The Reachability Problem for Vector Addition Systems is Ackermann-complete

Wojciech Czerwiński

Łukasz Orlikowski

Plan

Plan

- basic definitions

Plan

- basic definitions
- short history

Plan

- basic definitions
- short history
- proof

Plan

- basic definitions
- short history
- proof
  - known techniques

Plan

- basic definitions
- short history
- proof
  - known techniques
  - new technique (many zero-tests)

Vector Addition Systems with States

Vector Addition Systems with States

Nonnegative counters

Vector Addition Systems with States

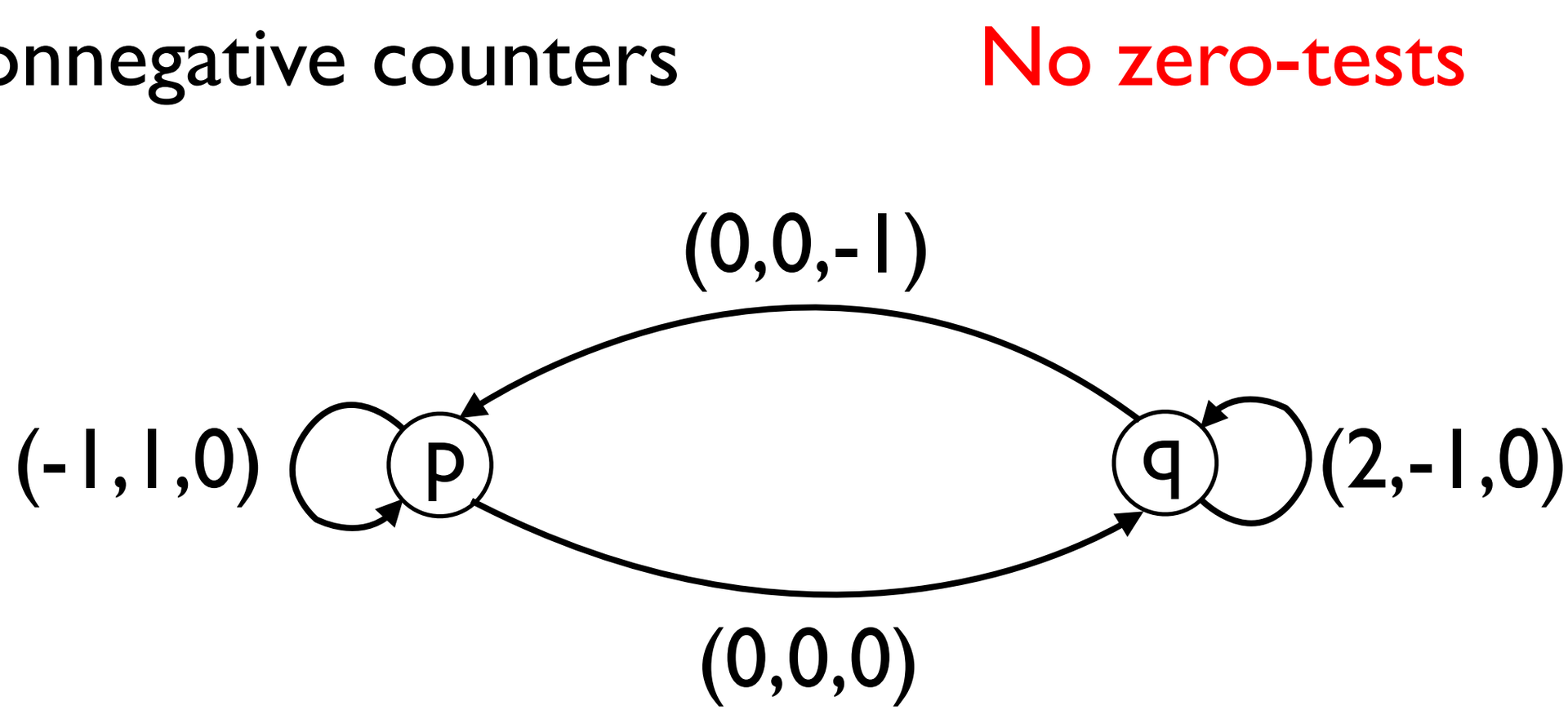
Nonnegative counters

No zero-tests

Vector Addition Systems with States

Nonnegative counters

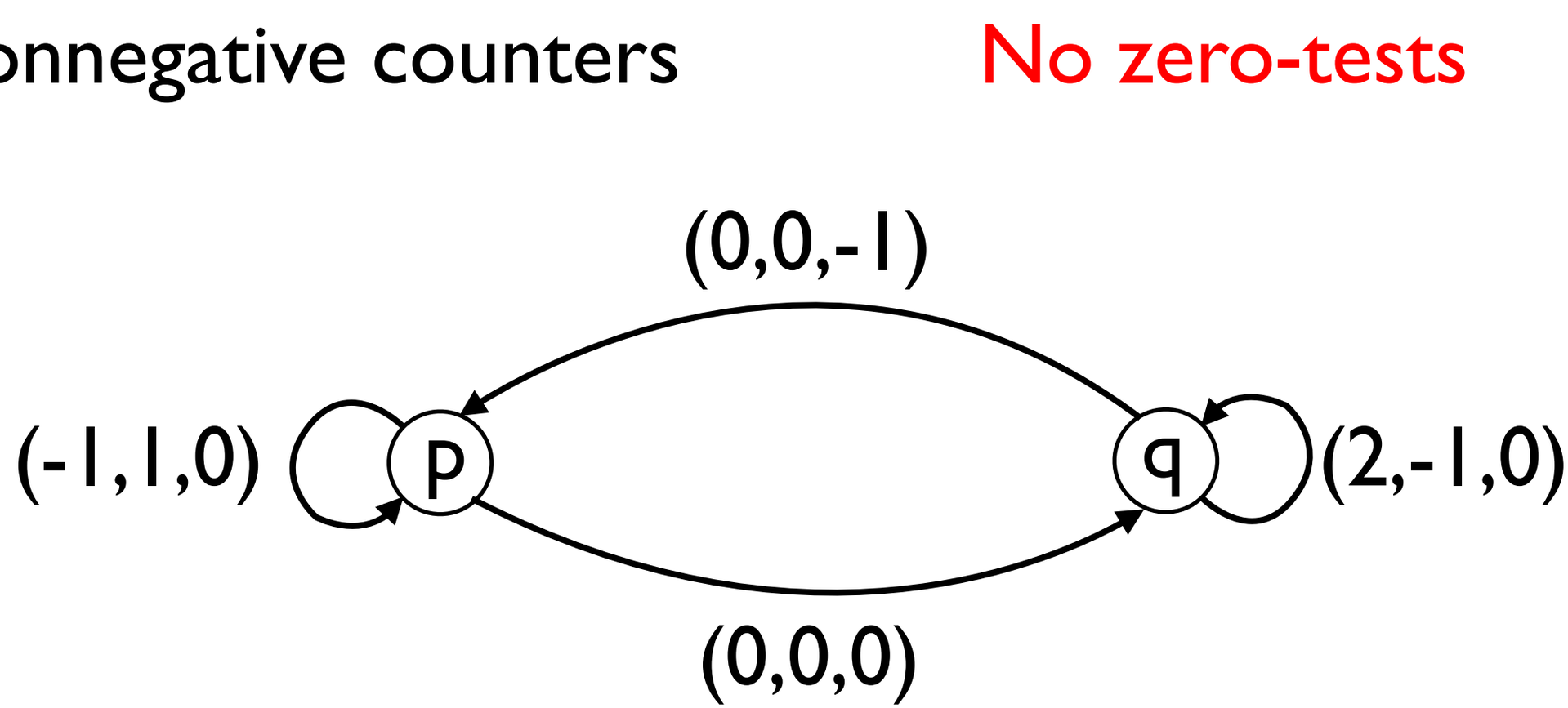
No zero-tests



Vector Addition Systems with States

Nonnegative counters

No zero-tests

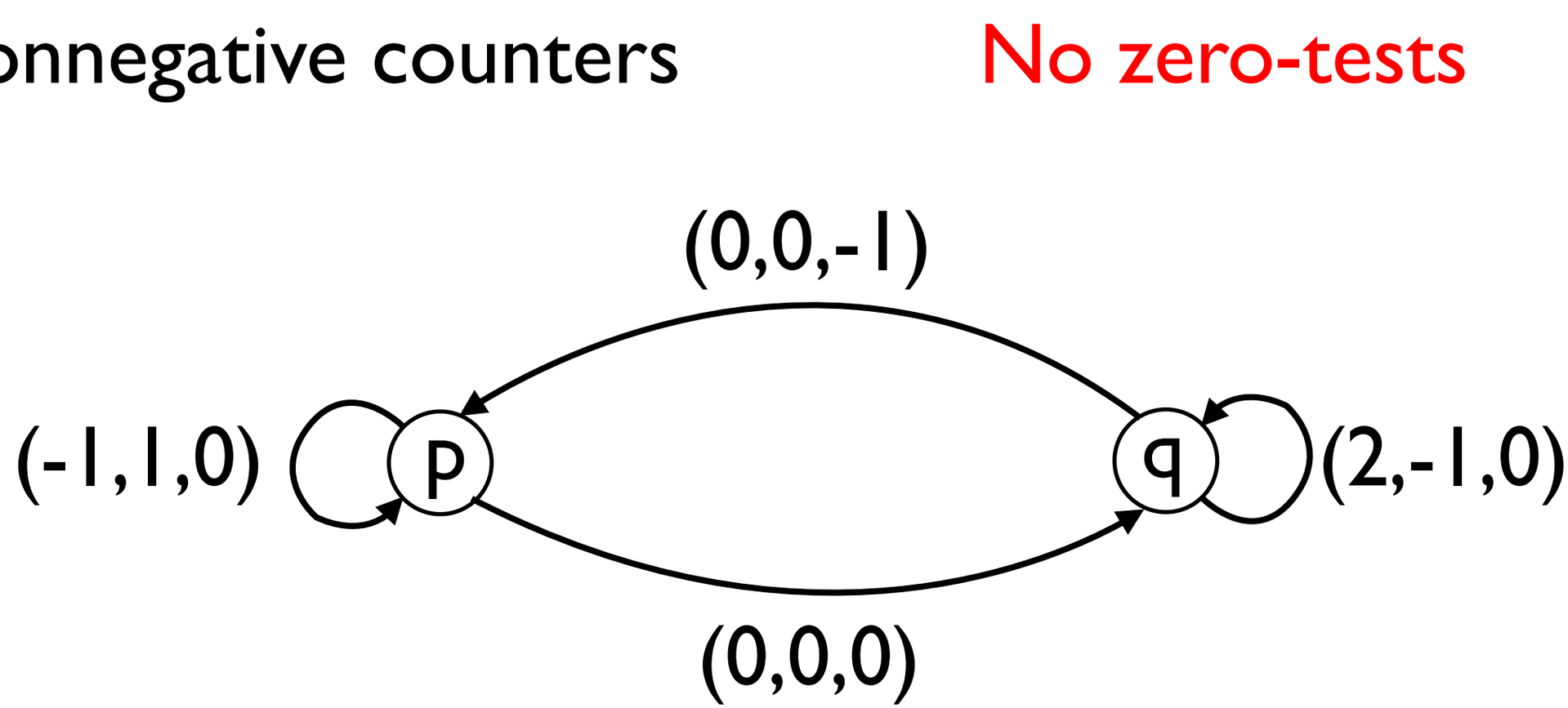


p(1,0,4)

Vector Addition Systems with States

Nonnegative counters

No zero-tests

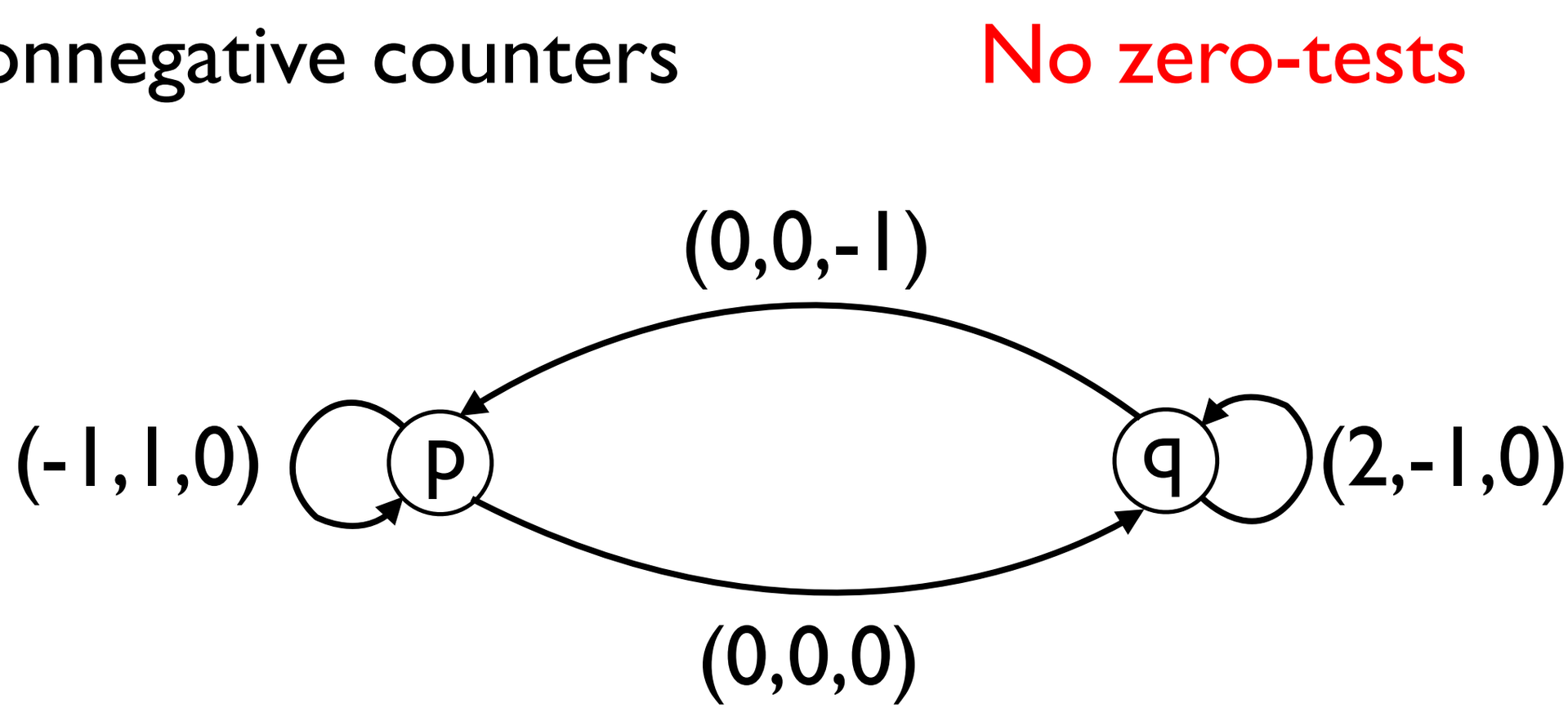


$p(1,0,4) \longrightarrow p(0,1,4)$

Vector Addition Systems with States

Nonnegative counters

No zero-tests

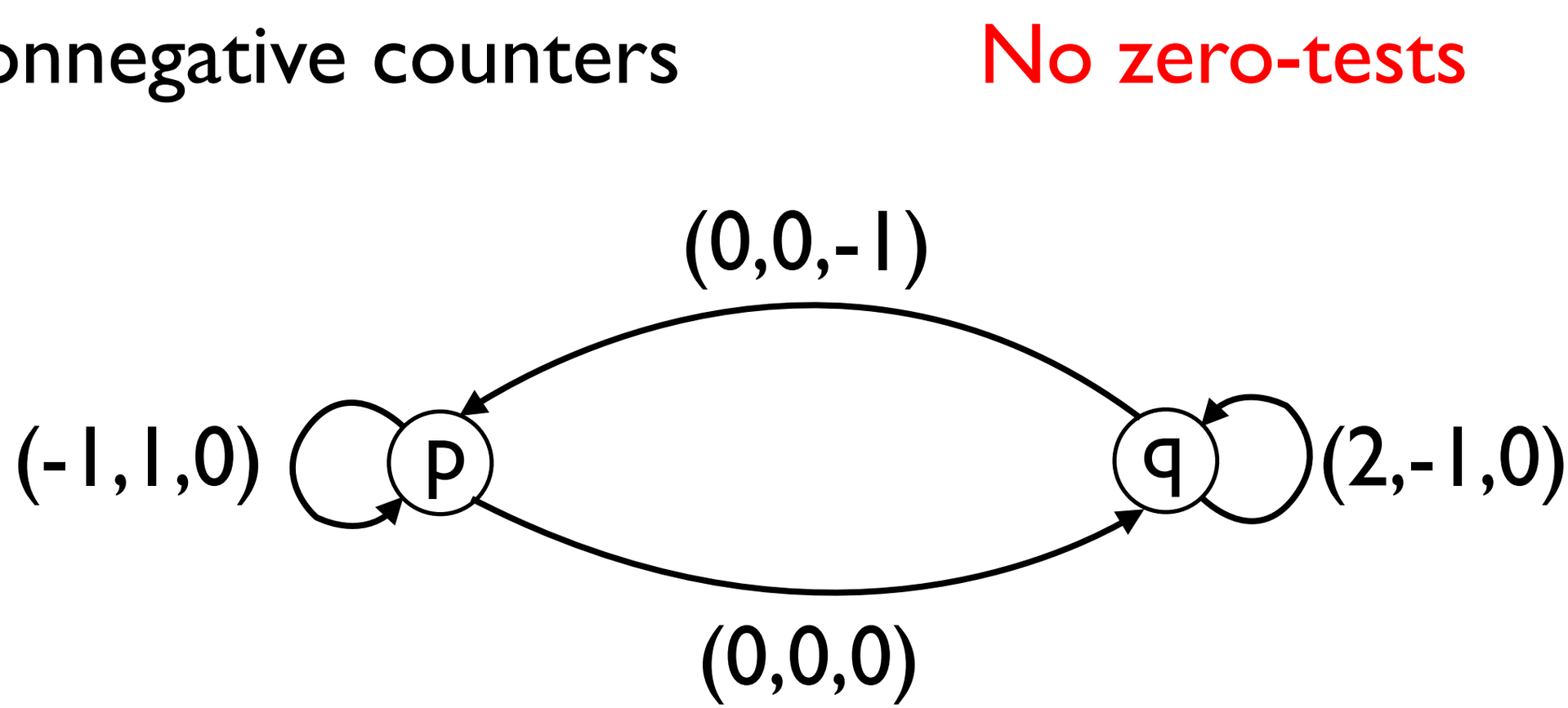


$p(1,0,4) \longrightarrow p(0,1,4) \longrightarrow q(0,1,4)$

Vector Addition Systems with States

Nonnegative counters

No zero-tests

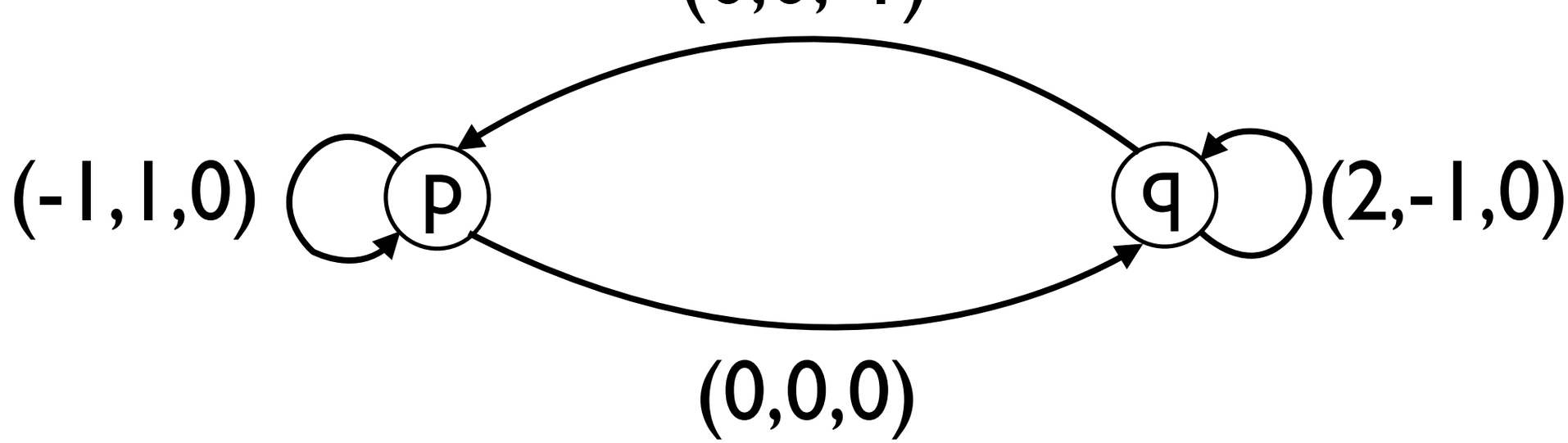


$p(1,0,4) \longrightarrow p(0,1,4) \longrightarrow q(0,1,4) \longrightarrow q(2,0,4)$

Vector Addition Systems with States

Nonnegative counters

No zero-tests



$p(1,0,4) \longrightarrow p(0,1,4) \longrightarrow q(0,1,4) \longrightarrow q(2,0,4) \longrightarrow p(2,0,3)$

Reachability problem

Reachability problem

Given: a VASS, two configurations s and t

Reachability problem

Given: a VASS, two configurations s and t

Question: is there a run from s to t?

Short history - reachability

Short history - reachability

Lipton `76: ExpSpace-hardness

Short history - reachability

Lipton `76: ExpSpace-hardness

Mayr `81: decidability

Short history - reachability

Lipton `76: ExpSpace-hardness

Mayr `81: decidability

Kosaraju `82, Lambert `92: simplifications

Short history - reachability

Lipton `76: ExpSpace-hardness

Mayr `81: decidability

Kosaraju `82, Lambert `92: simplifications

Leroux, Schmitz `15: cubic-Ackermann

Short history - reachability

Lipton `76: ExpSpace-hardness

Mayr `81: decidability

Kosaraju `82, Lambert `92: simplifications

Leroux, Schmitz `15: cubic-Ackermann

Leroux, Schmitz `19: Ackermann

Short history - reachability

Lipton `76: ExpSpace-hardness

Mayr `81: decidability

Kosaraju `82, Lambert `92: simplifications

Leroux, Schmitz `15: cubic-Ackermann

Leroux, Schmitz `19: Ackermann

Cz., Lasota, Lazic, Leroux, Mazowiecki `19:
Tower-hardness

Main result

Main result

Theorem

The Reachability Problem for Vector Addition Systems with States is Ackermann-hard.

Main result

Theorem

The Reachability Problem for Vector Addition Systems with States is Ackermann-hard.

Not in primitive recursive time

Main result

Theorem

The Reachability Problem for Vector Addition Systems with States is Ackermann-hard.

Not in primitive recursive time

Not in time $F_k(n)$

Functions F_k

Functions F_k

$F_1(n) = 2n$

Functions F_k

$F_1(n) = 2n$

$F_{k+1}(n) = F_k \circ \ldots \circ F_k(1)$

Functions F_k

$F_1(n) = 2n$

$F_{k+1}(n) = F_k \circ \ldots \circ F_k(1)$

composed n times

Functions F_k

$F_1(n) = 2n$

$F_{k+1}(n) = F_k \circ \ldots \circ F_k(1)$

composed n times

$F_2(n) = 2^n$

Functions F_k

$F_1(n) = 2n$

$F_{k+1}(n) = F_k \circ \ldots \circ F_k(1)$

composed n times

$F_2(n) = 2^n$

$F_3(n) = \text{Tower}(n)$

Functions F_k

$F_1(n) = 2n$

$F_{k+1}(n) = F_k \circ \ldots \circ F_k(1)$

composed n times

$F_2(n) = 2^n$ $F_3(n) = \text{Tower}(n)$

$\text{Ack}(n) = F_\omega(n) = F_n(n)$

Main result

Main result

Theorem

The Reachability Problem for 6k-VASSes is F_k-hard.

Main result

Theorem

The Reachability Problem for 6k-VASSes is F_k-hard.

Jerome: dimension 4k+9 is enough

Main result

Theorem

The Reachability Problem for 6k-VASSes is F_k-hard.

Jerome: dimension 4k+9 is enough

Sławek: 3k+2 is enough

Big counters

Big counters

The following problem is $\mathbb{F}_k$-complete

Big counters

The following problem is $\mathbb{F}_k$-complete

Given:

Big counters

The following problem is $\mathbb{F}_k$-complete

Given: a counter automaton A

with zero-tests, number n

Big counters

The following problem is $\mathbb{F}_k$-complete

Given: a counter automaton A with zero-tests, number n

Question:

Big counters

The following problem is $\mathbb{F}_k$-complete

Given: a counter automaton A with zero-tests, number n

Question: does A have an $F_k(n)$-bounded run?

Bounding counters

Bounding counters

Idea:

Bounding counters

Idea:

add counter: x’

Bounding counters

Idea:

add counter: x'

initialise: x = 0, x' = K

Bounding counters

Idea:

add counter: x'

initialise: x = 0, x' = K

keep invariant: x + x' = K

Bounding counters

Idea:

add counter: x’

initialise: x = 0, x’ = K

keep invariant: x + x’ = K

How to zero-test(x)?

Bounding counters

Idea:

add counter: x'

initialise: x = 0, x' = K

keep invariant: x + x' = K

How to zero-test(x)?

Idea: triples (K, n, Kn) allow for n/2 zero-tests

Triples

Triples

Assume (x', y, z) = (K, n, Kn)

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

loop {inc(x), dec(x'), dec(z)}

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

loop {inc(x), dec(x'), dec(z)}

loop {dec(x), inc(x'), dec(z)}

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

loop {inc(x), dec(x'), dec(z)}

loop {dec(x), inc(x'), dec(z)}

y := y-2

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

loop {inc(x), dec(x'), dec(z)}

loop {dec(x), inc(x'), dec(z)}

y := y-2

At the end check if y = z = 0

Triples

Assume (x', y, z) = (K, n, Kn)

Let x = 0

zero-test(x):

loop {inc(x), dec(x'), dec(z)}

y := y-2

At the end check if y = z = 0

Goal: compute ($F_k(n)$, m, $F_k(n)$ m)

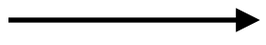
New technique

New technique

Assume we want to zero-test x three times

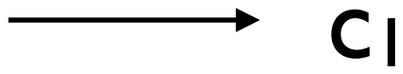
New technique

Assume we want to zero-test x three times



New technique

Assume we want to zero-test x three times

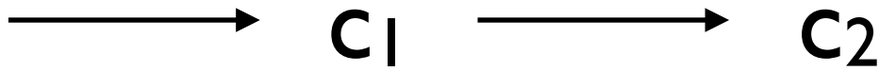


New technique

Assume we want to zero-test x three times

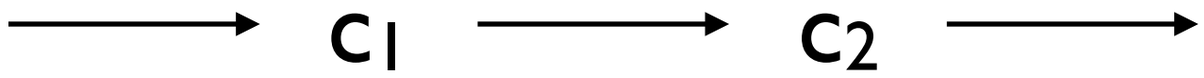
New technique

Assume we want to zero-test x three times



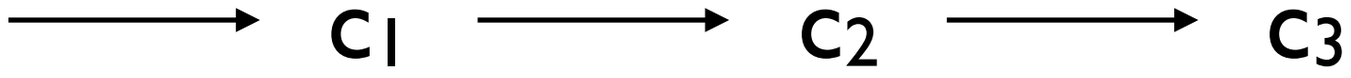
New technique

Assume we want to zero-test x three times



New technique

Assume we want to zero-test x three times

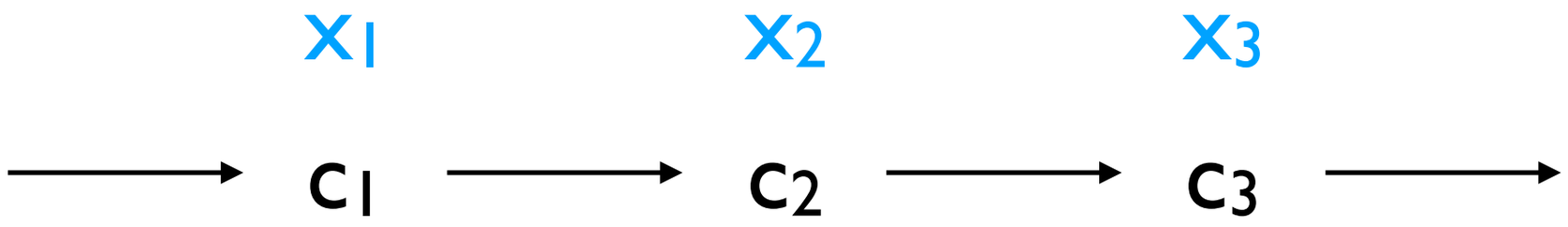


New technique

Assume we want to zero-test x three times

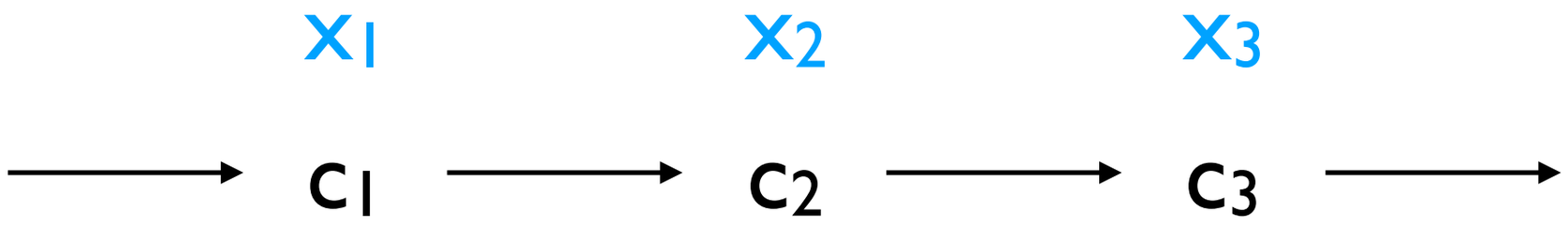
New technique

Assume we want to zero-test x three times



New technique

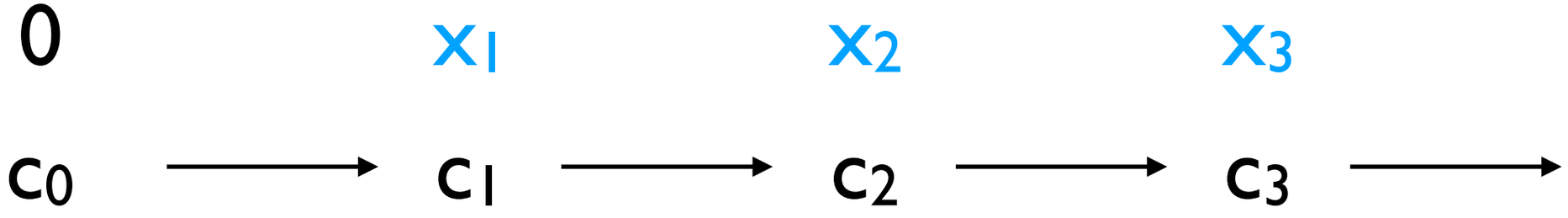
Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

New technique

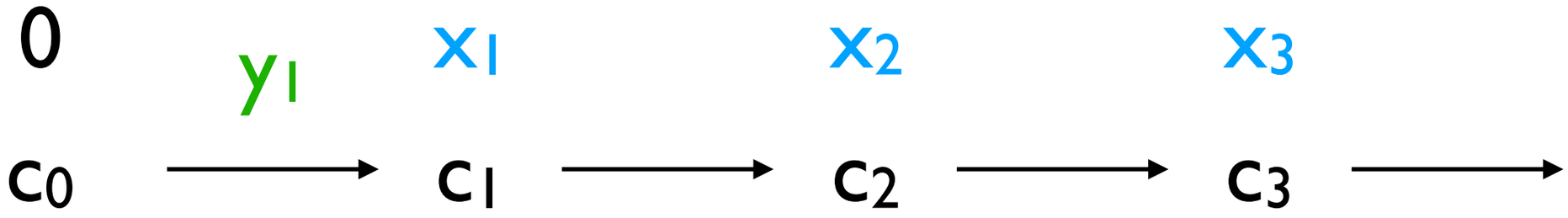
Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

New technique

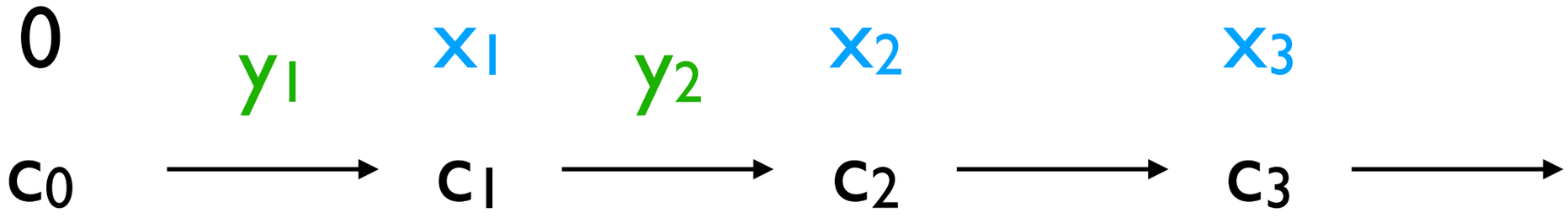
Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

New technique

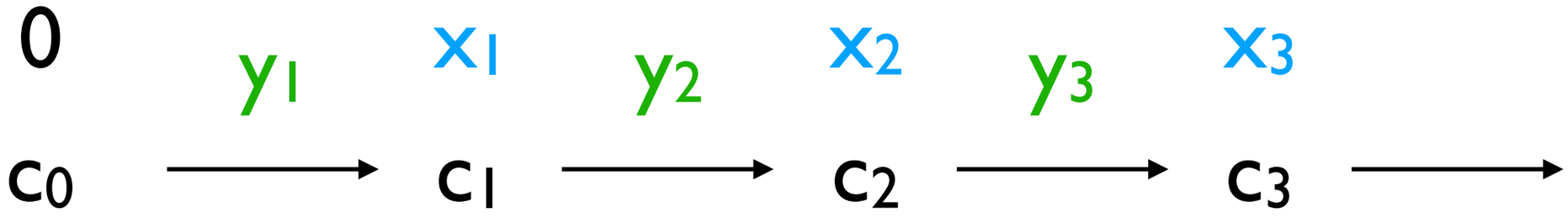
Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

New technique

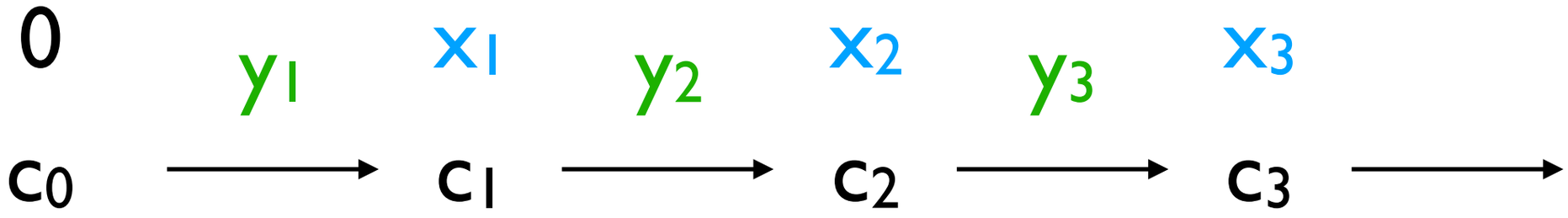
Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

New technique

Assume we want to zero-test x three times

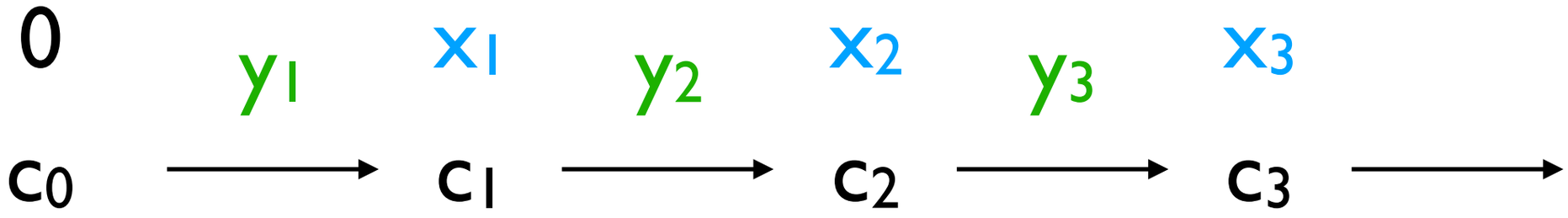


Enough to check if $x_1+x_2+x_3 = 0$

$x_1 = y_1$

New technique

Assume we want to zero-test x three times

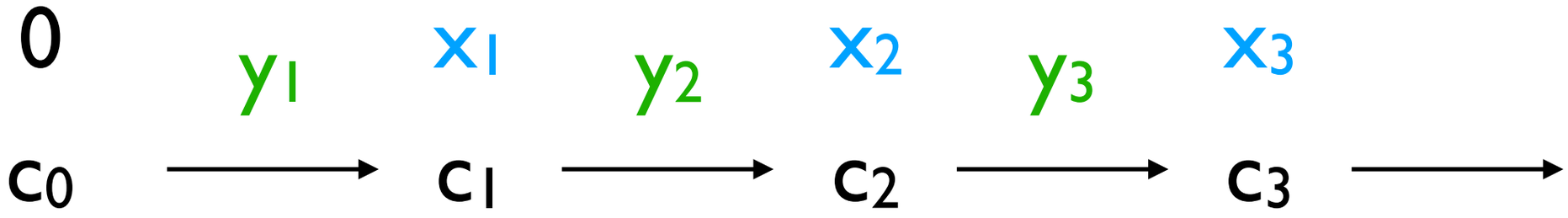


Enough to check if $x_1+x_2+x_3 = 0$

$x_1 = y_1$ $\qquad$ $x_2 = y_1+y_2$

New technique

Assume we want to zero-test x three times

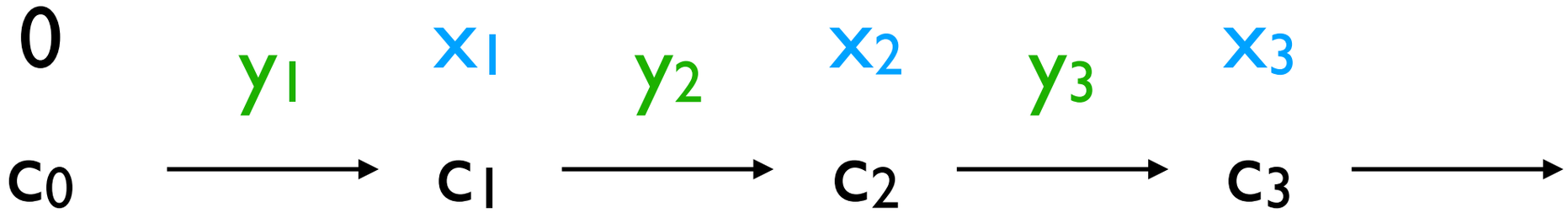


Enough to check if $x_1+x_2+x_3 = 0$

$x_1 = y_1$ $\quad$ $x_2 = y_1+y_2$ $\quad$ $x_3 = y_1+y_2+y_3$

New technique

Assume we want to zero-test x three times



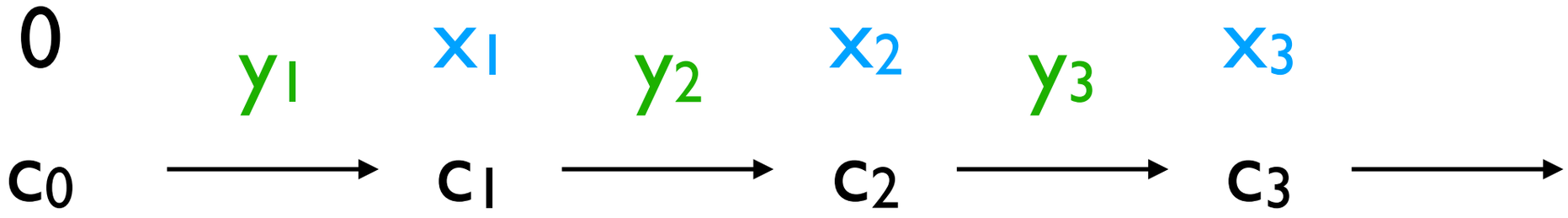
Enough to check if $x_1+x_2+x_3 = 0$

$x_1 = y_1$ $\quad$ $x_2 = y_1+y_2$ $\quad$ $x_3 = y_1+y_2+y_3$

Enough to check if $3y_1+2y_2+y_3 = 0$

New technique

Assume we want to zero-test x three times



Enough to check if $x_1+x_2+x_3 = 0$

$x_1 = y_1$ $\quad$ $x_2 = y_1+y_2$ $\quad$ $x_3 = y_1+y_2+y_3$

Enough to check if $3y_1+2y_2+y_3 = 0$

New counter c counts $3y_1+2y_2+y_3$

In general

In general

Many testing places for many counters

In general

Many testing places for many counters

One counter c

In general

Many testing places for many counters

One counter c

Counter c counts sum of all zero-tested places

In general

Many testing places for many counters

One counter c

Counter c counts sum of all zero-tested places

If x waits for k zero-tests
then inc(x) implies c+=k

Example 1

Example 1

```
for i=n downto 1 do
```

Example 1

```
for i=n downto 1 do
        loop
```

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1
```

Example 1

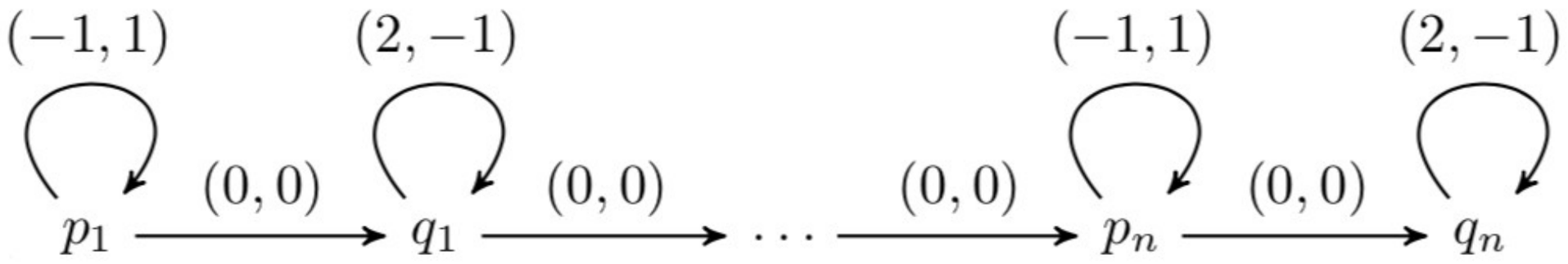
```
for i=n downto 1 do
        loop
            x-=1 y+=1
        loop
```

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
```

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
```



Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
```

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
```

from (1,0) we
can reach $(2^n,0)$

Example 1

```
for i=n downto 1 do
        loop
              x-=1 y+=1     c+=0
        loop
              x+=2 y-=1
```

from (1,0) we can reach $(2^n,0)$

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1     c+=0          0=i·(-1) + i·1
        loop
            x+=2 y-=1
```

from (1,0) we can reach $(2^n,0)$

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1     c+=0          0=i·(-1) + i·1
        loop
            x+=2 y-=1     c+=i-2
```

from (1,0) we can reach $(2^n,0)$

Example 1

```
for i=n downto 1 do
        loop
            x-=1 y+=1     c+=0          0=i·(-1) + i·1
        loop
            x+=2 y-=1     c+=i-2        i-2=(i-1)·2 + i·(-1)
```

from (1,0) we
can reach $(2^n,0)$

Example 1

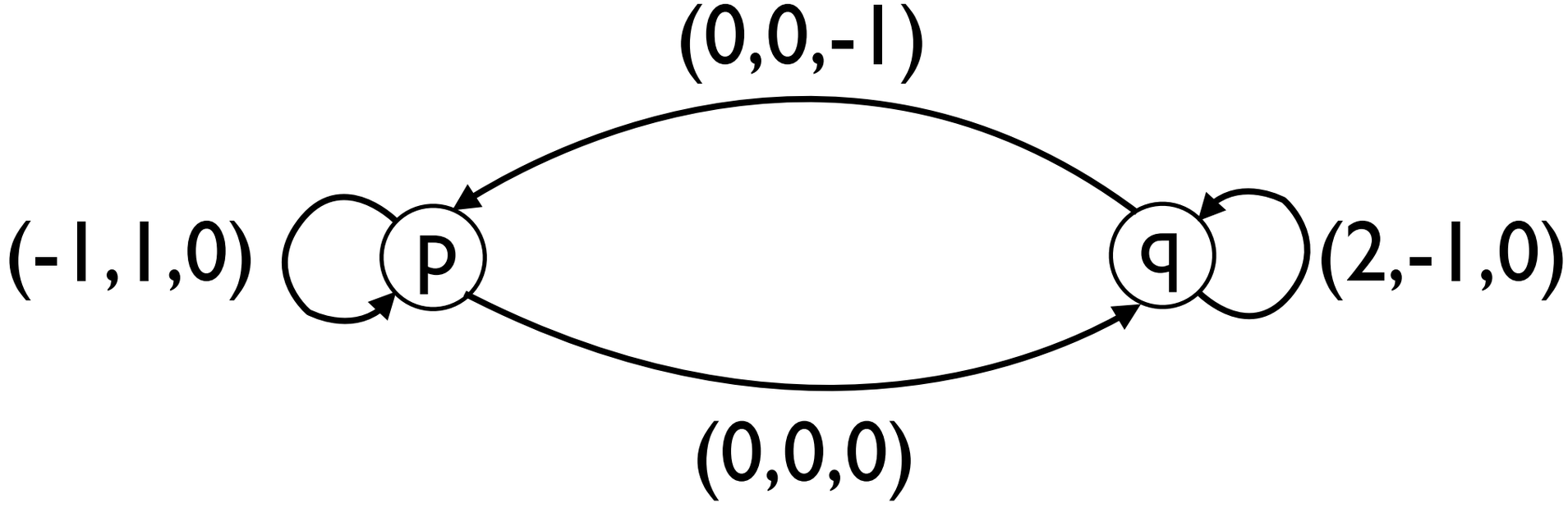
```
for i=n downto 1 do
        loop
            x-=1 y+=1     c+=0          0=i·(-1) + i·1
        loop
            x+=2 y-=1     c+=i-2        i-2=(i-1)·2 + i·(-1)
```

from (1,0) we
can reach $(2^n,0)$

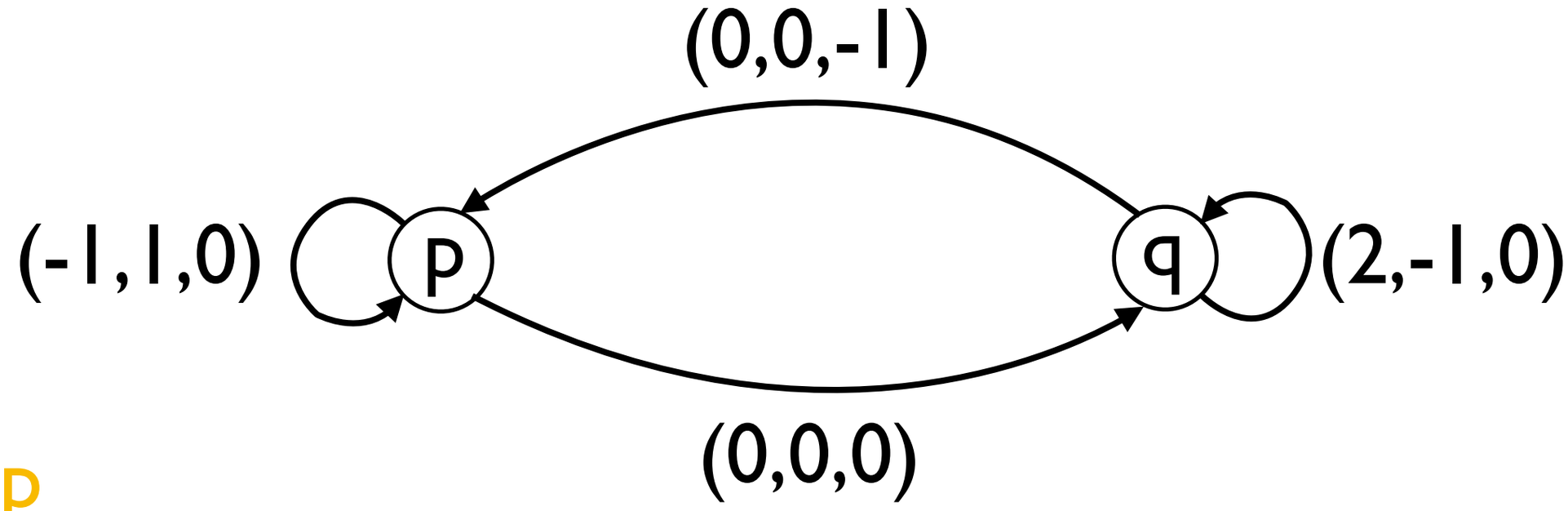
from (1,0,0) we
must reach $(2^n,0,0)$

Example 2

Example 2

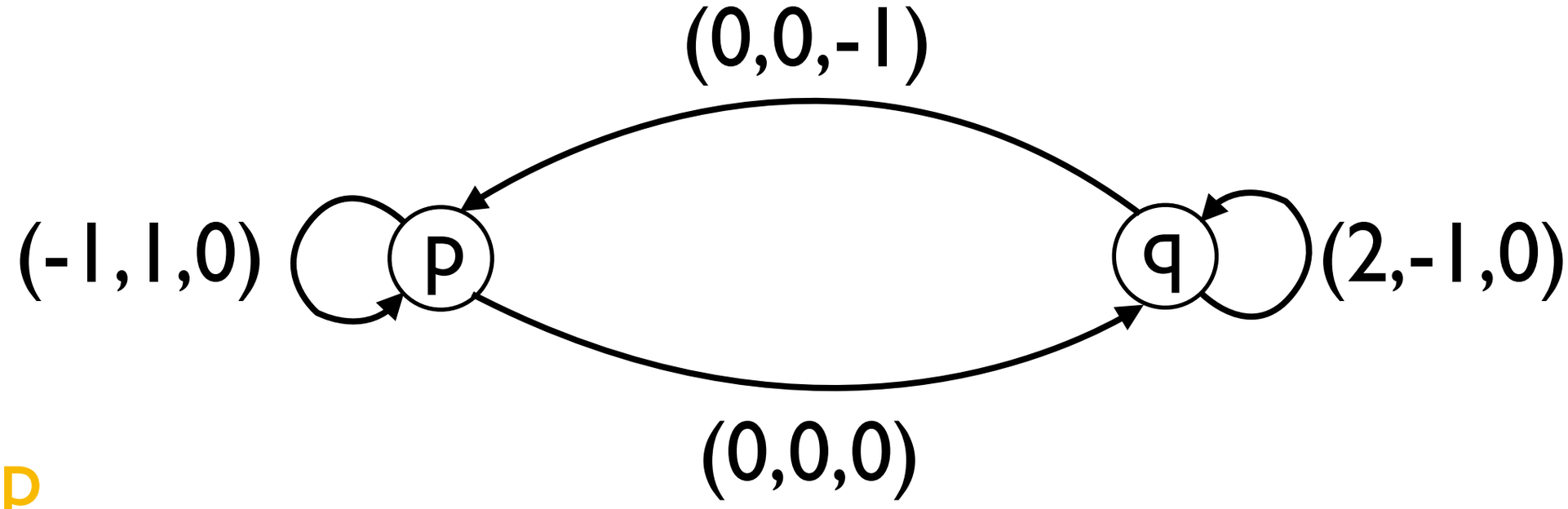


Example 2



loop

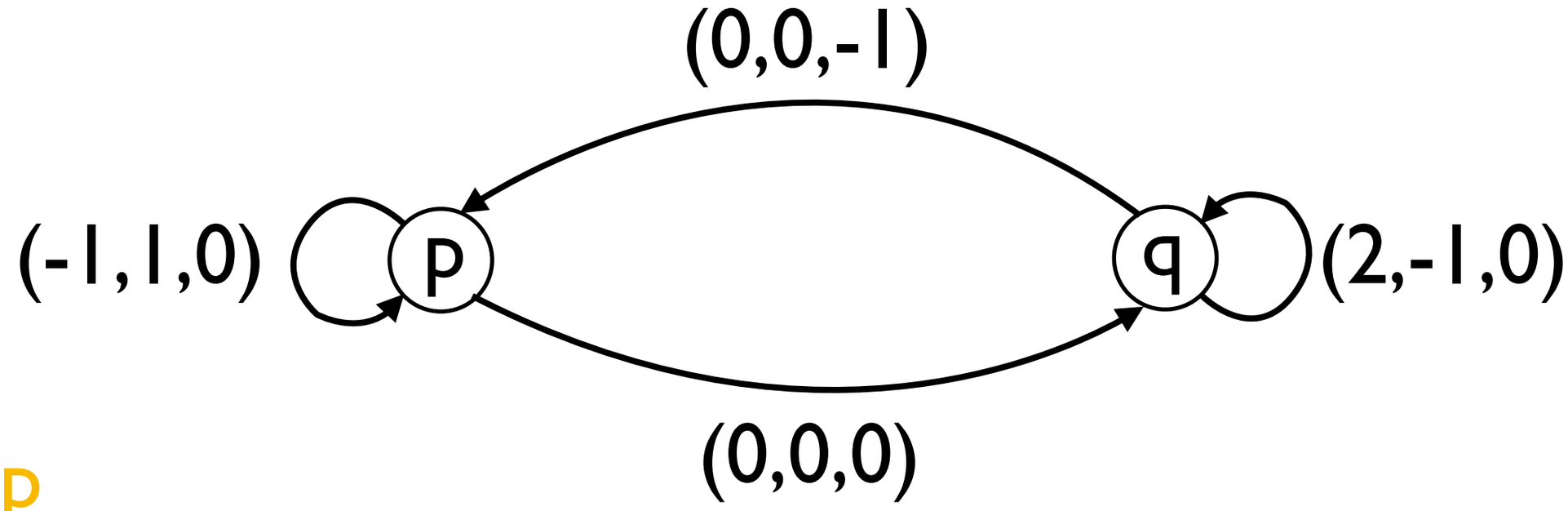
Example 2



loop

loop

Example 2

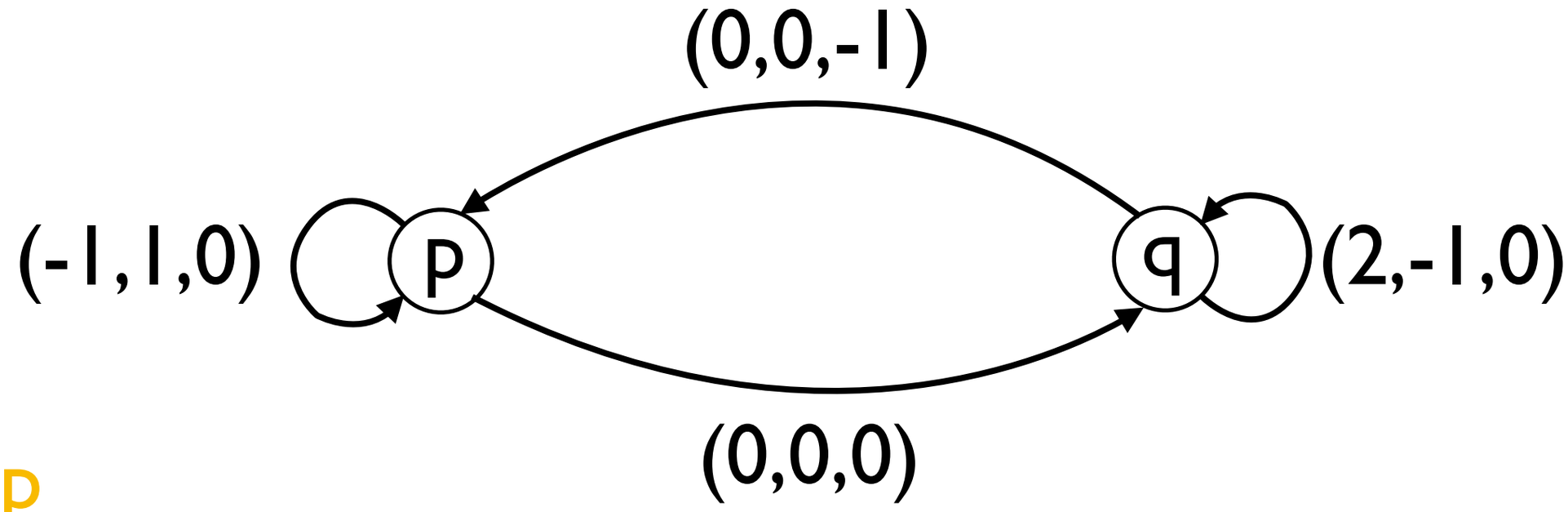


loop

loop

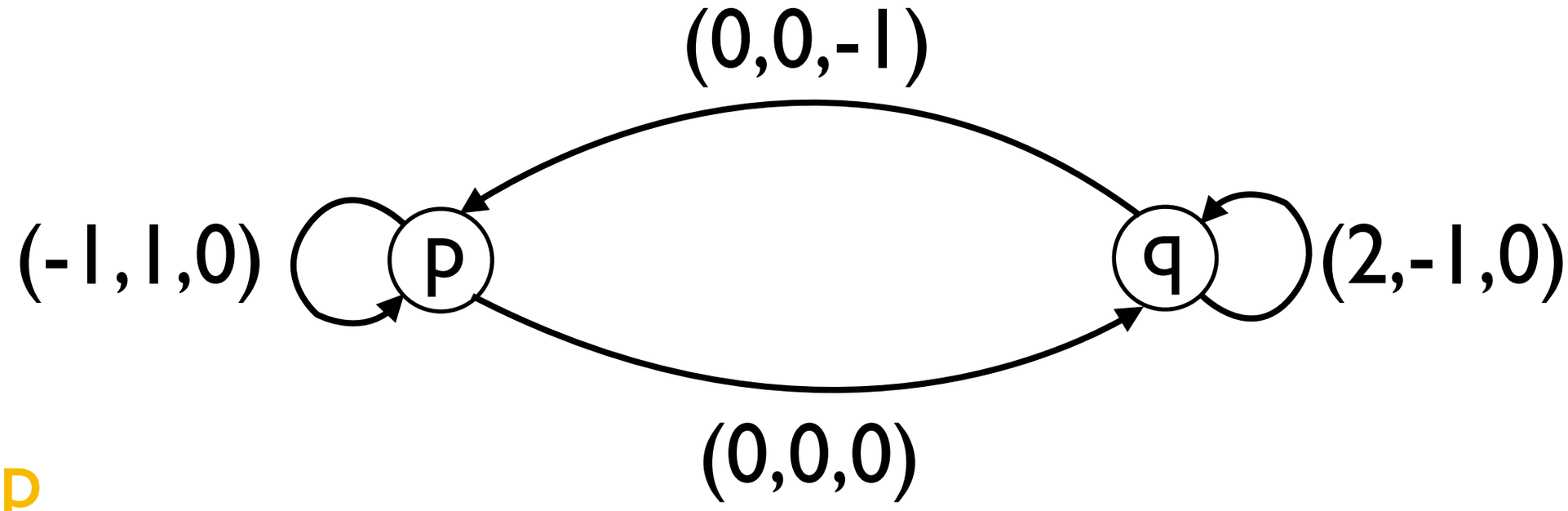
x-=1 y+=1

Example 2



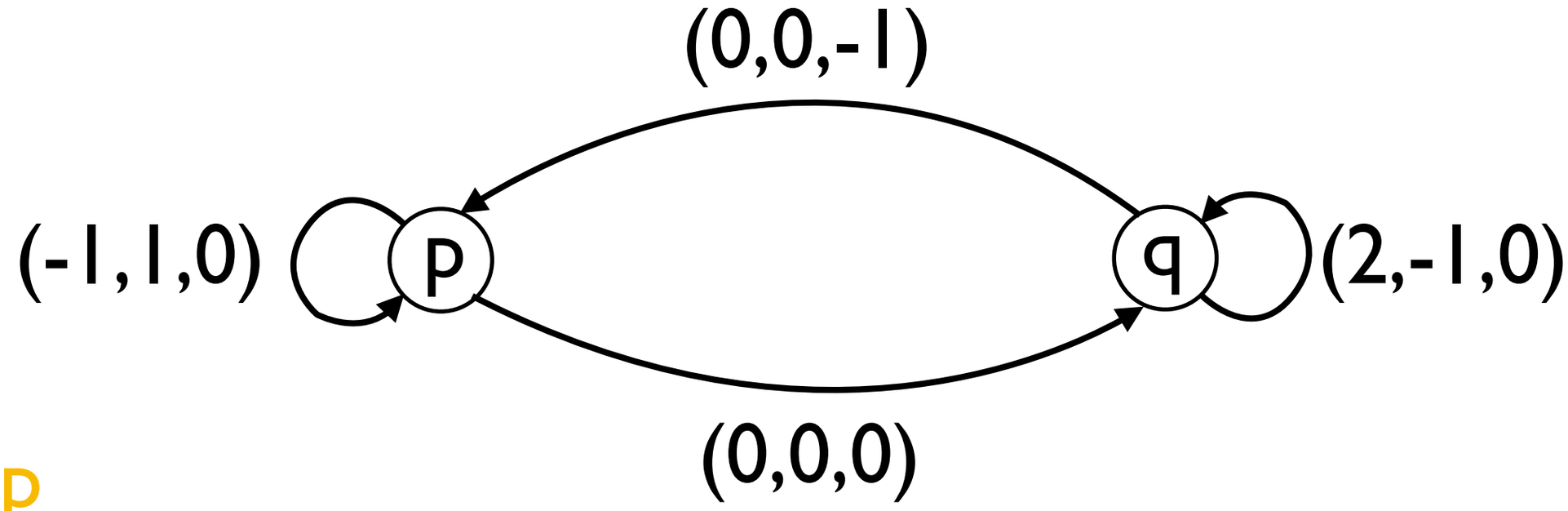

```
loop
    x-=1  y+=1
loop
```

Example 2



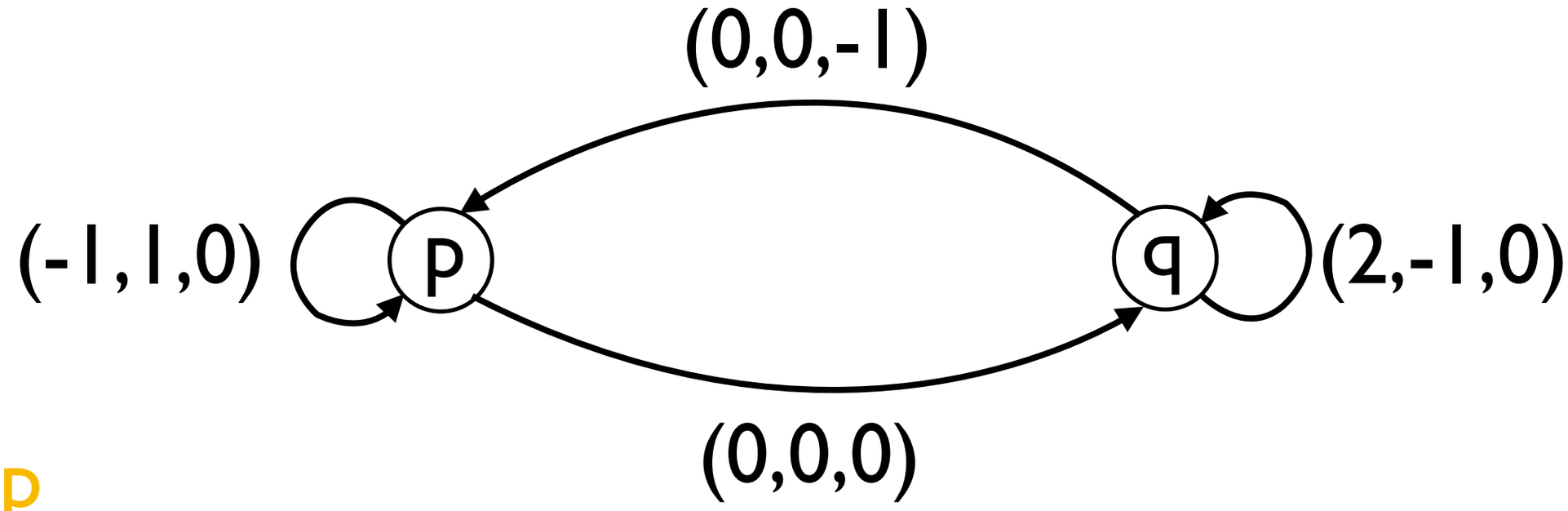

```
loop
    x-=1 y+=1
loop
    x+=2 y-=1
```

Example 2



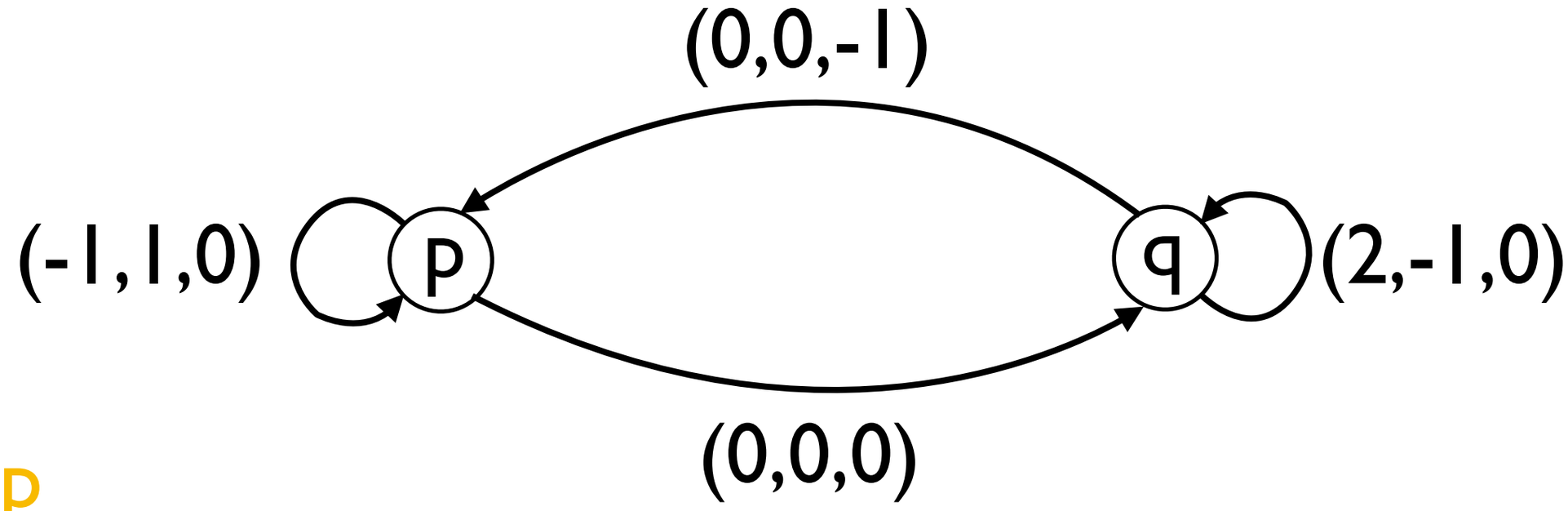

```
loop
      x-=1  y+=1
loop
      x+=2 y-=1
 z -=1
```

Example 2



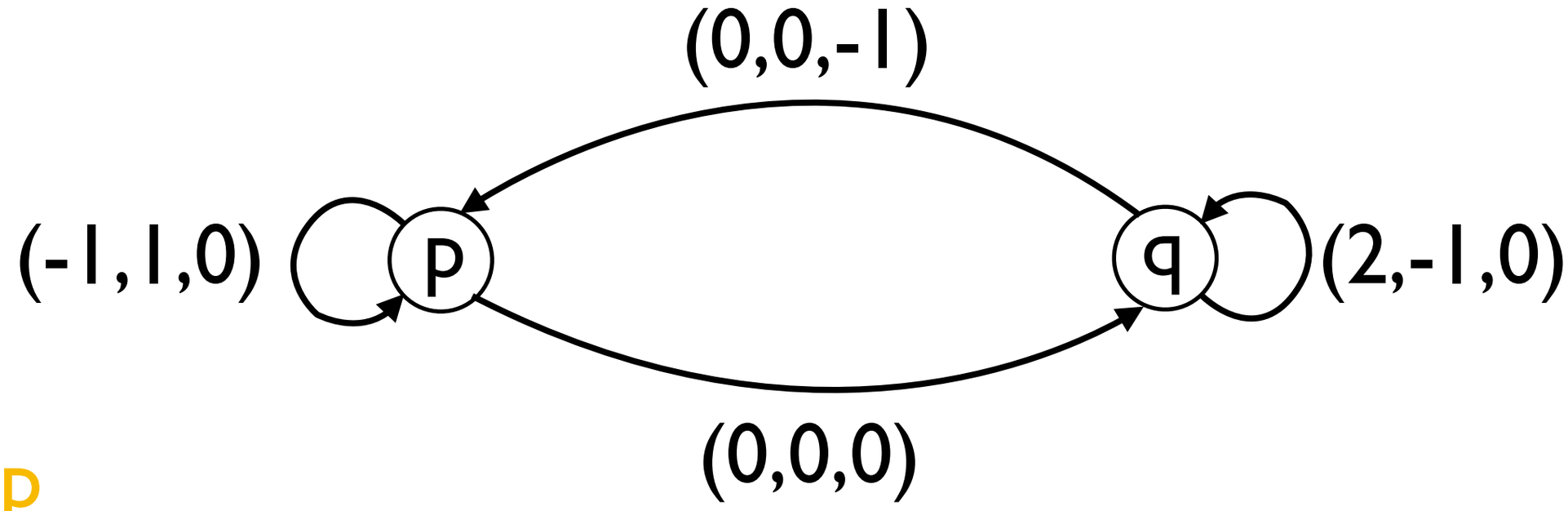

```
loop
      x-=1  y+=1     c+=0
loop
      x+=2 y-=1

 z -=1
```

Example 2




```
loop
    x-=1  y+=1     c+=0
loop
    x+=2 y-=1     c+=z-2
z -=1
```

Example 2



loop

loop

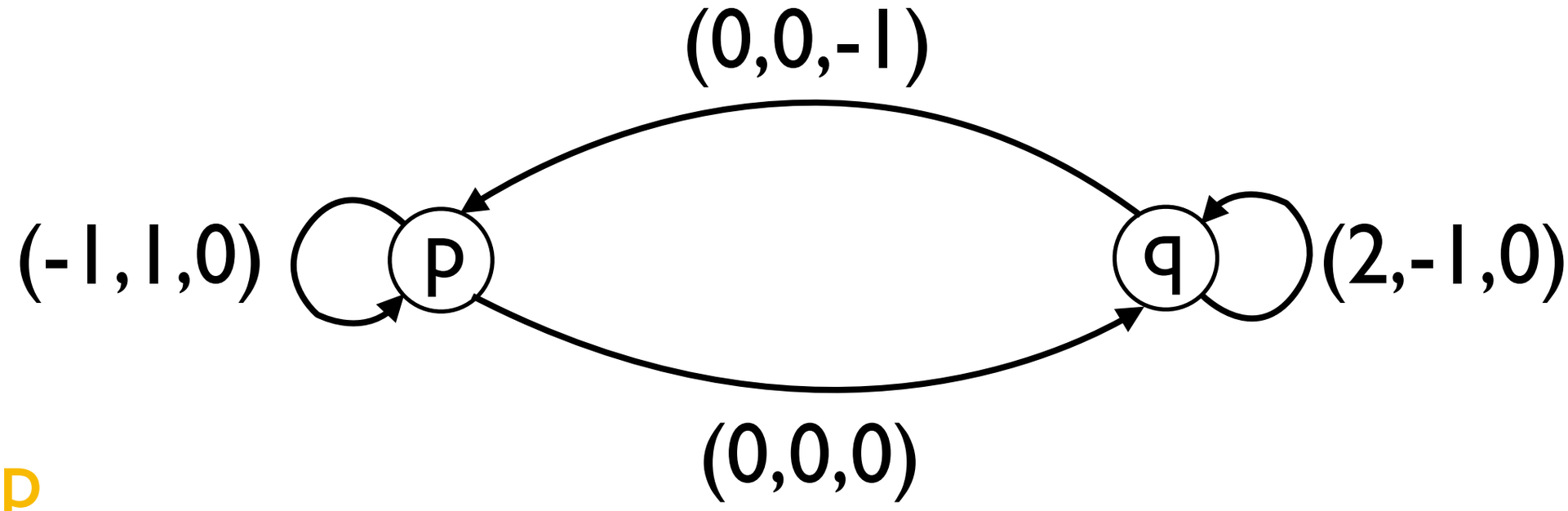
x-=1 y+=1 c+=0 ???

loop

x+=2 y-=1 c+=z-2

z -=1

Example 2



loop

loop

x-=1 y+=1 c+=0

loop

x+=2 y-=1 c+=z-2

z -=1

???

Start (1,0,n)

Finish (2^n,0,0)

Example 2

loop

Example 2

Example 2

```
loop
    loop
```

Example 2

```
loop
    loop
        x-=1 y+=1
```

Example 2

```
loop
    loop
        x-=1 y+=1
    loop
```

Example 2

```
loop
    loop
        x-=1 y+=1
    loop
        x+=2 y-=1
```

Example 2

```
loop
    loop
        x-=1 y+=1
    loop
        x+=2 y-=1
    z -=1
```

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1
    z -=1
```

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

Start (1,0,n)
Finish (2^n,0,0)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

Start (1,0,n)
Finish (2^n,0,0)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

```
    c+=z
```

Start (1,0,n)
Finish (2^n,0,0)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

```
loop
```

```
c+=z
```

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

zero-test(z)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

n-bounded

z'-= n

z'+= n

loop dec(z) inc(u) inc(c)

zero-test(z)

c+=z

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

zero-test(z)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

zero-test(z)

loop

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

zero-test(z)

loop inc(z) dec(u)

Example 2

```
loop
    loop
        x-=1 y+=1   c+=0
    loop
        x+=2 y-=1   c+=z-2
    z -=1
```

n-bounded

c+=z

Start (1,0,n)
Finish (2^n,0,0)

Add: z'=0, u=n, u'=0

Keep: z+z' = u+u' = n

loop dec(z) inc(u) inc(c)

zero-test(z)

loop inc(z) dec(u)

zero-test(u)

General strategy

General strategy

loop

General strategy

loop

loop

General strategy

loop

loop

loop

x-=1 y+=1

General strategy

```
loop
    loop
        loop
            x-=1 y+=1
    loop
```

General strategy

```
loop
    loop
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
```

General strategy

```
loop
    loop
        loop
            x-=1 y+=1
        loop
            x+=2 y-=1
z -=1
```

General strategy

loop

loop

x-=1 y+=1 modify(c)

loop

x+=2 y-=1

z -=1

General strategy

loop

loop

x-=1 y+=1 modify(c)

loop

x+=2 y-=1 modify(c)

z -=1

General strategy

loop

loop

x-=1 y+=1 modify(c)

loop

x+=2 y-=1 modify(c)

z -=1

n-bounded

General strategy

loop

loop

x-=1 y+=1 modify(c)

modify(c)

z -= 1

n-bounded

General strategy

loop

loop

x-=1 y+=1 modify(c)

x=2y if y:=0 modify(c)

z -=1

n-bounded

General strategy

loop

loop

x-=1 y+=1 modify(c)

modify(c)

z -= 1

n-bounded

General strategy

loop

loop

x-=1 y+=1 modify(c)

x=f(y) if y:=0 modify(c)

z -= 1

n-bounded

General strategy

loop

loop

x-=1 y+=1 modify(c)

x=f(y) if y:=0 modify(c)

z -=1

n-bounded

Start (1,0,n)

Finish ($f^n(1)$,0,0)

General strategy

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

from algorithm computing ($F_{k-1}(n)$, s, $F_{k-1}(n)$ s)

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

from algorithm computing ($F_{k-1}(n)$, s, $F_{k-1}(n)$ s)

$F_k(n) = F_{k-1} \circ \ldots \circ F_{k-1}(1)$

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

from algorithm computing ($F_{k-1}(n)$, s, $F_{k-1}(n)$ s)

$F_k(n) = F_{k-1} \circ \ldots \circ F_{k-1}(1)$

Idea:

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

from algorithm computing ($F_{k-1}(n)$, s, $F_{k-1}(n)$ s)

$F_k(n) = F_{k-1} \circ \ldots \circ F_{k-1}(1)$

Idea: start from triple (1, a, a)

General strategy

Goal: find algorithm computing ($F_k(n)$, m, $F_k(n)$ m)

from algorithm computing ($F_{k-1}(n)$, s, $F_{k-1}(n)$ s)

$F_k(n) = F_{k-1} \circ \ldots \circ F_{k-1}(1)$

Idea: start from triple (1, a, a)

n times apply F_{k-1} to 1

Thank you!