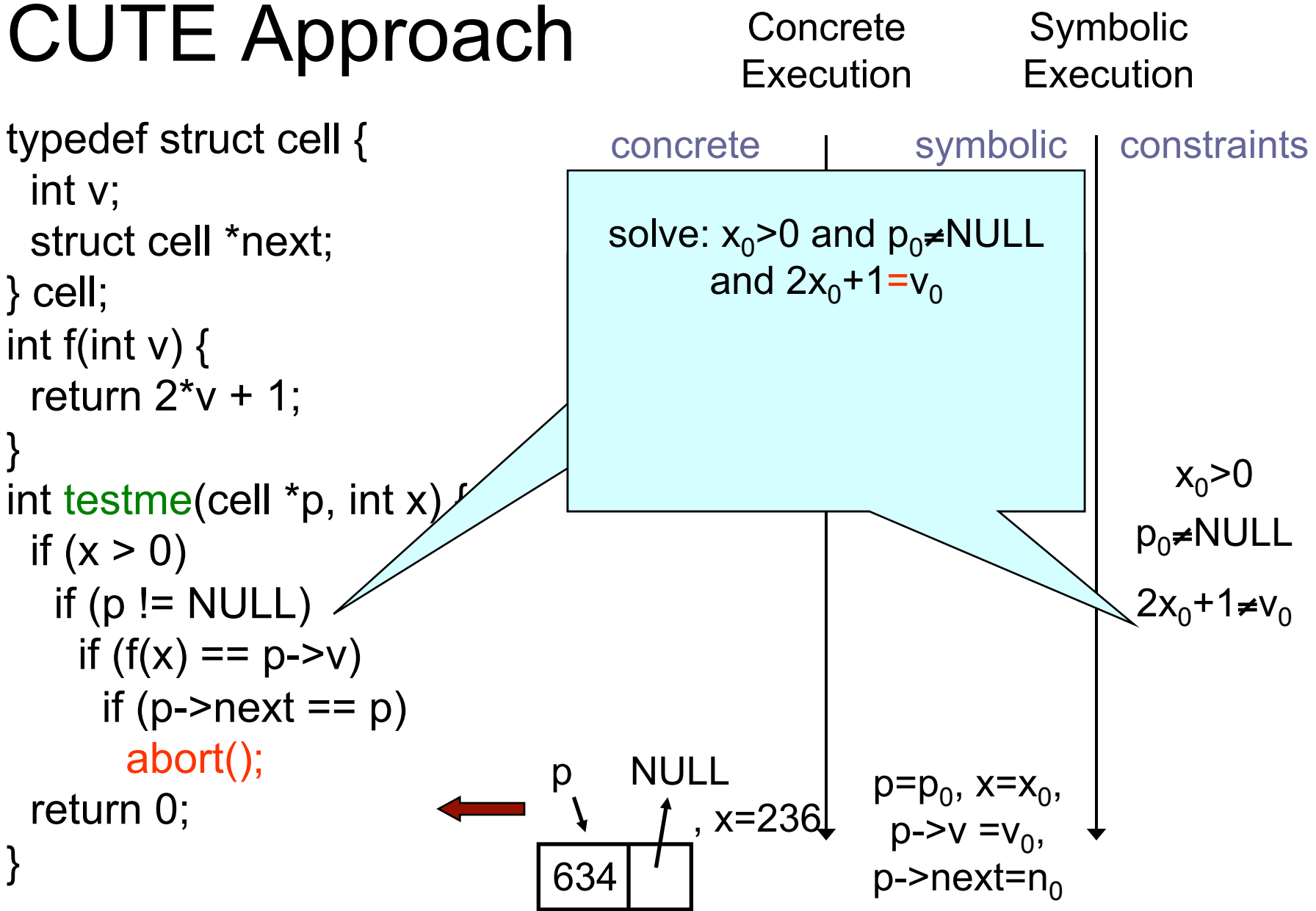


CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

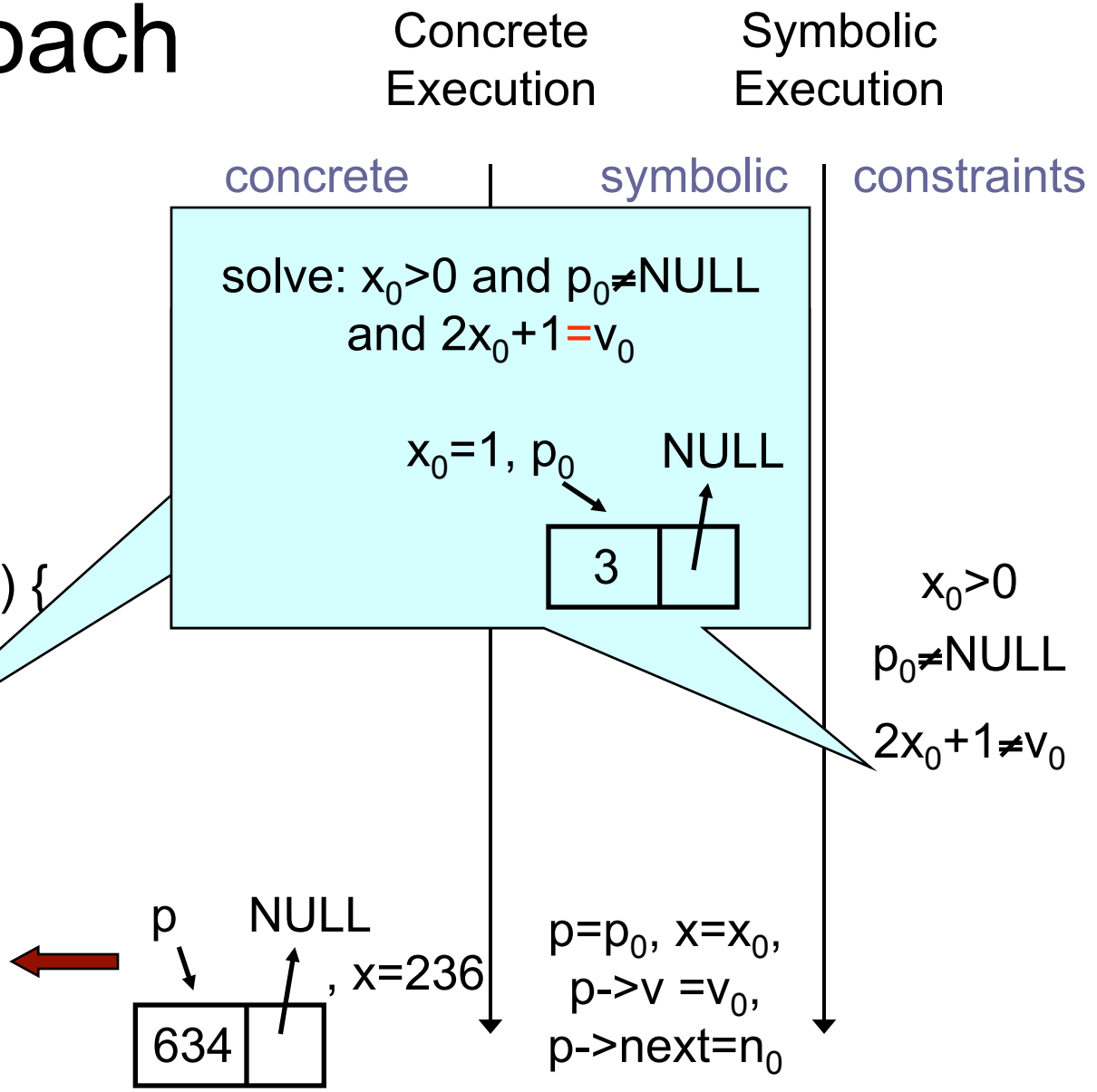


CUTE Approach



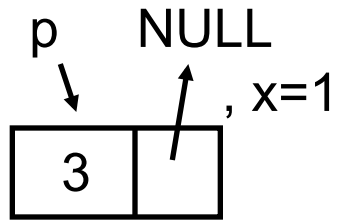
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

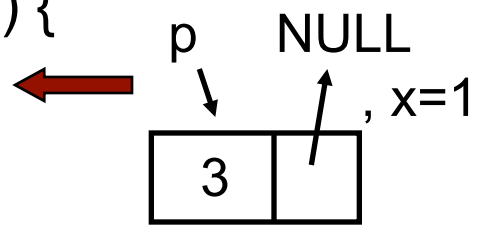
symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



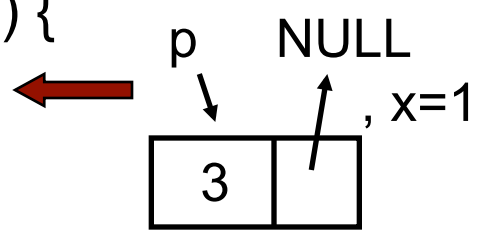
Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p->v=v_0$, $p->next=n_0$	$x_0>0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

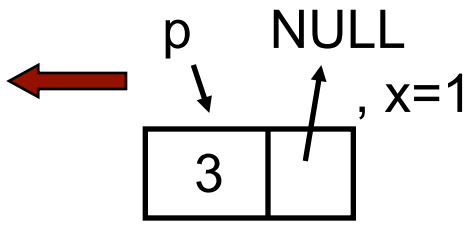
$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

$p_0 \neq NULL$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

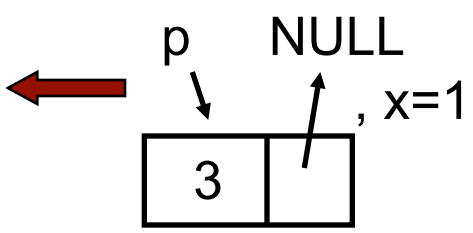
$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

$x_0>0$

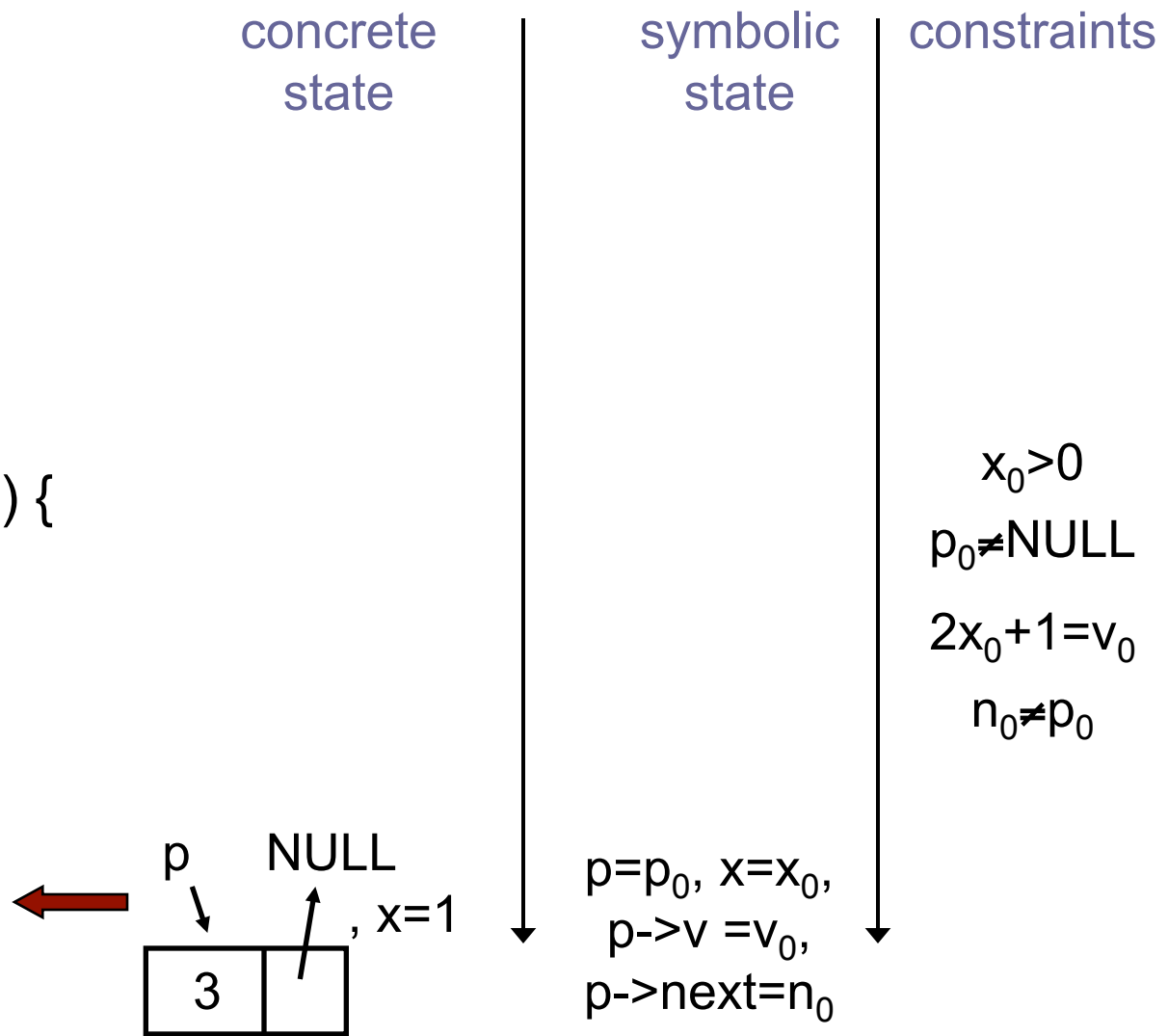
$p_0 \neq NULL$

$2x_0+1=v_0$

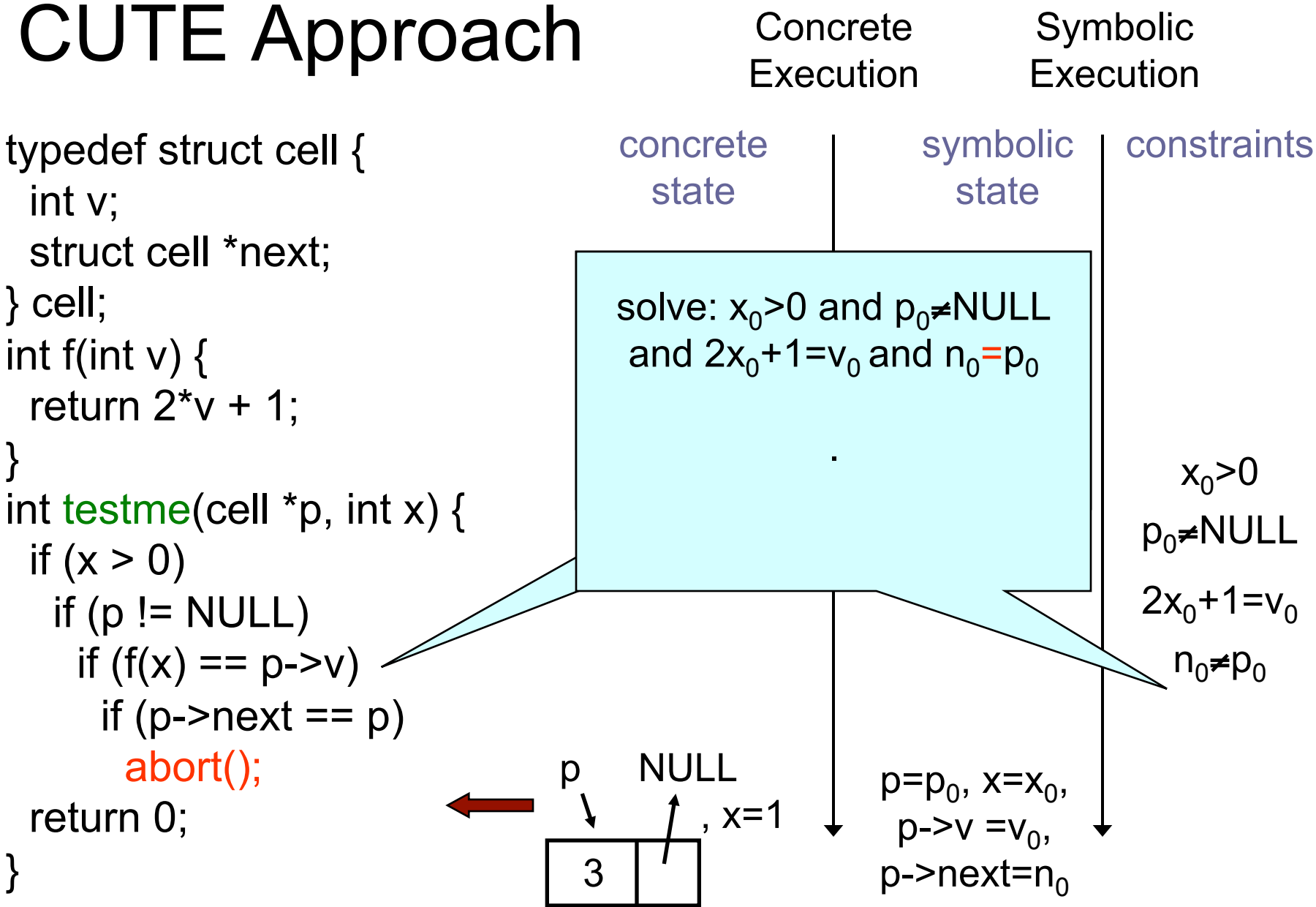
$n_0 \neq p_0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```

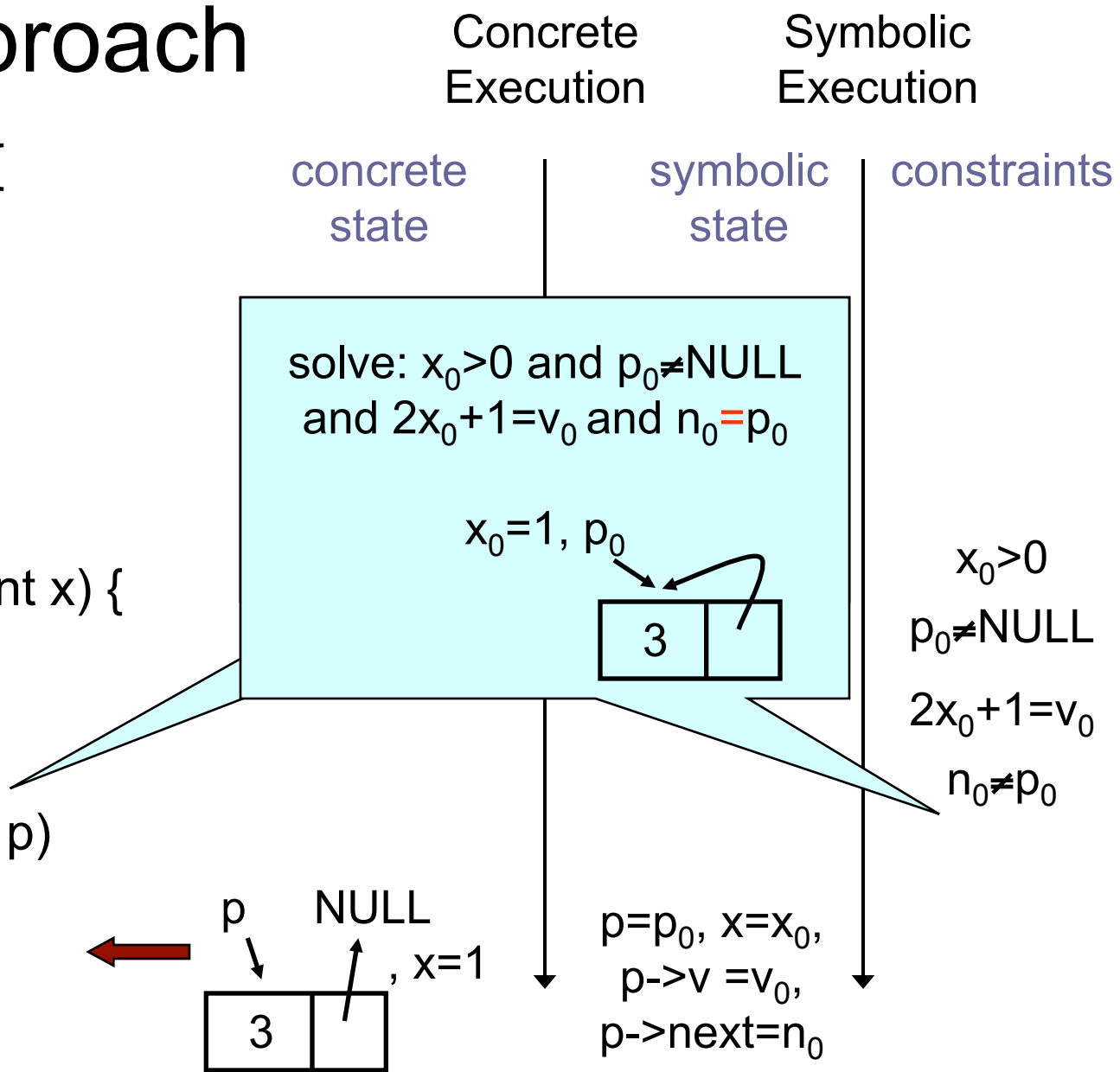


CUTE Approach



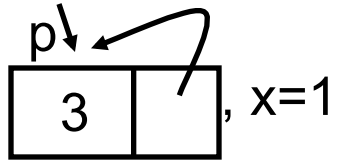
CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



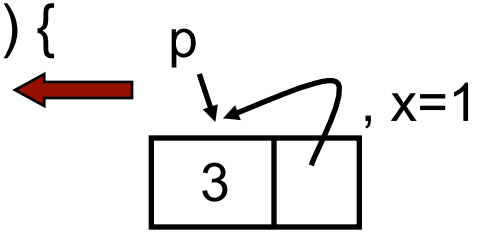
Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p->v=v_0$, $p->next=n_0$	

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



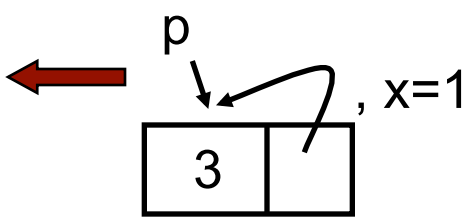
Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p\text{->}v=v_0$, $p\text{->}next=n_0$	$x_0>0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



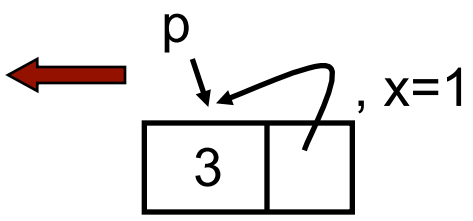
Concrete Execution

Symbolic Execution

concrete state	symbolic state	constraints
	$p=p_0$, $x=x_0$, $p->v=v_0$, $p->next=n_0$	$x_0>0$ $p_0 \neq NULL$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Concrete Execution

Symbolic Execution

concrete state

symbolic state

constraints

$p=p_0$, $x=x_0$,
$p->v = v_0$,
$p->next=n_0$

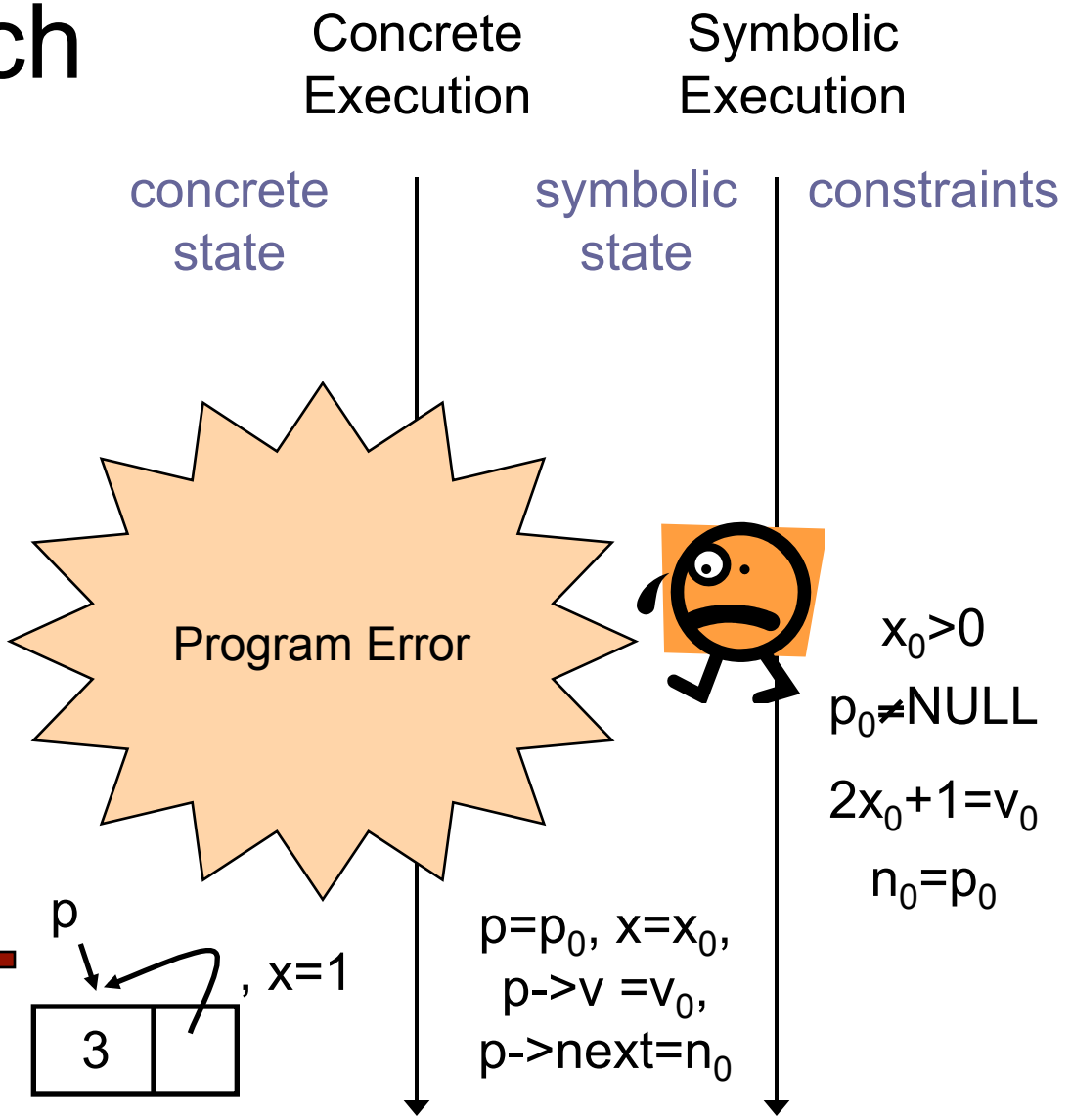
$x_0>0$

$p_0 \neq NULL$

$2x_0+1=v_0$

CUTE Approach

```
typedef struct cell {
  int v;
  struct cell *next;
} cell;
int f(int v) {
  return 2*v + 1;
}
int testme(cell *p, int x) {
  if (x > 0)
    if (p != NULL)
      if (f(x) == p->v)
        if (p->next == p)
          abort();
  return 0;
}
```



Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x +9
- take then branch with constraint x*x*x+ 3*x*x+9 ! = y

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - **Stuck ?**

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
- take then branch with constraint x*x*x+ 3*x*x+9 != y
- solve x*x*x+ 3*x*x+9 = y to take else branch
- Don't know how to solve !!
  - Stuck ?
  - NO : CUTE handles this smartly

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
          printf("Good branch");
    } else {
          printf("Bad branch");
          abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x +9
  - cannot handle symbolic value of z

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x +9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x +9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if(z != y){
        printf("Good branch");
    } else {
        printf("Bad branch");
        abort();
    }
}
```

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)

Simultaneous Symbolic & Concrete Execution

```
void again_test_me(int x,int
    y){
    z = x*x*x + 3*x*x + 9;
    if
    }
}
```

Replace symbolic expression by concrete value when symbolic expression becomes unmanageable (i.e. non-linear)

- Let initially x = -3 and y = 7 generated by random test-driver
- concrete z = 9
- symbolic z = x*x*x + 3*x*x+9
  - cannot handle symbolic value of z
  - make symbolic z = 9 and proceed
- take then branch with constraint 9 != y
- solve 9 = y to take else branch
- execute next run with x = -3 and y= 9
  - got error (reaches abort)