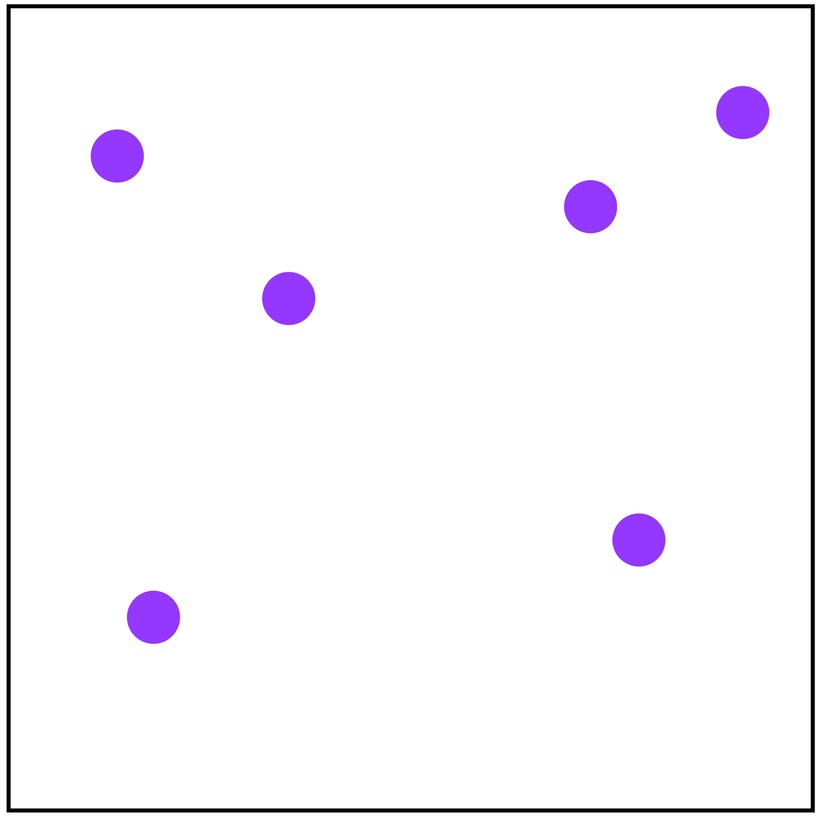
A Framework for Compositional Transformations of Recursion and Loops

Milind Kulkarni
Electrical and Computer Engineeri[illegible]sity

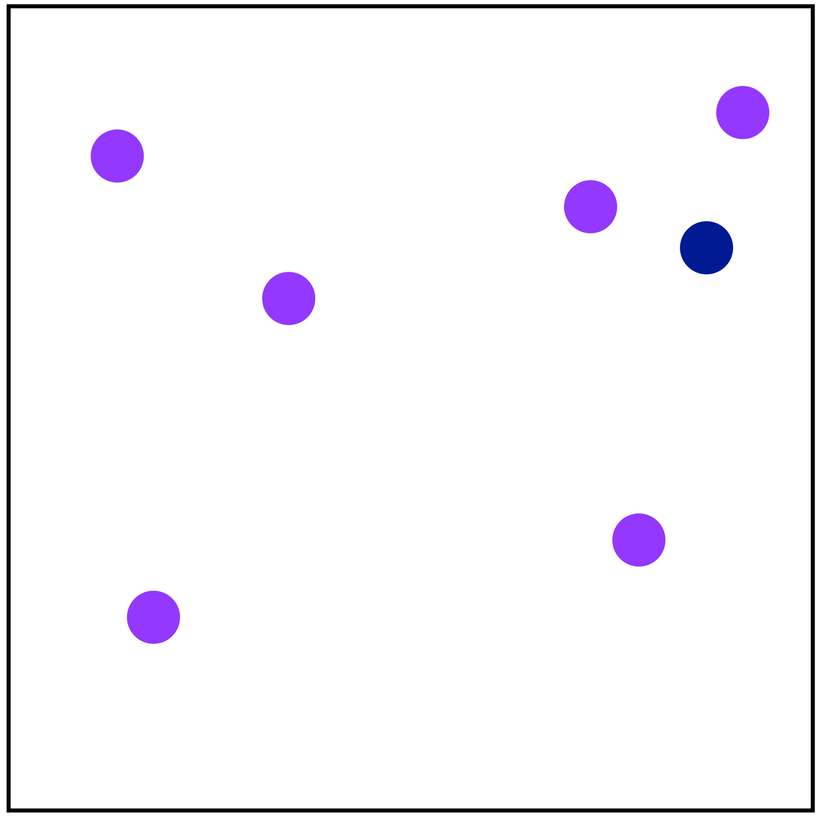
Joint work with
Kirshanthan Sundararajah

point correlation



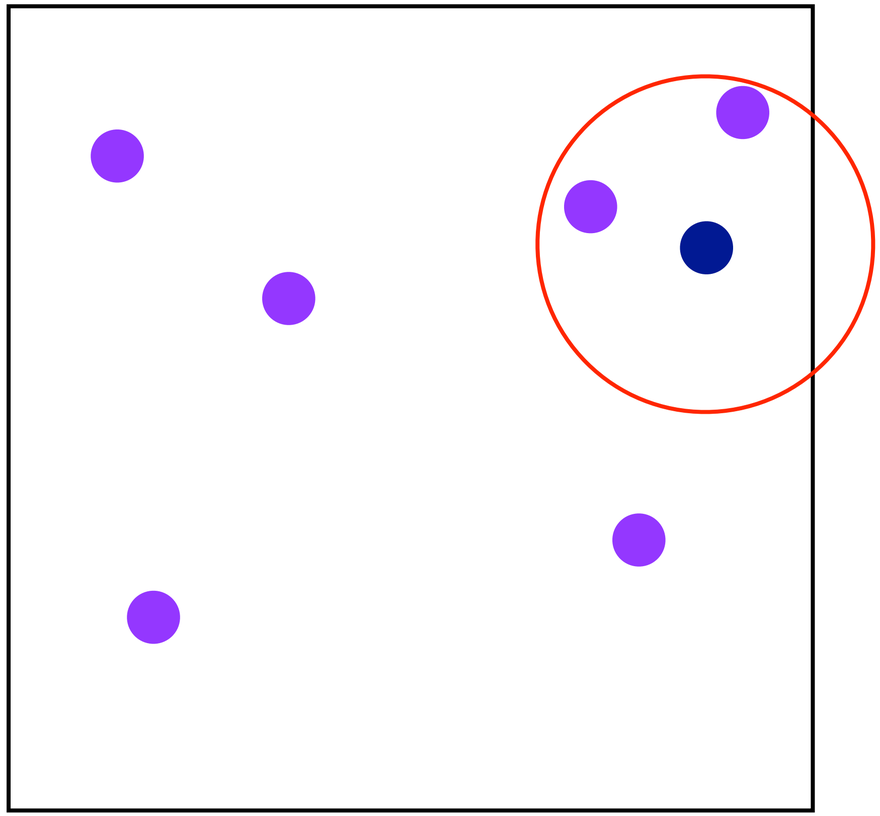
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

point correlation



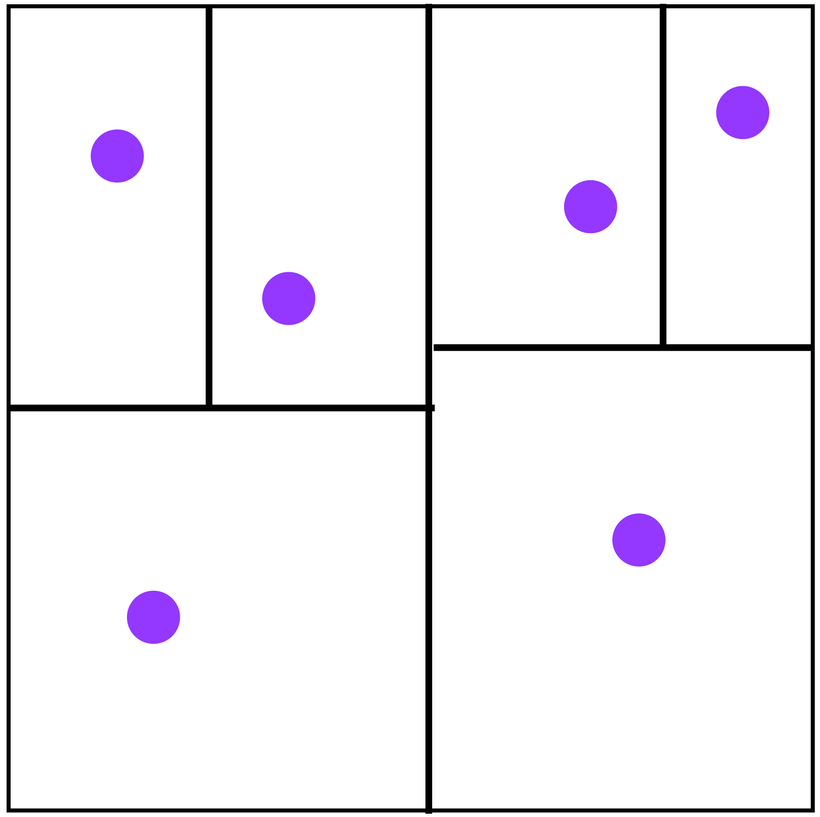
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

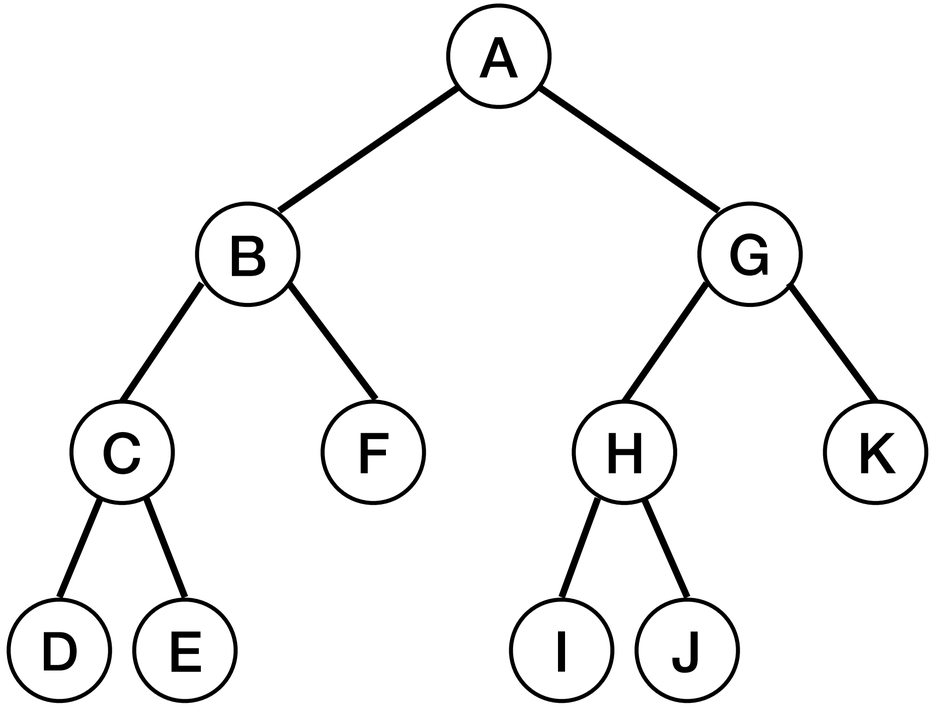
point correlation



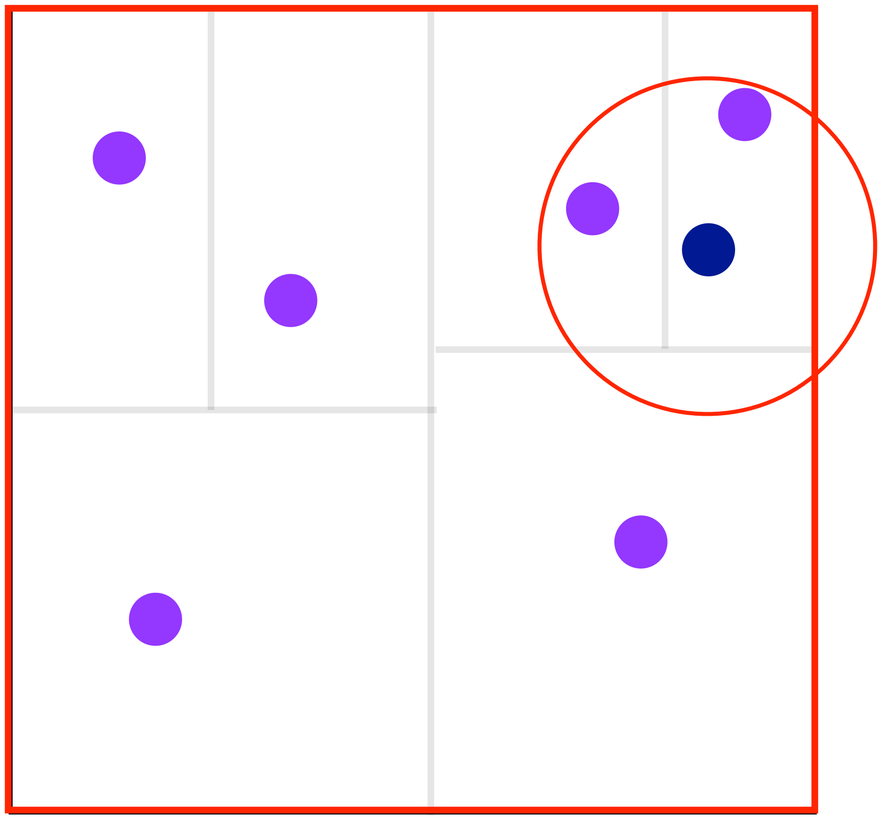
- Data mining algorithm
- Goal: given a set of ***N*** points in *k* dimensions and a point ***p***, find all points within a radius ***r*** of ***p***
- Naïve approach: compare all ***N*** points with ***p***
- Better approach: build *kd-tree* over points, traverse tree for point ***p***, prune subtrees that are far from ***p***

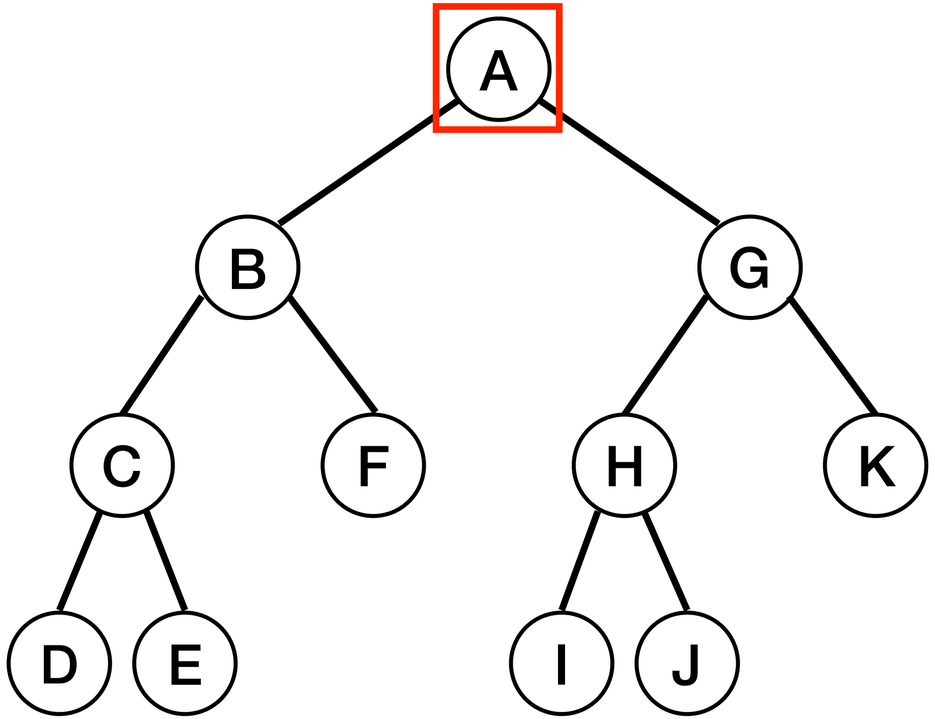
point correlation



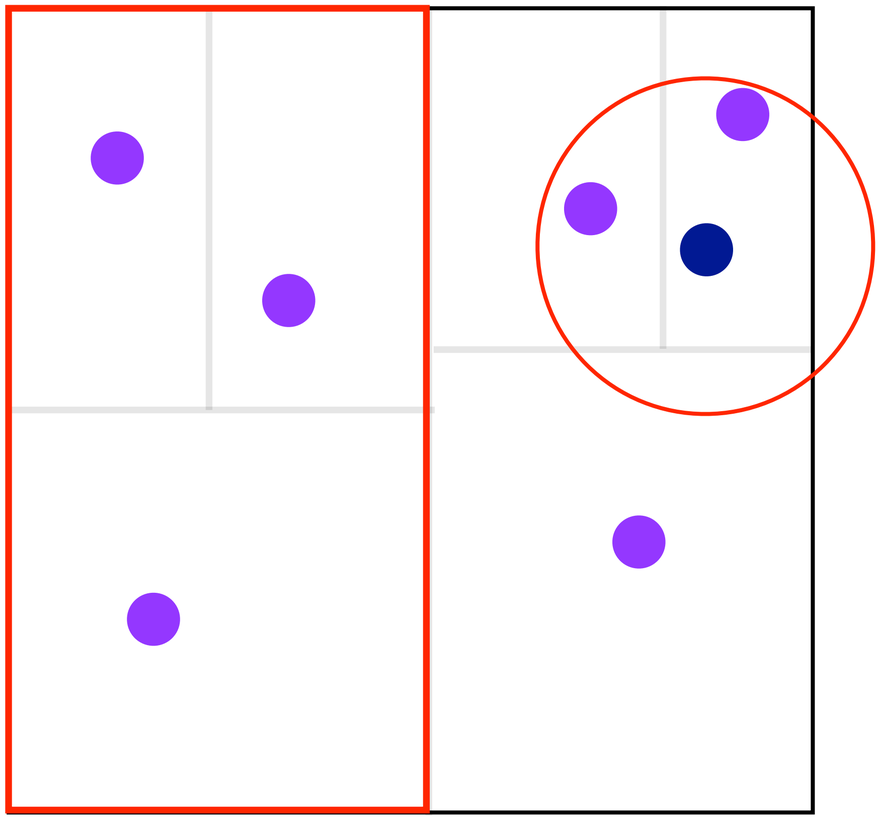


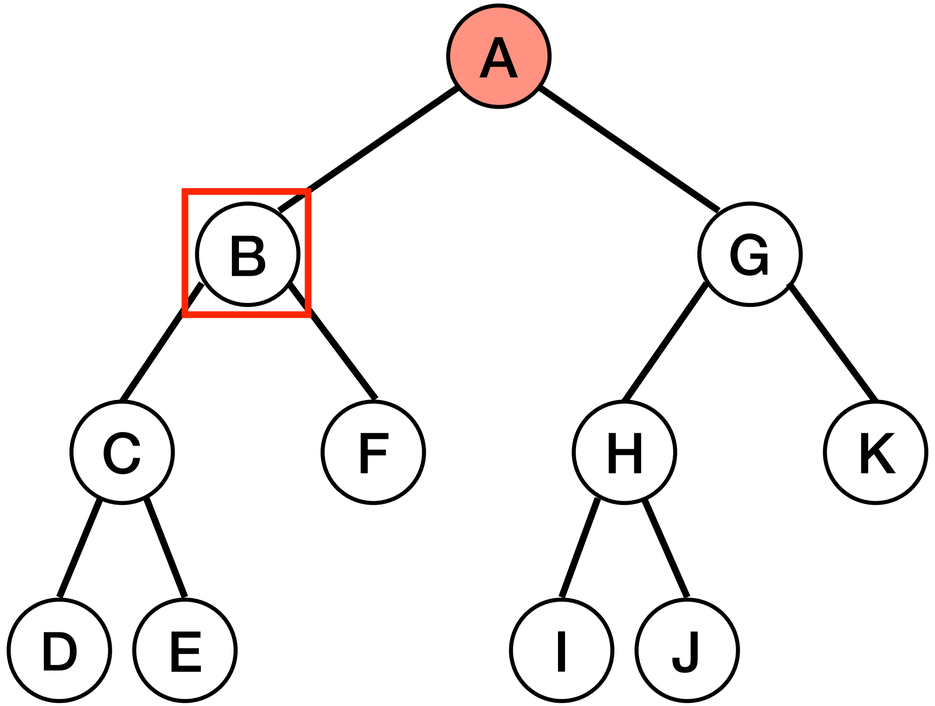
point correlation



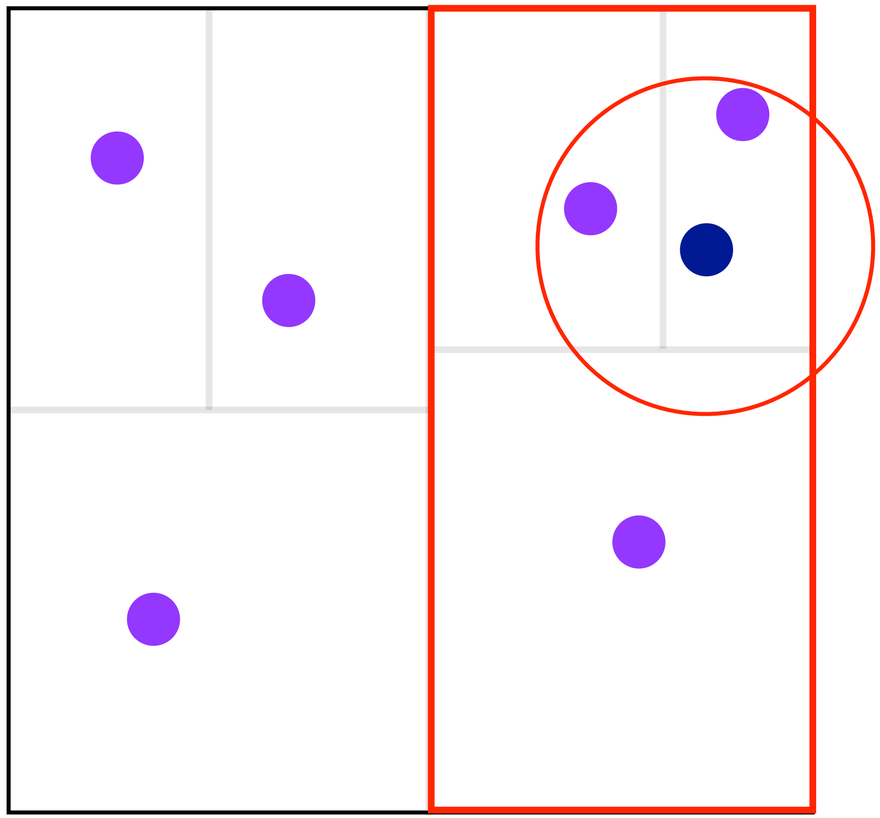


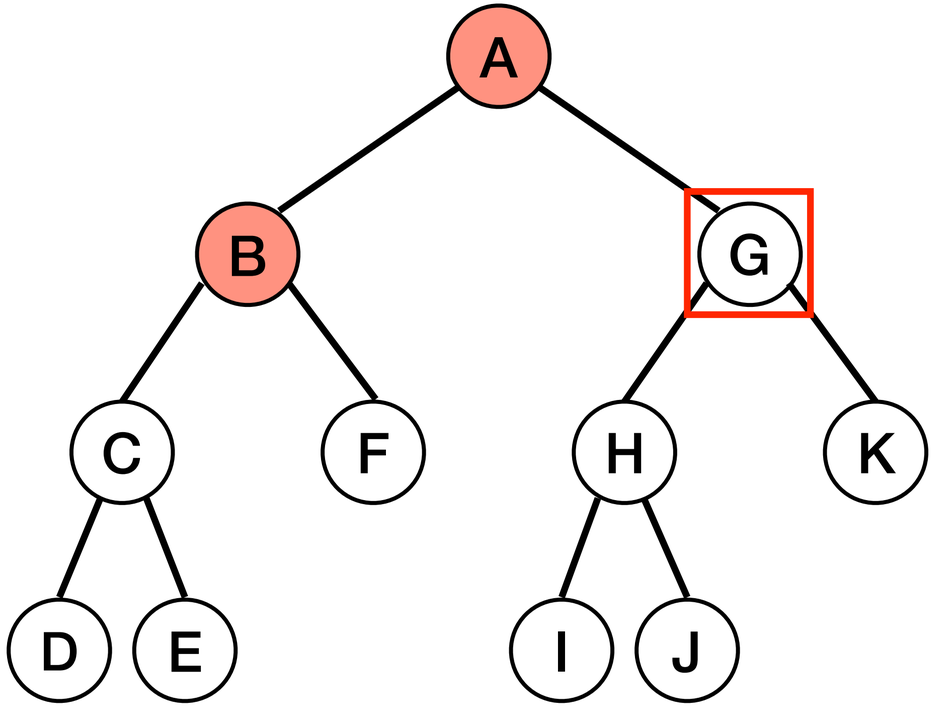
point correlation



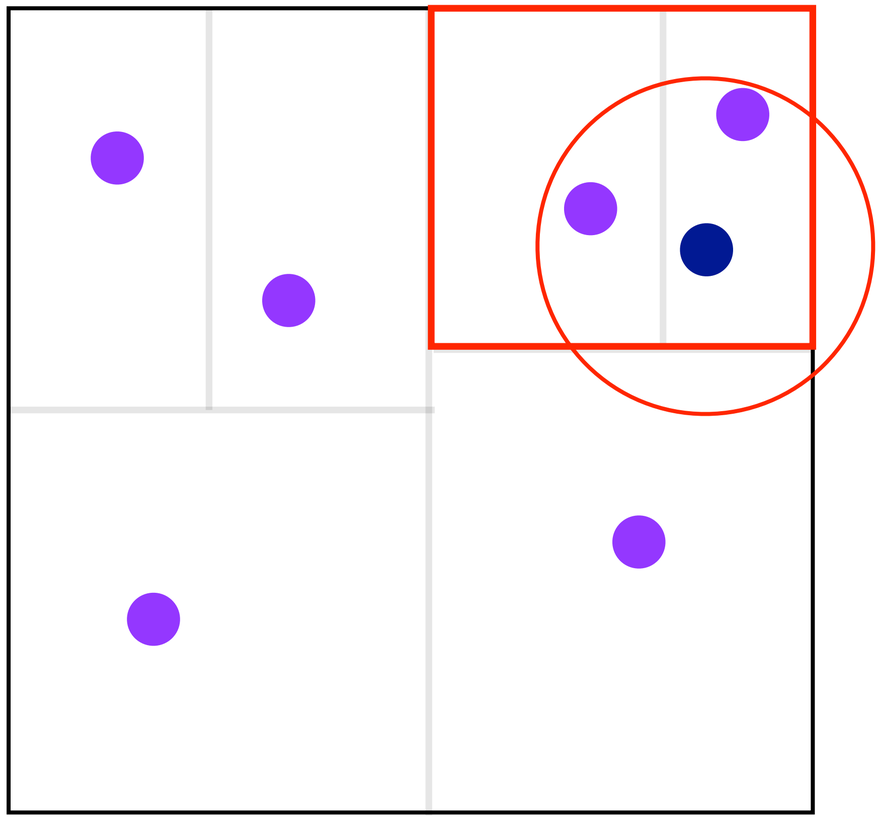


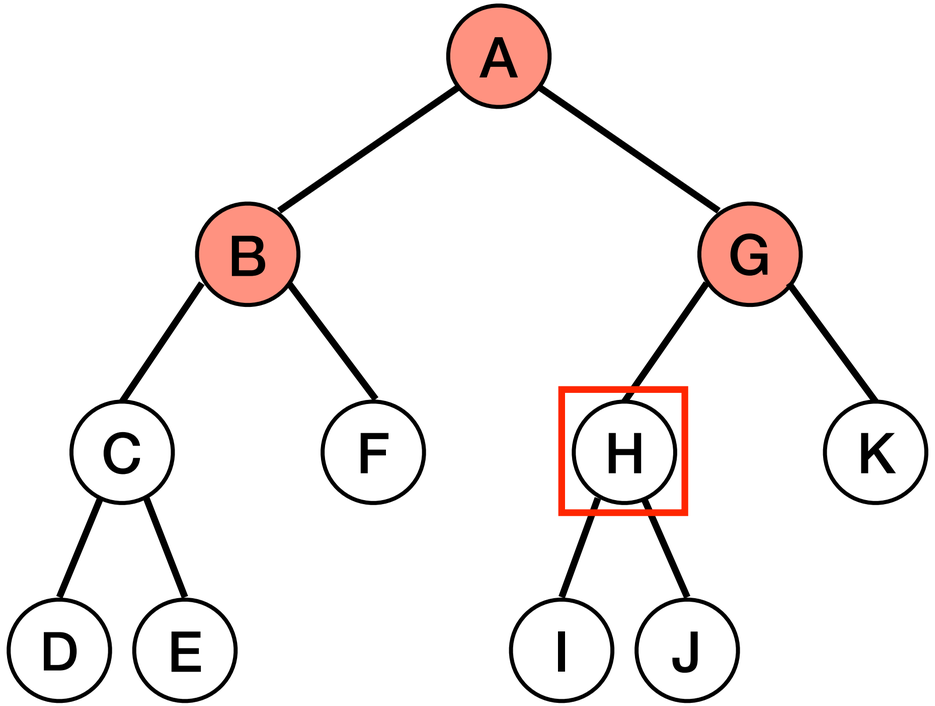
point correlation



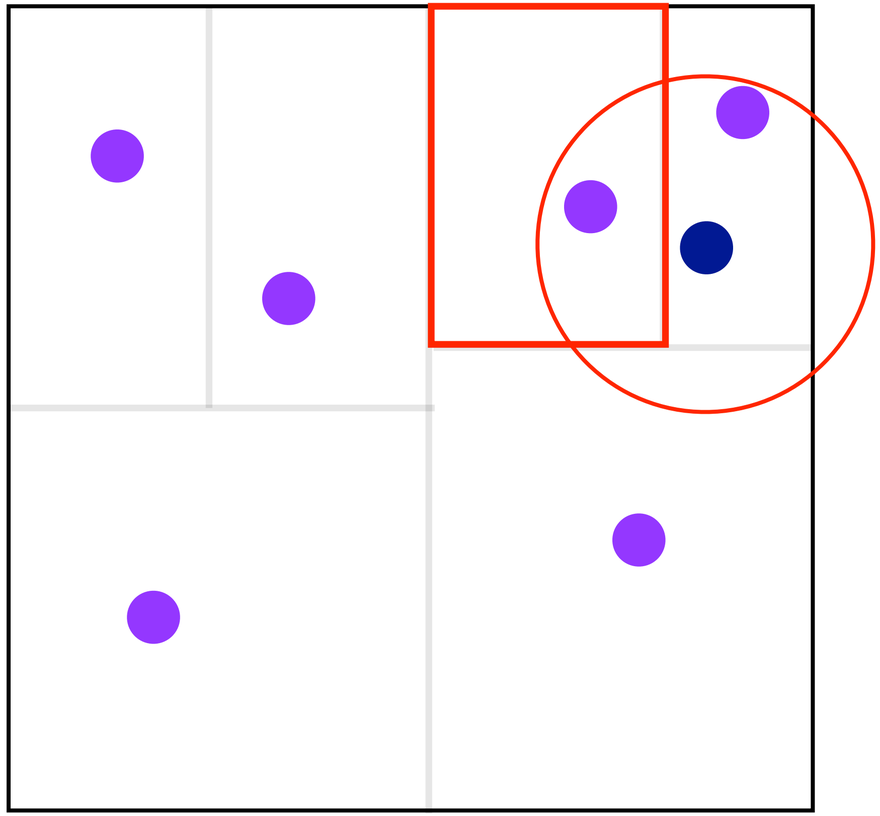


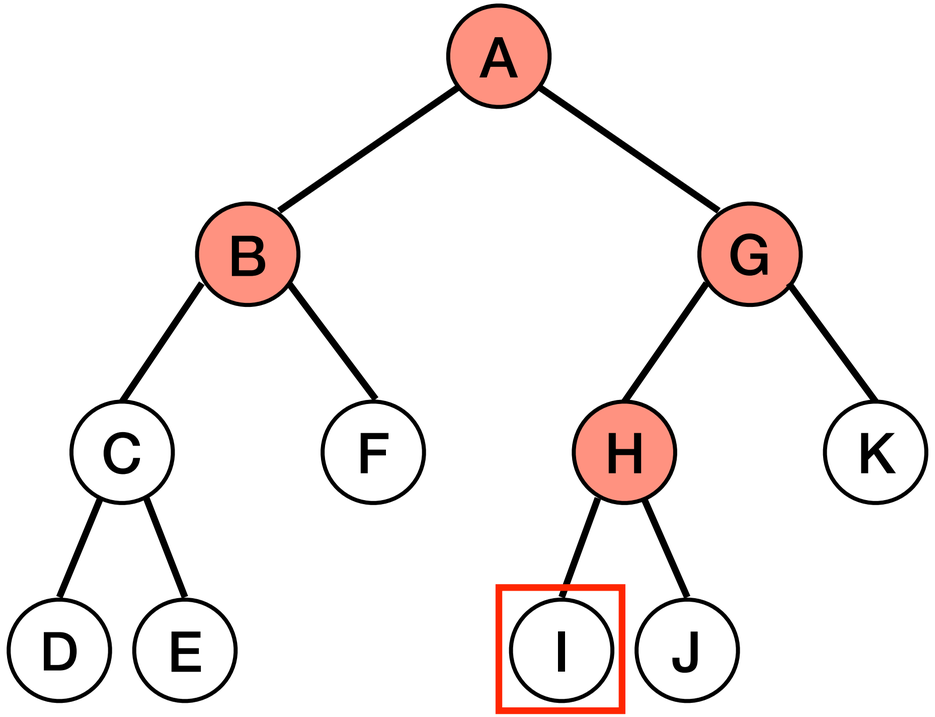
point correlation



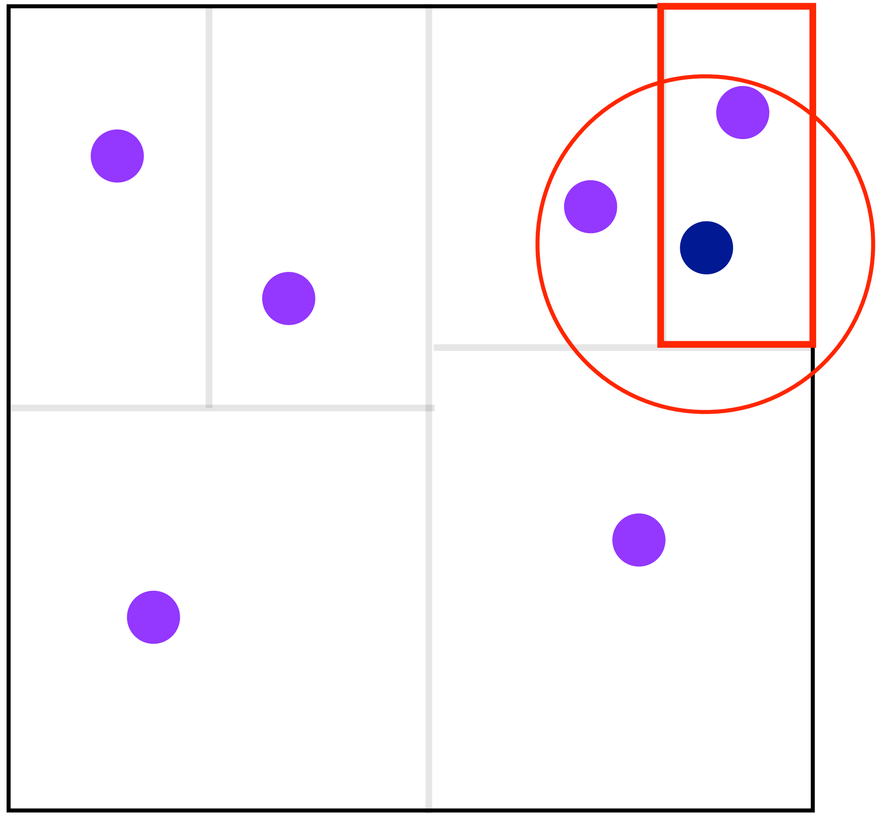


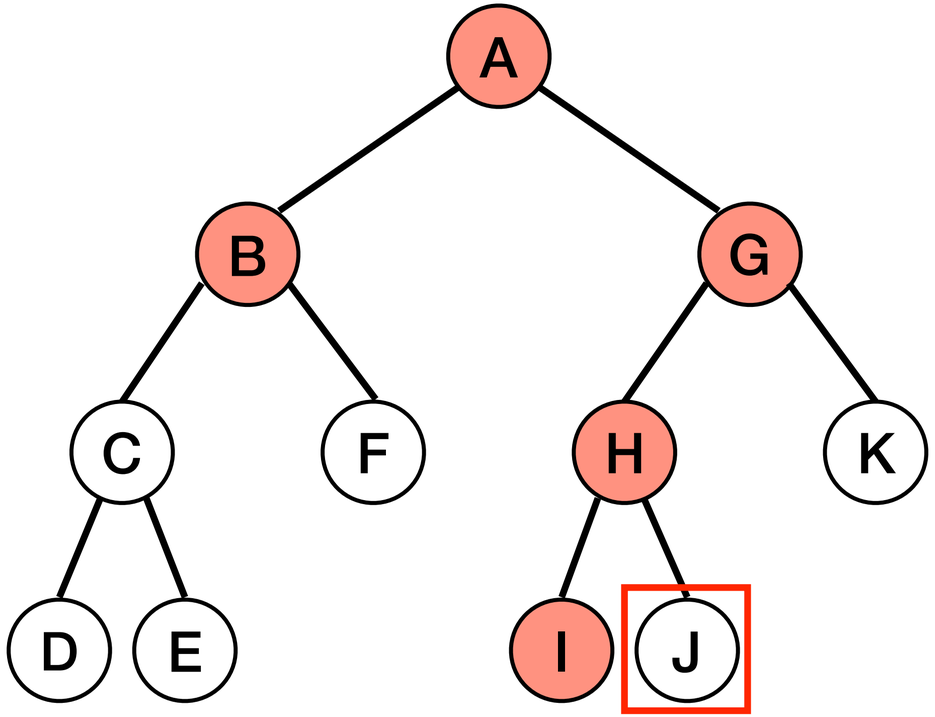
point correlation



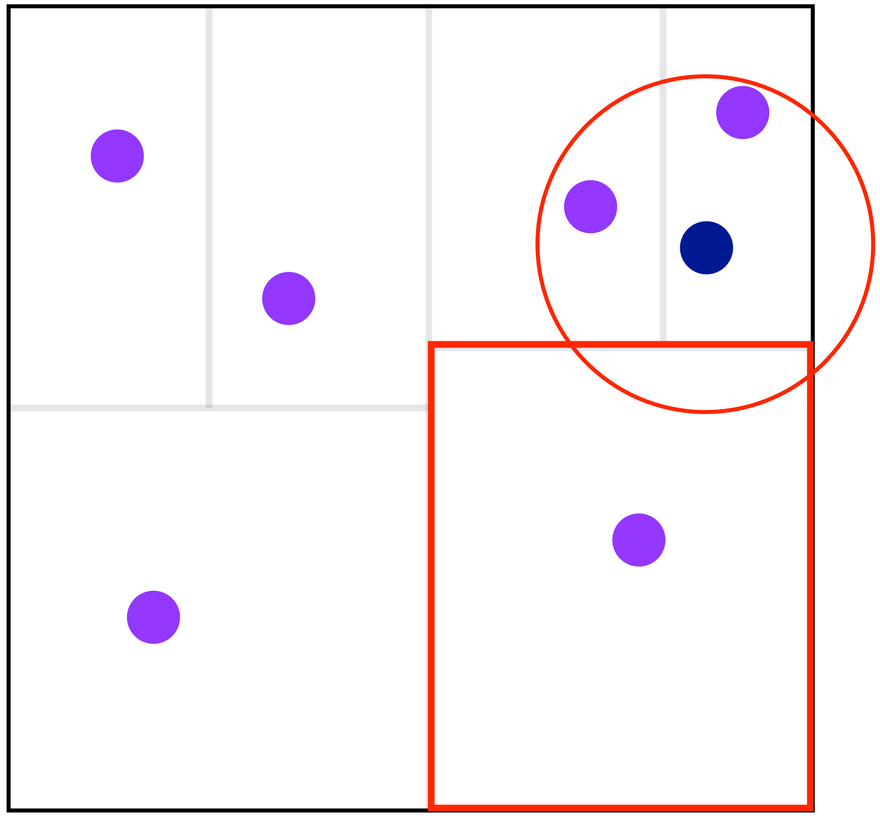


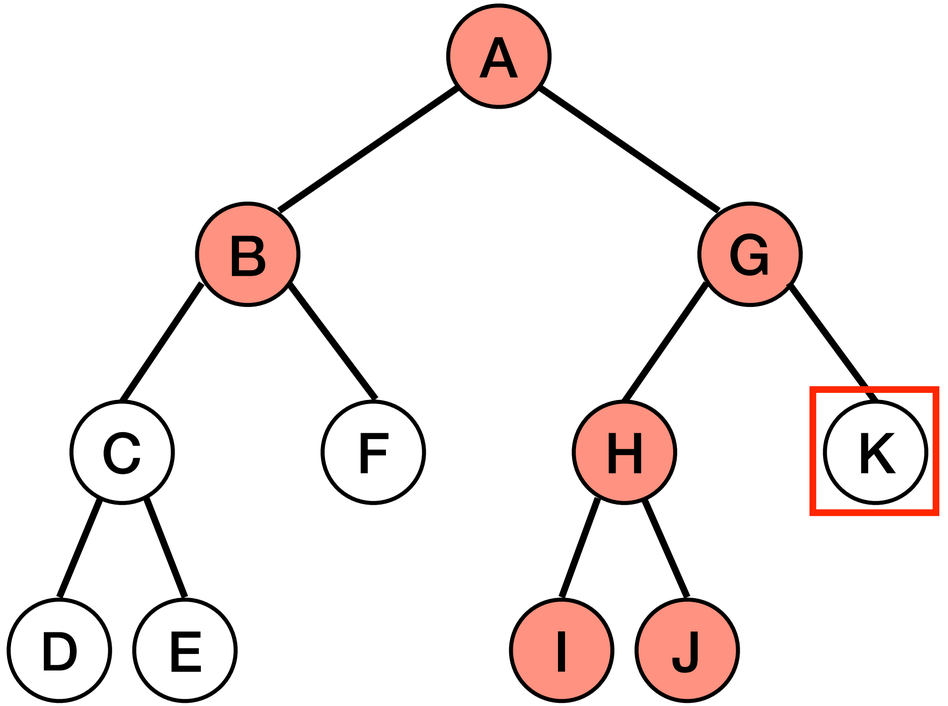
point correlation





point correlation





point correlation

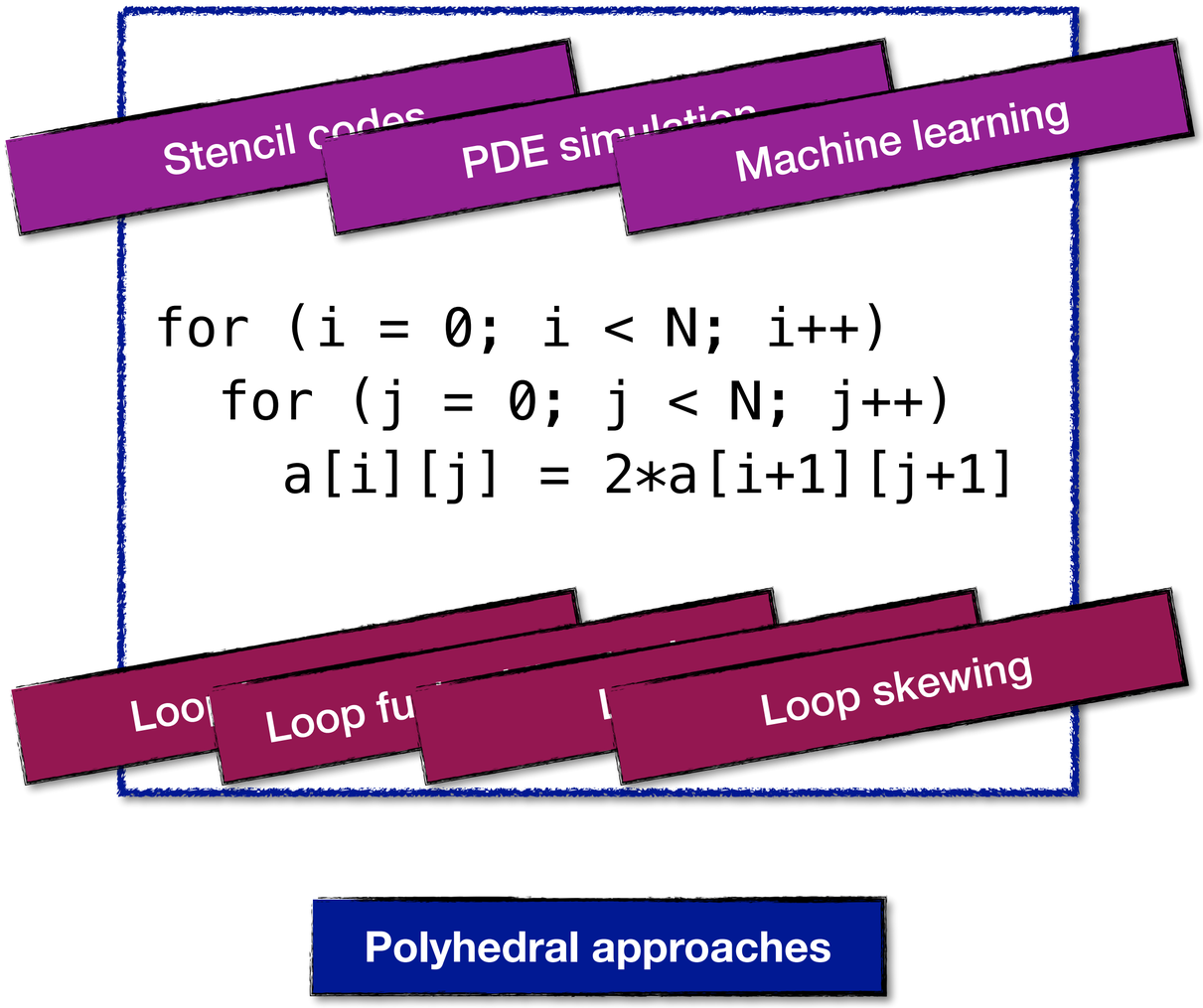
```
KDCell root = /* build kdtree */;
Set<Point> ps;
double radius;

foreach Point p in ps {
   recurse(p, root, radius);
}
...
void recurse(Point p, KDCell node, double r) {
   if (tooFar(p, node, r)) return;
   if (node.isLeaf() && (dist(node.point, p) < r))
      p.correlated++;
   else {
      recurse(p, node.left, r);
      recurse(p, node.right, r);
   }
}
```

other recursive applications

- Barnes-Hut
- K-nearest neighbor
- Collision detection
- Point cloud algorithms
- Tree joins
- Dual-tree data mining algorithms

```
for (i = 0; i < N; i++)
  traverse(i, root)

traverse(int i, node * n)
  if (!n) return;
  compute(i, n)
  traverse(i, n.l)
  traverse(i, n.r)
```

loops and recursion

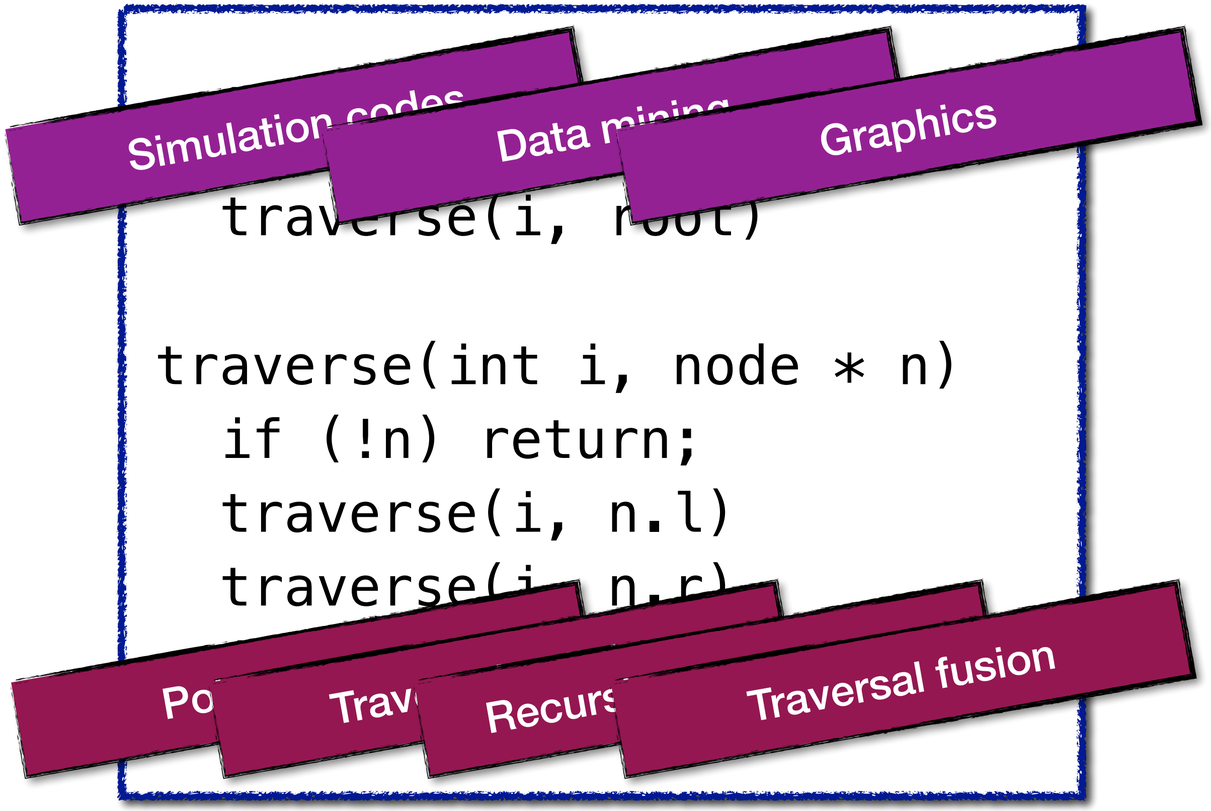


Polyhedral approaches

loops and recursion

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```

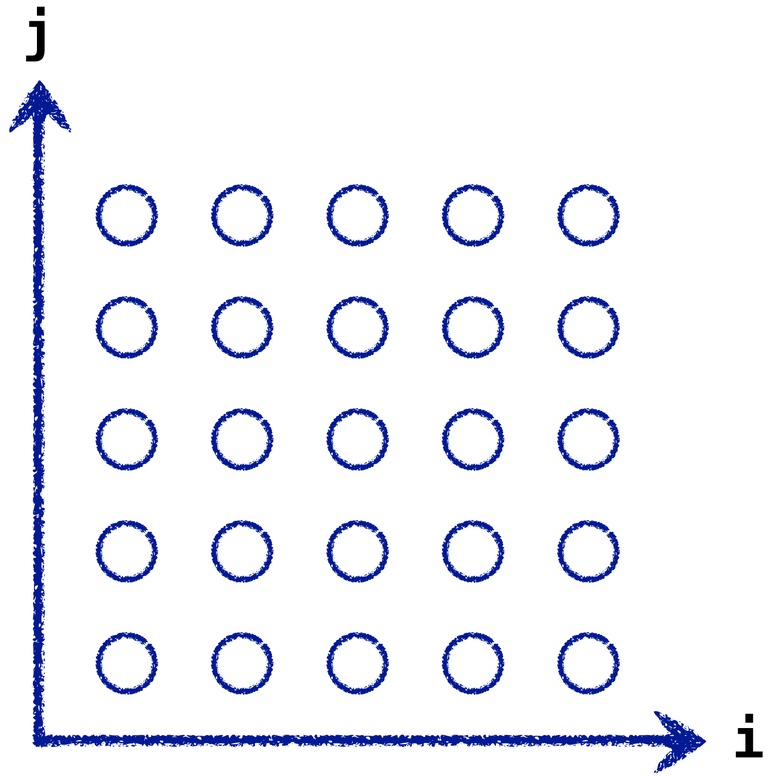
Polyhedral approaches



???

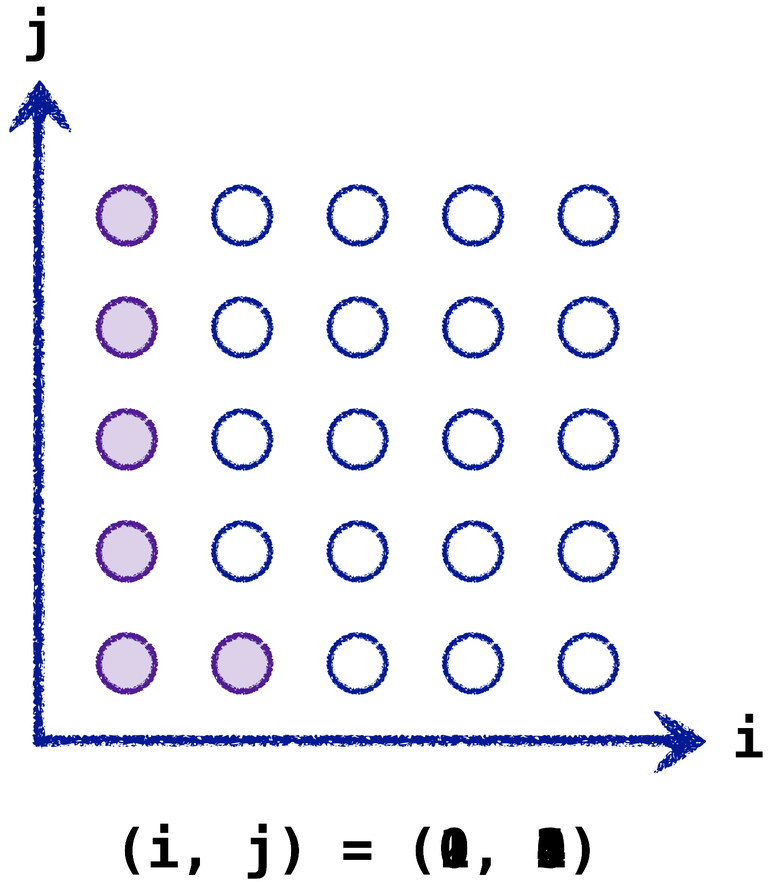
instance-wise reasoning

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```



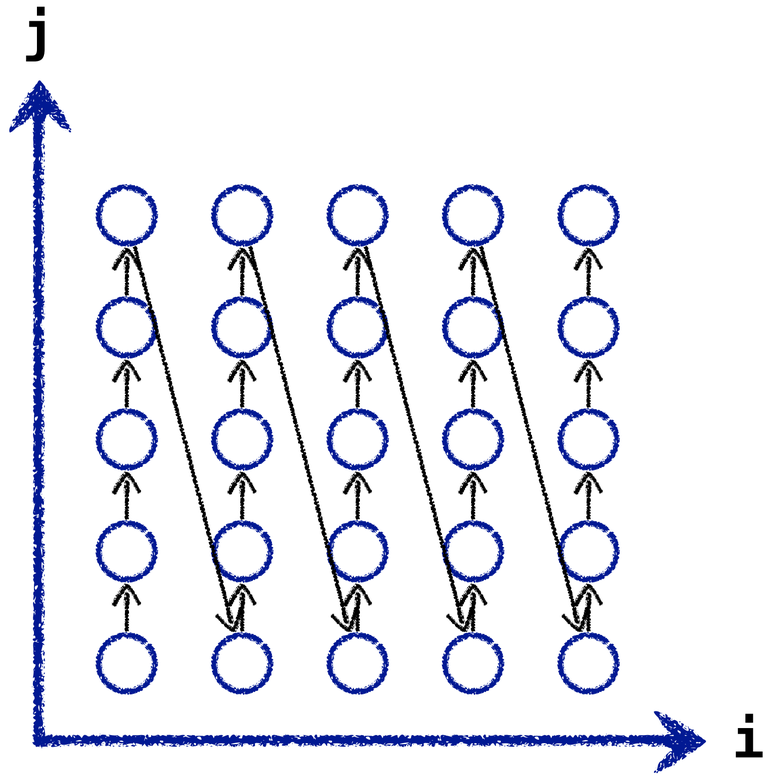
instance-wise reasoning

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```



instance-wise reasoning

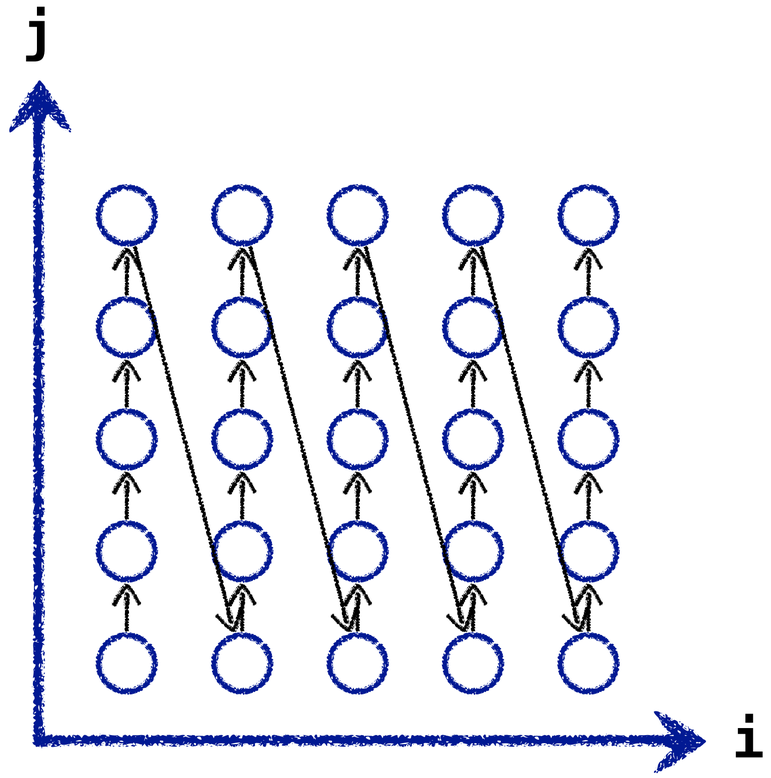
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```



instance-wise reasoning

```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    a[i][j] = 2*a[i+1][j+1]
```

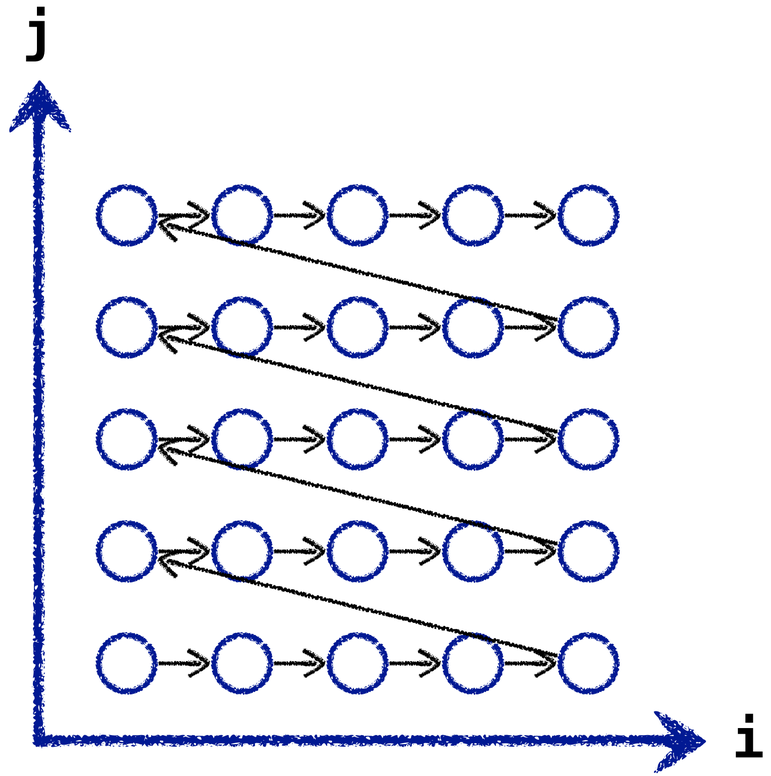
Loop interchange



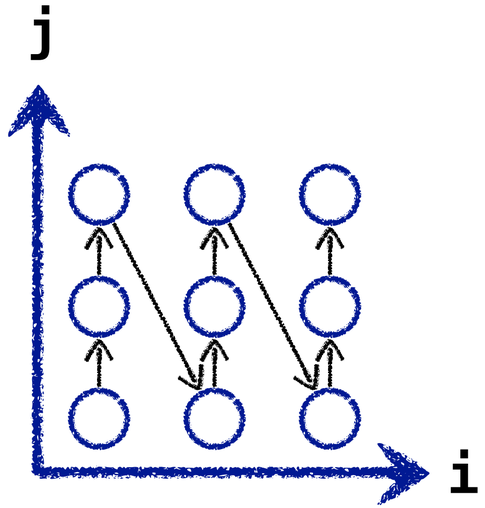
instance-wise reasoning

```
for (j = 0; j < N; j++)
  for (i = 0; i < N; i++)
    a[i][j] = 2*a[i+1][j+1]
```

Loop interchange

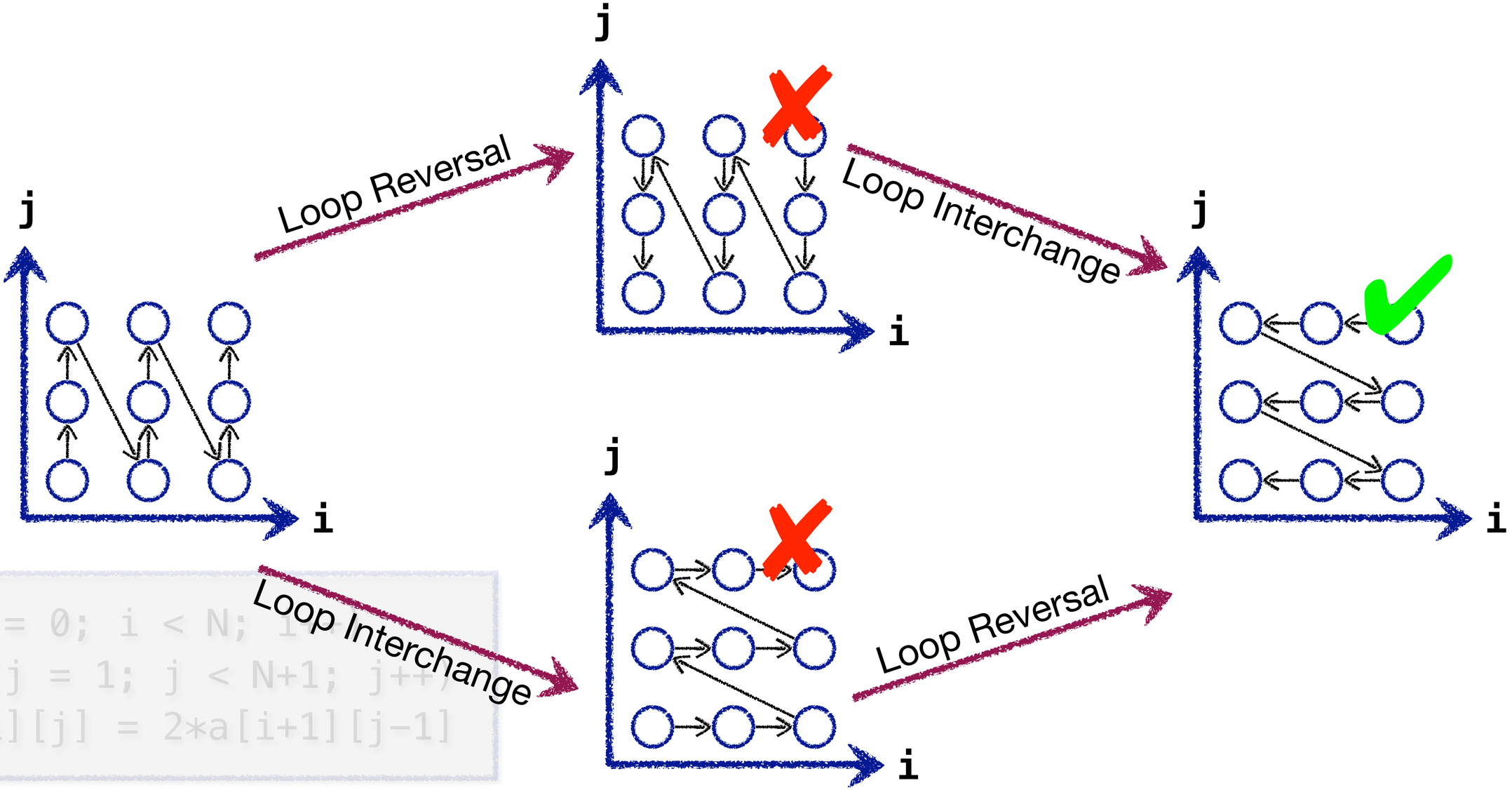


composing transformations




```
for (i = 0; i < N; i++)
  for (j = 1; j < N+1; j++)
    a[i][j] = 2*a[i+1][j-1]
```

composing transformations



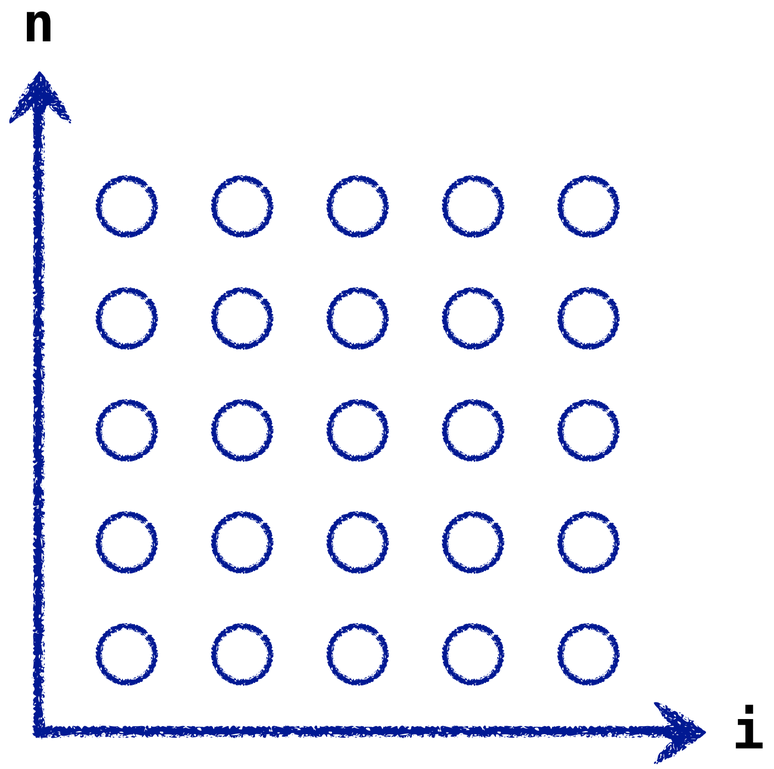

```
for (i = 0; i < N; i++)
  for (j = 1; j < N+1; j++)
    a[i][j] = 2*a[i+1][j-1]
```

what about recursion?

```
outer(int i, node n)
  if (i >= N) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```

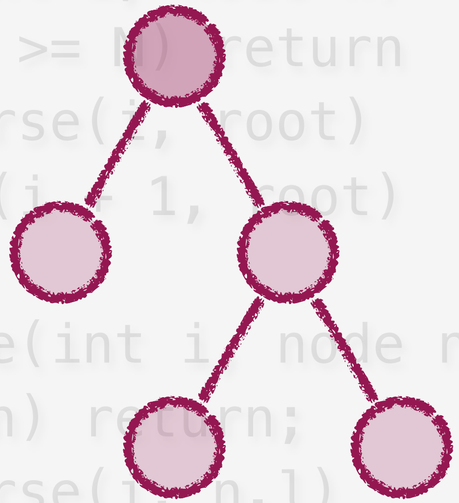
what about recursion?

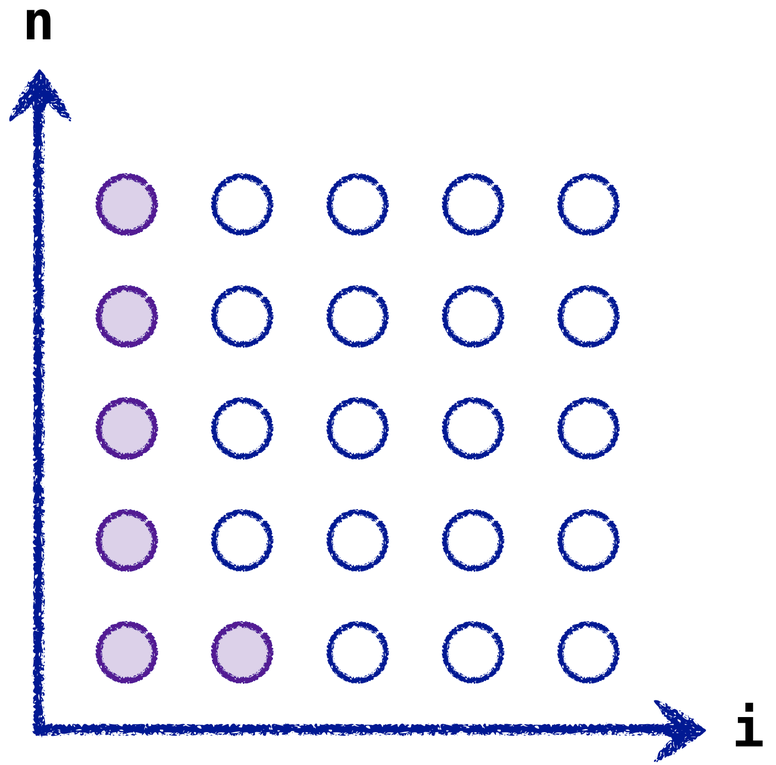
```
outer(int i, node n)
  if (i >= N) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```



what about recursion?

```
outer(int i, node n)
  if (i >= N) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```





transformations of recursion

```
outer(int i, node
  if (i >= N) retu
  traverse(i, root)
  ou                root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  trav
  n                 [i+1]
```

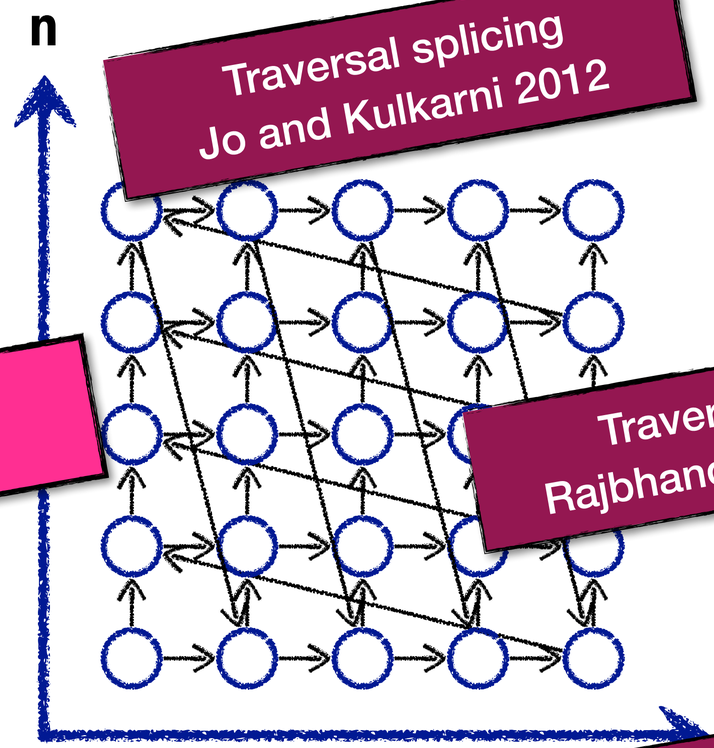
Point blocking
Jo and Kulkarni 2011

Dynamic pointer alignment
Zhang and Chien 1997

???

Grafter
Sakka et al 2019

TreeFuser
Sakka et al 2017

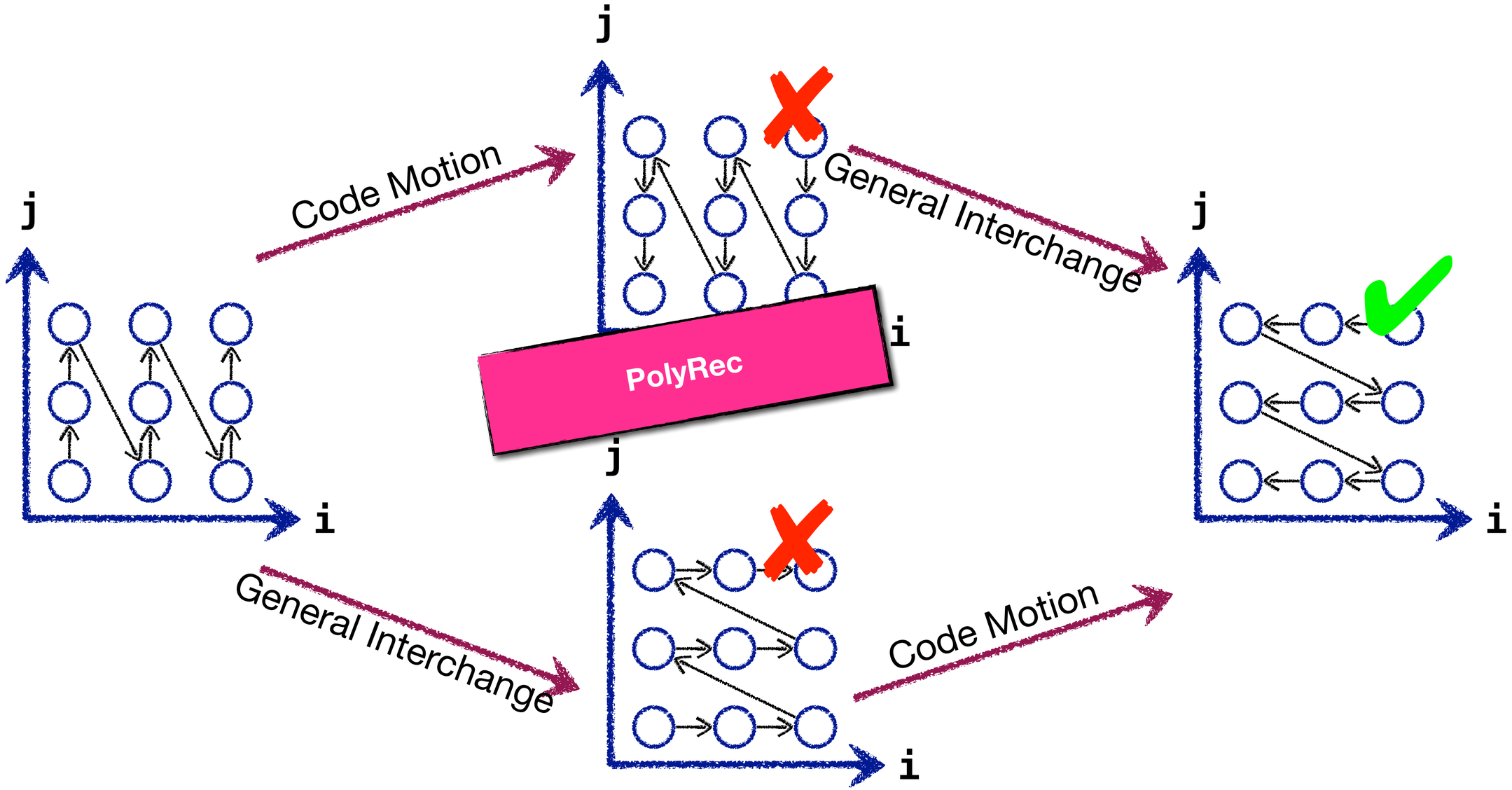


Traversal splicing
Jo and Kulkarni 2012

Traversal fusion
Rajbhandari et al 2016

Recursion twisting
Sundararajah et al. 2017

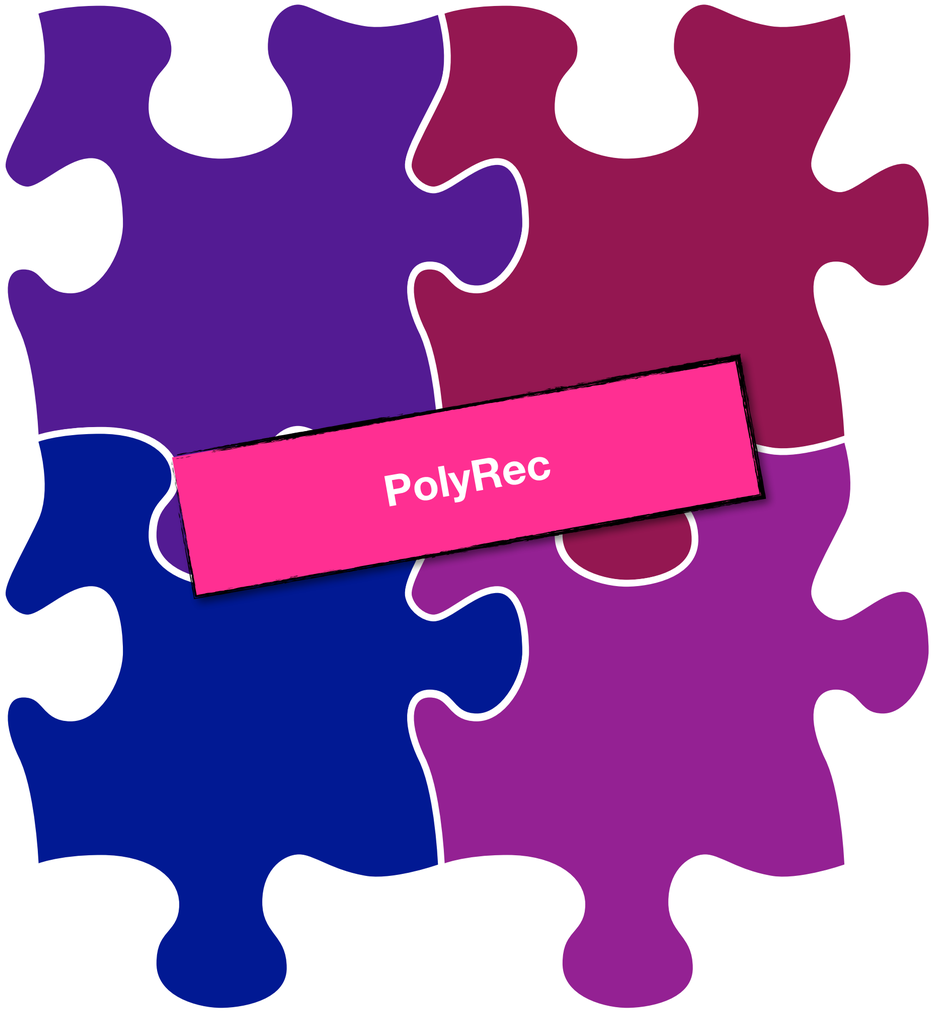
need composition



what is polyrec?

iteration space representation

soundness check

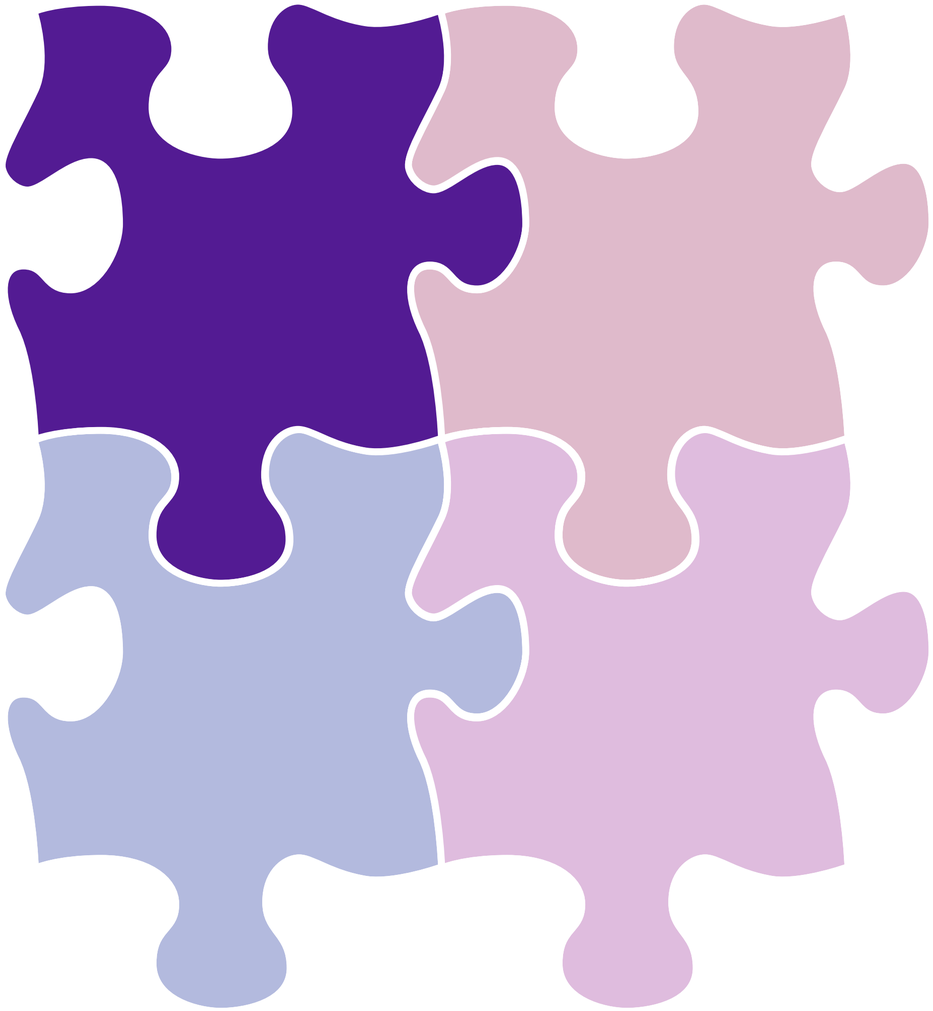


transformation representation

dependence representation

outline

iteration space representation



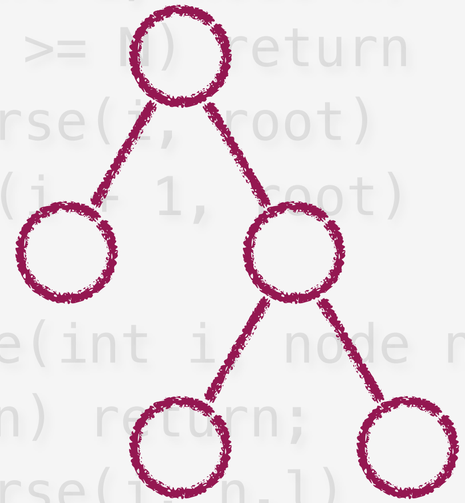
transformation representation

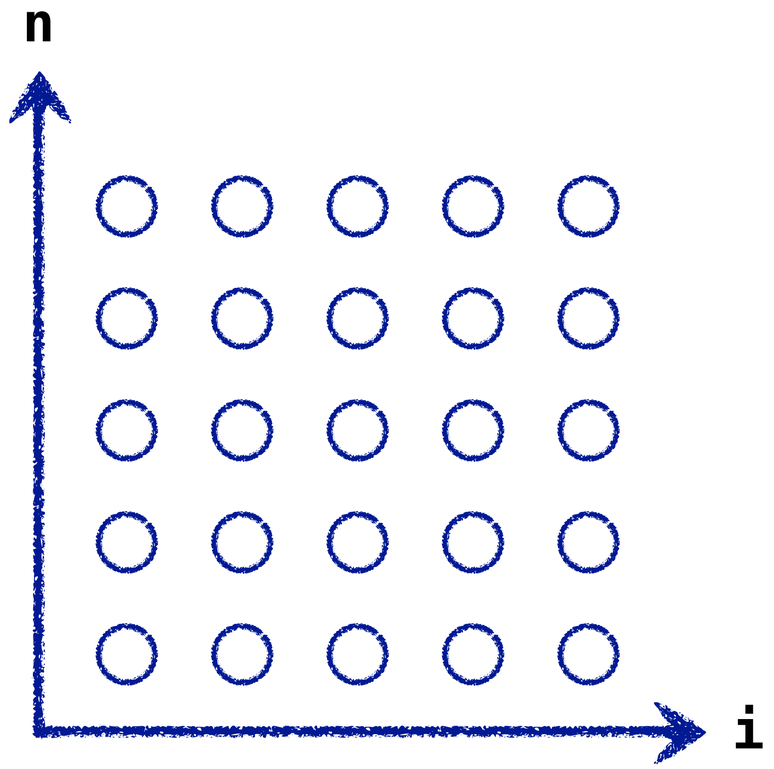
soundness check

dependence representation

naming dynamic instances

```
outer(int i, node n)
  if (i >= M) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```

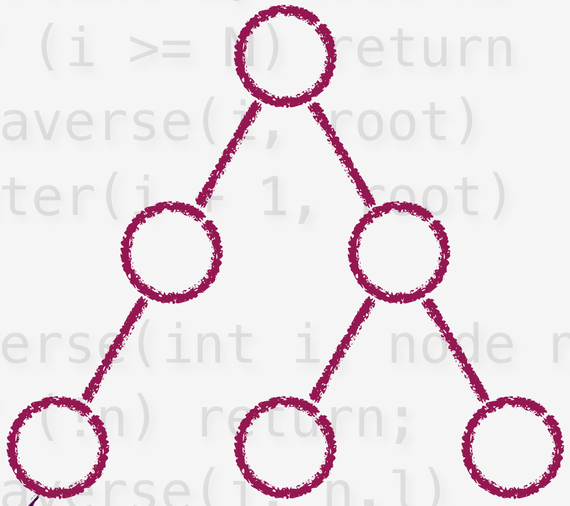




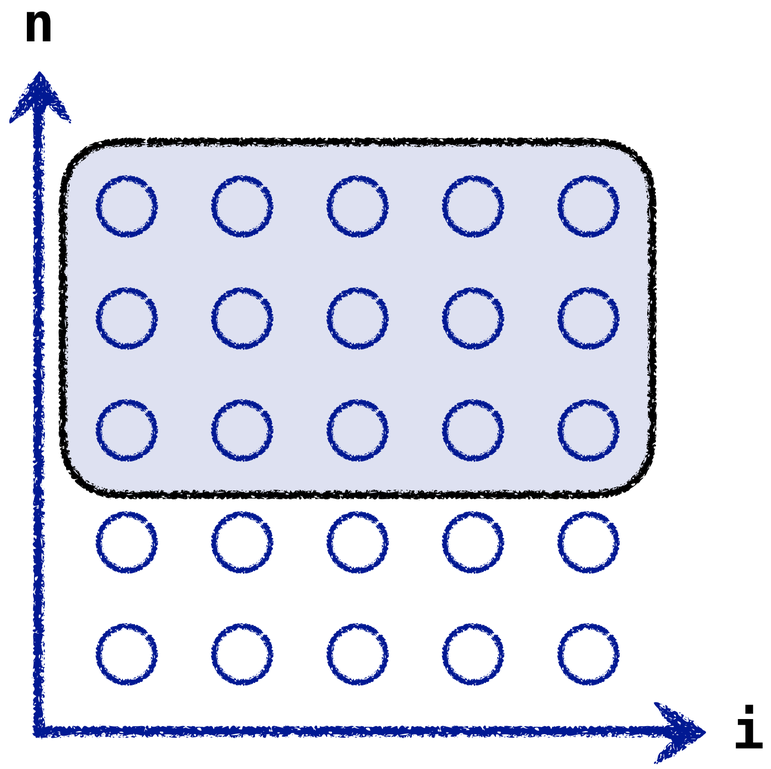
Challenge: Iteration space is not affine!

naming dynamic instances

```
outer(int i, node n)
  if (i >= M) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```



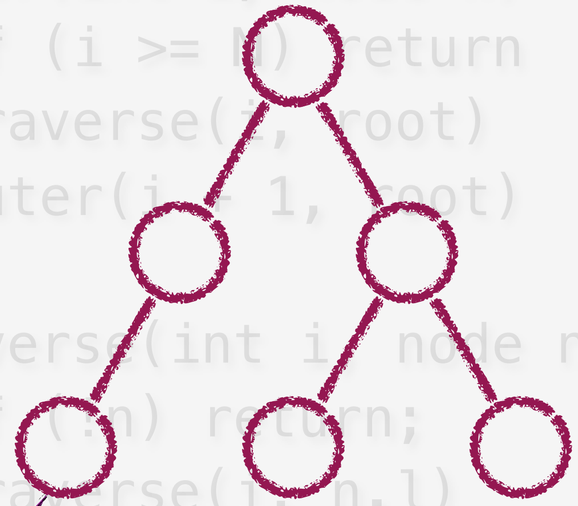
root.l.l



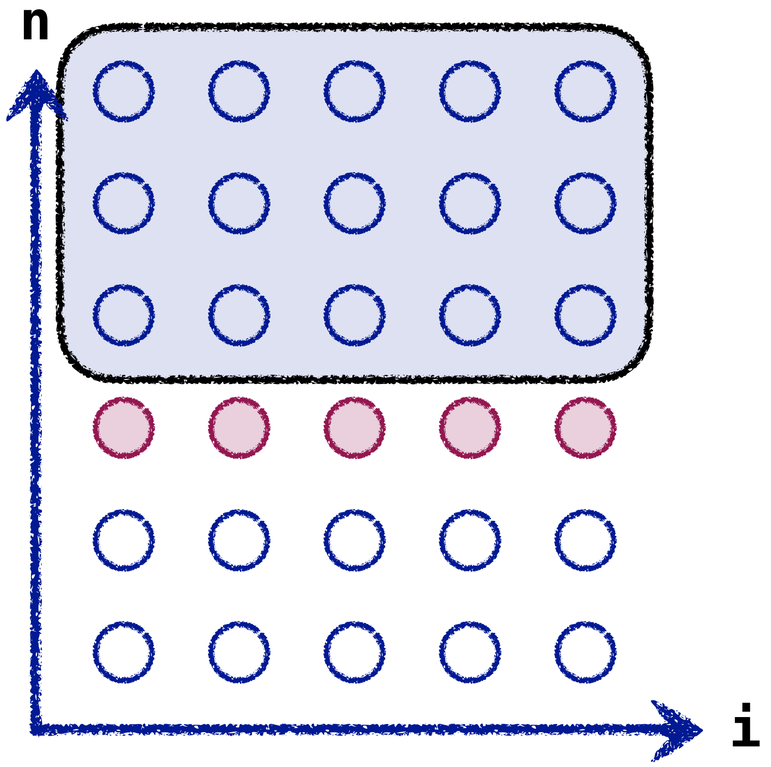
Challenge: Iteration space is not affine!

naming dynamic instances

```
outer(int i, node n)
  if (i >= M) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```



root.l.l



Challenge: Iteration space is not affine!

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2^l traverse(i, n.l)
r2^r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

outer(0, root)

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
  if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
r2^l traverse(i, n.l)
r2^r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

outer(0, root)

r_1 **outer(1, root)**

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
  if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
r2^l traverse(i, n.l)
r2^r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

outer(0, root)

r_1 **outer(1, root)**

t_1 **traverse(1, root)**

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

outer(0, root)

r_1 **outer(1, root)**

t_1 **traverse(1, root)**

r_2^l **traverse(1, root.l)**

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
  if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
r2^l traverse(i, n.l)
r2^r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

```
    outer(0, root)
r1  outer(1, root)
t1  traverse(1, root)
r2^l traverse(1, root.l)
r2^r traverse(1, root.l.r)
```

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

	outer(0, root)
r_1	**outer(1, root)**
t_1	**traverse(1, root)**
r_2^l	**traverse(1, root.l)**
r_2^r	**traverse(1, root.l.r)**
s_1	**root.l.r[1] = 2 * root.l.r.[2]**

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2^l traverse(i, n.l)
r2^r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

```
outer(0, root)
outer(1, root)
traverse(1, root)
traverse(1, root.l)
traverse(1, root.l.r)
root.l.r[1] = 2 * root.l.r.[2]
```

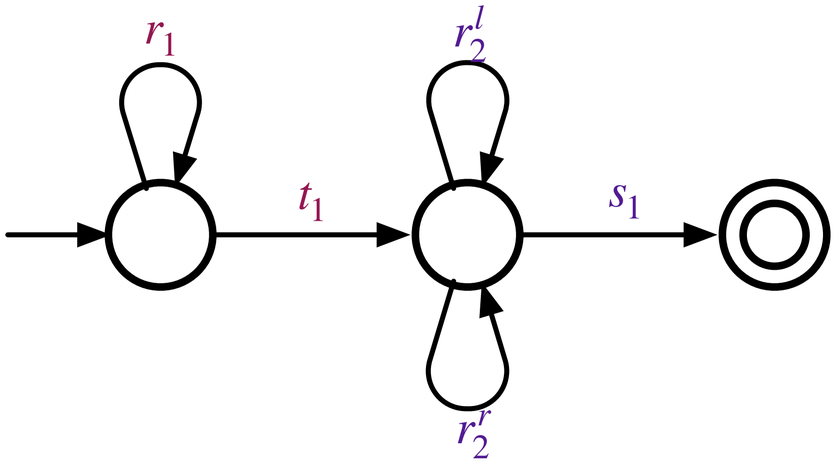
$r_1 \quad t_1 \quad r_2^l \quad r_2^r \quad s_1$

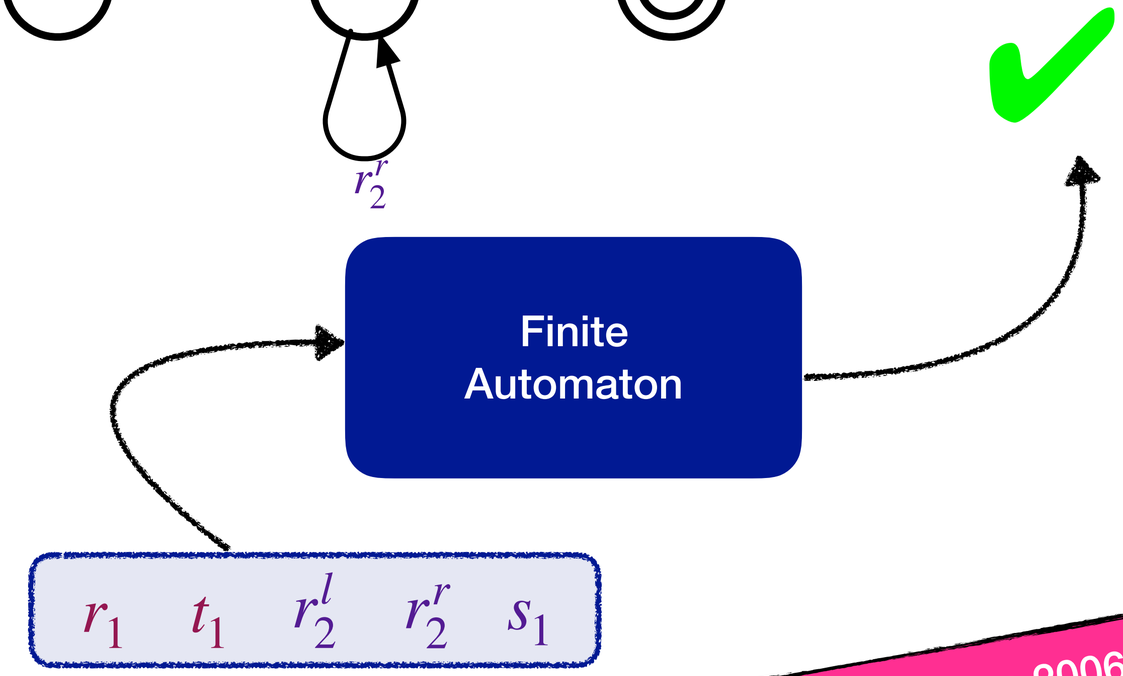
Amiranoff and Cohen 2006

(i,n) = (1, root.l.r)

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```





(i,n) = (1, root.l.r)

Amiranoff and Cohen 2006

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

$r_1 \quad t_1 \quad r_2^l \quad r_2^r \quad s_1 \prec r_1 \quad t_1 \quad r_2^l \quad s_1$

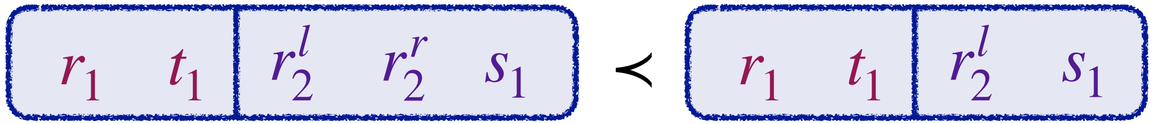
$$t_1 \prec r_1 \prec r_2^l \prec r_2^r \prec s_1$$

(i,n) = (1, root.l.r)

Amiranoff and Cohen 2006

naming dynamic instances

```
outer(int i, node n)
   if (i >= N) return
t1 traverse(i, root)
r1 outer(i + 1, root)

traverse(int i, node n)
   if (!n) return;
r2l traverse(i, n.l)
r2r traverse(i, n.r)
s1 n.x[i] = 2*n.x[i+1]
```

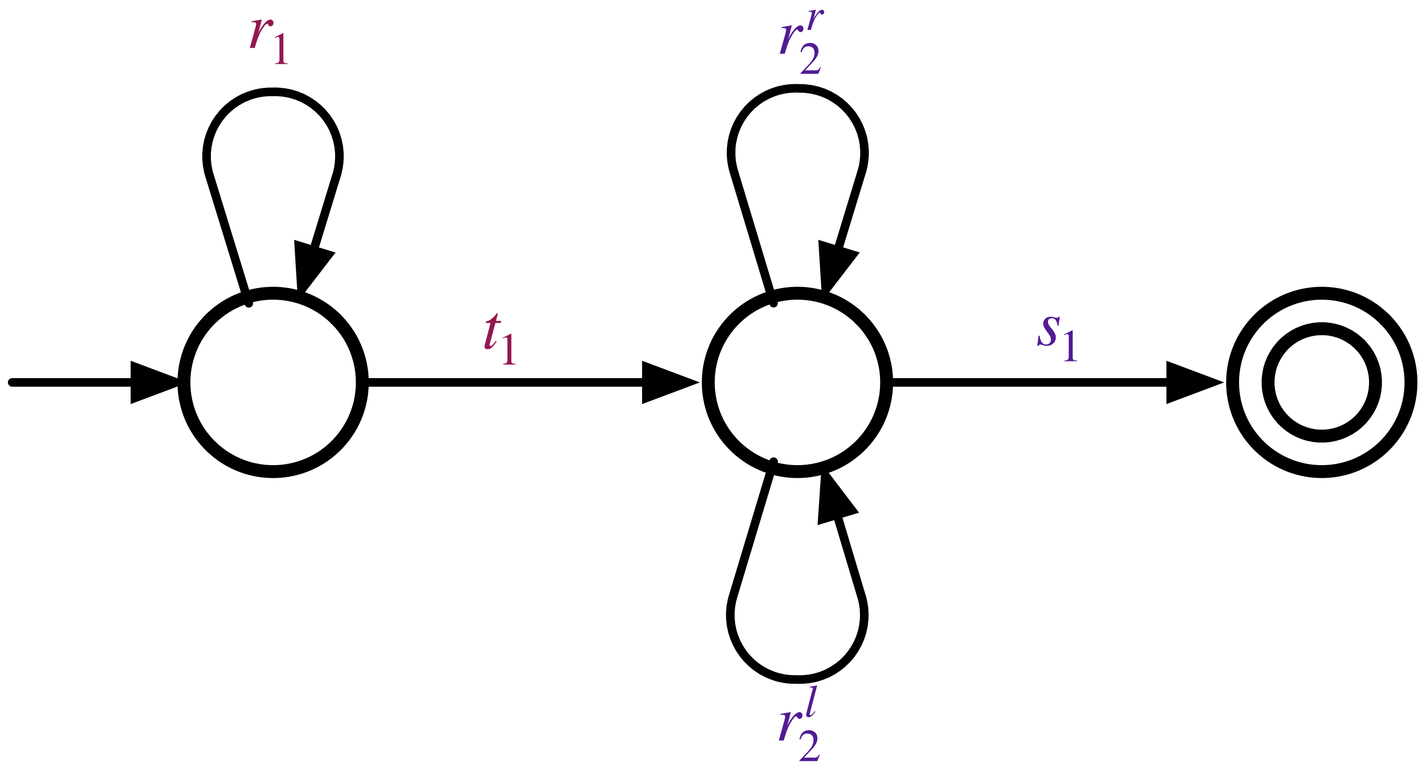


$$t_1 \prec r_1 \prec r_2^l \prec r_2^r \prec s_1$$

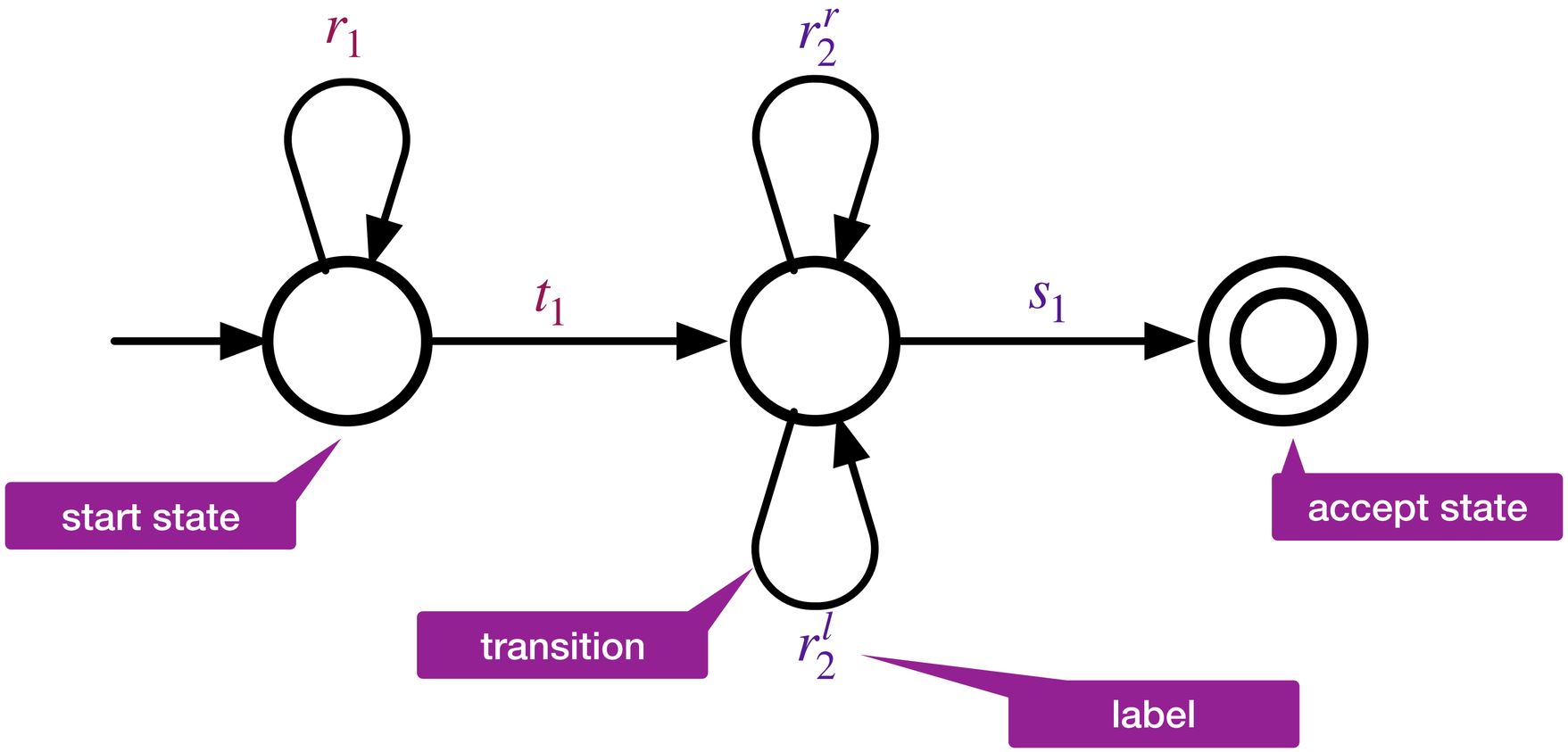
regular relation

(i,n) = (1, root.l.r)

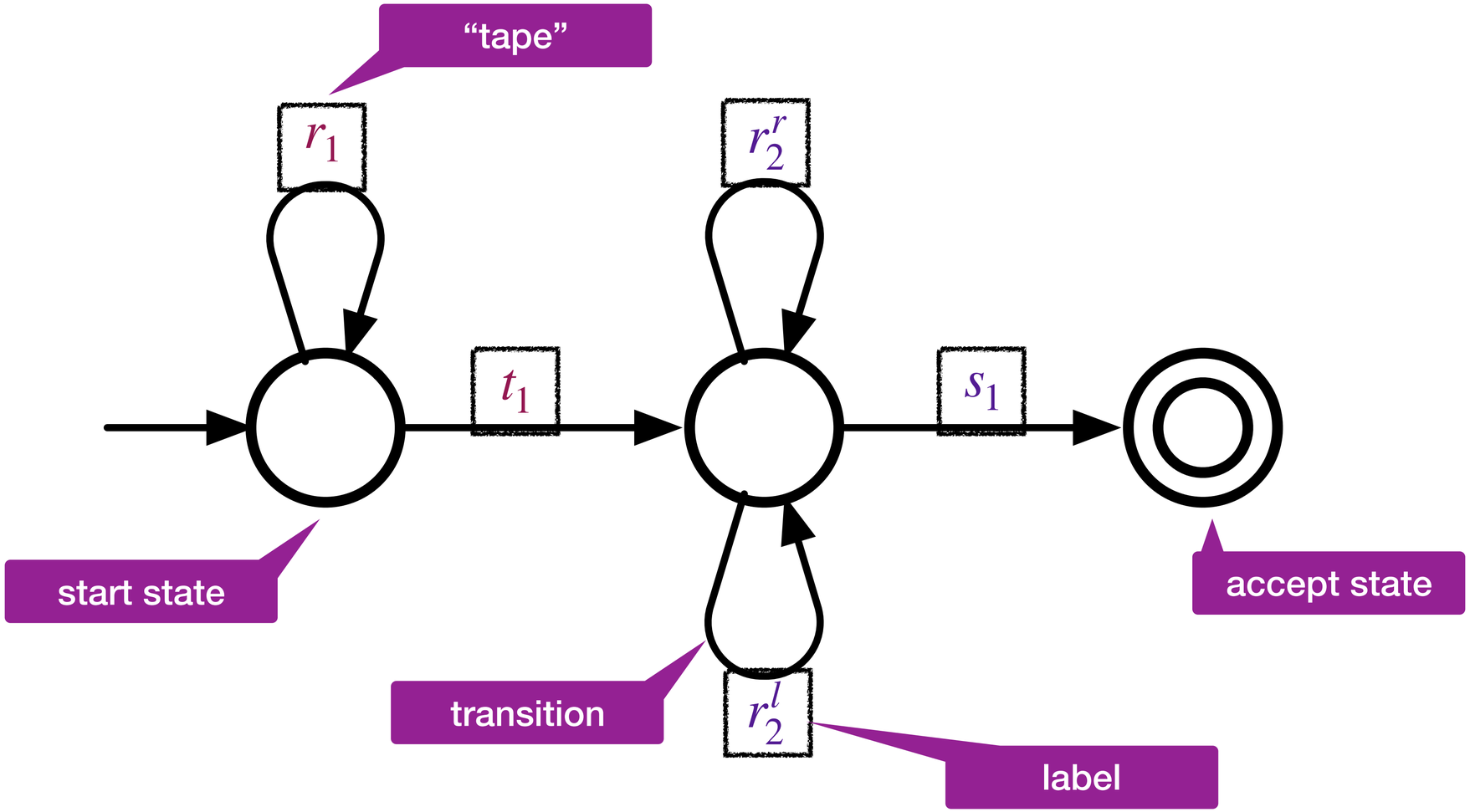
aside: multitape automata



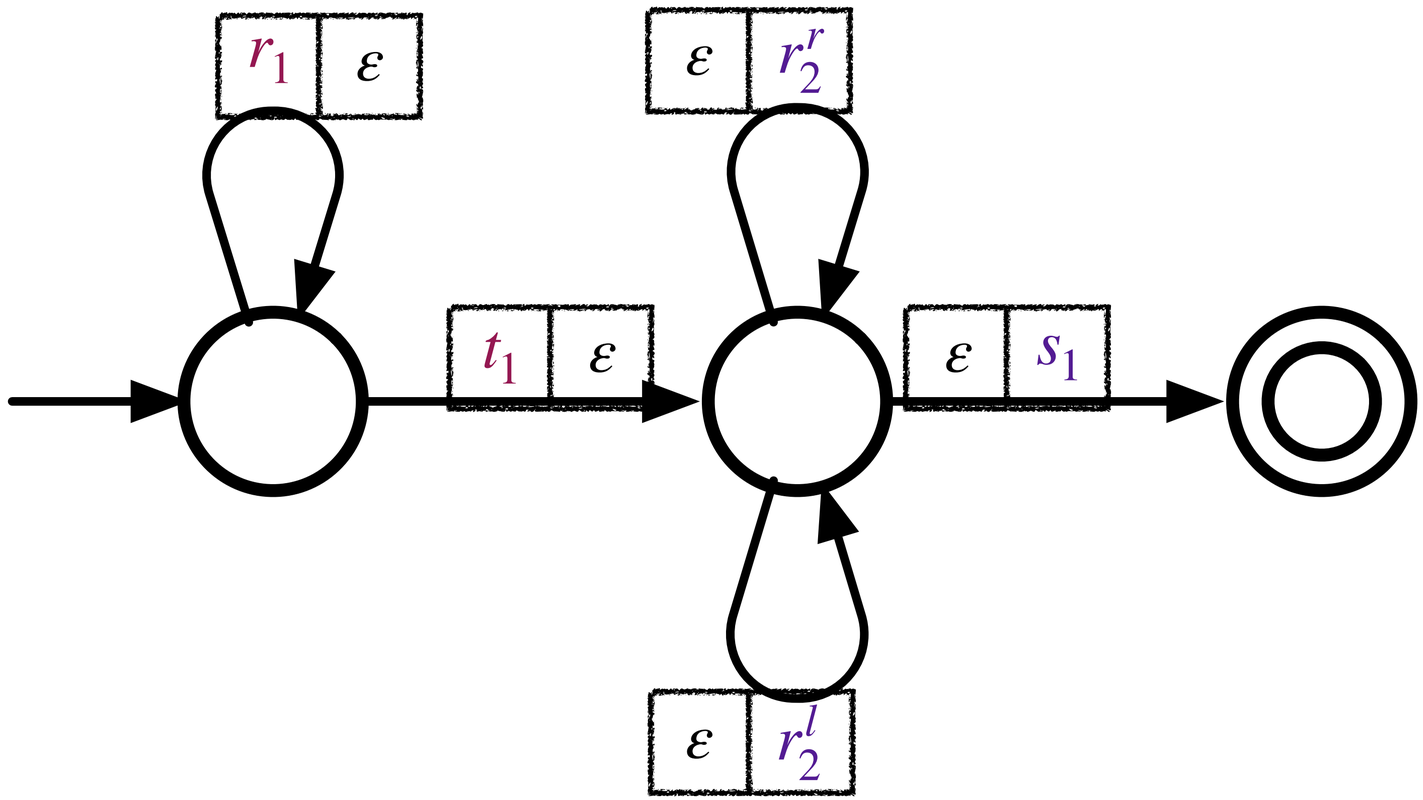
aside: multitape automata



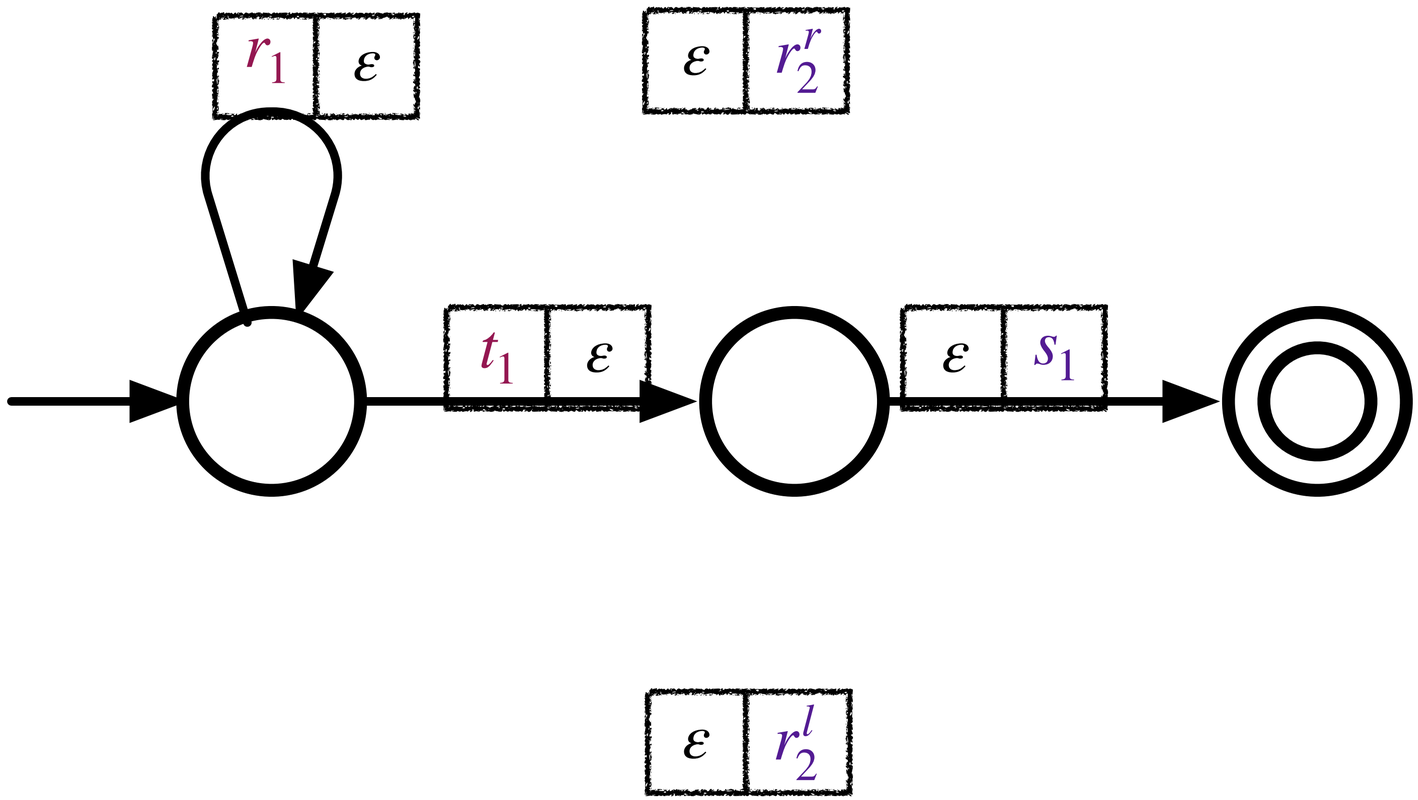
aside: multitape automata



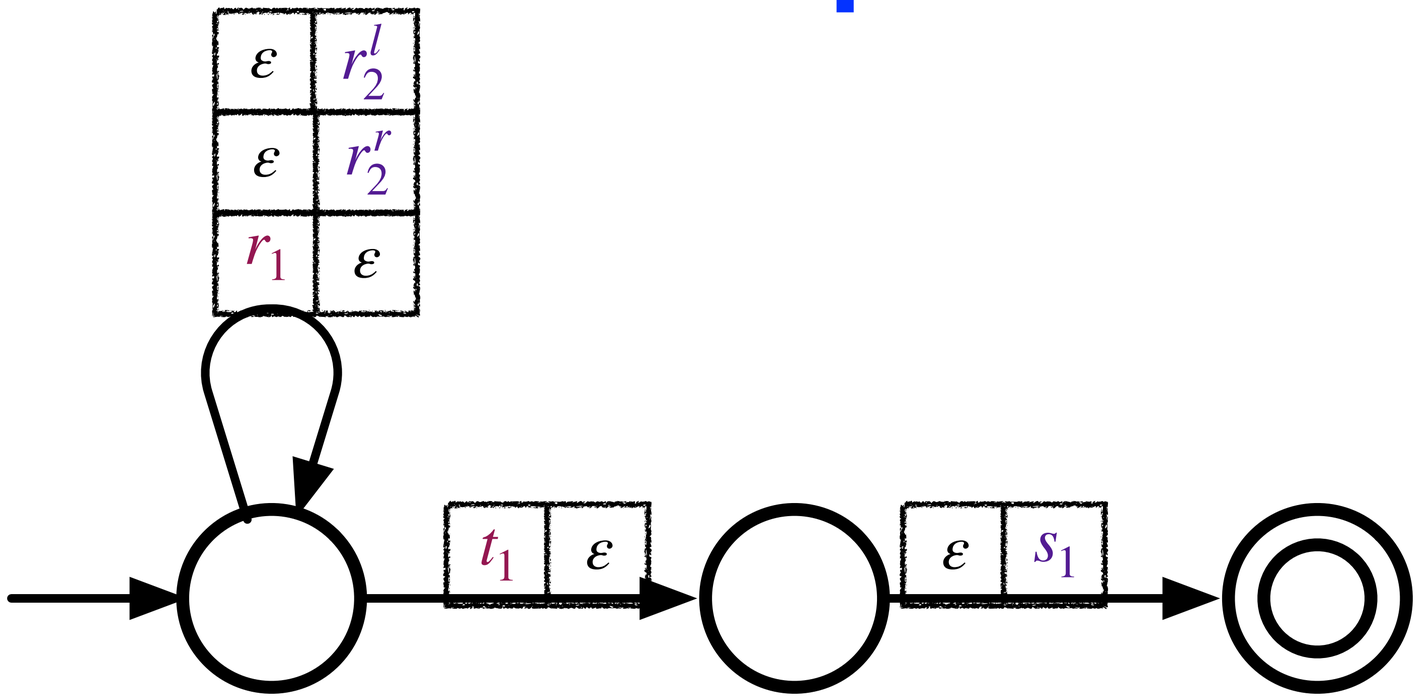
aside: multitape automata



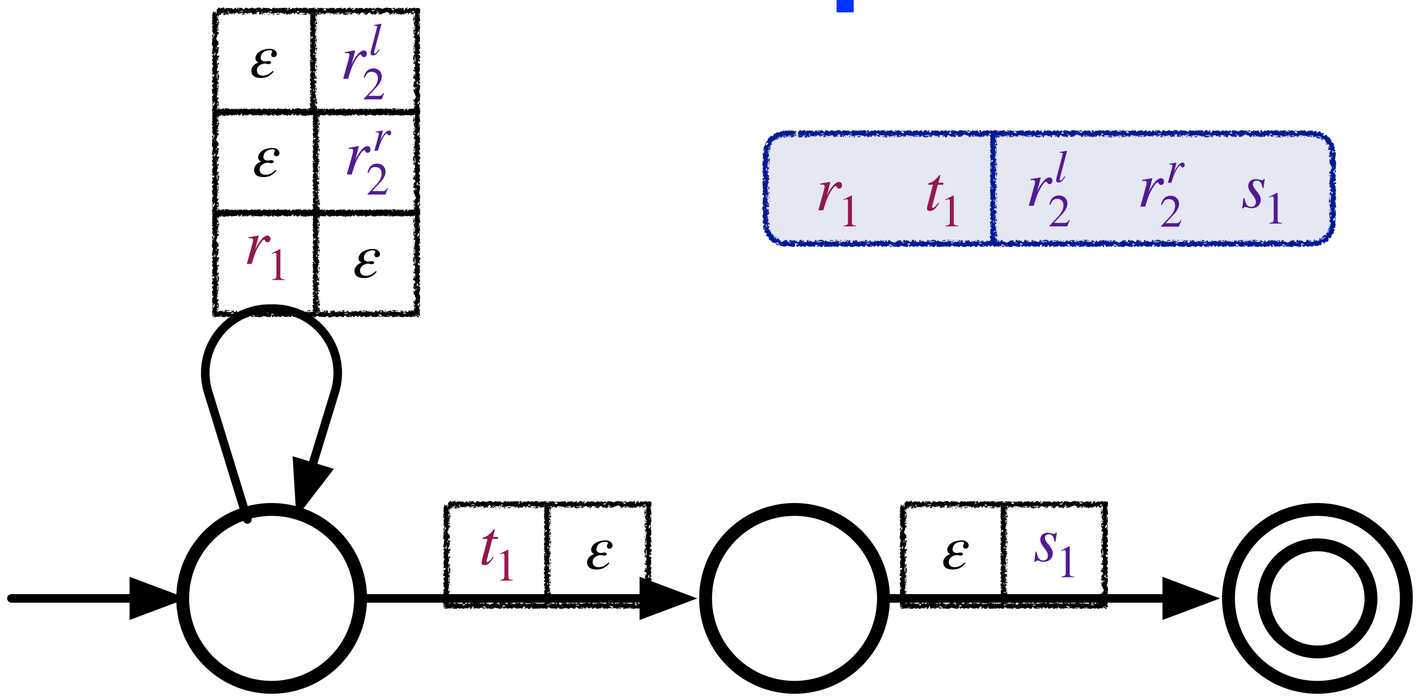
aside: multitape automata



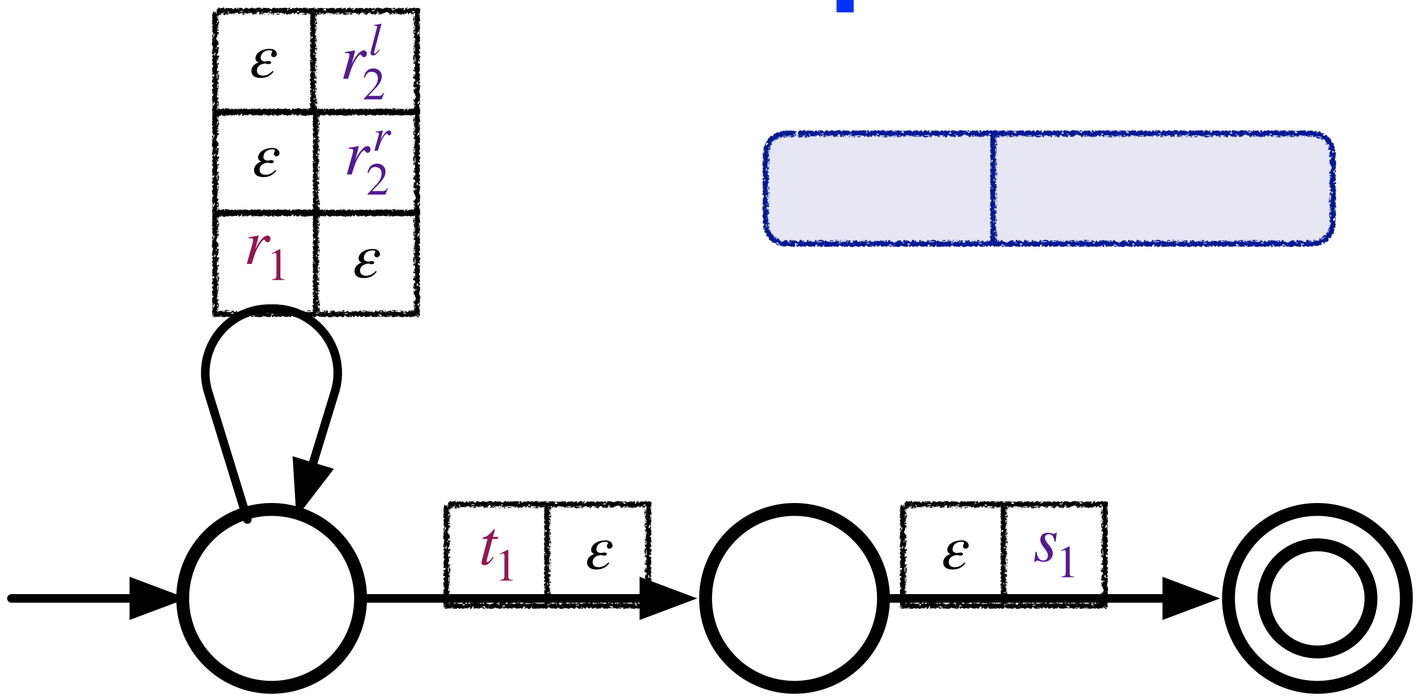
aside: multitape automata



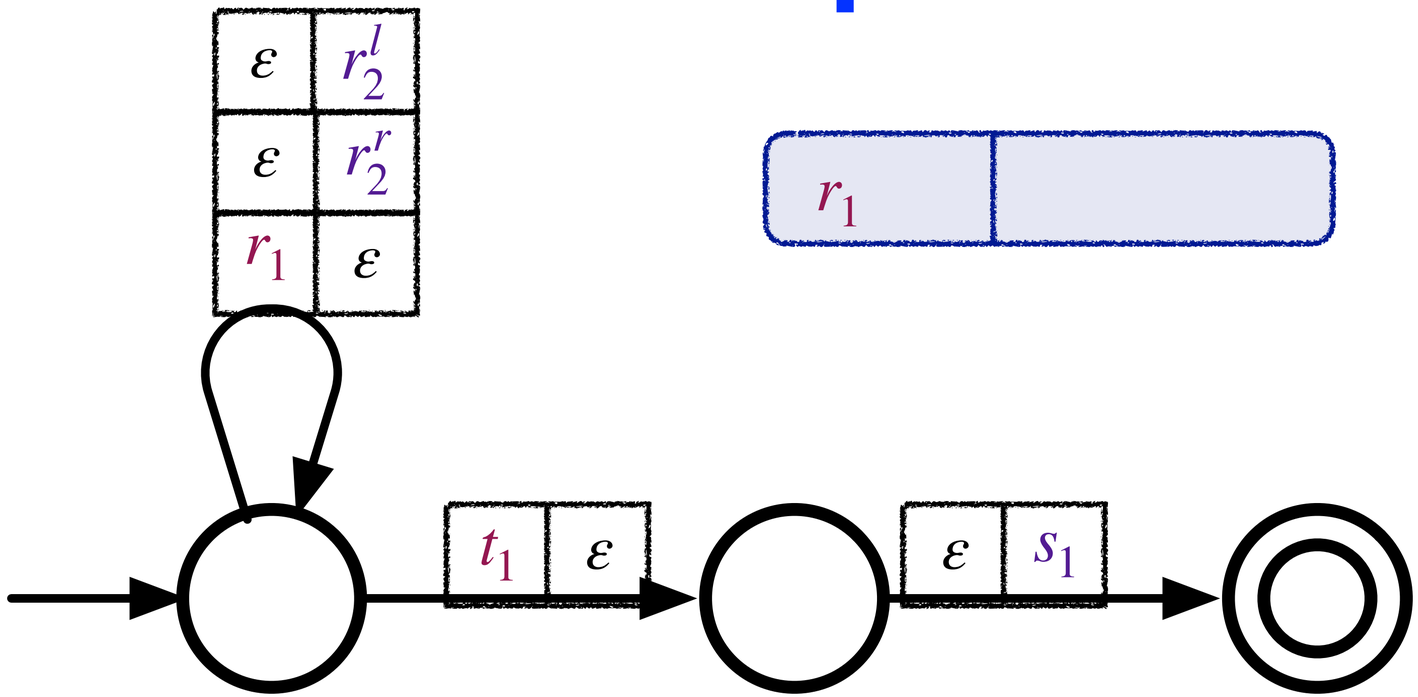
aside: multitape automata



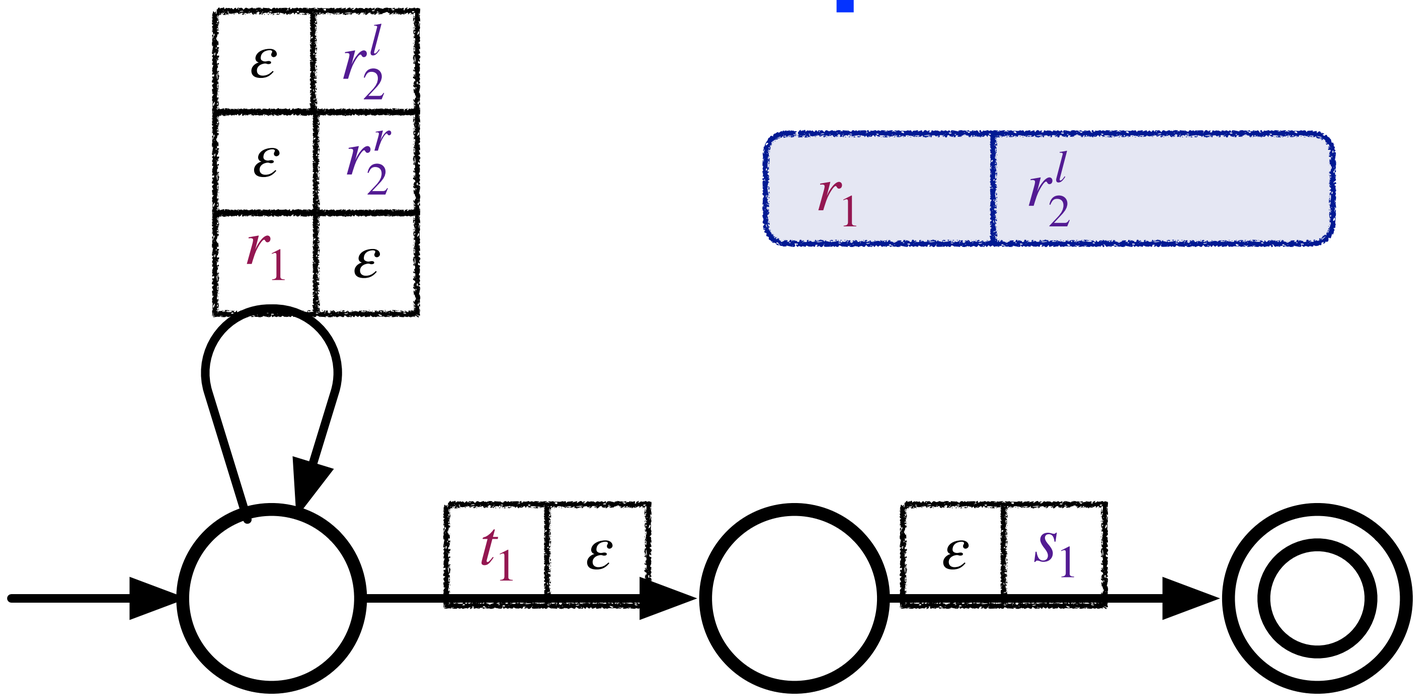
aside: multitape automata



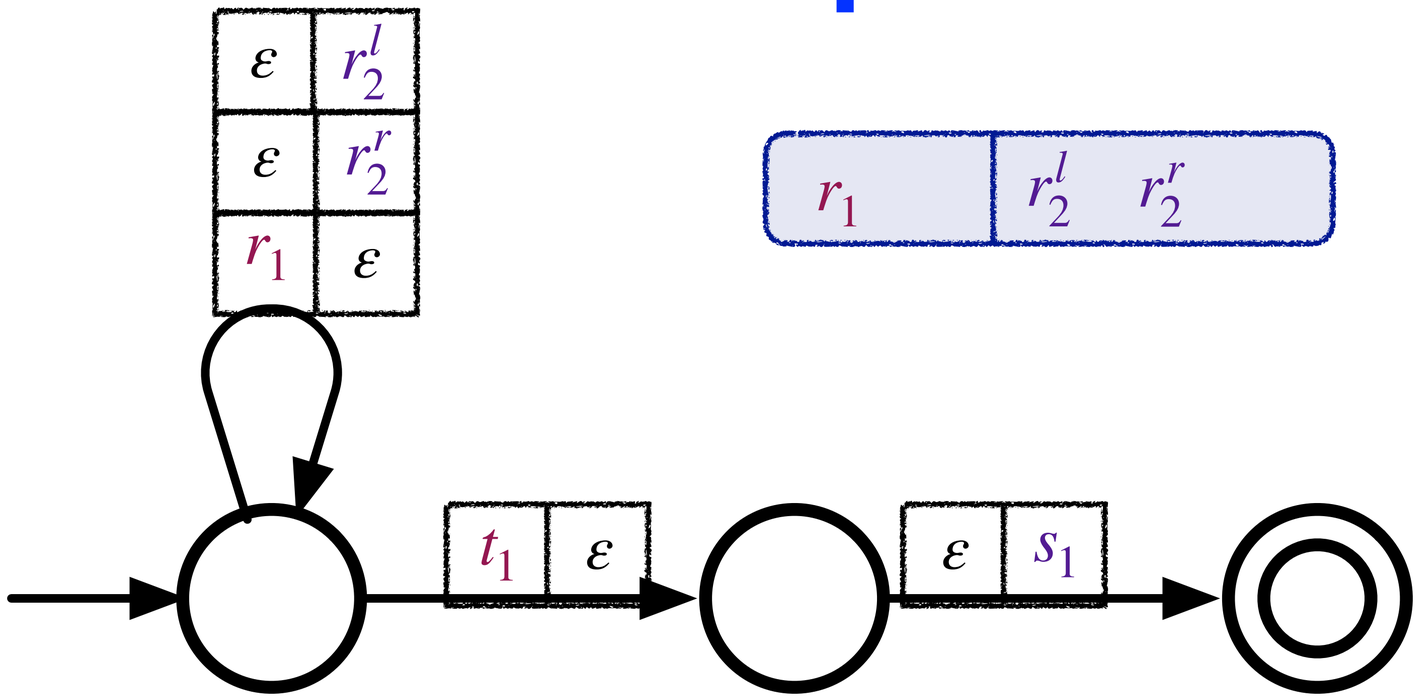
aside: multitape automata



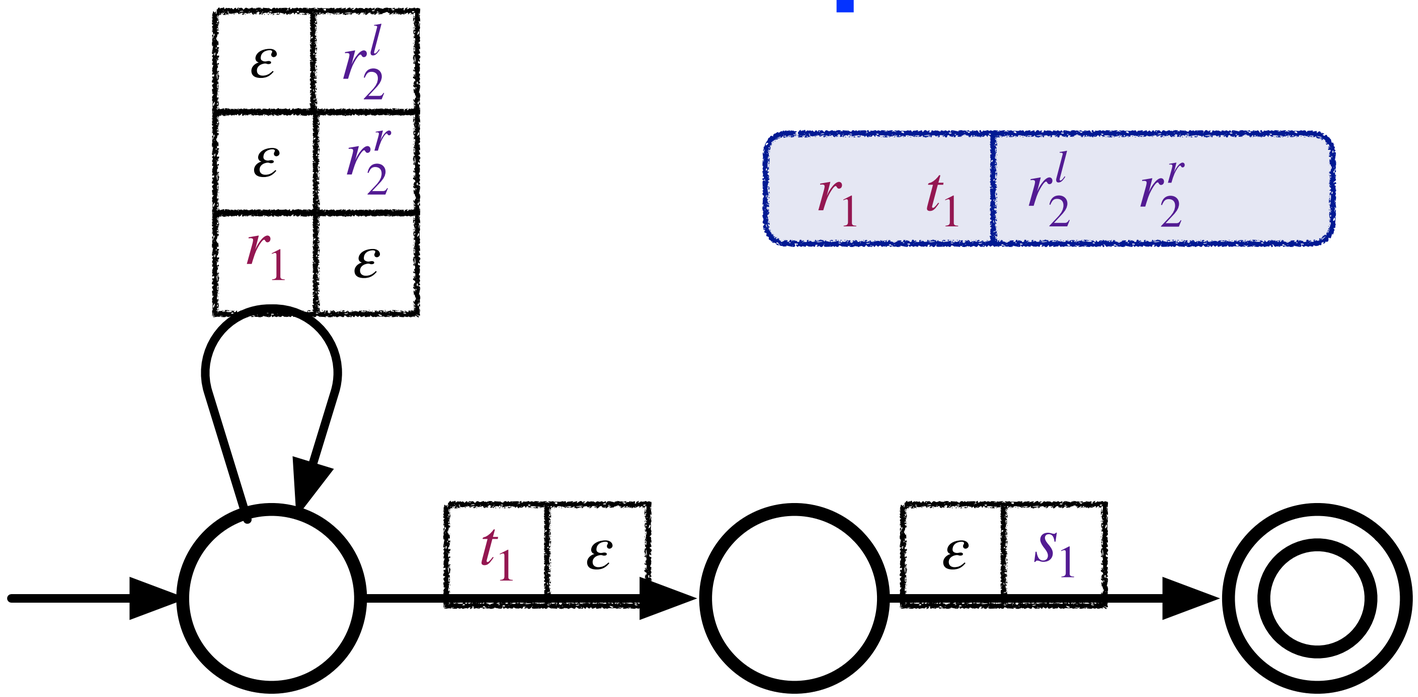
aside: multitape automata



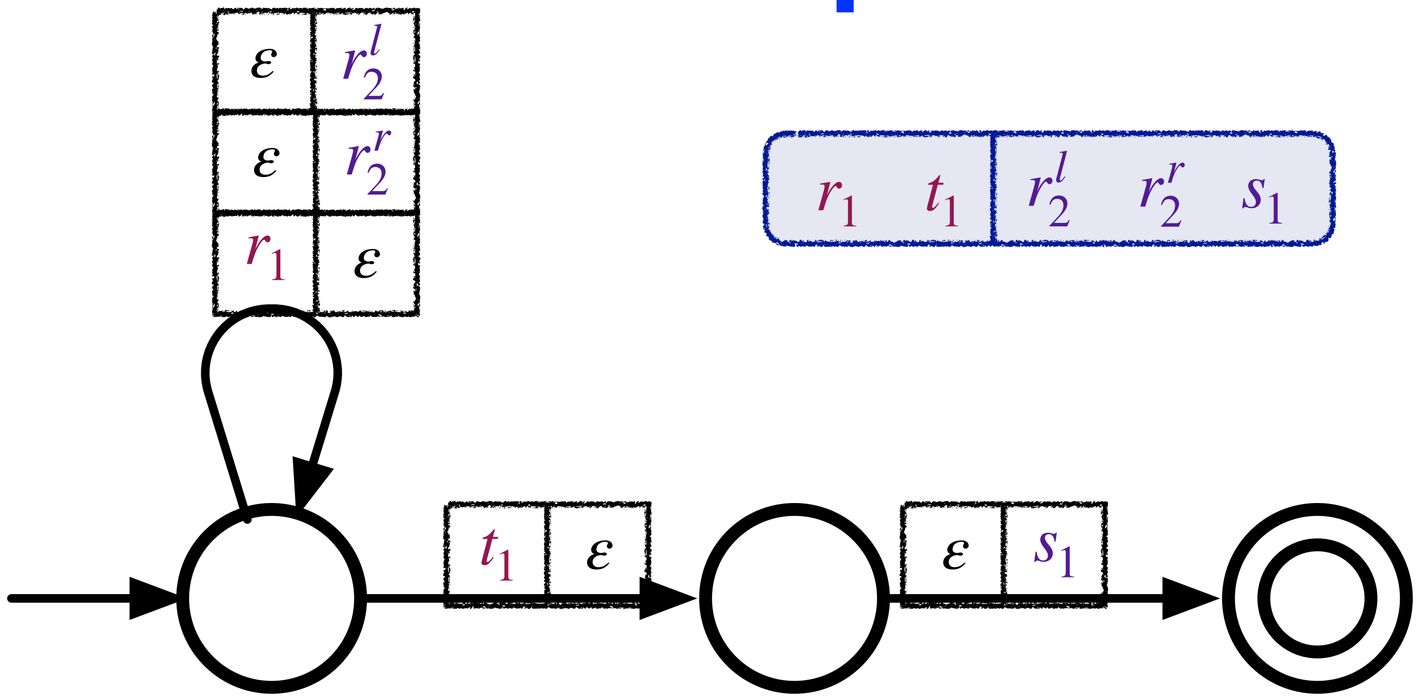
aside: multitape automata



aside: multitape automata

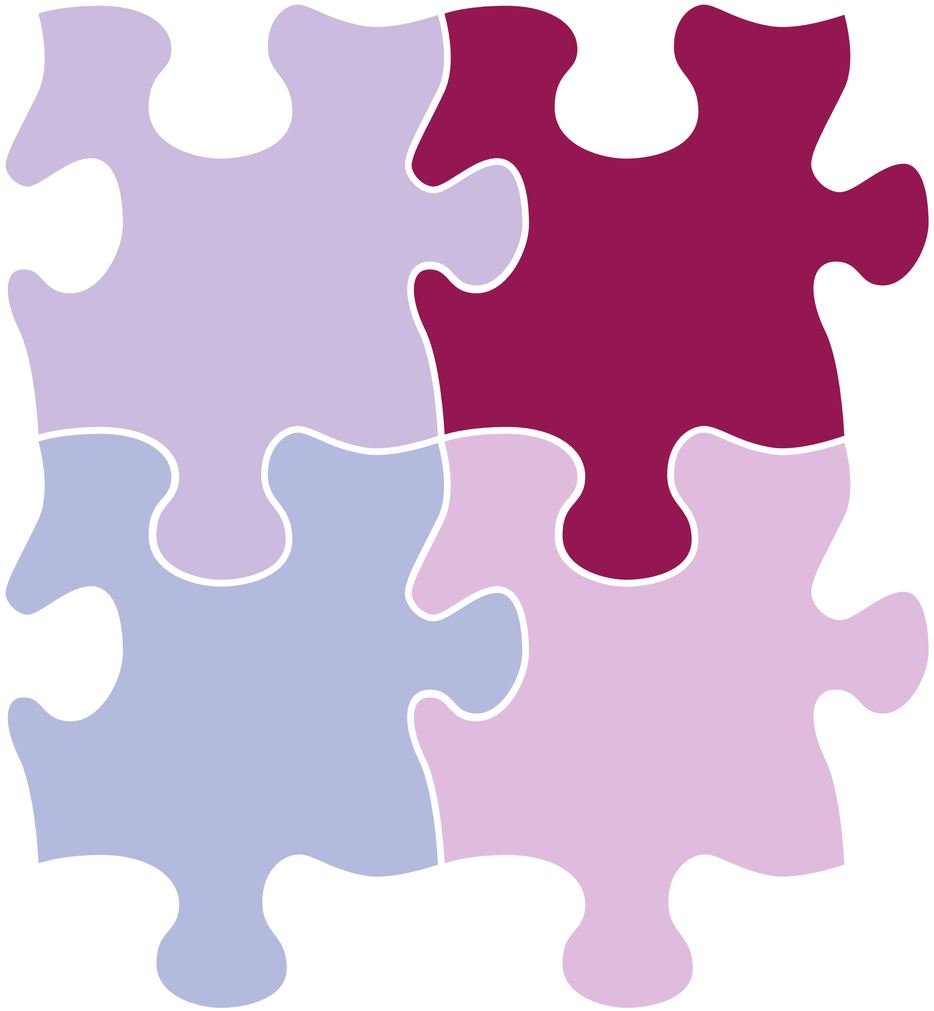


aside: multitape automata



outline

iteration space representation

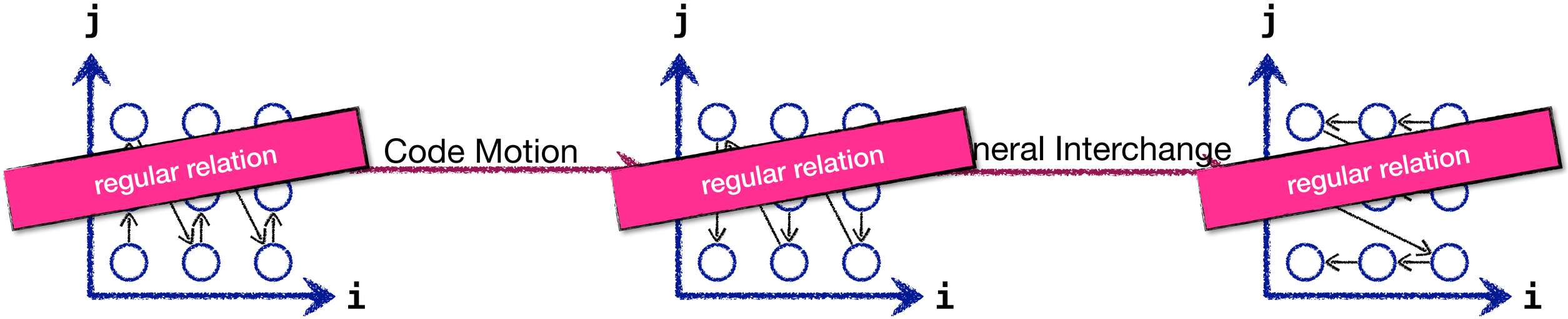


transformation representation

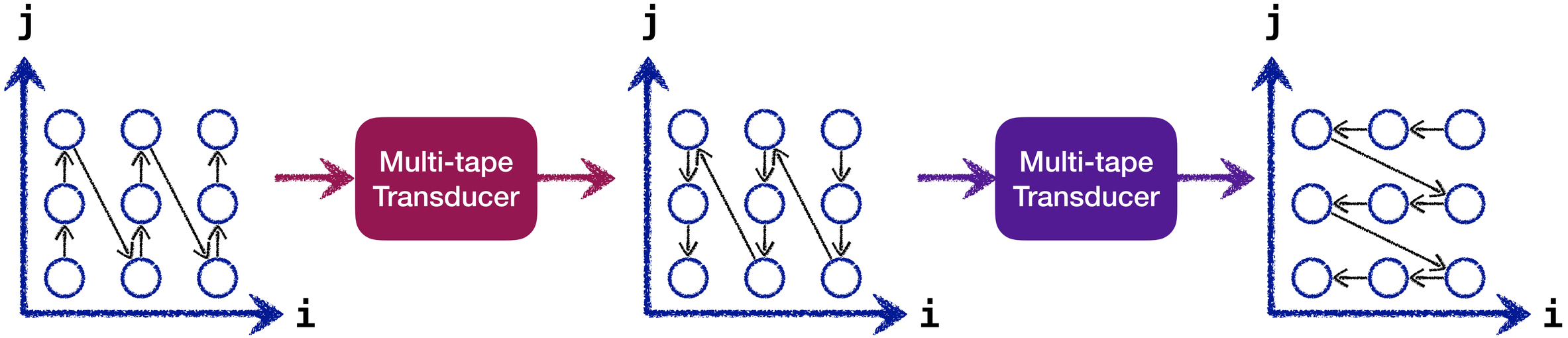
soundness check

dependence representation

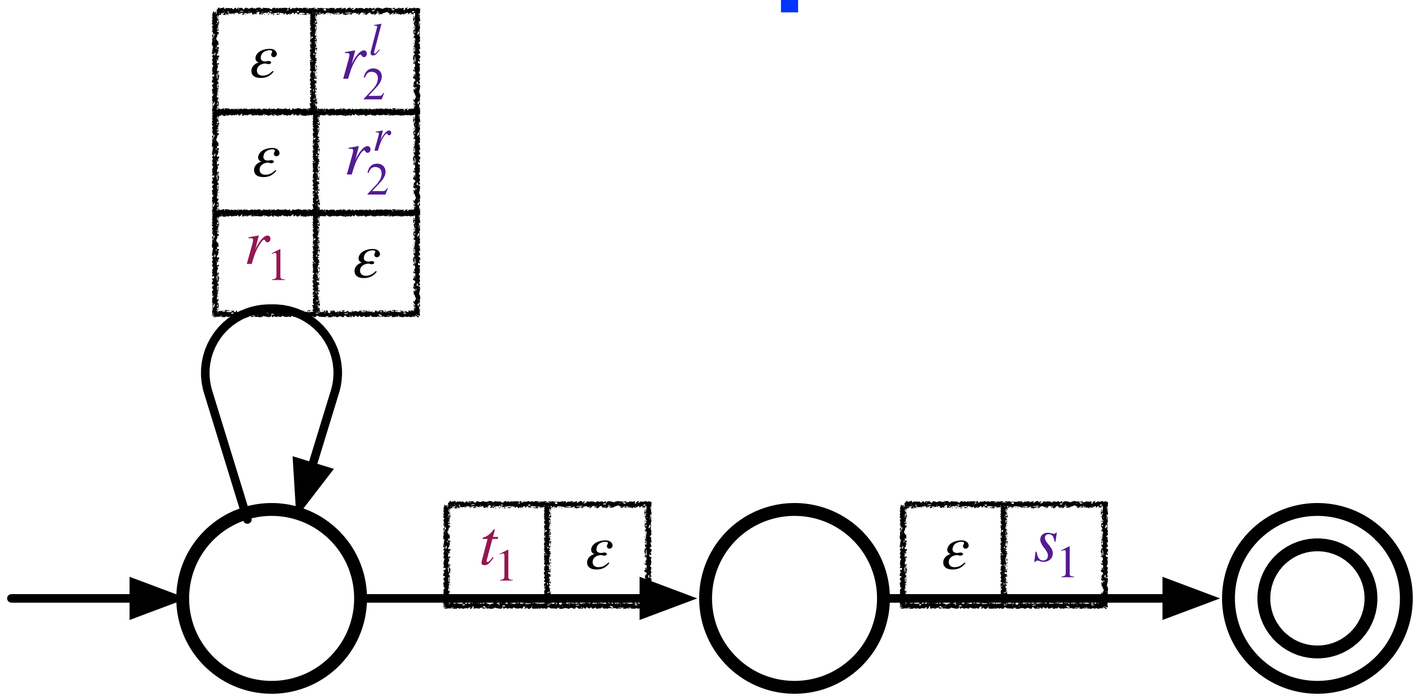
representing transformations



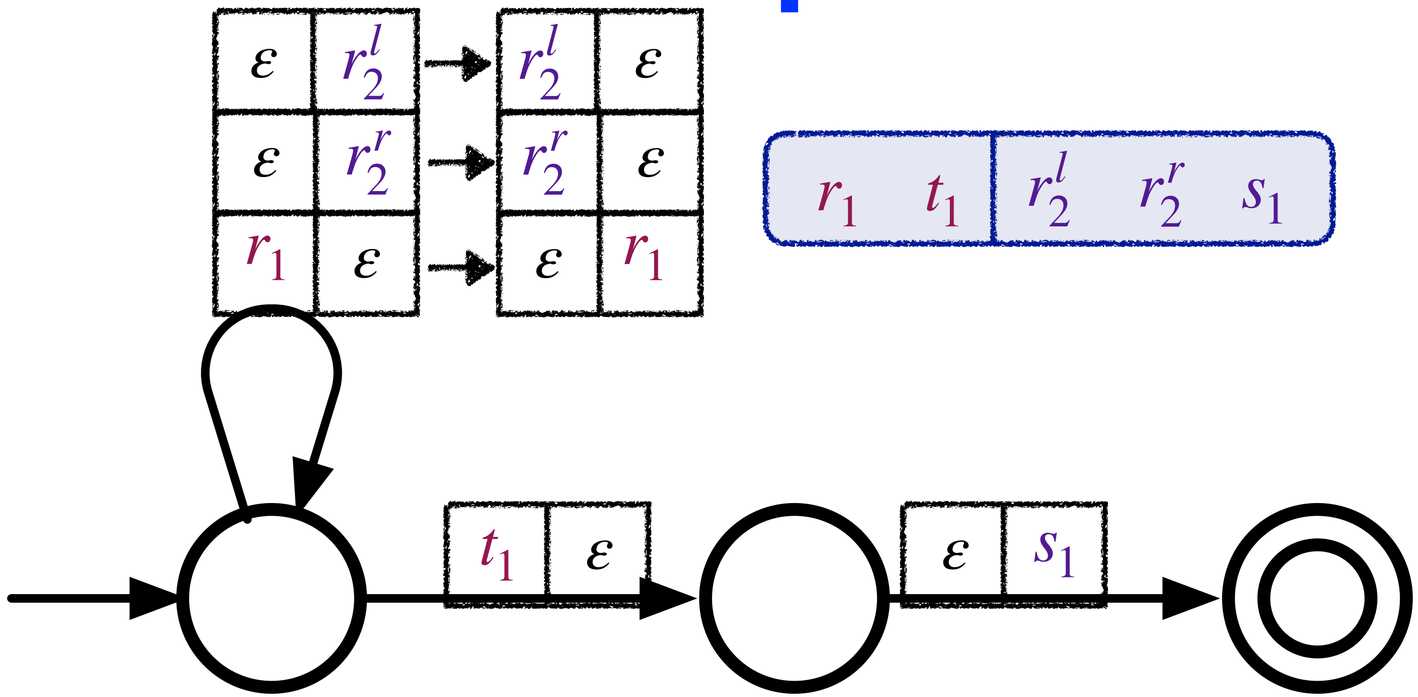
representing transformations



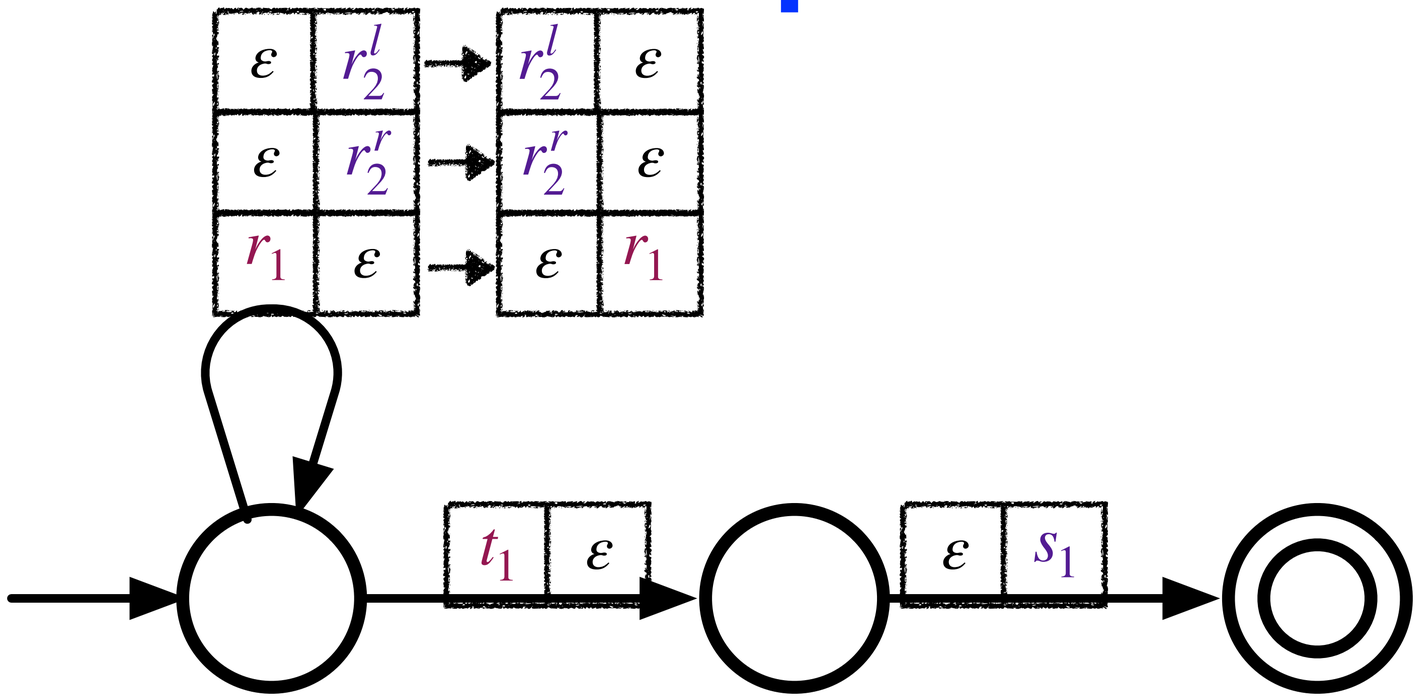
aside: multi-tape transducers



aside: multi-tape transducers

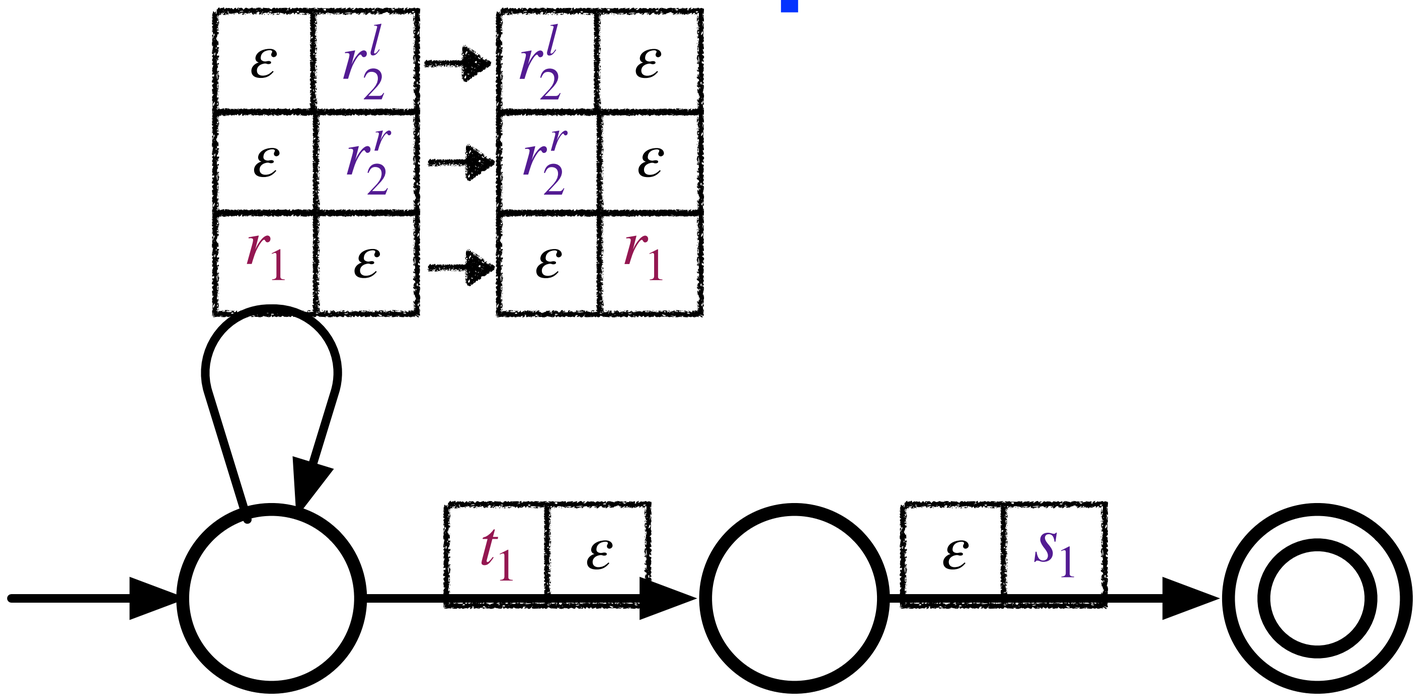


aside: multi-tape transducers

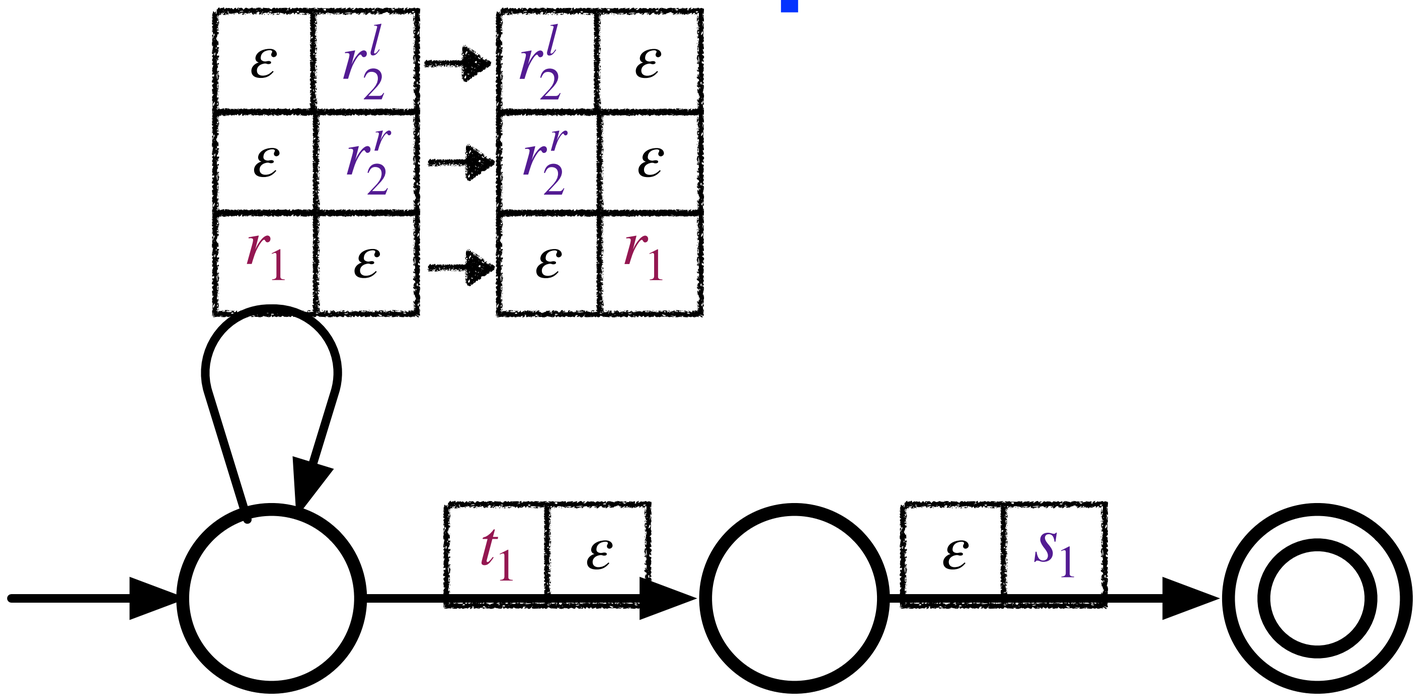


r_1	t_1	r_2^l	r_2^r	s_1

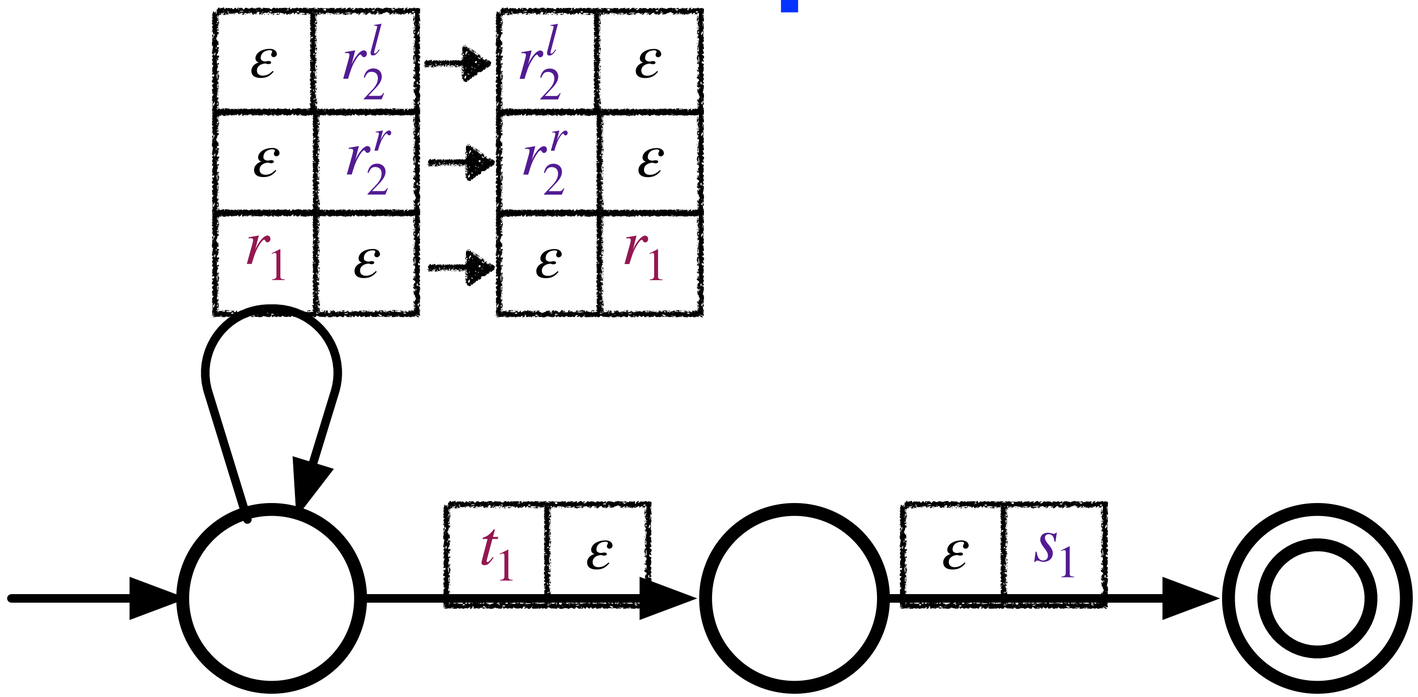
aside: multi-tape transducers



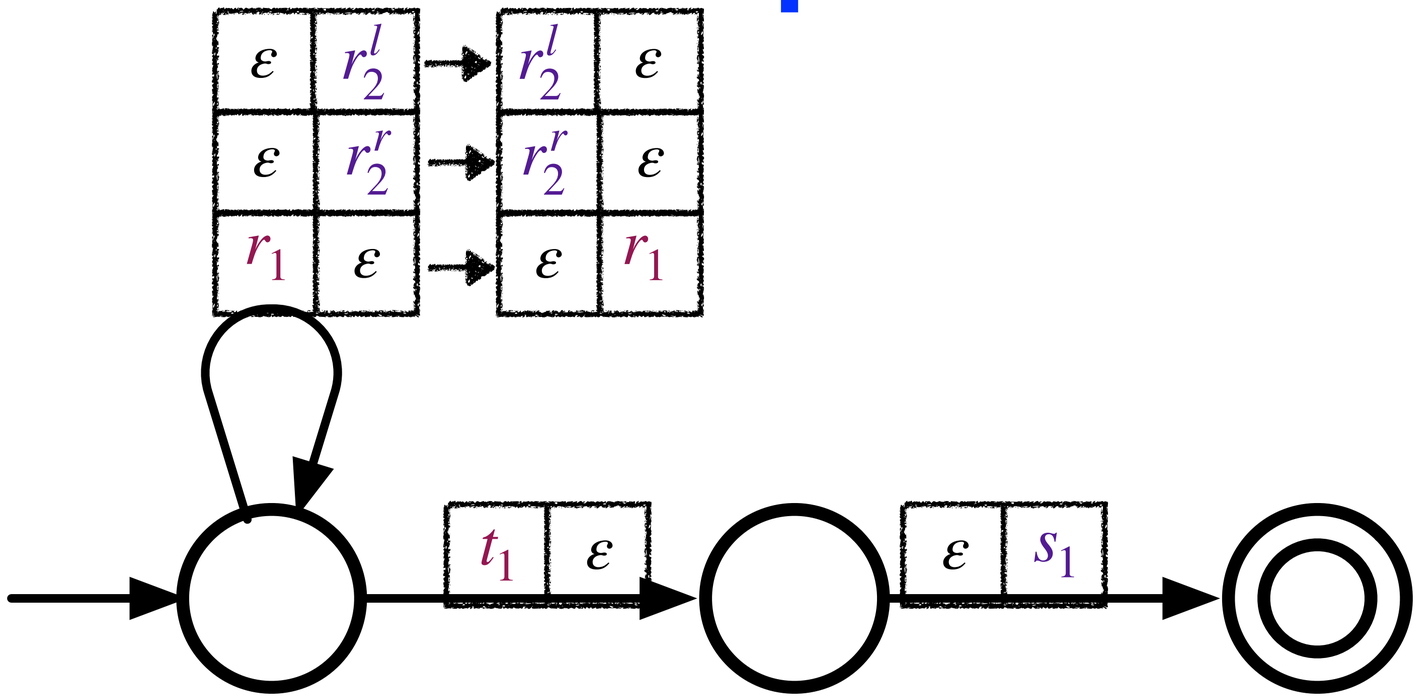
aside: multi-tape transducers



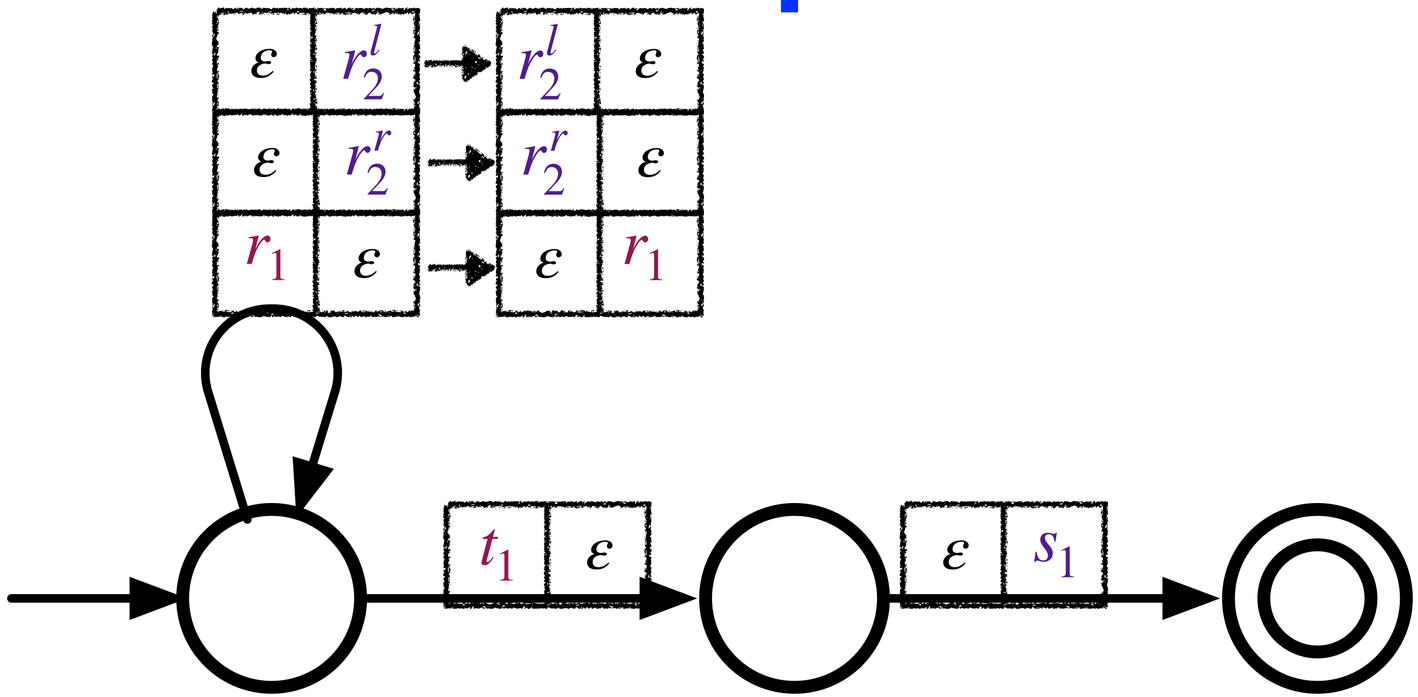
aside: multi-tape transducers



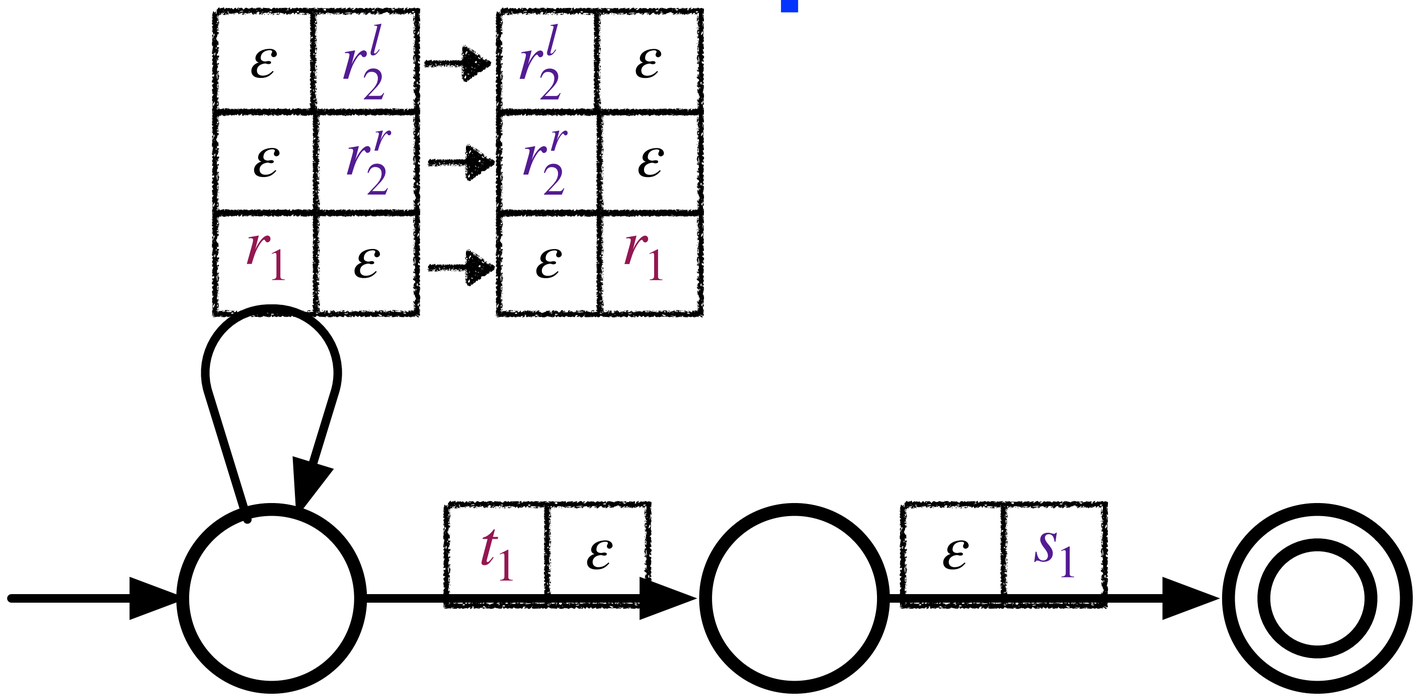
aside: multi-tape transducers



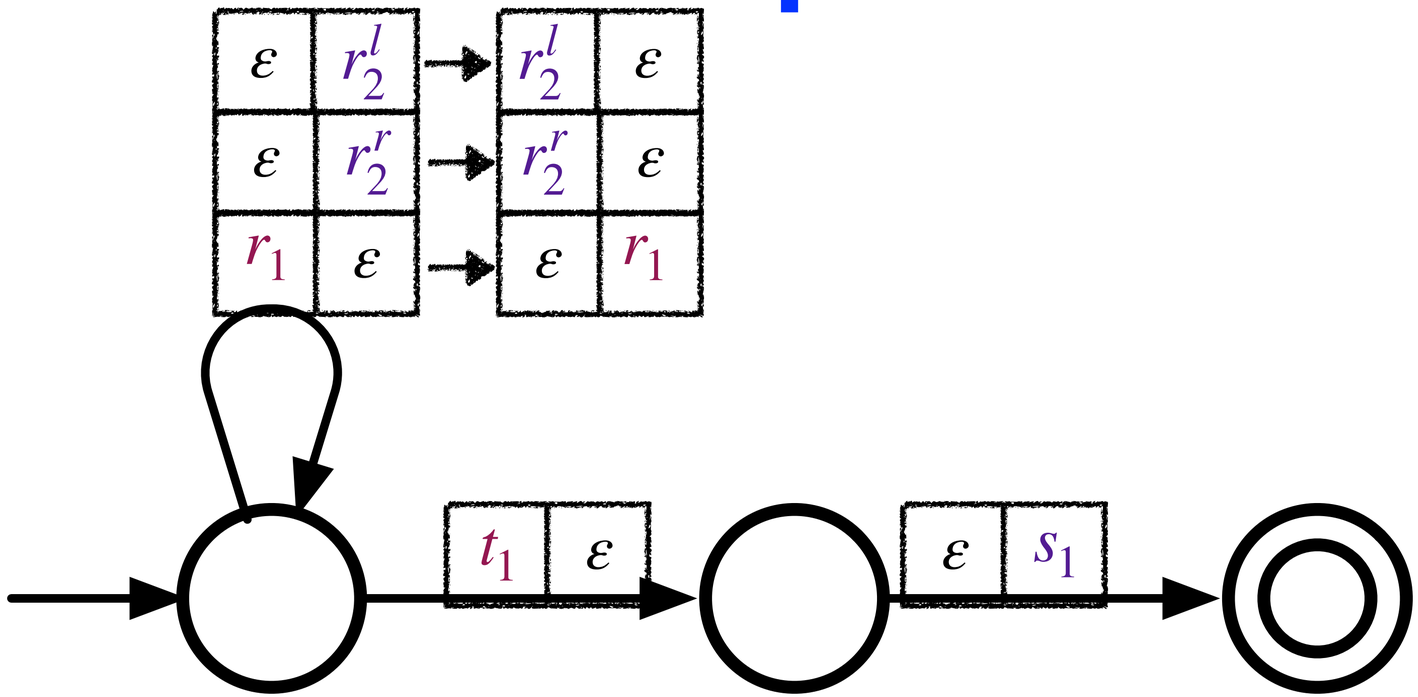
aside: multi-tape transducers



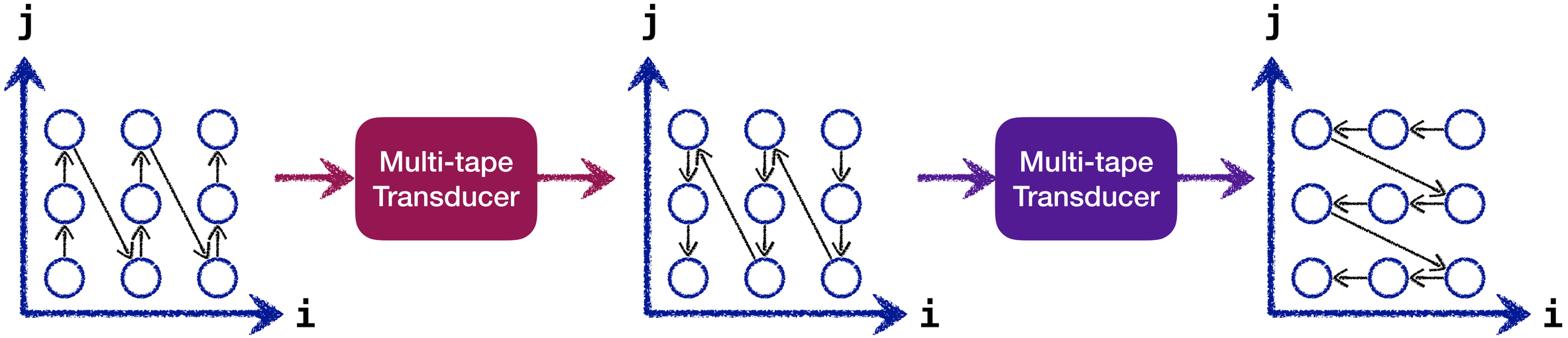
aside: multi-tape transducers



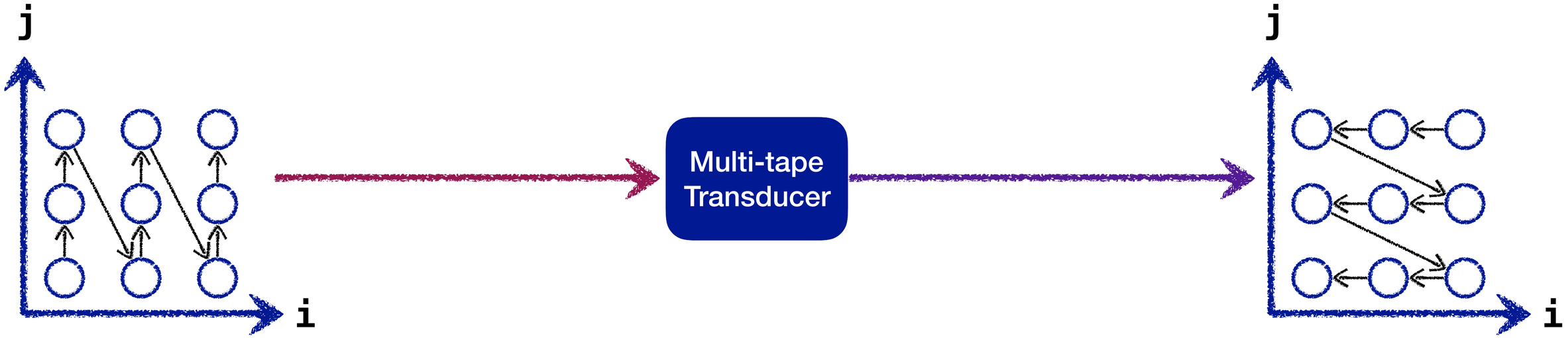
aside: multi-tape transducers



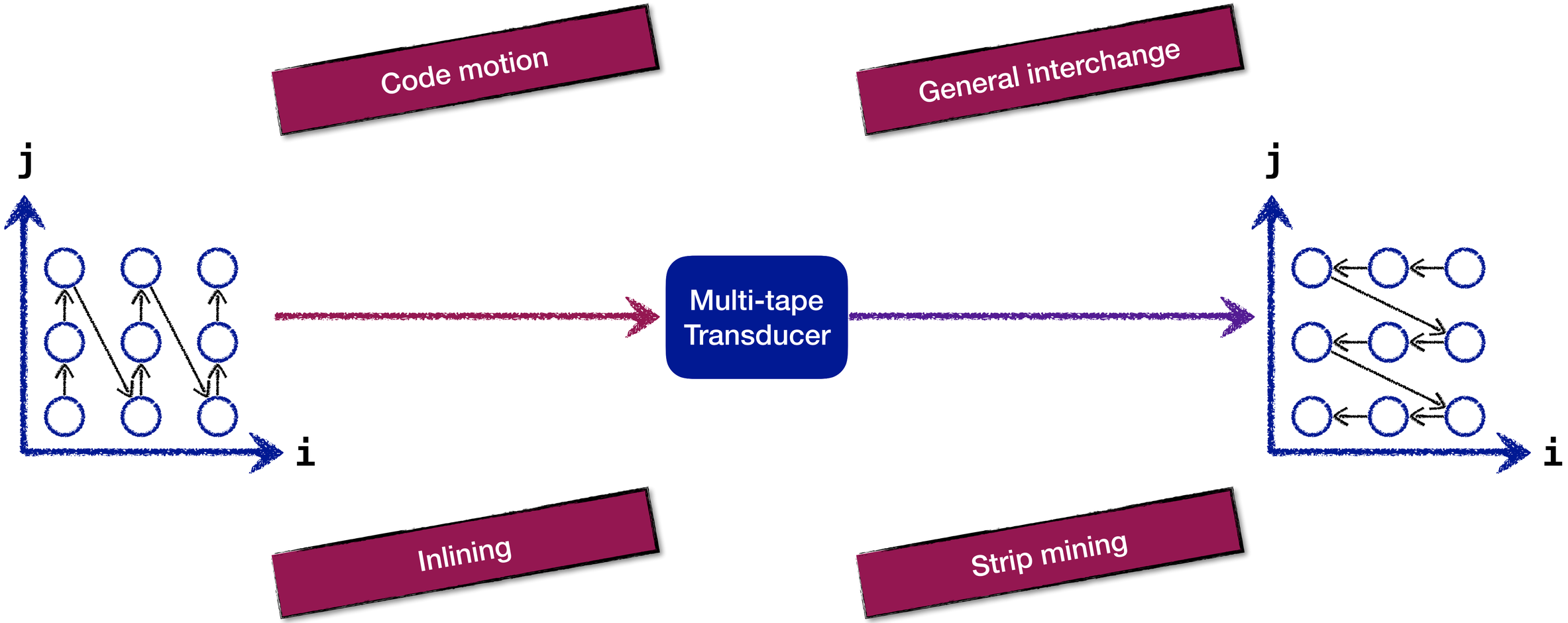
composing transformations



composing transformations



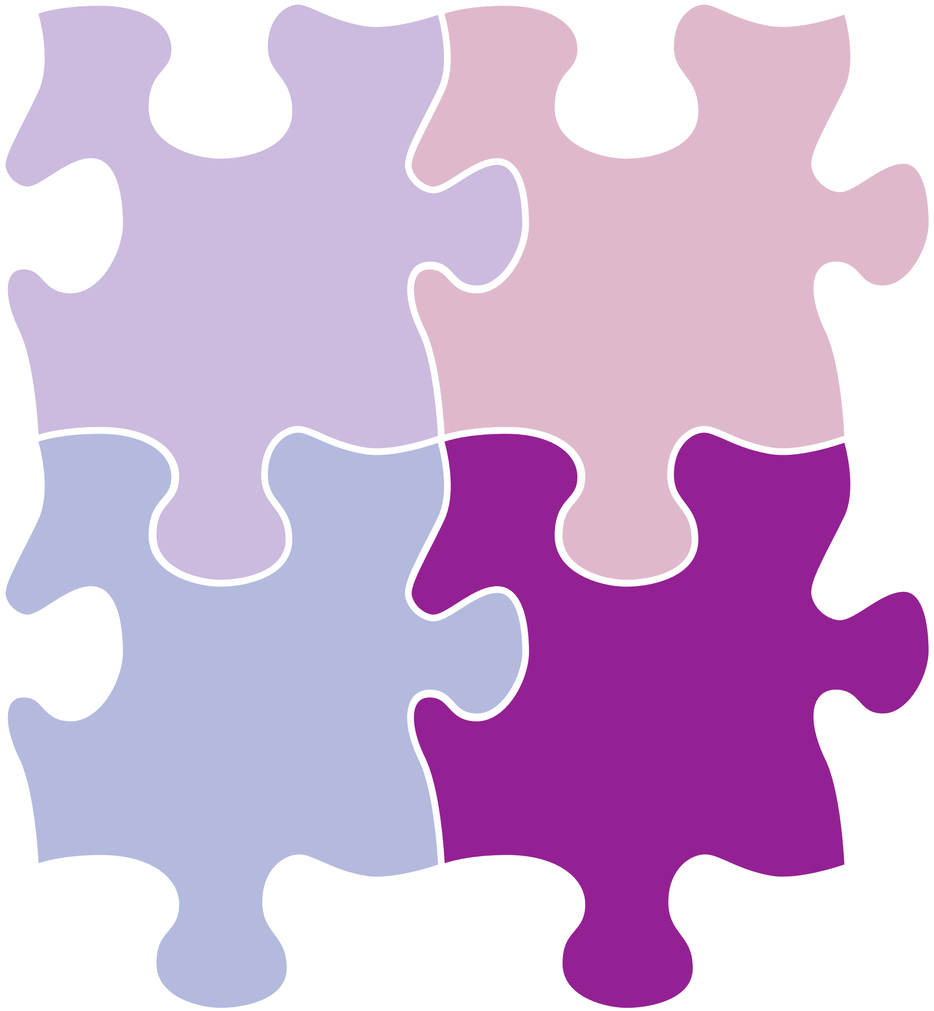
basic transformations



outline

iteration space representation

soundness check

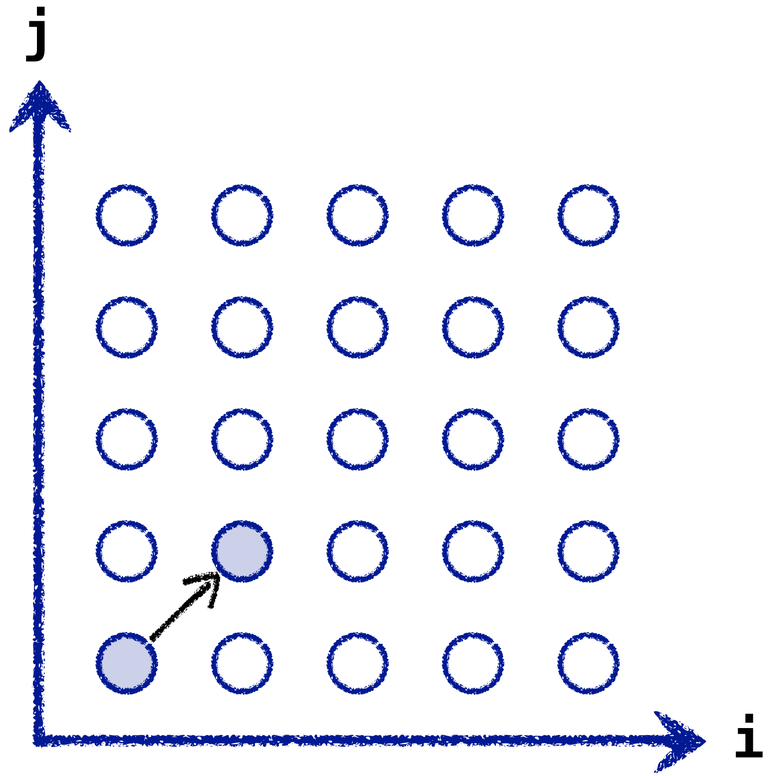


transformation representation

dependence representation

dependences

```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```

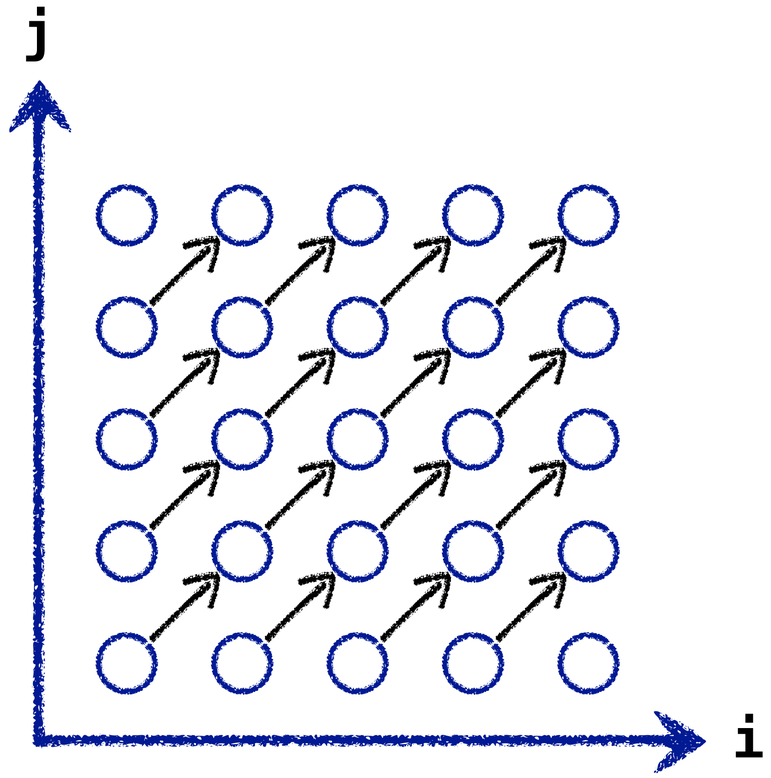


(i, j) = (0, 0): read A[1][1] write A[0][0]

(i, j) = (1, 1): read A[2][2] write A[1][1]

dependences

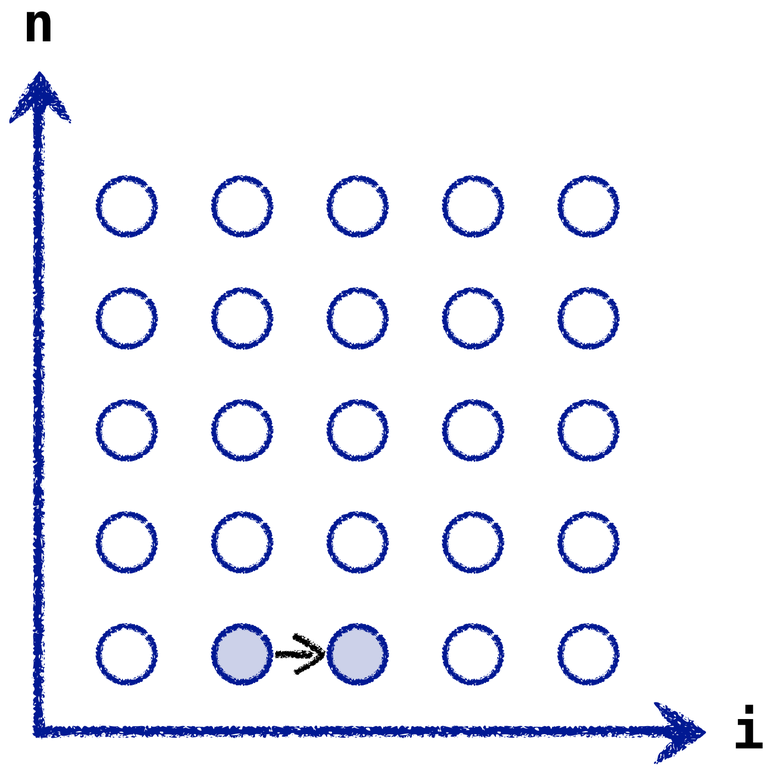
```
for (i = 0; i < N; i++)
  for (j = 0; j < N; j++)
    a[i][j] = 2*a[i+1][j+1]
```



Distance vector: (+1 , +1)

dependences

```
outer(int i, node n)
  if (i >= N) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```



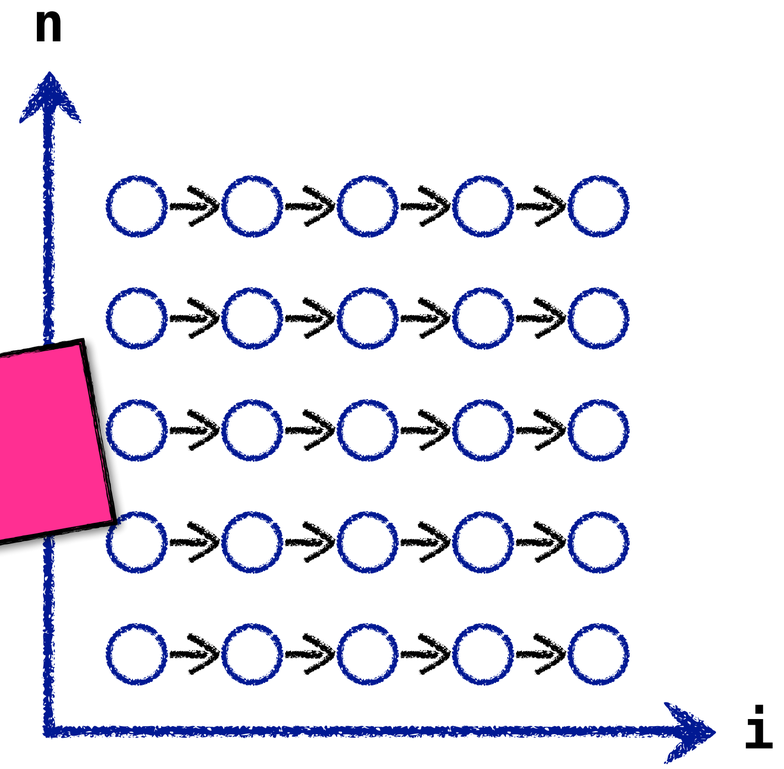
(i, n) = (0, root.l): read root.l.x[1] write root.l.x[0]

(i, n) = (1, root.l): read root.l.x[2] write root.l.x[1]

dependences

```
outer(int i, node n)
  if (i >= N) return
  traverse(i, root)
  outer(i + 1, root)

traverse(int i, node n)
  if (!n) return;
  traverse(i, n.l)
  traverse(i, n.r)
  n.x[i] = 2*n.x[i+1]
```

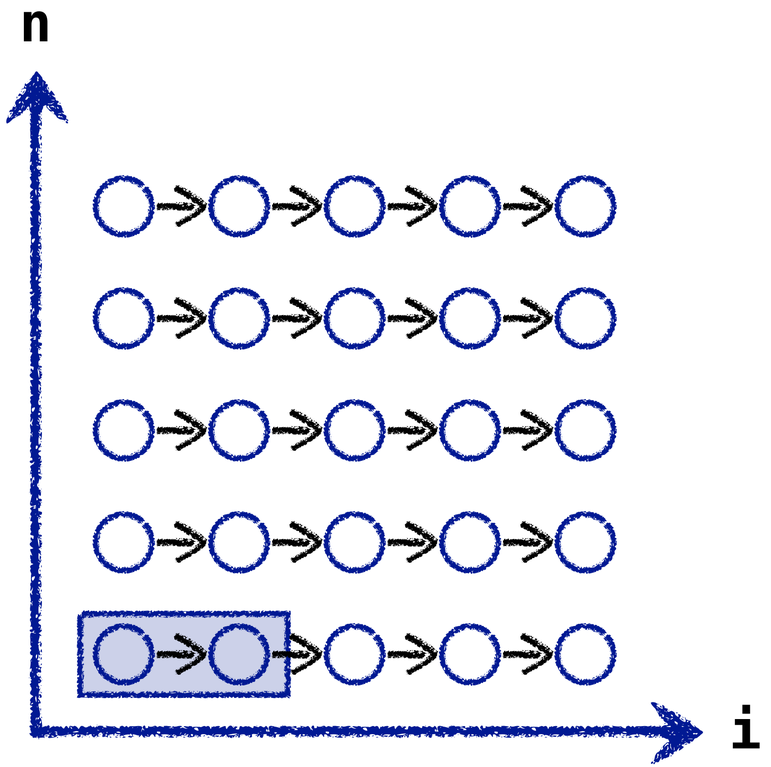
Need a succinct representation of dependences



dependences

source

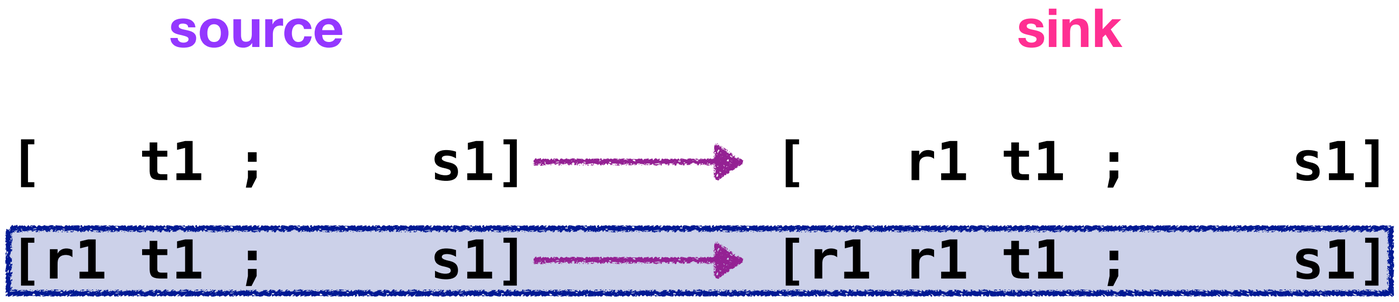
sink

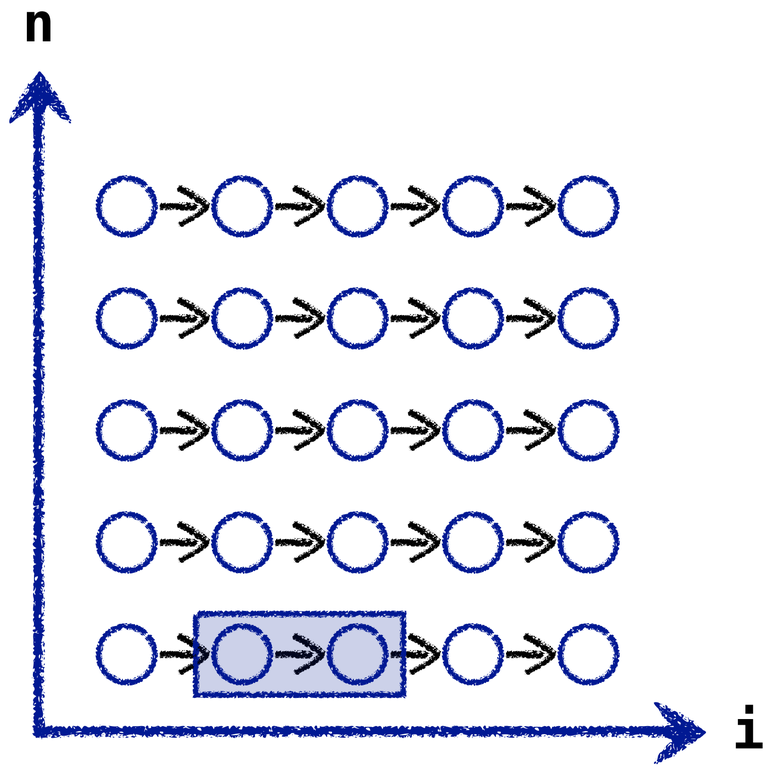


dependences

source

sink



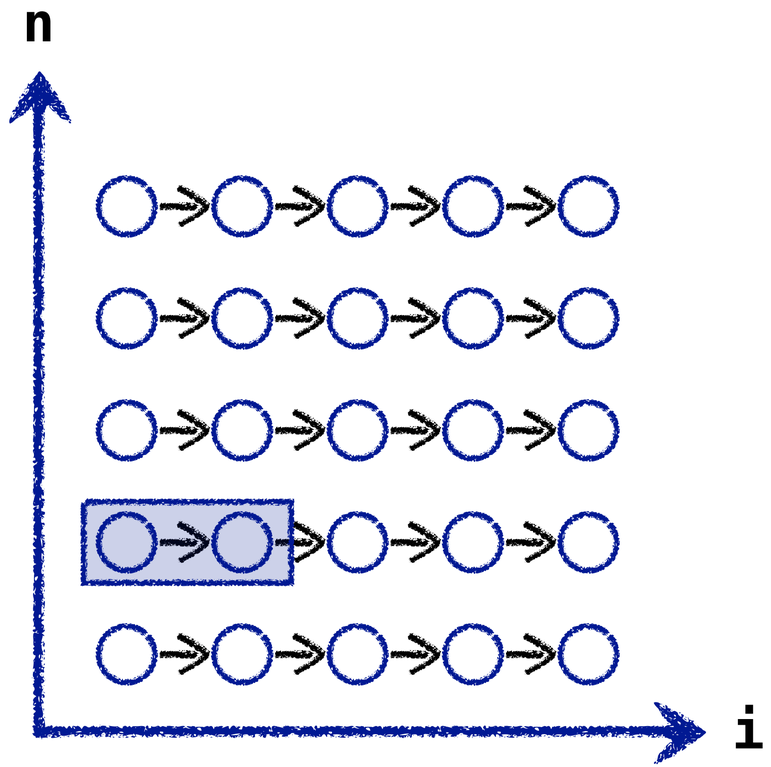


dependences

source

sink

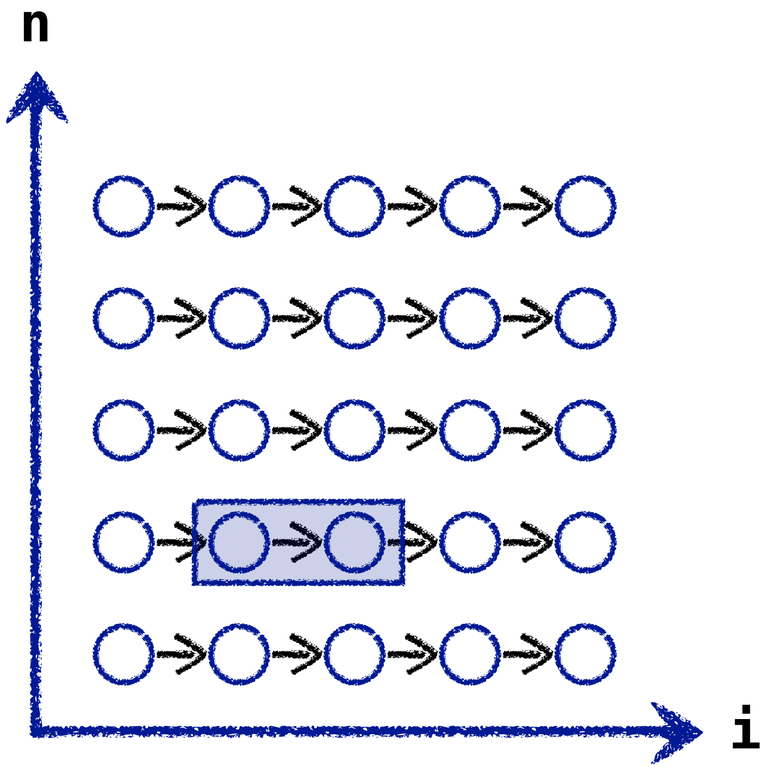
```
[    t1 ;      s1] ⟶ [    r1 t1 ;      s1]
[r1 t1 ;      s1] ⟶ [r1 r1 t1 ;      s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
```



dependences

source → sink

```
[    t1 ;       s1] ⟶ [    r1 t1 ;       s1]
[r1 t1 ;        s1] ⟶ [r1 r1 t1 ;        s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
[r1 t1 ; r2l s1] ⟶ [r1 r1 t1 ; r2l s1]
```

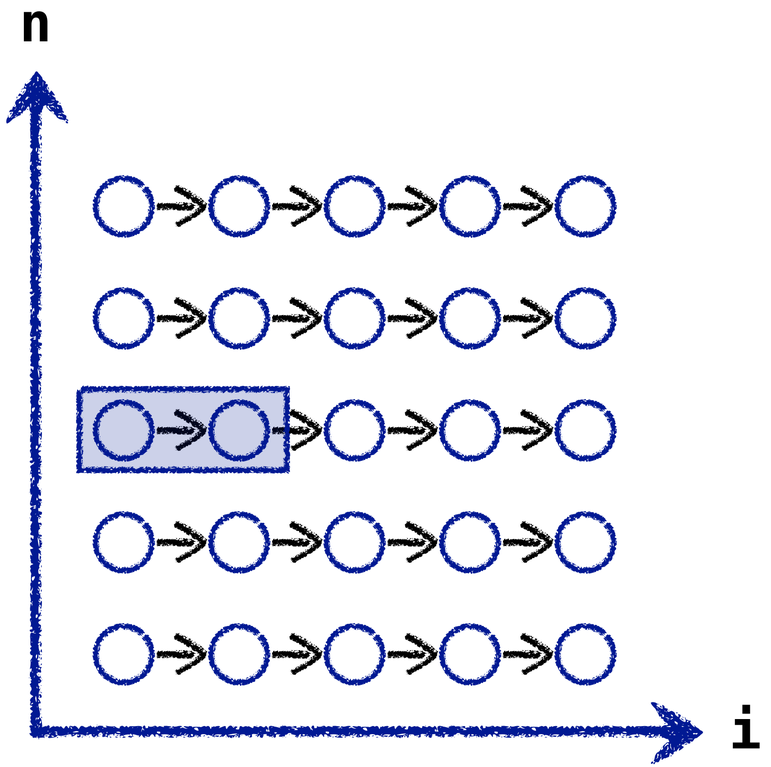


dependences

source

sink

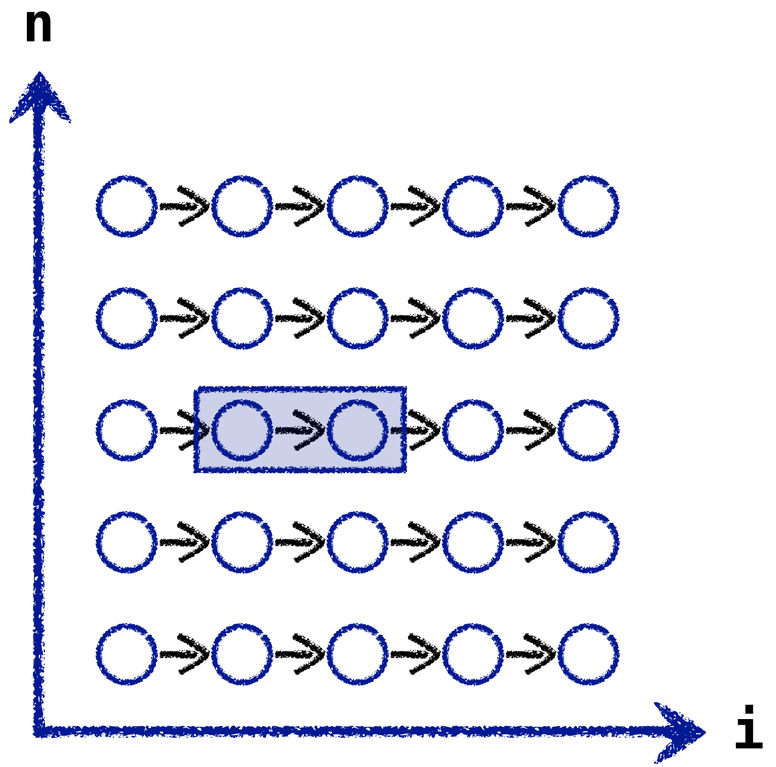
```
[    t1 ;       s1] ⟶ [    r1 t1 ;       s1]
[r1 t1 ;       s1] ⟶ [r1 r1 t1 ;       s1]

[    t1 ; r2l  s1] ⟶ [    r1 t1 ; r2l  s1]
[r1 t1 ; r2l  s1] ⟶ [r1 r1 t1 ; r2l  s1]

[    t1 ; r2r  s1] ⟶ [    r1 t1 ; r2r  s1]
```



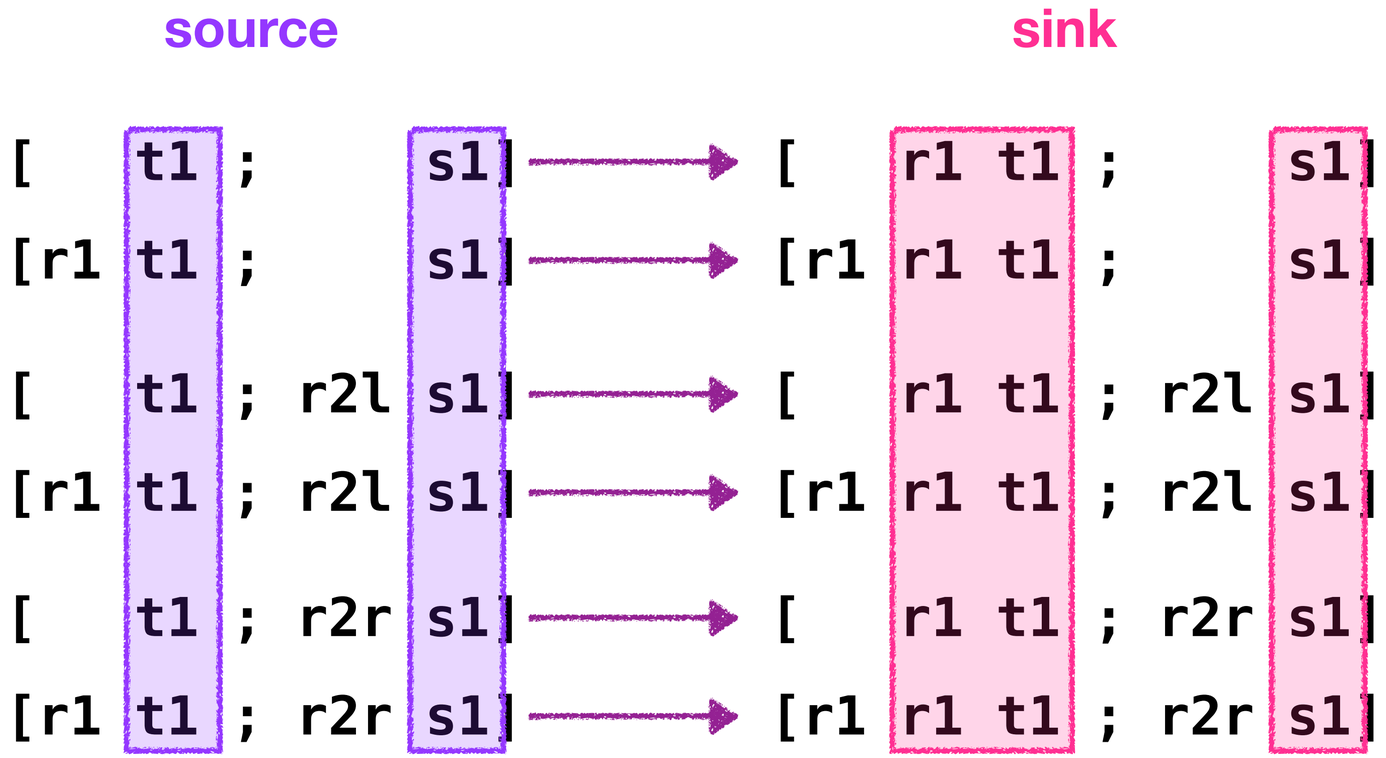
dependences

source → sink

```
[    t1 ;       s1] ⟶ [    r1 t1 ;       s1]
[r1 t1 ;       s1] ⟶ [r1 r1 t1 ;       s1]

[    t1 ; r2l  s1] ⟶ [    r1 t1 ; r2l  s1]
[r1 t1 ; r2l  s1] ⟶ [r1 r1 t1 ; r2l  s1]

[    t1 ; r2r  s1] ⟶ [    r1 t1 ; r2r  s1]
[r1 t1 ; r2r  s1] ⟶ [r1 r1 t1 ; r2r  s1]
```



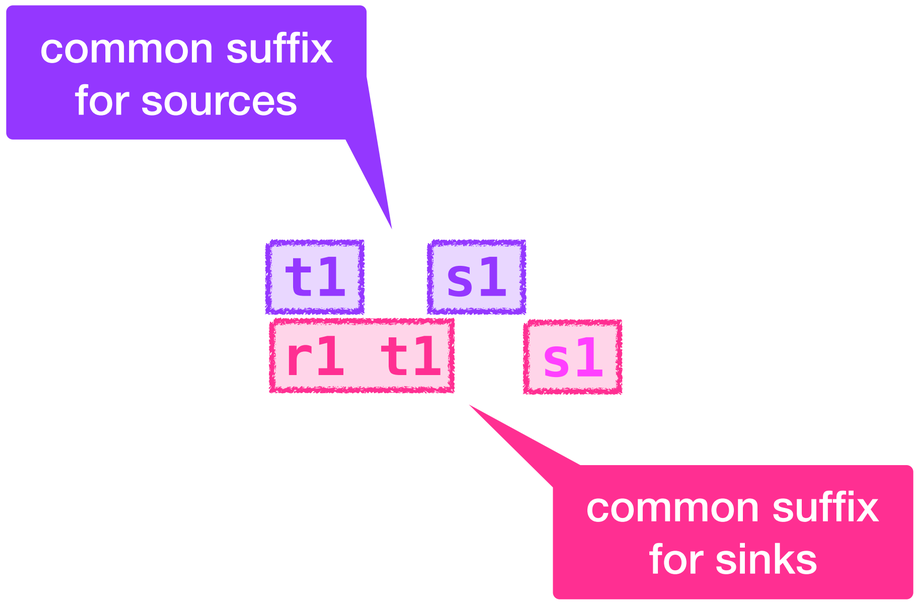
dependences



dependences

source → sink

```
[    t1 ;      s1] ⟶ [    r1 t1 ;      s1]
[r1 t1 ;      s1] ⟶ [r1 r1 t1 ;      s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
[r1 t1 ; r2l s1] ⟶ [r1 r1 t1 ; r2l s1]

[    t1 ; r2r s1] ⟶ [    r1 t1 ; r2r s1]
[r1 t1 ; r2r s1] ⟶ [r1 r1 t1 ; r2r s1]
```



dependences

source → sink

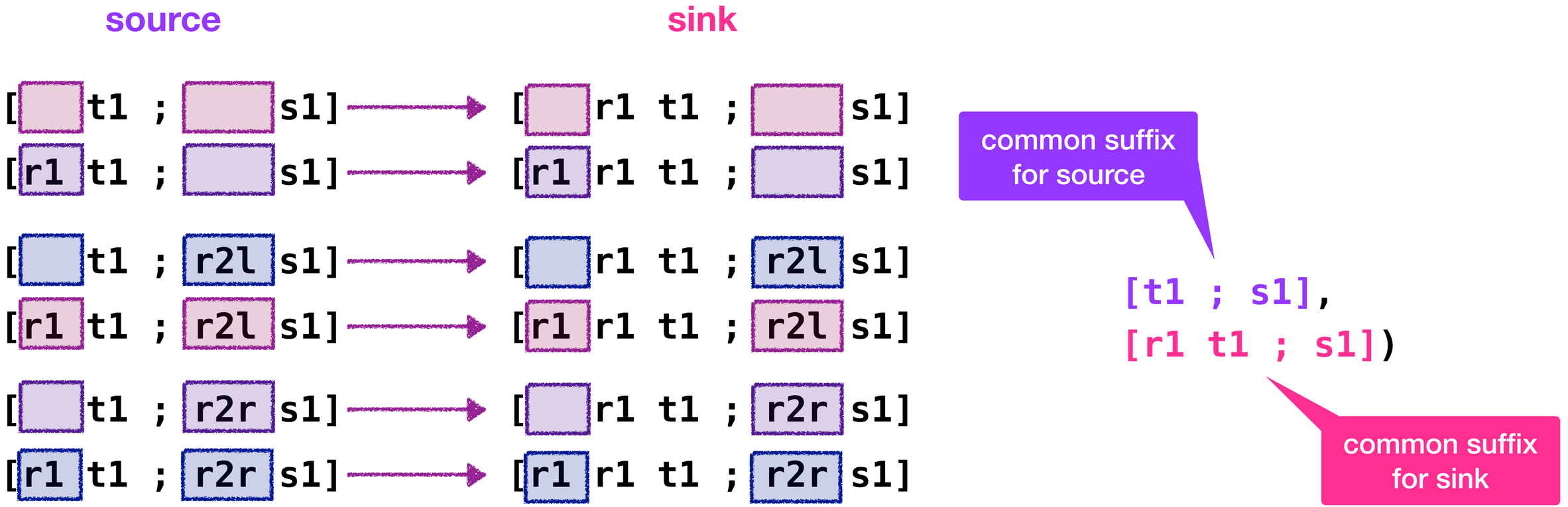
```
[    t1 ;      s1] ⟶ [    r1 t1 ;      s1]
[r1 t1 ;      s1] ⟶ [r1 r1 t1 ;      s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
[r1 t1 ; r2l s1] ⟶ [r1 r1 t1 ; r2l s1]

[    t1 ; r2r s1] ⟶ [    r1 t1 ; r2r s1]
[r1 t1 ; r2r s1] ⟶ [r1 r1 t1 ; r2r s1]
```

common suffix for source

```
[t1 ; s1],
[r1 t1 ; s1])
```

common suffix for sink

dependences



dependences

source → sink

```
[    t1 ;      s1] ⟶ [    r1 t1 ;      s1]
[r1 t1 ;      s1] ⟶ [r1 r1 t1 ;      s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
[r1 t1 ; r2l s1] ⟶ [r1 r1 t1 ; r2l s1]

[    t1 ; r2r s1] ⟶ [    r1 t1 ; r2r s1]
[r1 t1 ; r2r s1] ⟶ [r1 r1 t1 ; r2r s1]
```

regular relation prefix for each dependence

```
[r1* ; (r2l|r2r)*]
[t1 ; s1],
[r1 t1 ; s1])
```

dependences

source → sink

```
[    t1 ;      s1] ⟶ [    r1 t1 ;      s1]
[r1 t1 ;      s1] ⟶ [r1 r1 t1 ;      s1]

[    t1 ; r2l s1] ⟶ [    r1 t1 ; r2l s1]
[r1 t1 ; r2l s1] ⟶ [r1 r1 t1 ; r2l s1]

[    t1 ; r2r s1] ⟶ [    r1 t1 ; r2r s1]
[r1 t1 ; r2r s1] ⟶ [r1 r1 t1 ; r2r s1]
```

regular relation prefix for each dependence

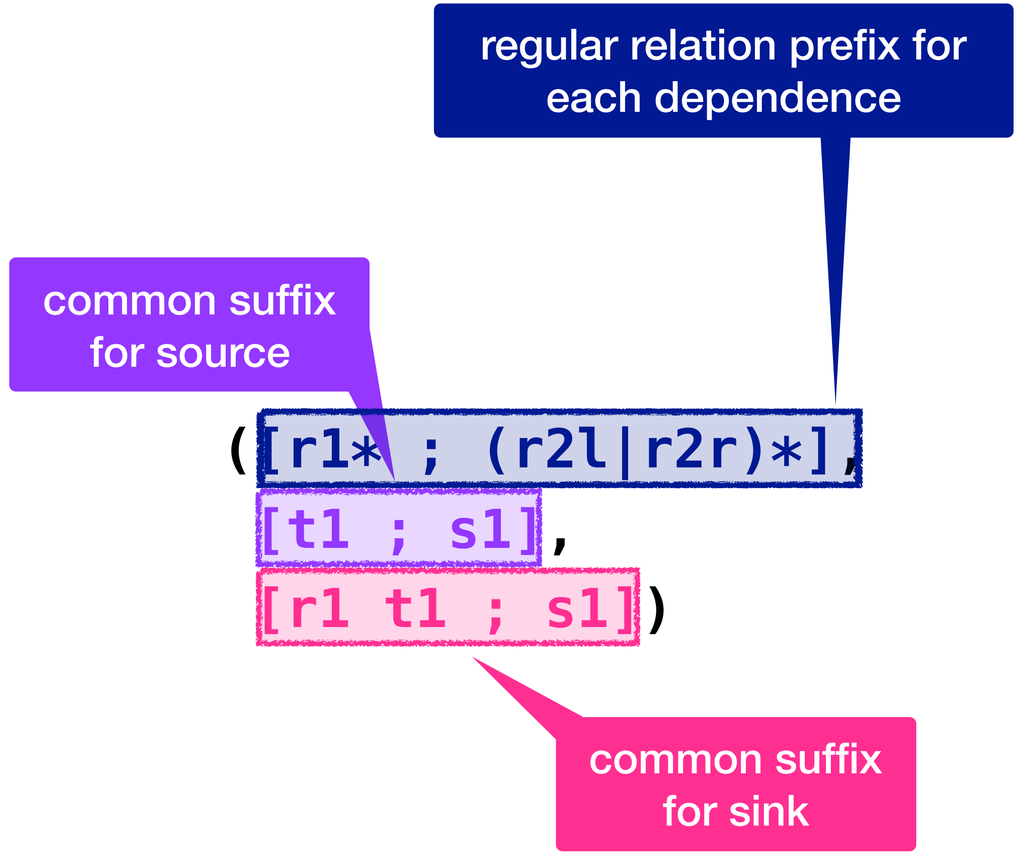
common suffix for source

common suffix for sink

```
([r1* ; (r2l|r2r)*],
 [t1 ; s1],
 [r1 t1 ; s1])
```

dependences

- A **witness tuple** defines a possibly infinite set of dependences:



dependences

- A **witness tuple** defines a possibly infinite set of dependences:

regular relation prefix for each dependence

common suffix for source

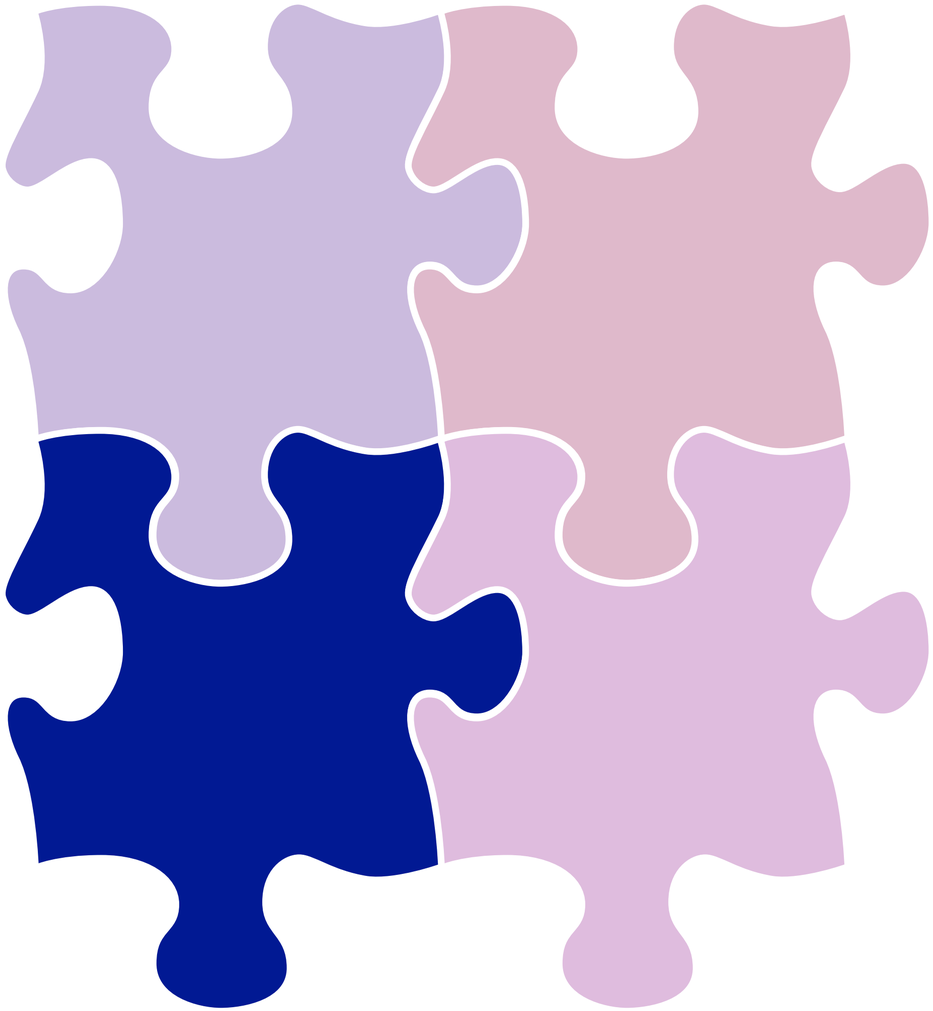
```
([r1* ; (r2l|r2r)*],
 [t1 ; s1],
 [r1 t1 ; s1])
```

common suffix for sink

outline

iteration space representation

transformation representation



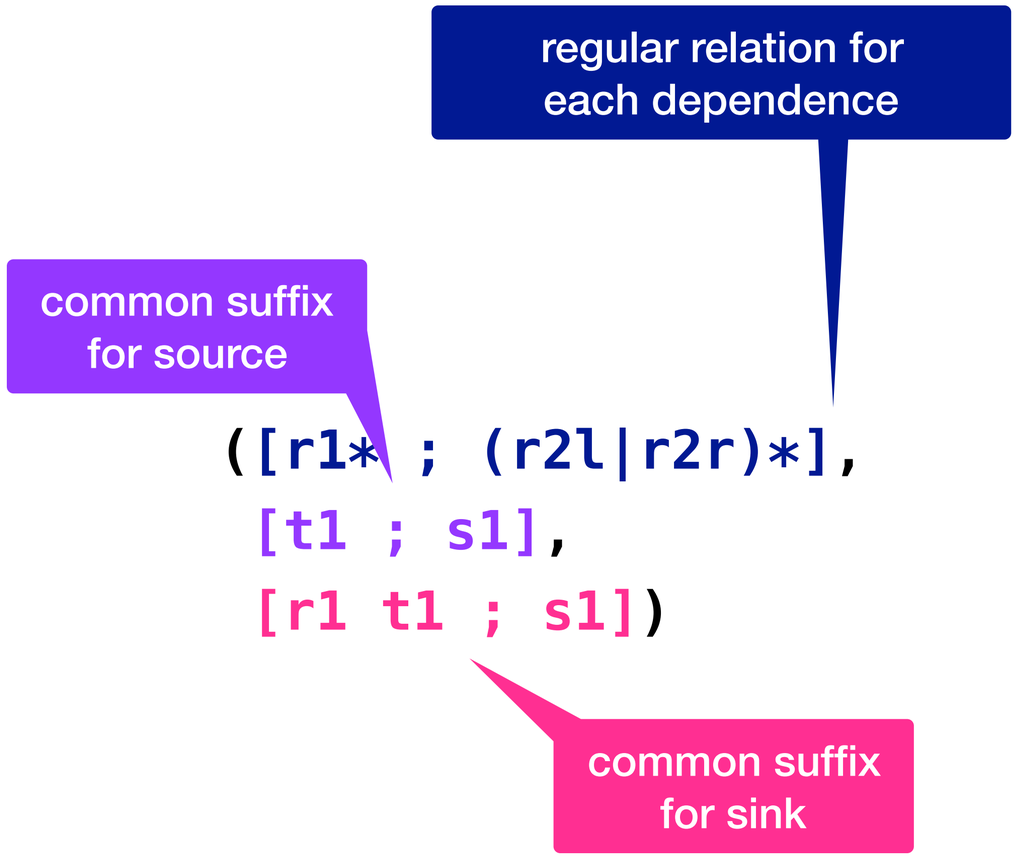
soundness check

dependence representation

soundness check

- **Goal:** determine whether any dependences are broken by transformations
  - **Sound:** must always tell when a dependence is broken (while avoiding false positives)
  - **Compositional:** must check soundness of overall transformation
  - **Decidable:** procedure must terminate

soundness check



check one prefix

regular relation for each dependence

common suffix for source

```
([r1* ; (r2l|r2r)*],
 [t1 ; s1],
 [r1 t1 ; s1])
```

common suffix for sink

Multi-tape Transducer

check one prefix

regular relation for each dependence

common suffix for source

```
([r1* ; (r2l|r2r)*],
 [t1 ; s1],
 [r1 t1 ; s1])
```

common suffix for sink

Prefix Transducer

check one prefix

regular relation for each dependence

common suffix for source

```
([r1* ; (r2l|r2r)*],
 [t1 ; s1],
 [r1 t1 ; s1])
```

common suffix for sink

transformed version of source

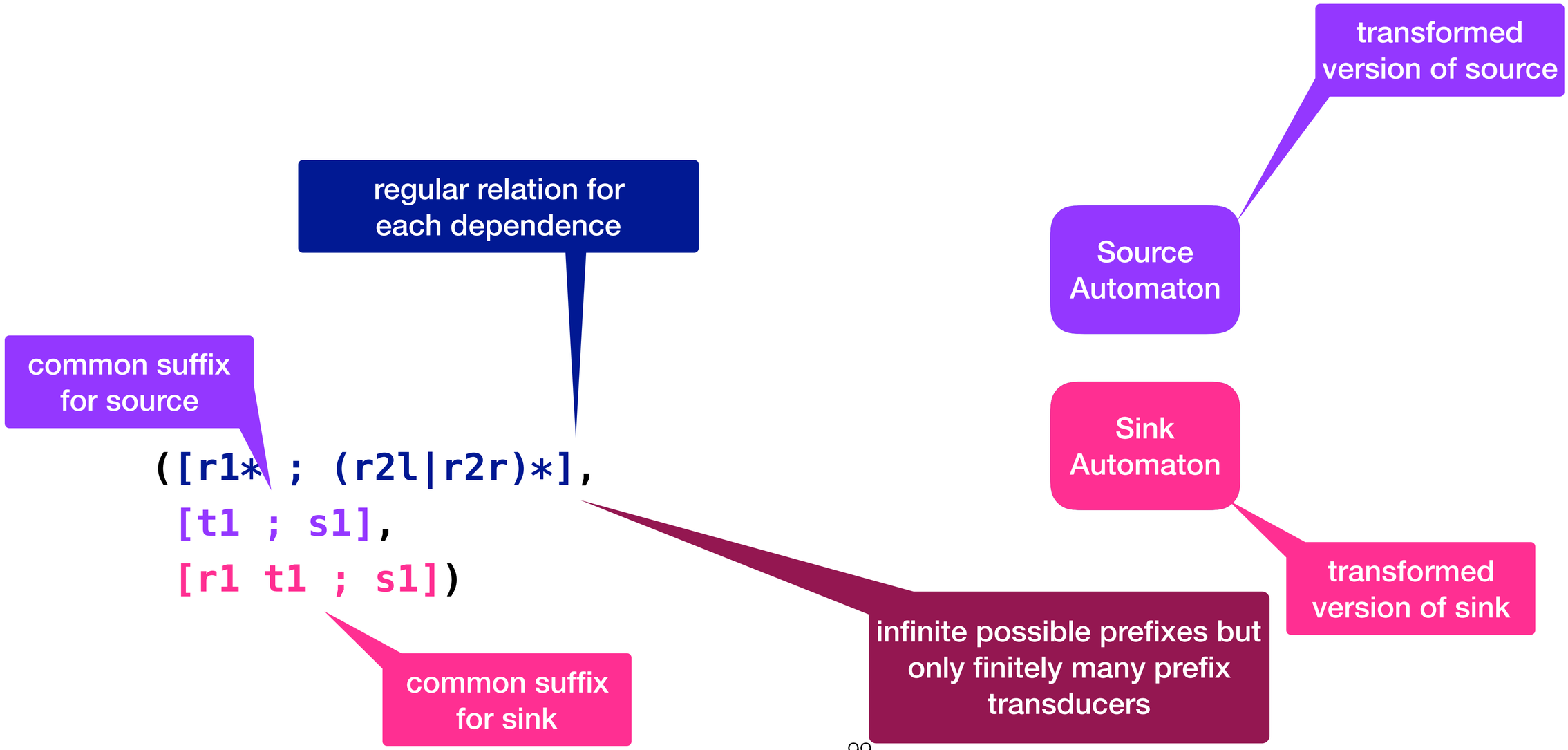
Source Automaton

dependences preserved if all transformed sources precede transformed sinks

Sink Automaton

transformed version of sink

decidability



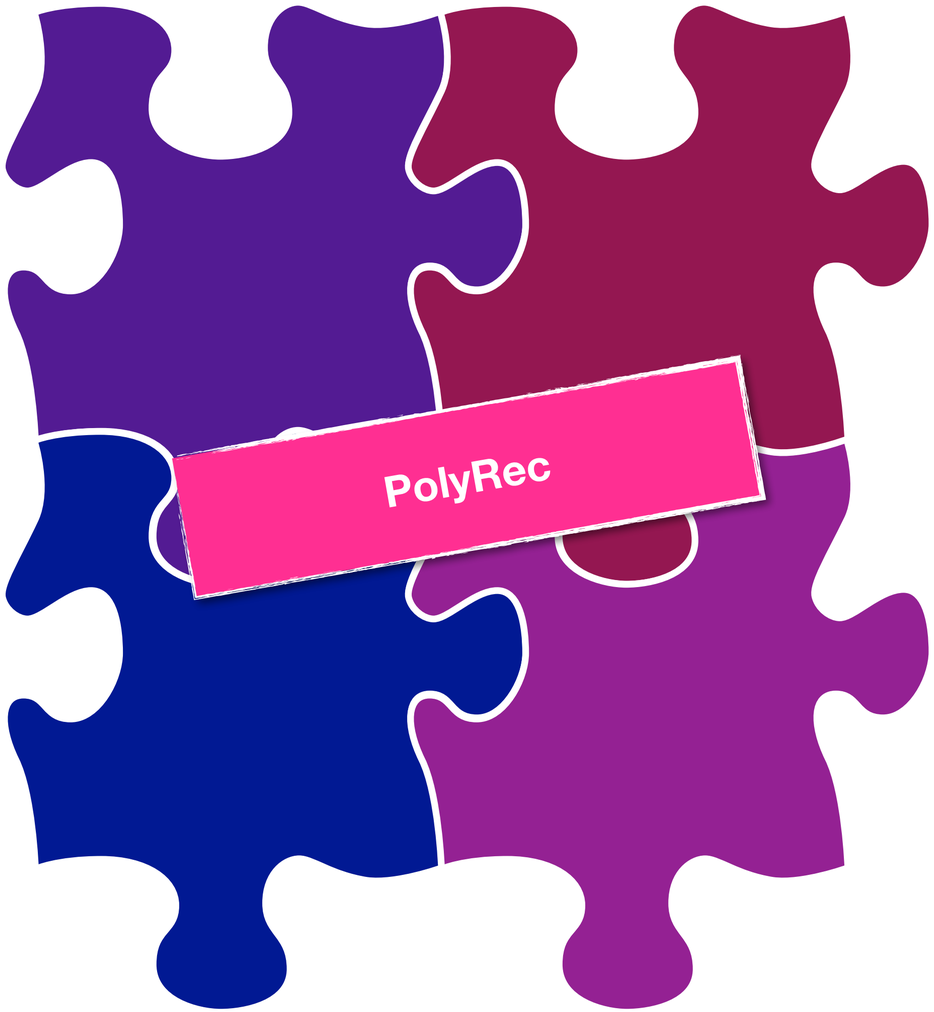
demo

- TODO

conclusions

iteration space representation

transformation representation



soundness check

dependence representation

A Framework for Compositional Transformations of Recursion and Loops

Milind Kulkarni
Electrical and Computer Engineering, Purdue University