# Verifying Concurrent Graph Algorithms

**Azalea Raad**    Aquinas Hobor    Jules Villard    Philippa Gardner

Imperial College London
National University of Singapore

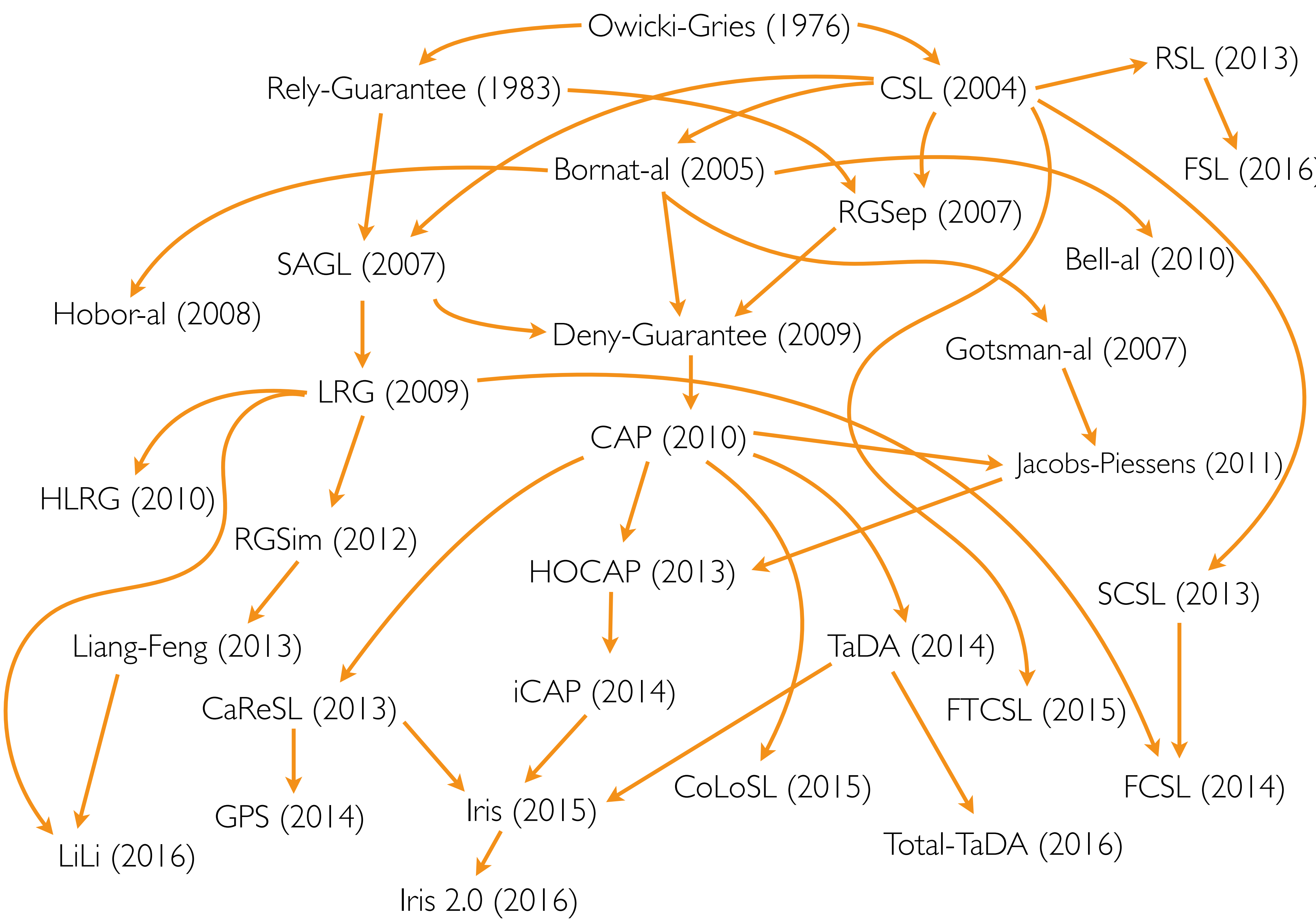
Northern Concurrency Meeting
13 January 2017

# Concurrent Program Logic Genealogy

Verifying **concurrent** algorithms is difficult…

# Concurrent Program Logic Genealogy

Verifying **concurrent** algorithms is difficult…




Graph credit: Ilya Sergey

# Graph Algorithms

Verifying **graph** algorithms is difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)

# Graph Algorithms

Verifying **graph** algorithms is difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)

**Local reasoning, separation and aliasing**


Richard Bornat
School of Computing Science
Middlesex University
R.Bornat@mdx.ac.uk

Cristiano Calcagno
Department of Computing
Imperial College
University of London
ccris@doc.ic.ac.uk

Peter O'Hearn
Department of Computer Science
Queen Mary College
University of London
ohearn@dcs.qmul.ac.uk



**Abstract**

Structures built by pointer aliasing, such as DAGs and graphs, are notoriously tricky to deal with. The mechanisms of separation logic can deal with these structures, but so far this has been done by the maintenance of a global invariant. Specifications and proofs which use local reasoning, and which may point the way to a structured programming for pointers, are discussed. An idiom for inclusion – where one structure is included in another, is presented. A …


Pointers have been thought of as the gotos of data structuring, and so they can be, but if we can organise things so that our data structures and our programs come apart neatly in similar ways – structured reasoning for the heap – we shall perhaps be able to use pointers reliably. Separation logic facilitates local reasoning about parts of data structures: if an assignment or a procedure call alters only one corner of one data structure, then we can easily express and exploit the fact that it doesn't alter any other part of it nor any part of any separate data structure. The advantages can be seen most clearly in Yang's proof of the Schorr-Waite algorithm [12, 13], which elegantly avoids the struggles of [1] to show the same result using Hoare logic and array assignment. Apart from that work and Torp-Smith's proof of the Cheney algorithm [11], most work using separation logic has focussed on lists and trees, where separation is easily described.

In this paper we discuss three formal proofs (alas, not yet mechanically checked) of algorithms which deal with internal sharing of structure; two of the proofs deal with aliasing as well. Our aim has been to make the reasoning as local as possible. Although we can see considerable possibilities for further improvement, we can already claim significant success in extending the range of separation logic to new and difficult areas.

# Graph Algorithms

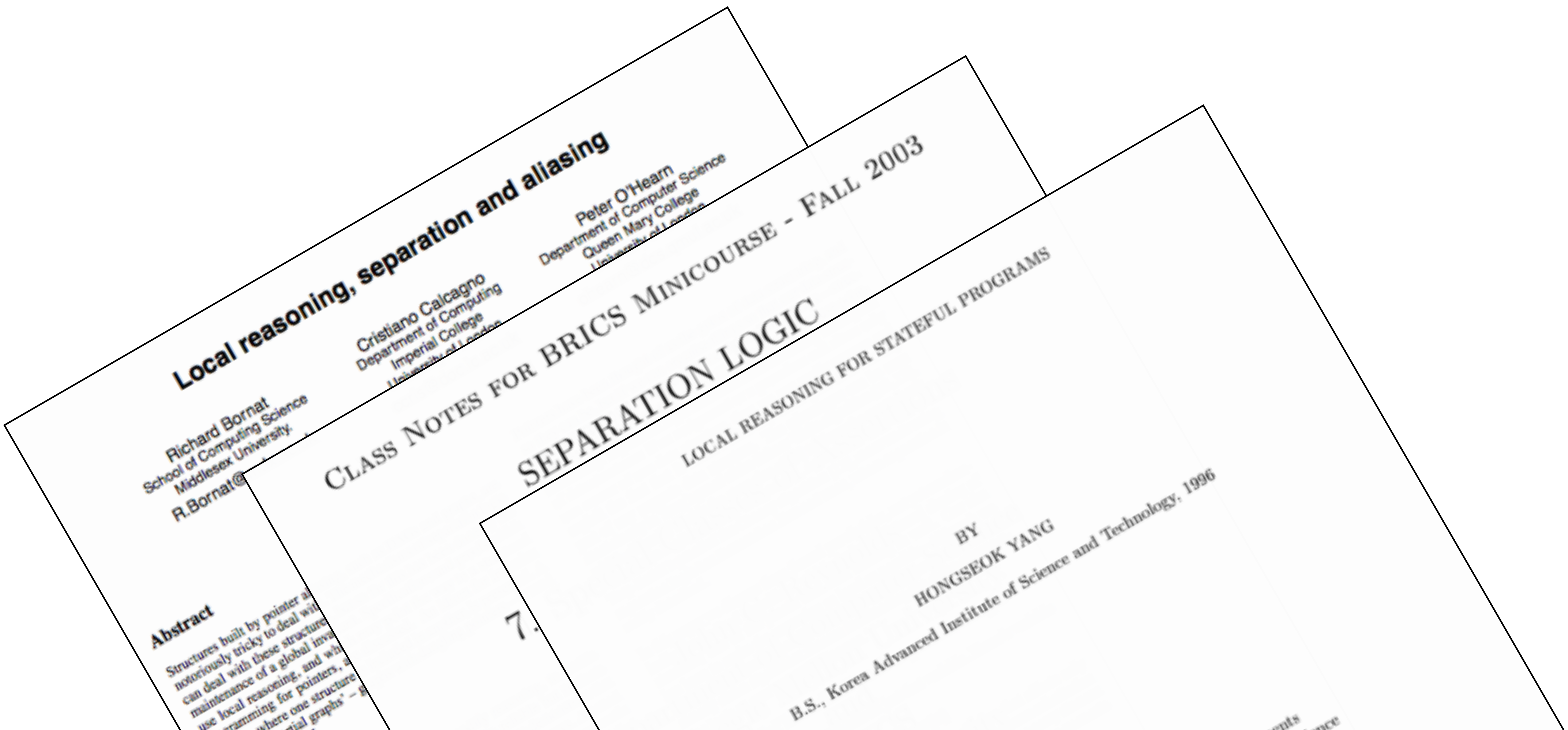
Verifying **graph** algorithms is difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)

# Graph Algorithms

Verifying **graph** algorithms is difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)



# Graph Algorithms

Verifying **graph** algorithms is difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)

# Concurrent Graph Algorithms

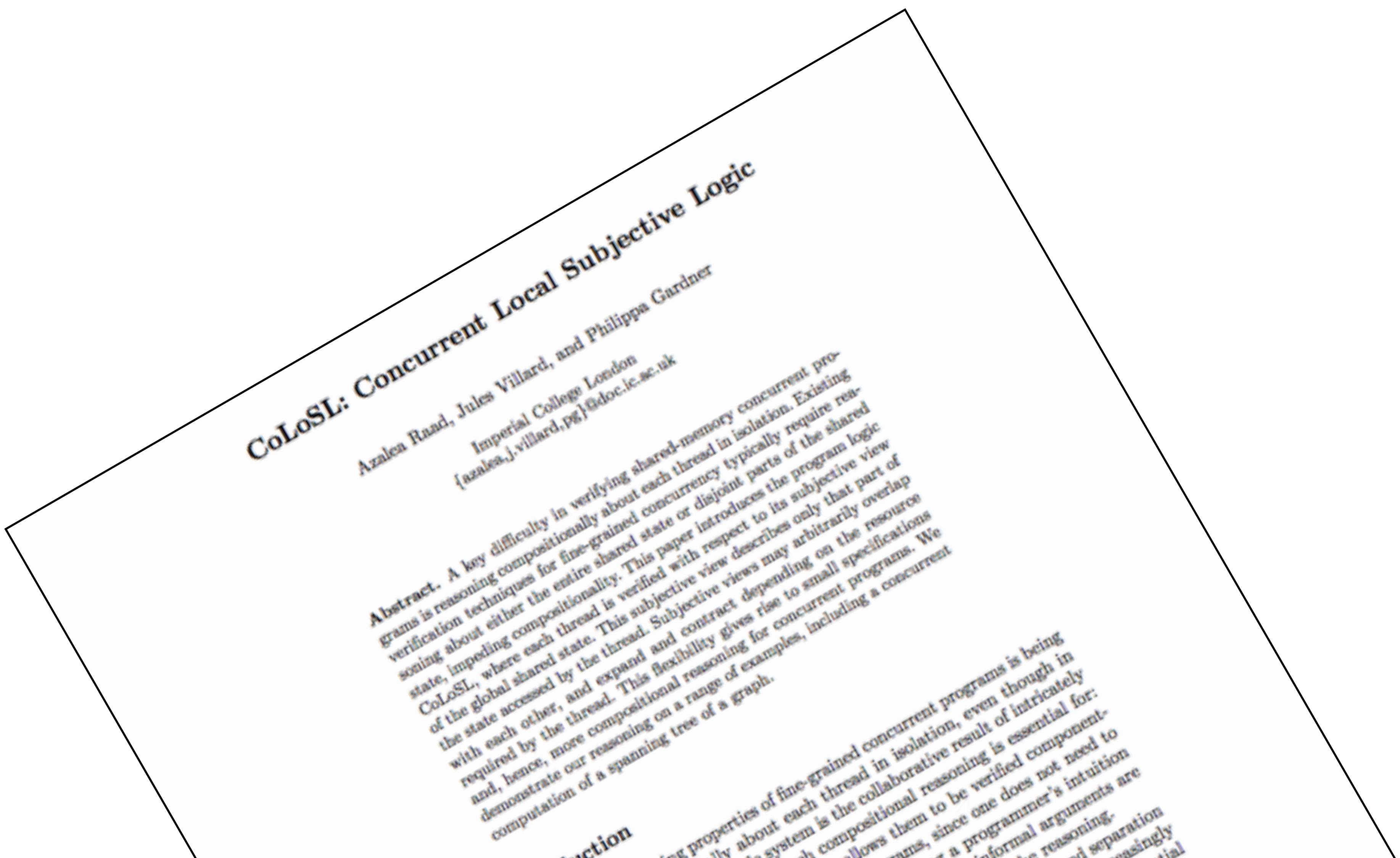
Verifying **concurrent graph** algorithms is even more difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)
- Reasoning about each thread in isolation

# Concurrent Graph Algorithms

Verifying **concurrent graph** algorithms is even more difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)
- Reasoning about each thread in isolation



**CoLoSL: Concurrent Local Subjective Logic**

Azalea Raad, Jules Villard, and Philippa Gardner

Imperial College London
{azalea,j.villard,pg}@doc.ic.ac.uk

**Abstract.** A key difficulty in verifying shared-memory concurrent programs is reasoning compositionally about each thread in isolation. Existing verification techniques for fine-grained concurrency typically require reasoning about either the entire shared state or disjoint parts of the shared state, impeding compositionality. This paper introduces the program logic CoLoSL, where each thread is verified with respect to its subjective view of the global shared state. This subjective view describes only that part of the state accessed by the thread. Subjective views may arbitrarily overlap with each other, and expand and contract depending on the resource required by the thread. This flexibility gives rise to small specifications and, hence, more compositional reasoning for concurrent programs. We demonstrate our reasoning on a range of examples, including a concurrent computation of a spanning tree of a graph.

# Concurrent Graph Algorithms

Verifying **concurrent graph** algorithms is even more difficult…

- Subtle correctness argument
- Overlapping structure (unspecified sharing via pointer aliasing)
  - Non-compositional reasoning (preventing the use of the frame rule)
- Reasoning about each thread in isolation

# Contributions

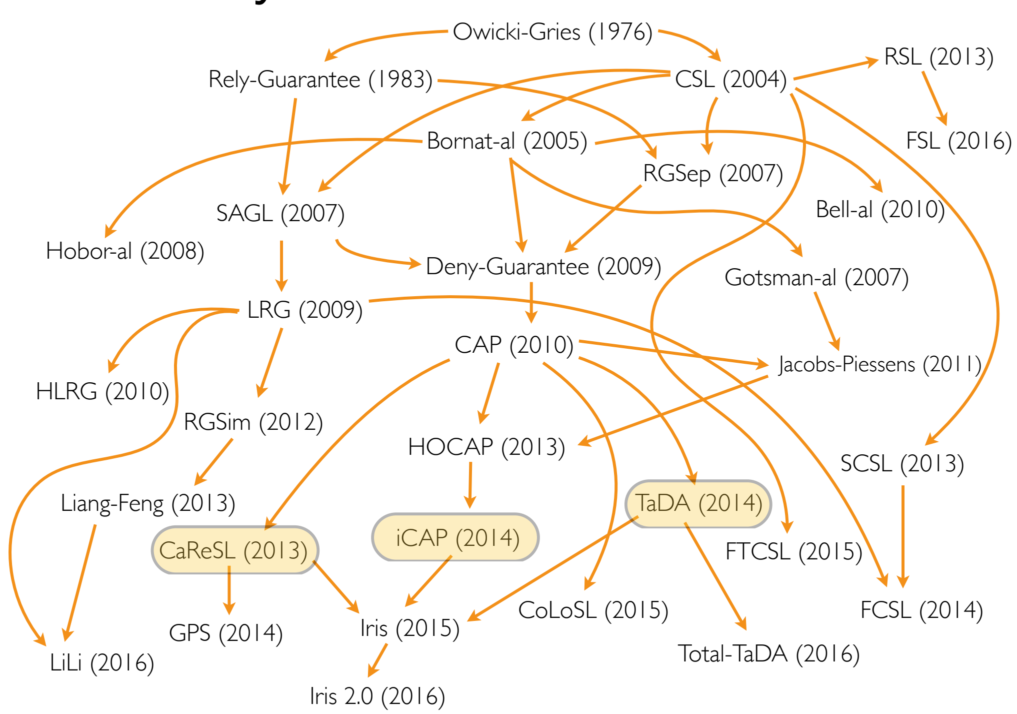
- Verified 4 concurrent fine-grained graph algorithms
  - Copying dags (directed acyclic graphs)
  - Speculative variant of Dijkstra's shortest path
  - Computing the spanning tree of a graph
  - Marking a graph

# Contributions

- Verified 4 concurrent fine-grained graph algorithms
  - Copying dags (directed acyclic graphs)
  - Speculative variant of Dijkstra’s shortest path
  - Computing the spanning tree of a graph
  - Marking a graph
- Presented a common proof pattern for graph algorithms
  - *Abstract mathematical* graphs for functional correctness
  - *Concrete spatial (heap-represented)* graphs for memory safety
  - Combined reasoning for full proof

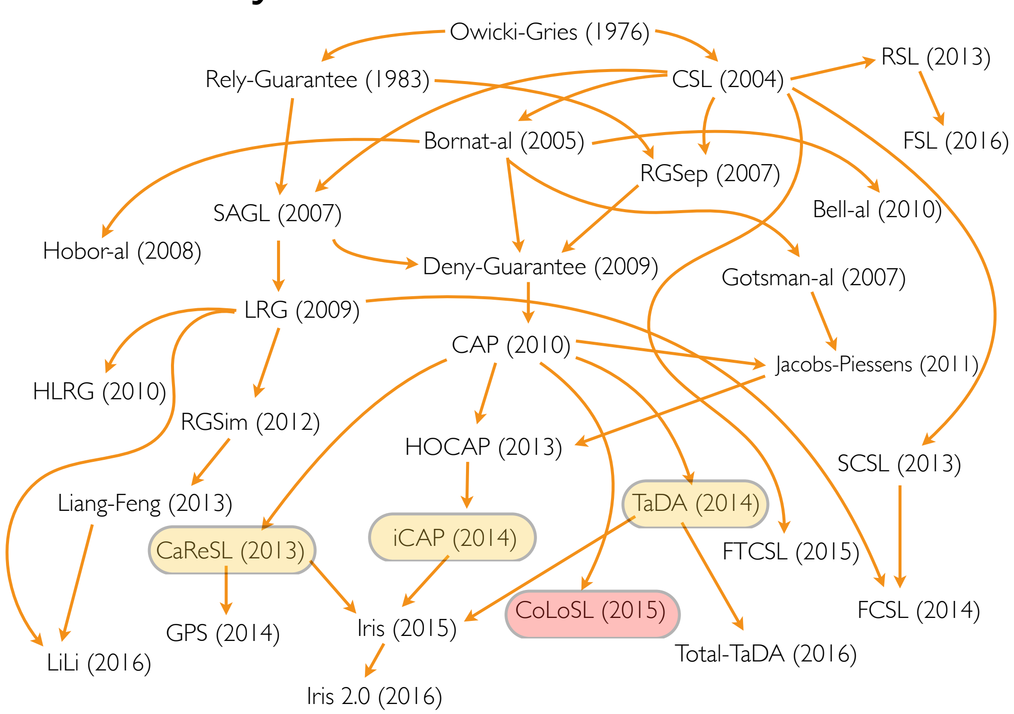
# Contributions

- Verified 4 concurrent fine-grained graph algorithms
  - Copying dags (directed acyclic graphs)
  - Speculative variant of Dijkstra's shortest path
  - Computing the spanning tree of a graph
  - Marking a graph
- Presented a common proof pattern for graph algorithms
  - <u>*Abstract mathematical*</u> graphs for functional correctness
  - <u>*Concrete spatial (heap-represented)*</u> graphs for memory safety
  - Combined reasoning for full proof
  - Inspired by existing logics where this pattern is "baked-in" to the model
  - "Baking-in" is unnecessary



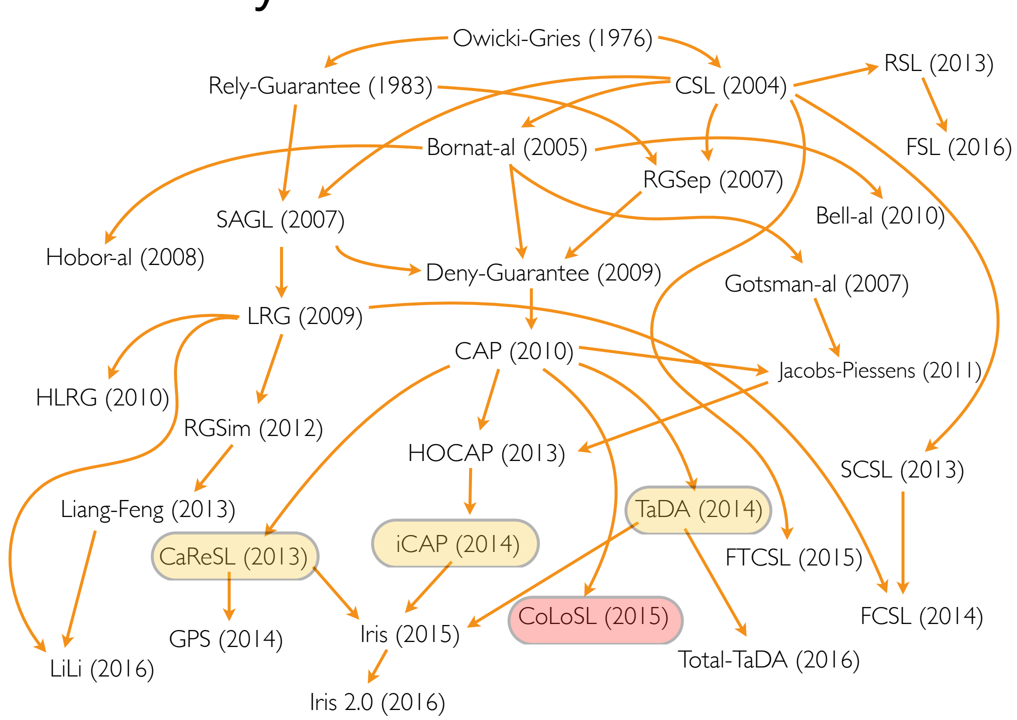
# Contributions

- Verified 4 concurrent fine-grained graph algorithms
  - Copying dags (directed acyclic graphs)
  - Speculative variant of Dijkstra's shortest path
  - Computing the spanning tree of a graph
  - Marking a graph
- Presented a common proof pattern for graph algorithms
  - *Abstract mathematical* graphs for functional correctness
  - *Concrete spatial (heap-represented)* graphs for memory safety
  - Combined reasoning for full proof
  - Inspired by existing logics where this pattern is "baked-in" to the model
  - "Baking-in" is unnecessary



# This Talk

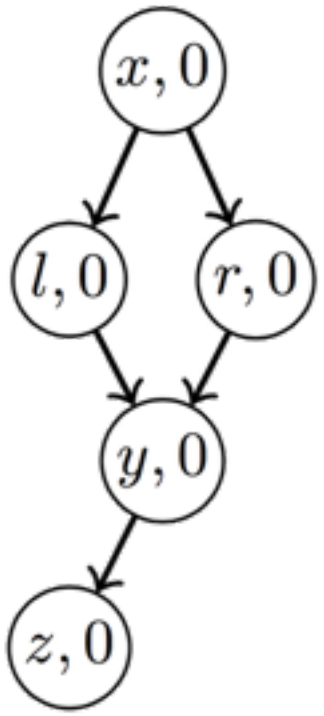
- Verified 4 concurrent fine-grained graph algorithms
  - ☑ Copying dags (directed acyclic graphs)
  - Speculative variant of Dijkstra's shortest path
  - Computing the spanning tree of a graph
  - Marking a graph
- ☑ Presented a common proof pattern for graph algorithms
  - *Abstract mathematical* graphs for functional correctness
  - *Concrete spatial (heap-represented)* graphs for memory safety
  - Combined reasoning for full proof
  - Inspired by existing logics where this pattern is "baked-in" to the model
  - "Baking-in" is unnecessary



# Copying Binary DAGs

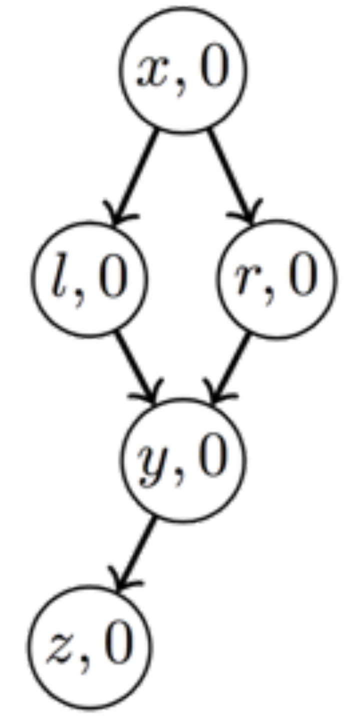
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

atomic blocks



# `copy_dag(x)` Specification

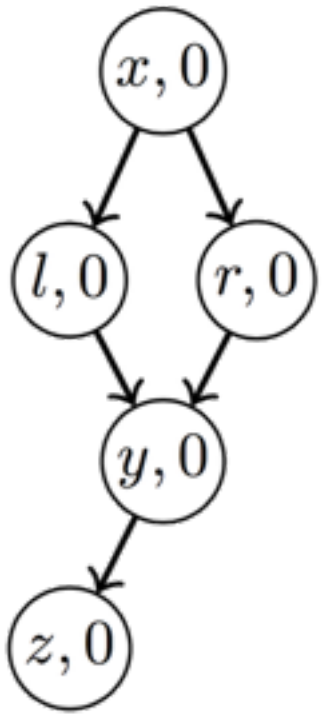
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```




- Specification challenges
  - When **`copy_dag(x)`** returns, **`x`** is copied but its children may not be
  - If **`x`** is already copied, **`copy_dag(x)`** simply returns: the thread that copied **`x`** has made a *promise* to visit **`x`**'s children and ensure they are copied

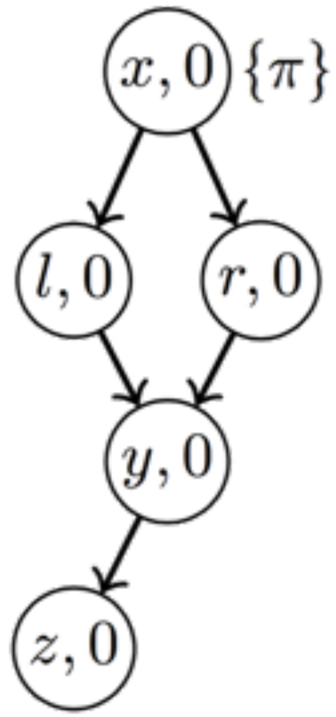
# `copy_dag(x)`: A Trace

```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```



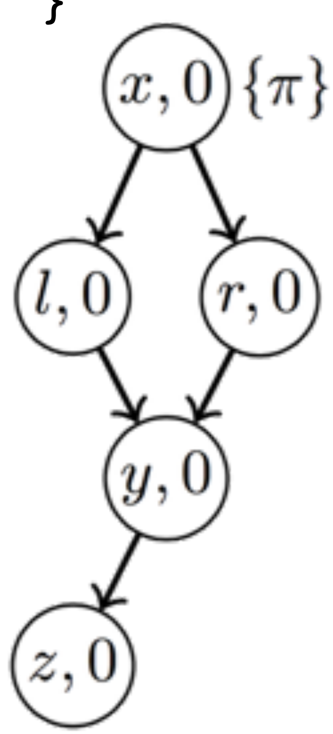
# `copy_dag(x)`: A Trace

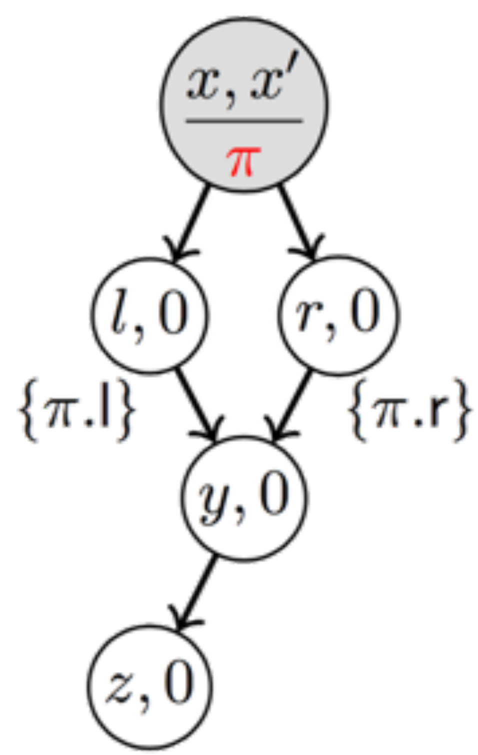
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```



# `copy_dag(x)`: A Trace

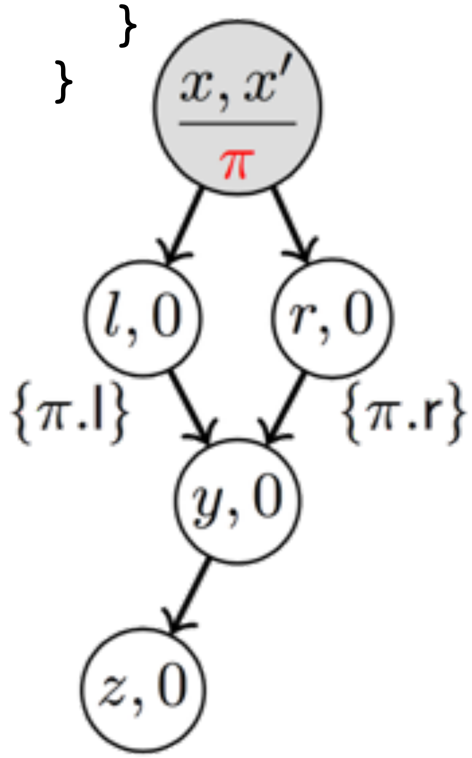
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

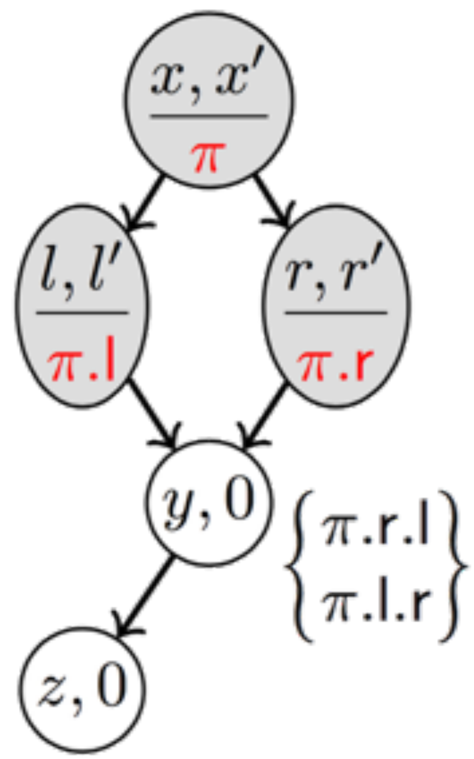


# copy_dag(x): A Trace

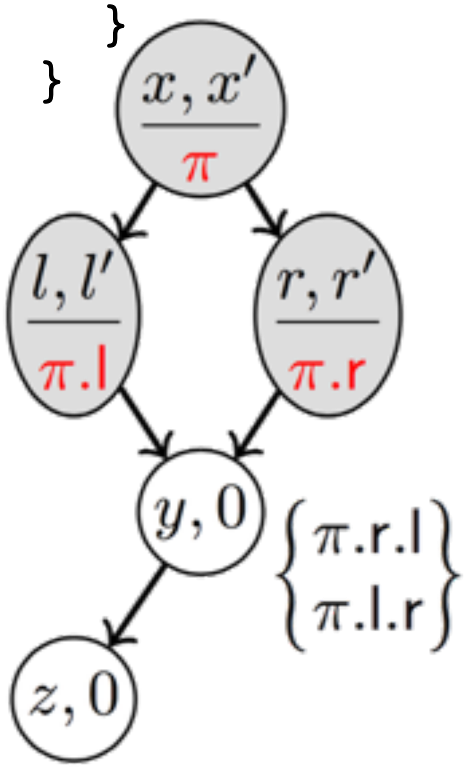
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x’;  bool b;
  if (!x) {return 0;}
  x’ = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x’)>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x’->l = ll>; <x’->r = rr>;
    return x’;
  } else {
    free(x’, sizeof(struct node));  return x->c;
  }
}
```

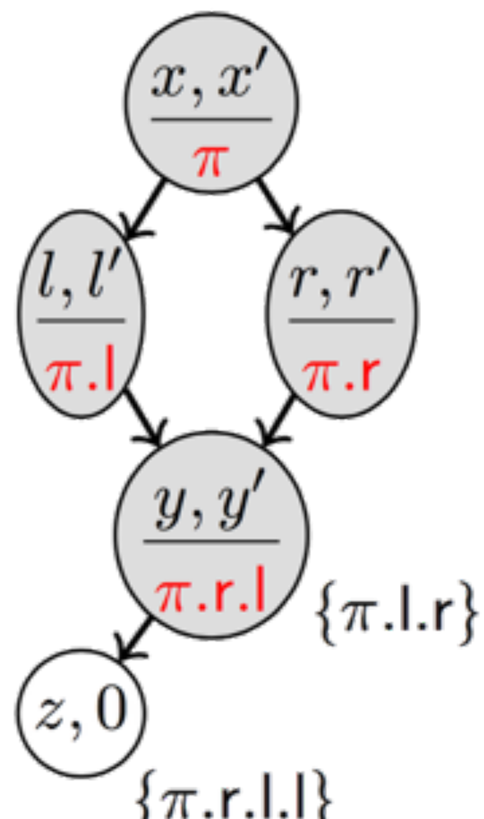


# `copy_dag(x)`: A Trace

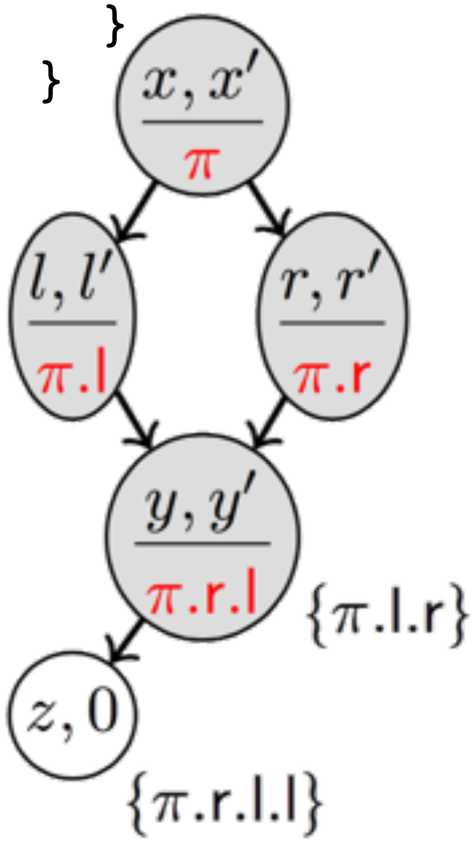
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

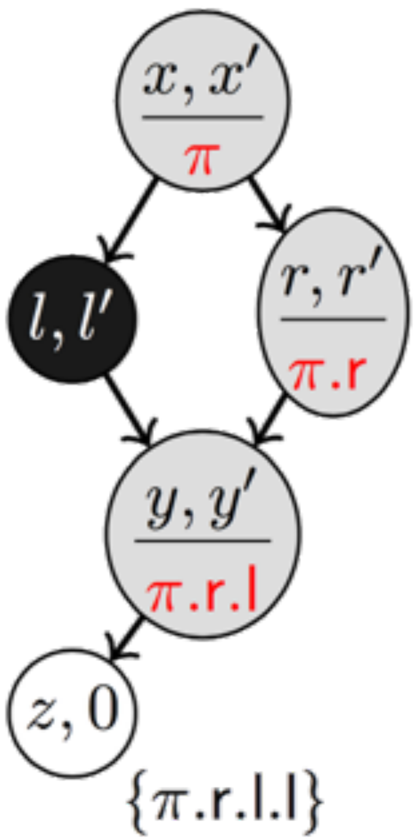


# copy_dag(x): A Trace

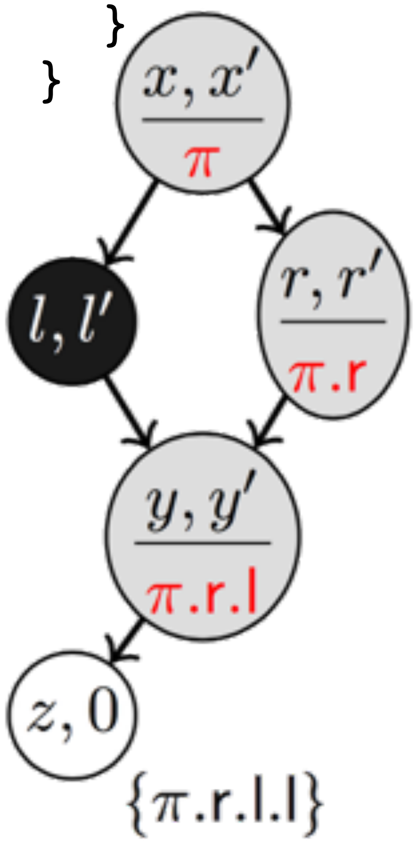
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

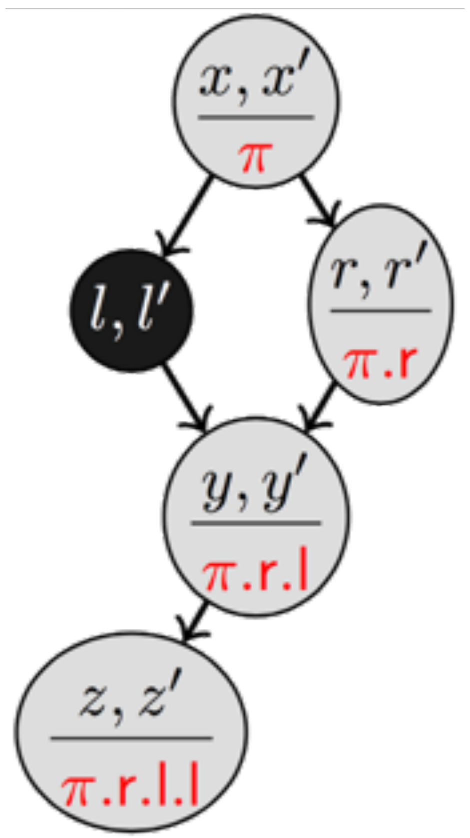


# copy_dag(x): A Trace

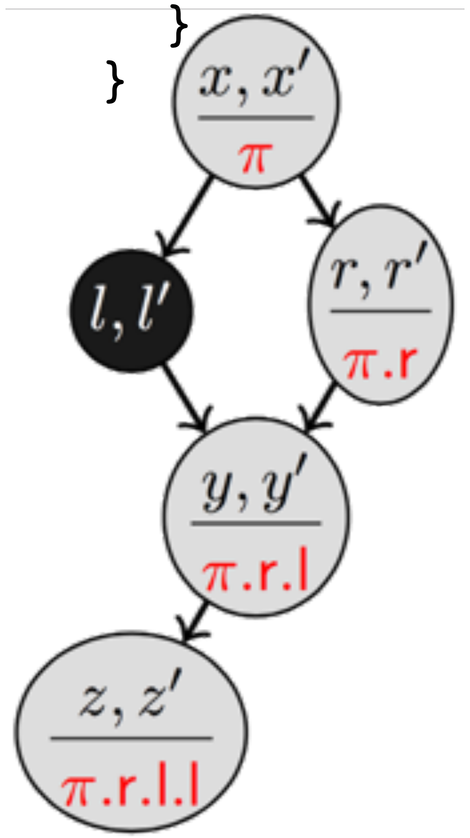
```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

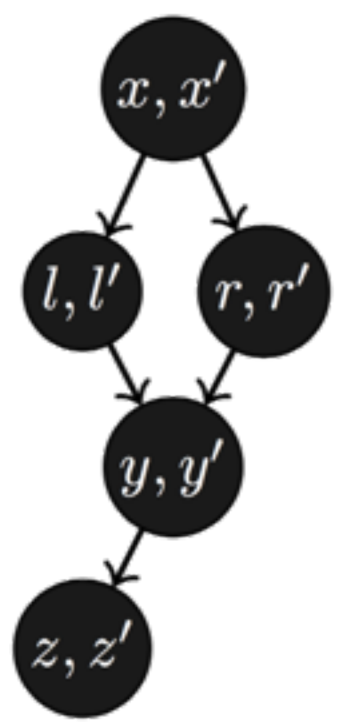


# copy_dag(x): A Trace

```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```



# 1. Tokens

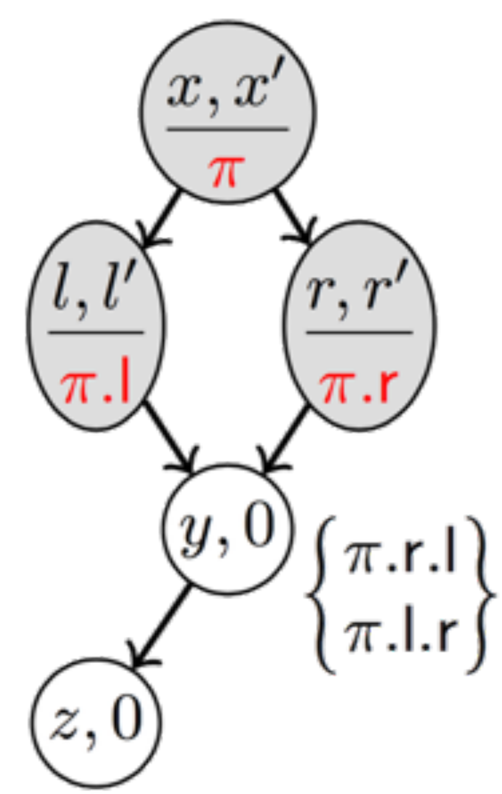
A token mechanism for

- Thread identification
- Thread progress tracking

# 1. Tokens

The `copy_dag` token mechanism for

- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets

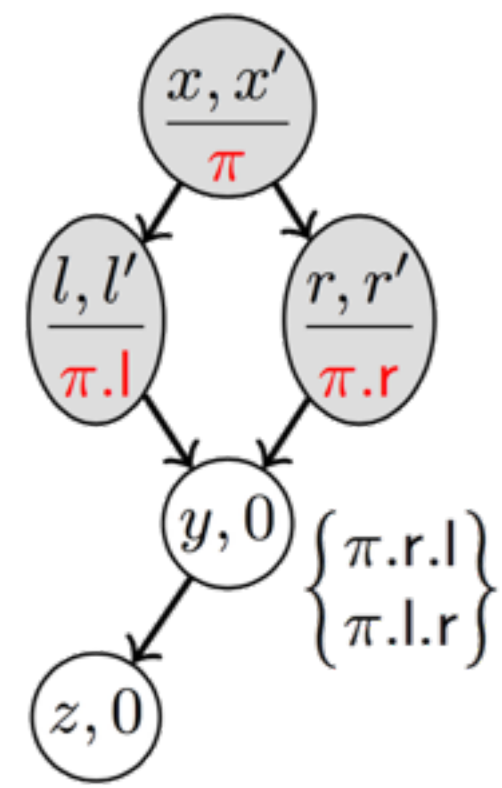


# 1. Tokens

The **`copy_dag`** token mechanism for

- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets

$$\pi ::= \bullet \mid \widehat{\circ\ \pi} \mid \widehat{\pi\ \circ}$$



# 1. Tokens

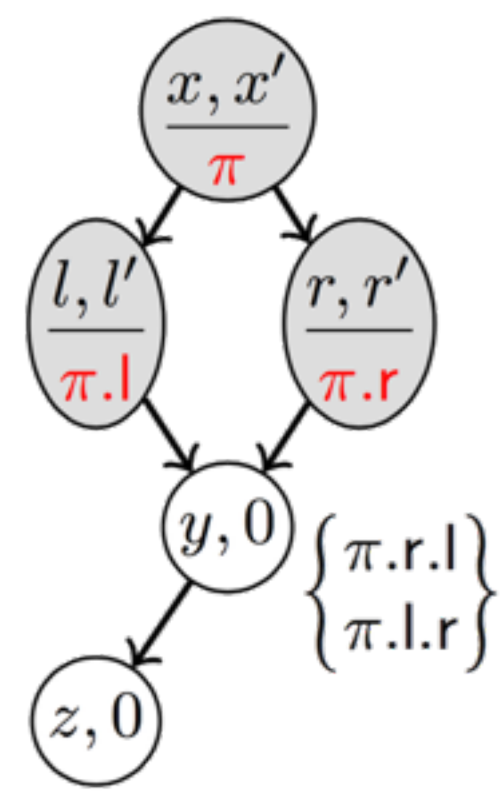
The `copy_dag` token mechanism for

- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets

$$\pi ::= \bullet \mid \widehat{\circ\,\pi} \mid \widehat{\pi\,\circ}$$

$$\bullet.\mathsf{l} = \widehat{\bullet\,\circ} \qquad (\widehat{\circ\,\pi}).\mathsf{l} = \widehat{\circ\,\pi.\mathsf{l}} \qquad (\widehat{\pi\,\circ}).\mathsf{l} = \widehat{\pi.\mathsf{l}\,\circ}$$

$$\bullet.\mathsf{r} = \widehat{\circ\,\bullet} \qquad (\widehat{\circ\,\pi}).\mathsf{r} = \widehat{\circ\,\pi.\mathsf{r}} \qquad (\widehat{\pi\,\circ}).\mathsf{r} = \widehat{\pi.\mathsf{r}\,\circ}$$



# 1. Tokens

The **`copy_dag`** token mechanism for

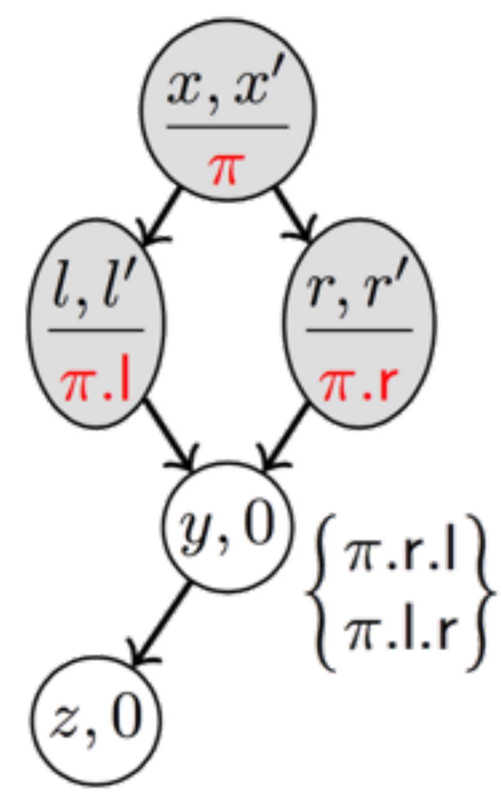
- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets

$$\pi ::= \bullet \mid \widehat{\circ\;\pi} \mid \widehat{\pi\;\circ}$$

$$\bullet.\mathsf{l} = \widehat{\bullet\;\circ} \qquad (\widehat{\circ\;\pi}).\mathsf{l} = \widehat{\circ\;\pi.\mathsf{l}} \qquad (\widehat{\pi\;\circ}).\mathsf{l} = \widehat{\pi.\mathsf{l}\;\circ}$$

$$\bullet.\mathsf{r} = \widehat{\circ\;\bullet} \qquad (\widehat{\circ\;\pi}).\mathsf{r} = \widehat{\circ\;\pi.\mathsf{r}} \qquad (\widehat{\pi\;\circ}).\mathsf{r} = \widehat{\pi.\mathsf{r}\;\circ}$$

$\sqsubset\; = \{(\pi.\mathsf{l}, \pi), (\pi.\mathsf{r}, \pi)\}^{+}$ sub-thread relation



# 1. Tokens

The `copy_dag` token mechanism for

- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets

top-most (maximal) token

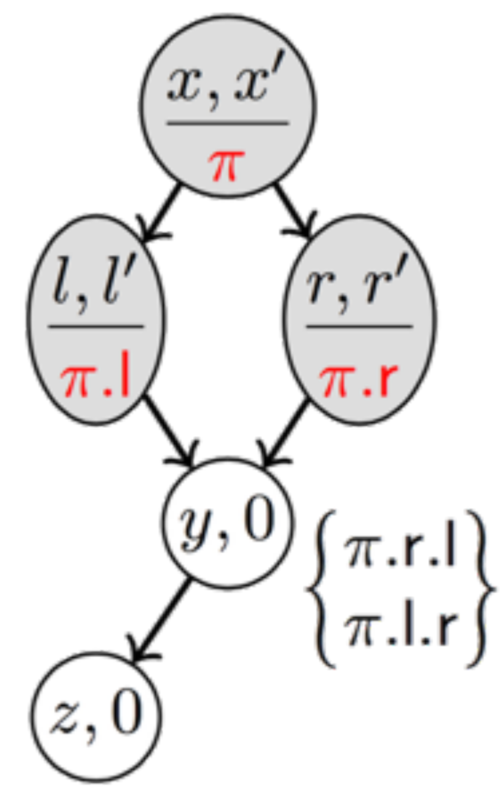
$$\pi ::= \bullet \mid \widehat{\circ\;\pi} \mid \widehat{\pi\;\circ}$$

$$\bullet.\mathsf{l} = \widehat{\bullet\;\circ} \qquad (\widehat{\circ\;\pi}).\mathsf{l} = \widehat{\circ\;\pi.\mathsf{l}} \qquad (\widehat{\pi\;\circ}).\mathsf{l} = \widehat{\pi.\mathsf{l}\;\circ}$$

$$\bullet.\mathsf{r} = \widehat{\circ\;\bullet} \qquad (\widehat{\circ\;\pi}).\mathsf{r} = \widehat{\circ\;\pi.\mathsf{r}} \qquad (\widehat{\pi\;\circ}).\mathsf{r} = \widehat{\pi.\mathsf{r}\;\circ}$$

$$\sqsubset \; = \{(\pi.\mathsf{l}, \pi), (\pi.\mathsf{r}, \pi)\}^{+}$$

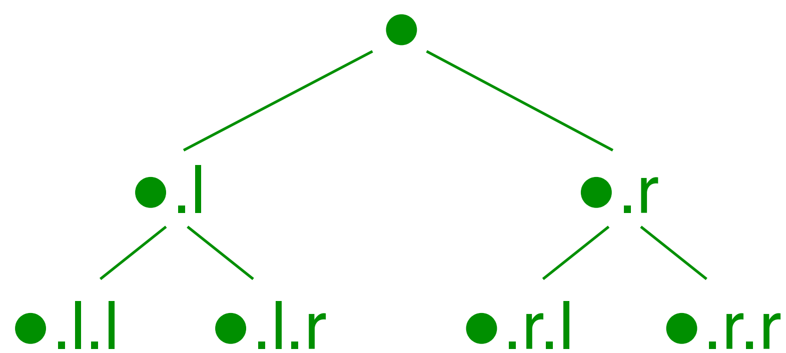
sub-thread relation



# 1. Tokens

The `copy_dag` token mechanism for

- Thread identification
  - distinguish one token (thread) from another
  - identify two distinct sub-tokens given any token (at recursive call points)
  - model a parent-child relation (spawner-spawnee)
- Thread progress tracking
  - marking thread ids as tokens
  - promise sets as token sets




top-most (maximal) token

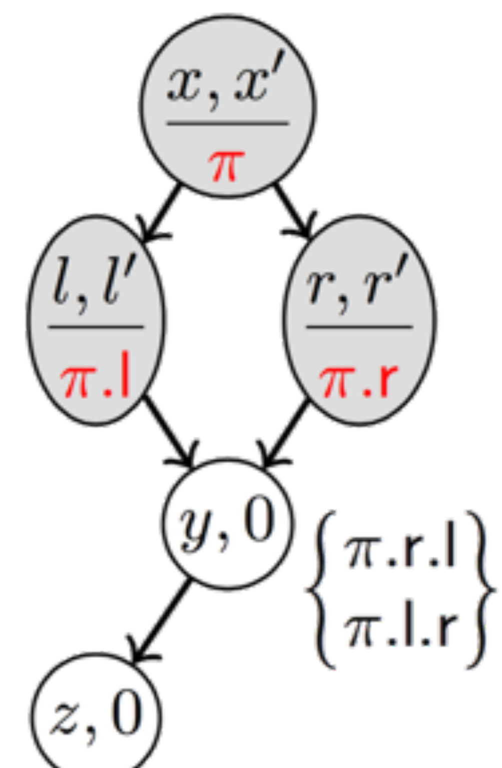
$$\pi ::= \bullet \mid \widehat{\circ\ \pi} \mid \widehat{\pi\ \circ}$$

$$\bullet.\mathsf{l} = \widehat{\bullet\ \circ} \qquad (\widehat{\circ\ \pi}).\mathsf{l} = \widehat{\circ\ \pi.\mathsf{l}} \qquad (\widehat{\pi\ \circ}).\mathsf{l} = \widehat{\pi.\mathsf{l}\ \circ}$$

$$\bullet.\mathsf{r} = \widehat{\circ\ \bullet} \qquad (\widehat{\circ\ \pi}).\mathsf{r} = \widehat{\circ\ \pi.\mathsf{r}} \qquad (\widehat{\pi\ \circ}).\mathsf{r} = \widehat{\pi.\mathsf{r}\ \circ}$$

$$\sqsubset\ = \{(\pi.\mathsf{l},\ \pi),\ (\pi.\mathsf{r},\ \pi)\}^{+}$$

sub-thread relation



# 2. Mathematical Objects

# 2. Mathematical Objects

- An abstract representation of the underlying data structure

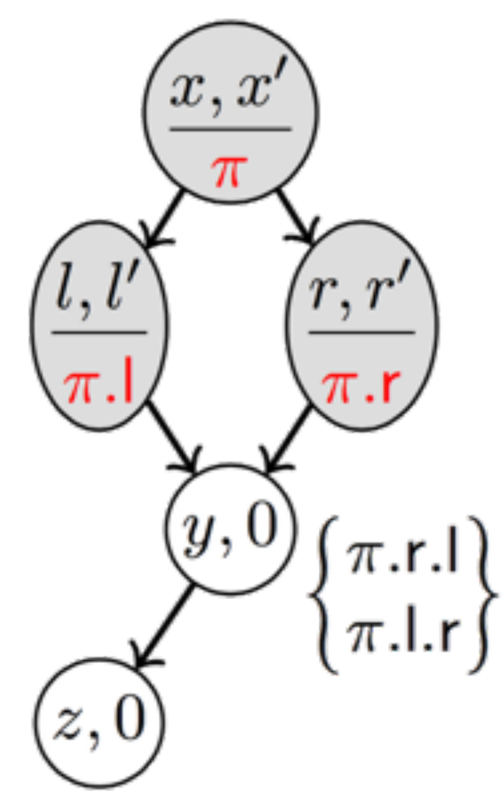
# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags $(\delta, \delta_c)$

# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags ($\delta$, $\delta_c$)
  - each dag is a triple:

$$\delta = ( V , E , L )$$



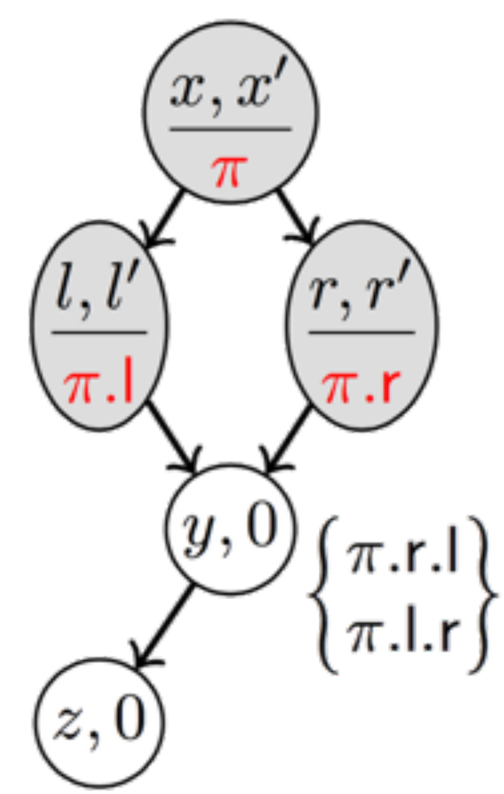
# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags $(\delta, \delta_c)$
  - each dag is a triple:

Vertices

$$\delta = (V, E, L)$$

$$V = \{x,\ l,\ r,\ y,\ z\}$$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags ($\delta$, $\delta_c$)
  - each dag is a triple:

Edges

$$\delta = ( V , E , L )$$

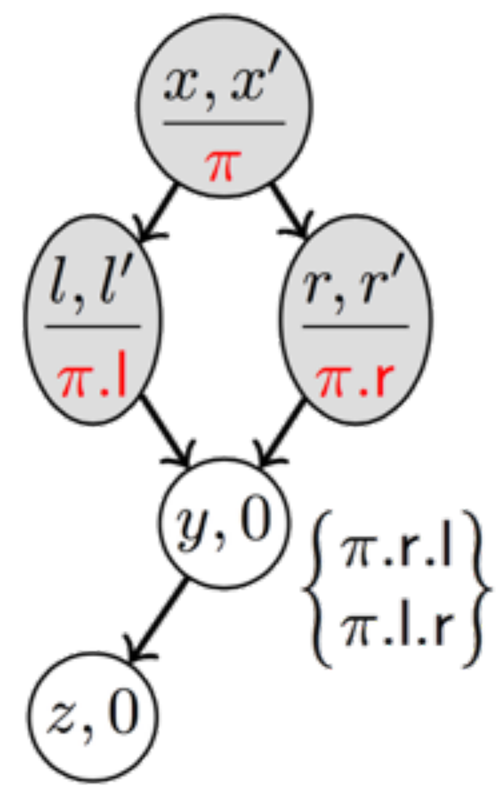
$$E(x) = l,\ r$$

$$E(l) = 0,\ y$$

$$E(r) = y,\ 0$$

$$E(y) = z,\ 0$$

$$E(z) = 0,\ 0$$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags ($\delta$, $\delta_c$)
  - each dag is a triple:

Labels

$$\delta = ( V , E , L )$$

copy

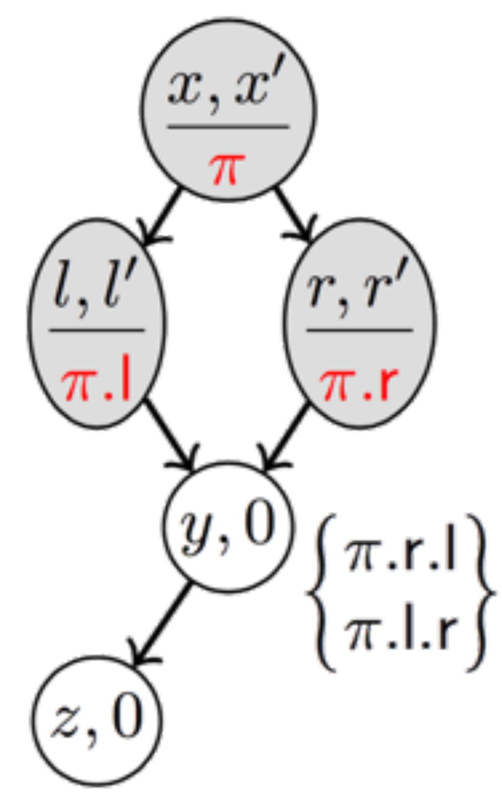
$$L\,(x) = x',\ \pi,\ \{\ \}$$

$$L\,(l) = l',\ \pi.\text{l},\ \{\ \}$$

$$L\,(r) = r',\ \pi.\text{r},\ \{\ \}$$

$$L\,(y) = 0,\ 0,\ \{\pi.\text{r.l},\ \pi.\text{l.r}\}$$

$$L\,(z) = 0,\ 0,\ \{\ \}$$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags $(\delta, \delta_c)$
  - each dag is a triple:

Labels

$$\delta = (V, E, L)$$

copy    copying thread

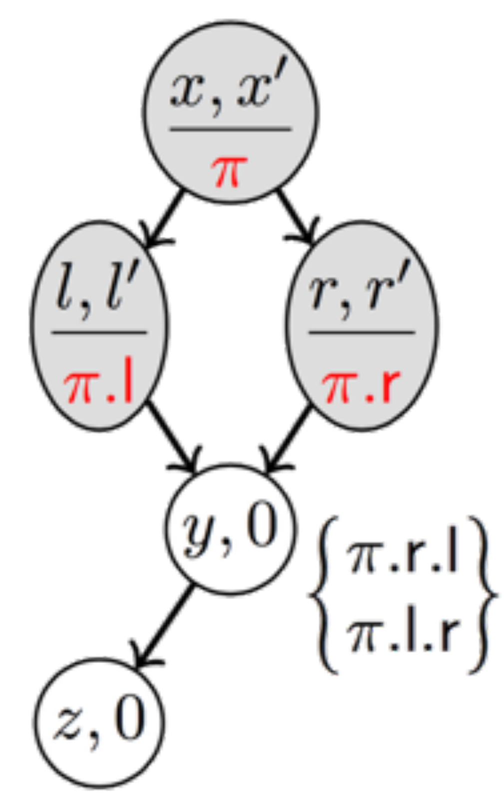
$$L(x) = x', \pi, \{\}$$

$$L(l) = l', \pi.\text{l}, \{\}$$

$$L(r) = r', \pi.\text{r}, \{\}$$

$$L(y) = 0, 0, \{\pi.\text{r.l}, \pi.\text{l.r}\}$$

$$L(z) = 0, 0, \{\}$$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags ($\delta$, $\delta_c$)
  - each dag is a triple:

Labels

$$\delta = ( V , E , L )$$

copy　copying thread　promise set

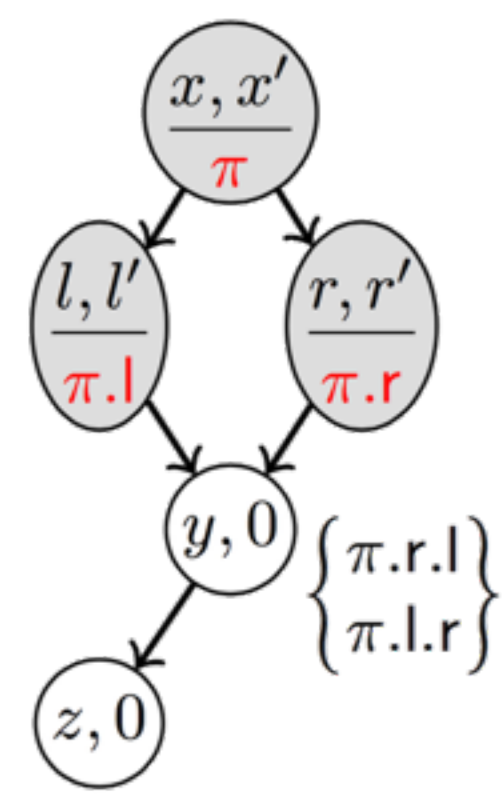
$L(x) = x', \pi, \{\ \}$

$L(l) = l', \pi.\mathrm{l}, \{\ \}$

$L(r) = r', \pi.\mathrm{r}, \{\ \}$

$L(y) = 0, 0, \{\pi.\mathrm{r.l}, \pi.\mathrm{l.r}\}$

$L(z) = 0, 0, \{\ \}$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags ($\delta$, $\delta_c$)
  - each dag is a triple:

Labels

$$\delta = (V, E, L)$$

copy  copying thread  promise set

ghost components

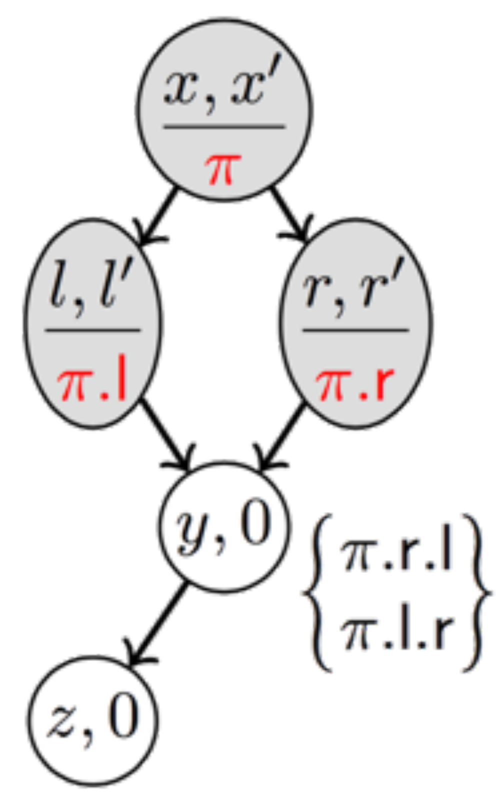
$$L(x) = x', \pi, \{\}$$

$$L(l) = l', \pi.\mathrm{l}, \{\}$$

$$L(r) = r', \pi.\mathrm{r}, \{\}$$

$$L(y) = 0, 0, \{\pi.\mathrm{r.l}, \pi.\mathrm{l.r}\}$$

$$L(z) = 0, 0, \{\}$$



# 2. Mathematical Objects

- An abstract representation of the underlying data structure
  - e.g. a pair of mathematical dags $(\delta, \delta_c)$
  - each dag is a triple:

Labels

$$\delta = (V, E, L)$$

copy($x$) thread($x$) promise($x$)

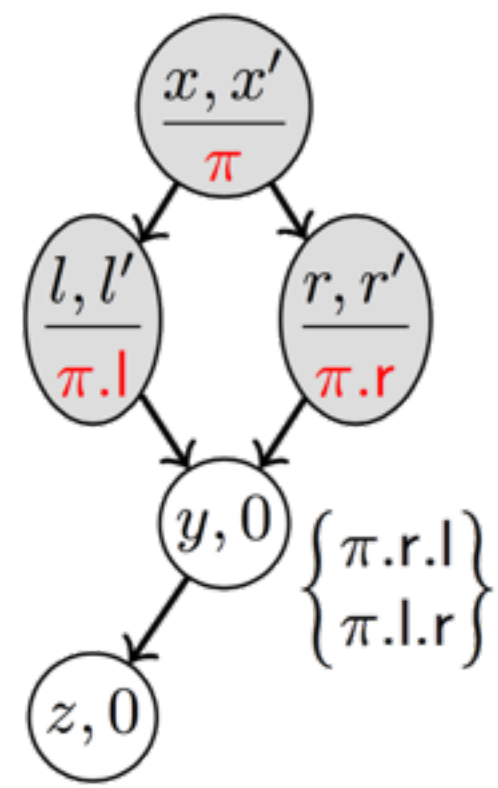
$$L(x) = x', \ \pi, \ \{\ \}$$

$$L(l) = l', \ \pi.\mathrm{l}, \ \{\ \}$$

$$L(r) = r', \ \pi.\mathrm{r}, \ \{\ \}$$

$$L(y) = 0, \ 0, \ \{\pi.\mathrm{r.l}, \ \pi.\mathrm{l.r}\}$$

$$L(z) = 0, \ 0, \ \{\ \}$$



# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^\pi$ denotes the actions of thread $\pi$

# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^{\pi}$ denotes the actions of thread $\pi$

```
struct node {struct node *c, *l, *r}
copy_dag(struct node *x) {
  struct node *l, *r, *ll, *rr, *x';  bool b;
  if (!x) {return 0;}
  x' = malloc(sizeof(struct node));
  b = <CAS(x->c, 0, x')>;
  if (b) {
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>;
    return x';
  } else {
    free(x', sizeof(struct node));  return x->c;
  }
}
```

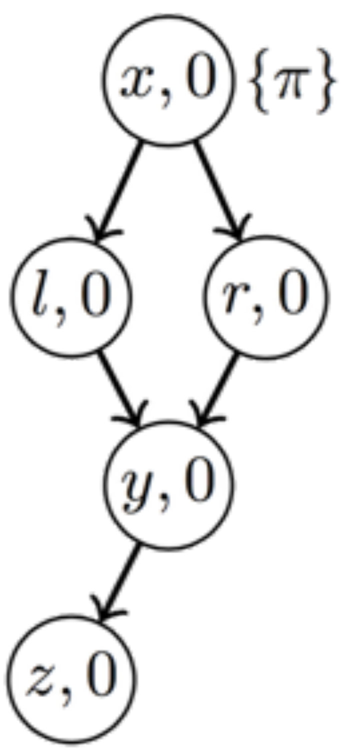
# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^{\pi}$ denotes the actions of thread $\pi$

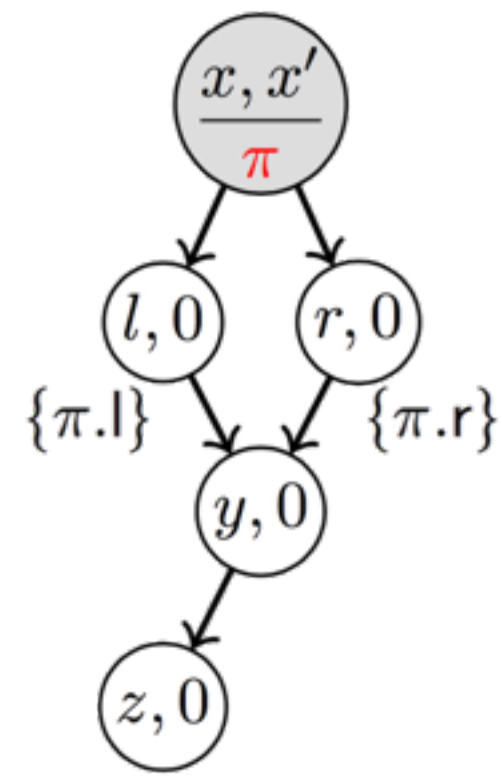
```
struct
copy_dag(
  struct
  if
  x' =
  b = <CAS(x->c, 0, x')>;
  if
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>

  } else {

  }
}
```

# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^{\pi}$ denotes the actions of thread $\pi$




```
b = <CAS(x->c, 0, x')>;
```

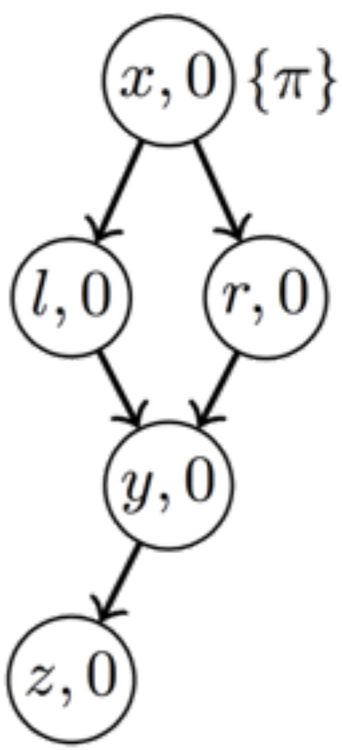



$(\delta, \delta_c) = ((V, E, L), (V_c, E_c, L_c))$ $\overset{A^{\pi}}{\rightsquigarrow}$ $(\delta', \delta'_c) = ((V, E, L'), (V'_c, E'_c, L'_c))$

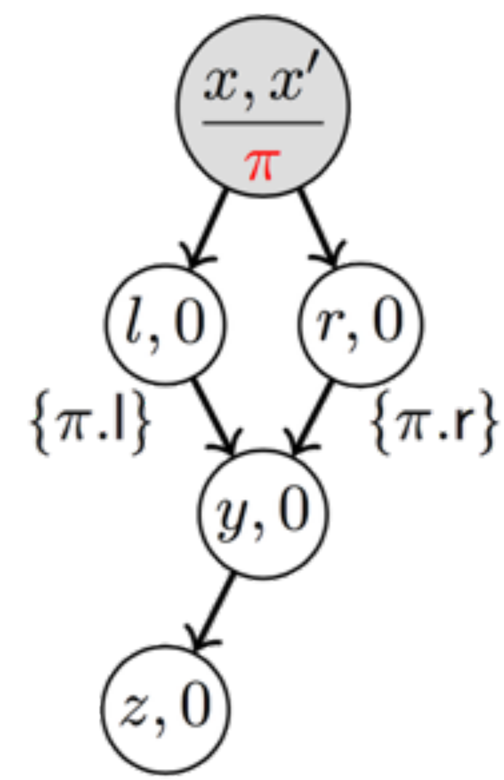
$L(x) = 0,\ 0,\ \{\pi\}$
$L(l) = 0,\ 0,\ \{\ \}$
$L(r) = 0,\ 0,\ \{\ \}$

# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^{\pi}$ denotes the actions of thread $\pi$




```
struct
copy_dag(
  struct
  if
  x' =
  b = <CAS(x->c, 0, x')>;
  if
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>

  } else {

  }
}
```



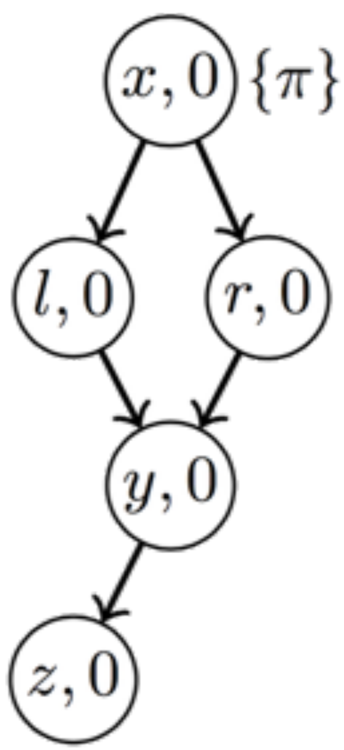

$(\delta, \delta_C) = ((V, E, L), (V_C, E_C, L_C))$ $\overset{A^{\pi}}{\rightsquigarrow}$ $(\delta', \delta'_C) = ((V, E, L'), (V'_C, E'_C, L'_C))$

$L(x) = 0,\ 0,\ \{\pi\}$
$L(l) = 0,\ 0,\ \{\ \}$
$L(r) = 0,\ 0,\ \{\ \}$

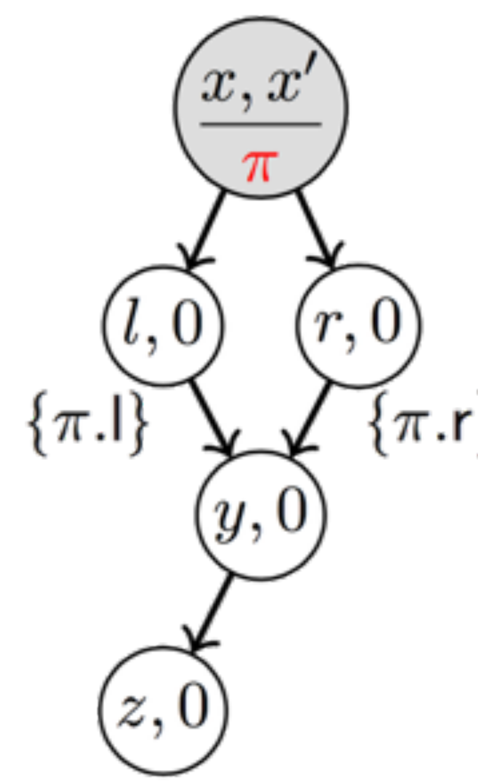
$L' = L[x \mapsto x',\ \pi,\ \{\ \}][l \mapsto 0, 0, \{\pi.\mathrm{l}\}][r \mapsto 0, 0, \{\pi.\mathrm{r}\}]$

# 3. Mathematical Actions

- An abstraction of thread actions (on mathematical objects)
  - atomic blocks as well as ghost actions
  - $A^{\pi}$ denotes the actions of thread $\pi$




```
b = <CAS(x->c, 0, x')>;
```




$(\delta, \delta_C) = ( (V, E, L), (V_C, E_C, L_C) )$ $\overset{A^{\pi}}{\rightsquigarrow}$ $(\delta', \delta'_C) = ( (V, E, L'), (V'_C, E'_C, L'_C) )$

$L(x) = 0,\ 0,\ \{\pi\}$
$L(l) = 0,\ 0,\ \{\ \}$
$L(r) = 0,\ 0,\ \{\ \}$

$L' = L[x \mapsto x',\ \pi,\ \{\ \}][l \mapsto 0, 0, \{\pi.\mathsf{l}\}][r \mapsto 0, 0, \{\pi.\mathsf{r}\}]$

$V'_C = V_C \uplus \{x'\}$ $\quad E'_C = E_C \uplus [x' \mapsto \cdots]$ $\quad L'_C = L_C \uplus [x' \mapsto \cdots]$

# 4. Mathematical Specification

Inv(δ, $\delta_c$) ≜ δ and $\delta_c$ are both acyclic;
every node $x$' in the copy $\delta_c$ corresponds to a unique node $x$ in the source δ;
every node $x$ in the source δ has some copy value $x$'

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge$

every node $x'$ in the copy $\delta_c$ corresponds to a unique node $x$ in the source $\delta$;
every node $x$ in the source $\delta$ has some copy value $x'$

# 4. Mathematical Specification

Inv(δ, $\delta_c$) ≜ acyc(δ) ∧ acyc($\delta_c$) ∧ $\forall x' \in \delta_c.\ \exists! x \in \delta.$ copy($x$)= $x'$ ∧

every node $x$ in the source δ has some copy value $x'$

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \land \mathrm{acyc}(\delta_c) \land \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\quad \land \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \land \mathrm{ic}(x, x', \delta, \delta_c)$

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\quad \wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

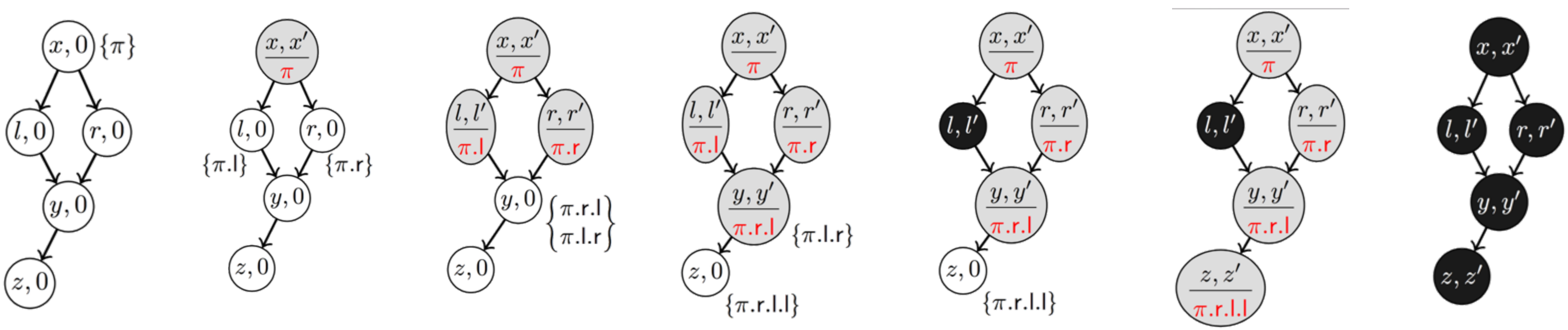
$\mathrm{ic}(x, x', \delta, \delta_c) \triangleq$ if $x'$ is 0 ($x$ is not copied yet), then $x$ will eventually be copied:

# 4. Mathematical Specification

Inv(δ, $\delta_c$) ≜ acyc(δ) ∧ acyc($\delta_c$) ∧ ∀$x$' ∈ $\delta_c$. ∃!$x$ ∈ δ. copy($x$)= $x$'
∧ ∀$x$ ∈ δ. ∃$x$'. copy($x$)=$x$' ∧ ic($x$, $x$', δ, $\delta_c$)

ic($x$, $x$', δ, $\delta_c$) ≜ if $x$' is 0 ($x$ is not copied yet), then $x$ will eventually be copied:

there exists some $y$ in δ s.t.
1) the promise set of $y$ is non-empty; 2) $y$ can reach $x$ along a path p;
and 3) every node along the path p is not copied
⇒ when $y$ is eventually copied, it'll visit $x$ along p and copy it too



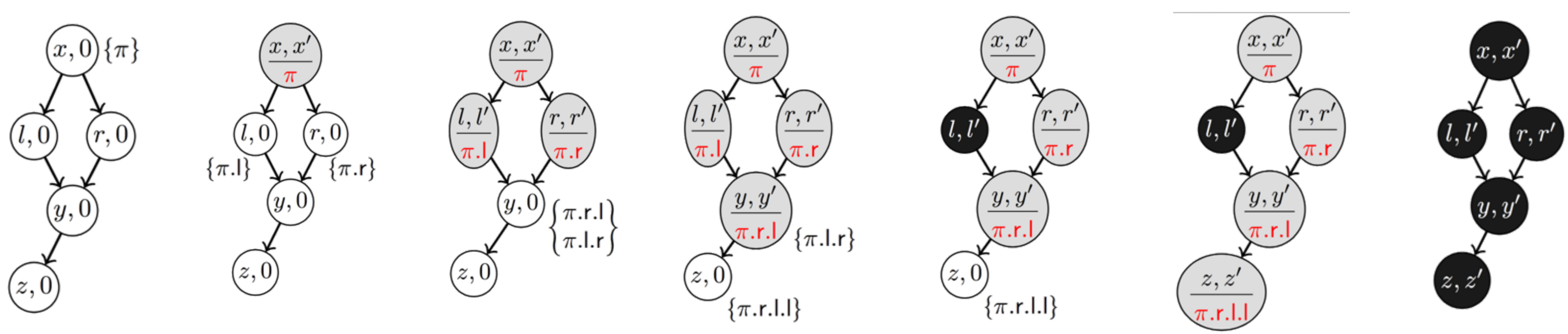
# 4. Mathematical Specification

Inv(δ, $\delta_c$) ≜ acyc(δ) ∧ acyc($\delta_c$) ∧ $\forall x' \in \delta_c.\ \exists! x \in \delta.$ copy($x$)= $x'$
∧ $\forall x \in \delta.\ \exists x'.$ copy($x$)=$x'$ ∧ ic($x$, $x'$, δ, $\delta_c$)

ic($x$, $x'$, δ, $\delta_c$) ≜ if $x'$ is 0 ($x$ is not copied yet), then $x$ will eventually be copied:

there exists some $y$ in δ s.t.
1) the promise set of $y$ is non-empty; 2) $y$ can reach $x$ along a path p; and 3) every node along the path p is not copied

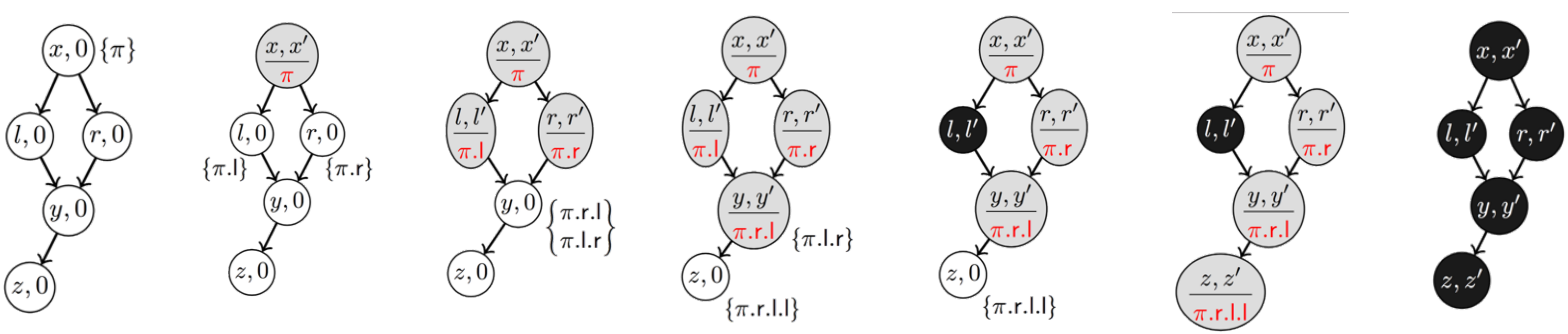
otherwise, $x'$ is a node in $\delta_c$ and
the children of $x$, ($l$,$r$), are also copied to some ($l'$,$r'$):
ic($l$, $l'$, δ, $\delta_c$) and ic($r$, $r'$, δ, $\delta_c$)



# 4. Mathematical Specification

$\mathsf{Inv}(\delta, \delta_c) \triangleq \mathsf{acyc}(\delta) \wedge \mathsf{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathsf{copy}(x) = x'$
$\quad \wedge\ \forall x \in \delta.\ \exists x'.\ \mathsf{copy}(x) = x' \wedge \mathsf{ic}(x, x', \delta, \delta_c)$

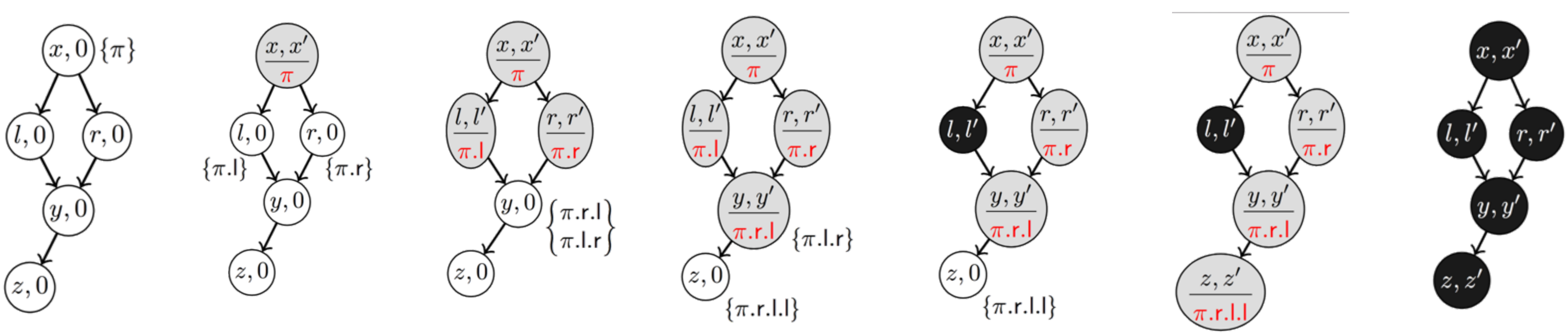
$$
\begin{aligned}
\mathsf{ic}(x, x', \delta, \delta_c) \triangleq\ & (x{=}0 \wedge x'{=}0) \vee \\
& \Big(x{\neq}0 \wedge \Big[(x'{=}0 \wedge \delta^{c}(x){=}x' \wedge \exists y.\ \delta^{p}(y){\neq}\emptyset \wedge y \overset{\delta}{\rightsquigarrow}{}^{*}_{0}\ x) \\
& \quad \vee (x'{\neq}0 \wedge x' \in \delta' \wedge \exists \pi, l, r, l', r'.\ \delta(x){=}((x', \pi, -), l, r) \wedge \delta'(x'){=}(-, l', r') \\
& \qquad \wedge (l'{\neq}0 \Rightarrow \mathsf{ic}(l, l', \delta, \delta')) \wedge (r'{\neq}0 \Rightarrow \mathsf{ic}(r, r', \delta, \delta'))) \\
& \quad \vee (x'{\neq}0 \wedge x' \in \delta' \wedge \exists l, r, l', r'.\ \delta(x){=}((x', 0, -), l, r) \wedge \delta'(x'){=}(-, l', r') \\
& \qquad \wedge \mathsf{ic}(l, l', \delta, \delta') \wedge \mathsf{ic}(r, r', \delta, \delta'))\Big]\Big)
\end{aligned}
$$



# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

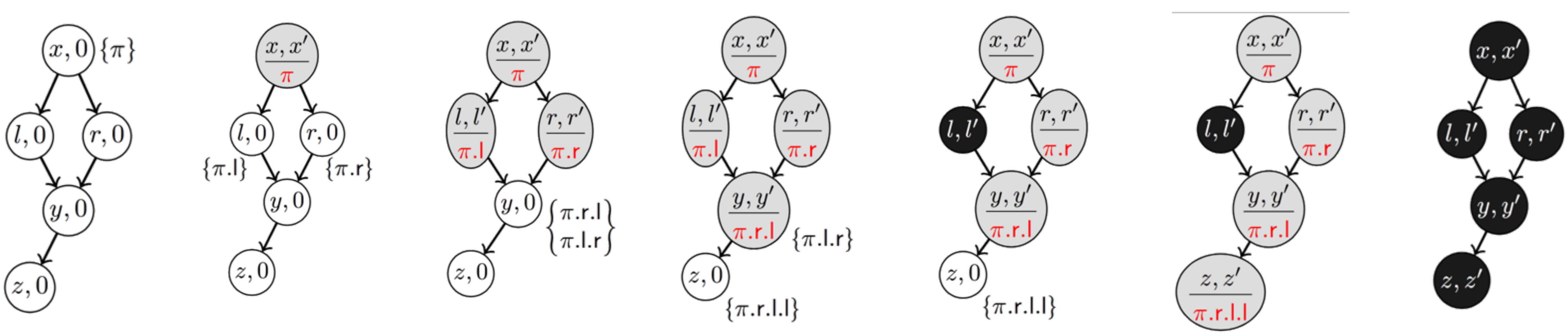
$P^{\pi}(x, \delta) \triangleq$ $\pi$ has made a promise to visit $x$; $\pi$ has made a promise to $x$ only; and $\pi$ has not spawned any threads yet:
its subthreads are not in the graph (in promise sets or as copying thread)



# 4. Mathematical Specification

Inv(δ, $\delta_c$) ≜ acyc(δ) ∧ acyc($\delta_c$) ∧ $\forall x' \in \delta_c.\ \exists! x \in \delta.$ copy($x$)= $x'$
∧ $\forall x \in \delta.\ \exists x'.$ copy($x$)=$x'$ ∧ ic($x$, $x'$, δ, $\delta_c$)

$P^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in$ promise($x$) ∧ $\pi$ has made a promise to $x$ only; and
$\pi$ has not spawned any threads yet:
its subthreads are not in the graph (in promise sets or as copying thread)



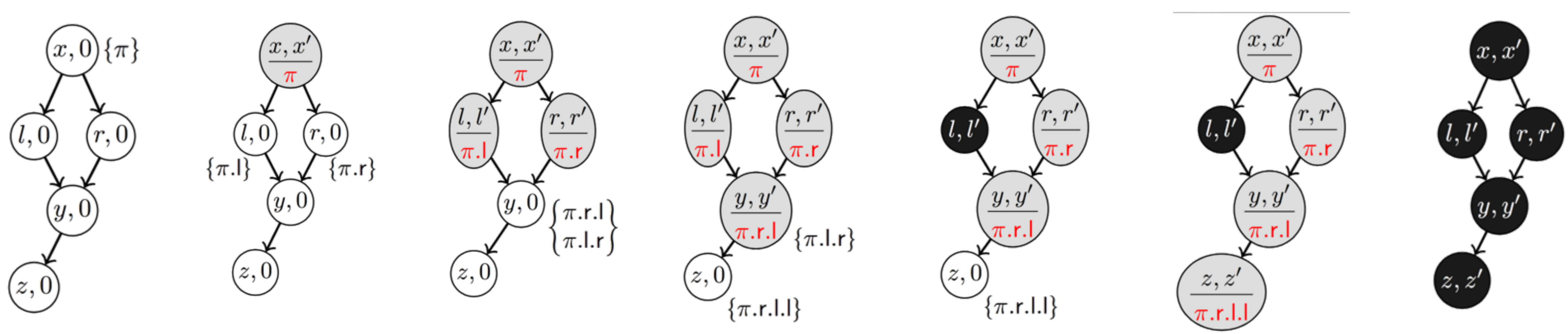
# 4. Mathematical Specification

$\text{Inv}(\delta, \delta_c) \triangleq \text{acyc}(\delta) \wedge \text{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \text{copy}(x) = x'$
$\quad \wedge\ \forall x \in \delta.\ \exists x'.\ \text{copy}(x) = x' \wedge \text{ic}(x, x', \delta, \delta_c)$

$P^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in \text{promise}(x) \quad \wedge \quad \forall z \in \delta.\ \pi \in \text{promise}(z) \Rightarrow x = z$

$\pi$ has not spawned any threads yet:

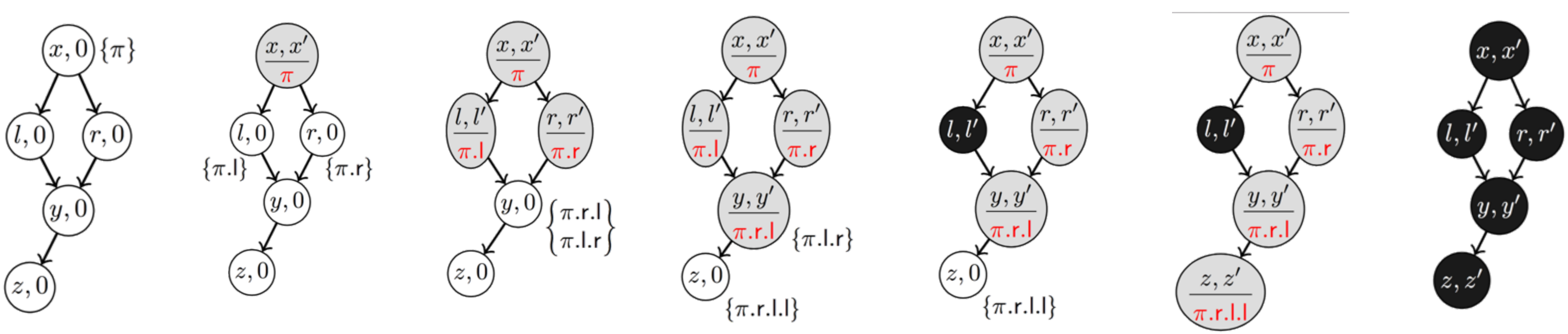
its subthreads are not in the graph (in promise sets or as copying thread)



# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

$\mathrm{P}^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in \mathrm{promise}(x) \wedge \forall z \in \delta.\ \pi \in \mathrm{promise}(z) \Rightarrow x = z$
$\forall z \in \delta.\ \forall \pi' \sqsubset \pi.\ \pi' \notin \mathrm{promise}(z) \wedge \pi' \neq \mathrm{thread}(z)$

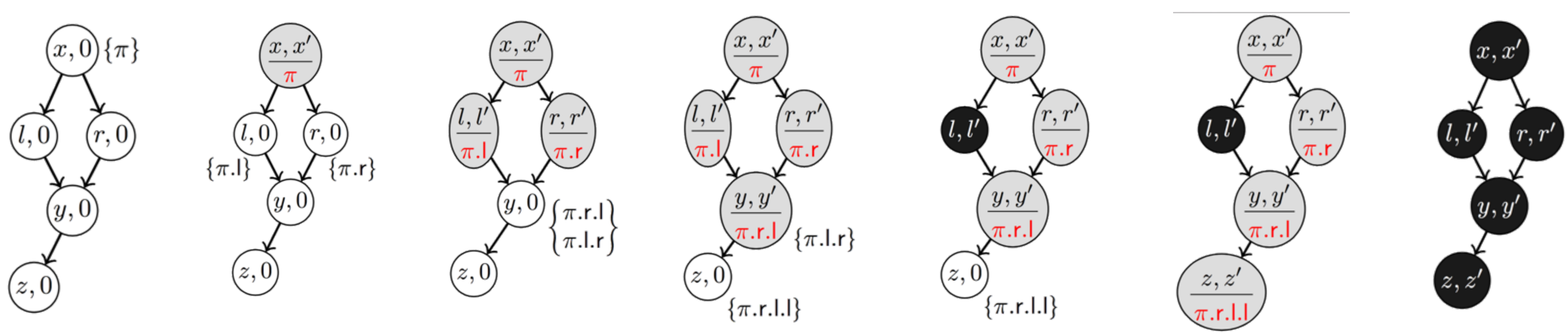


# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

$P^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in \mathrm{promise}(x) \wedge \forall z \in \delta.\ \pi \in \mathrm{promise}(z) \Rightarrow x = z$
$\forall z \in \delta.\ \forall \pi' \sqsubset \pi.\ \pi' \notin \mathrm{promise}(z) \wedge \pi' \neq \mathrm{thread}(z)$

$Q^{\pi}(x, x', \delta, \delta_c) \triangleq x$ is copied to $x'$ in $\delta_c$ ; and
$\pi$ and all its subthreads have finished executing (have joined):
they are not in the graph (in promise sets or as copying thread)

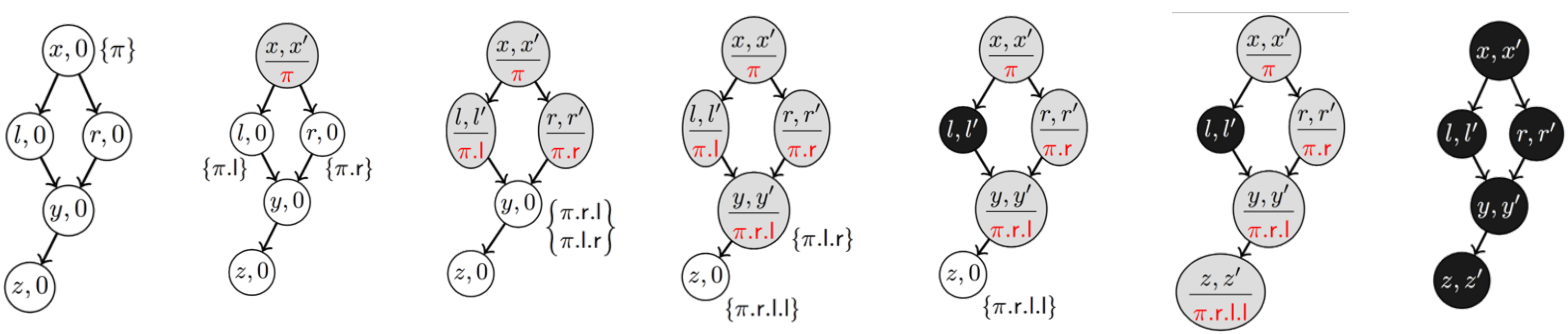


# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

$P^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in \mathrm{promise}(x) \wedge \forall z \in \delta.\ \pi \in \mathrm{promise}(z) \Rightarrow x = z$
$\forall z \in \delta.\ \forall \pi' \sqsubset \pi.\quad \pi' \notin \mathrm{promise}(z) \wedge \pi' \neq \mathrm{thread}(z)$

$Q^{\pi}(x, x', \delta, \delta_c) \triangleq \mathrm{copy}(x) = x' \wedge x' \in \delta_c \wedge$
$\pi$ and all its subthreads have finished executing (have joined):
they are not in the graph (in promise sets or as copying thread)

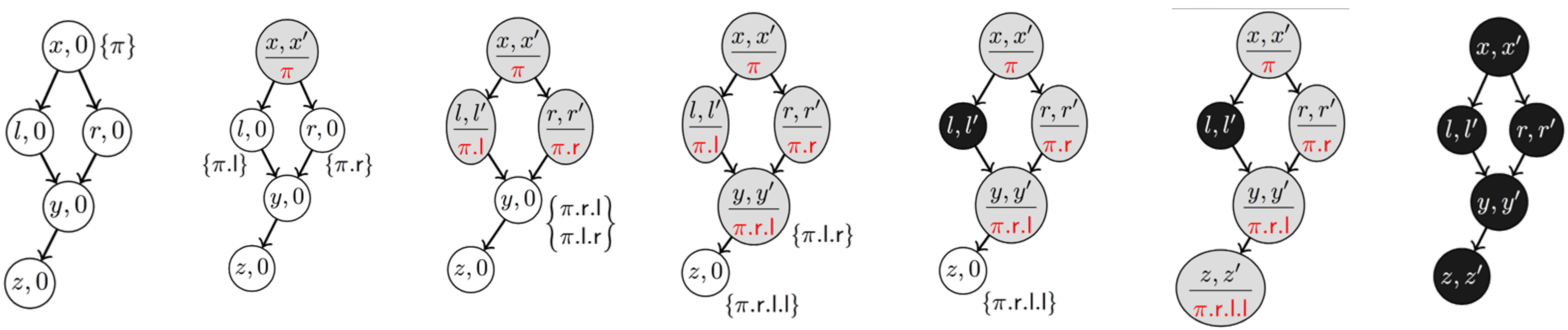


# 4. Mathematical Specification

$\text{Inv}(\delta, \delta_c) \triangleq \text{acyc}(\delta) \wedge \text{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \text{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \text{copy}(x) = x' \wedge \text{ic}(x, x', \delta, \delta_c)$

$P^{\pi}(x, \delta) \triangleq x \neq 0 \Rightarrow \pi \in \text{promise}(x) \wedge \forall z \in \delta.\ \pi \in \text{promise}(z) \Rightarrow x = z$
$\forall z \in \delta.\ \forall \pi' \sqsubset \pi.\ \ \pi' \notin \text{promise}(z) \wedge \pi' \neq \text{thread}(z)$

$Q^{\pi}(x, x', \delta, \delta_c) \triangleq \text{copy}(x) = x' \wedge x' \in \delta_c \wedge$
$\forall z \in \delta.\ \forall \pi' \sqsubseteq \pi.\ \ \pi' \notin \text{promise}(z) \wedge \pi' \neq \text{thread}(z)$



# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \land \mathrm{acyc}(\delta_c) \land \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\land\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \land \mathrm{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \mathrm{copy}(x) = x' \land x' \in \delta_c \land$
$\forall z \in \delta.\ \forall \pi' \sqsubseteq \bullet.\ \ \pi' \notin \mathrm{promise}(z) \land \pi' \neq \mathrm{thread}(x)$

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \mathrm{copy}(x) = x' \wedge x' \in \delta_c\ \wedge$
$\forall z \in \delta.\ \forall \pi' \sqsubseteq \bullet.\ \ \pi' \notin \mathrm{promise}(z) \wedge \pi' \neq \mathrm{thread}(x)$

$\forall\ \pi.\ \pi \sqsubseteq \bullet$

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \land \mathrm{acyc}(\delta_c) \land \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$

$\land\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \land \mathrm{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \mathrm{copy}(x) = x' \land x' \in \delta_c \land$

$\forall z \in \delta.\ \forall \pi'.\quad \pi' \notin \mathrm{promise}(z) \land \pi' \neq \mathrm{thread}(x)$

# 4. Mathematical Specification

$\text{Inv}(\delta, \delta_c) \triangleq \text{acyc}(\delta) \wedge \text{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \text{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \text{copy}(x) = x' \wedge \text{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \text{copy}(x) = x' \wedge x' \in \delta_c \wedge$
$\forall z \in \delta.\ \forall \pi'.\quad \pi' \notin \text{promise}(z) \wedge \pi' \neq \text{thread}(x)$

all promise sets are empty

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_c) \triangleq \mathrm{acyc}(\delta) \land \mathrm{acyc}(\delta_c) \land \forall x' \in \delta_c.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\land\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \land \mathrm{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \mathrm{copy}(x) = x' \land x' \in \delta_c \land$
$\forall z \in \delta.\ \forall \pi'.\quad \pi' \notin \mathrm{promise}(z) \land \pi' \neq \mathrm{thread}(x)$

all promise sets are empty

$\mathrm{ic}(z, z', \delta, \delta_c)$ ≜ if $z'$ is 0 ($z$ is not copied yet), then $z$ will eventually be copied:
there exists some $y$ in δ s.t.
1) the promise set of $y$ is non-empty; 2) $y$ can reach $z$ along a path p;
and 3) every node along the path p is not copied
otherwise, $z'$ is a node in $\delta_c$ and the children of $z$ are also copied

# 4. Mathematical Specification

$\mathrm{Inv}(\delta, \delta_C) \triangleq \mathrm{acyc}(\delta) \wedge \mathrm{acyc}(\delta_C) \wedge \forall x' \in \delta_C.\ \exists! x \in \delta.\ \mathrm{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \mathrm{copy}(x) = x' \wedge \mathrm{ic}(x, x', \delta, \delta_C)$

$Q^{\bullet}(x, x', \delta, \delta_C) \triangleq \mathrm{copy}(x) = x' \wedge x' \in \delta_C \wedge$
$\forall z \in \delta.\ \forall \pi'.\ \pi' \notin \mathrm{promise}(z) \wedge \pi' \neq \mathrm{thread}(x)$

all promise sets are empty

$\mathrm{ic}(z, z', \delta, \delta_C) \triangleq$ if $z'$ is 0 ($z$ is not copied yet), then $z$ will eventually be copied:
there exists some $y$ in $\delta$ s.t.
1) the promise set of $y$ is non-empty; 2) $y$ can reach $z$ along a path p;
and 3) every node along the path p is not copied
otherwise, $z'$ is a node in $\delta_C$ and the children of $z$ are also copied

# 4. Mathematical Specification

$\text{Inv}(\delta, \delta_c) \triangleq \text{acyc}(\delta) \wedge \text{acyc}(\delta_c) \wedge \forall x' \in \delta_c.\ \exists! x \in \delta.\ \text{copy}(x) = x'$
$\wedge\ \forall x \in \delta.\ \exists x'.\ \text{copy}(x) = x' \wedge \text{ic}(x, x', \delta, \delta_c)$

$Q^{\bullet}(x, x', \delta, \delta_c) \triangleq \text{copy}(x) = x' \wedge x' \in \delta_c \wedge$
$\forall z \in \delta.\ \forall \pi'.\quad \pi' \notin \text{promise}(z) \wedge \pi' \neq \text{thread}(x)$

$\text{ic}(z, z', \delta, \delta_c) \triangleq$ $z'$ is a node in $\delta_c$ and the children of $z$ are also copied

# 4. Mathematical Specification

Inv(δ, $δ_c$) ≜ acyc(δ) ∧ acyc($δ_c$) ∧ $\forall x' \in δ_c$. $\exists! x \in δ$. copy($x$)= $x'$
∧ $\forall x \in δ$. $\exists x'$. copy($x$)=$x'$ ∧ ic($x$, $x'$, δ, $δ_c$)

$Q^{\bullet}$($x$, $x'$, δ, $δ_c$) ≜ copy($x$) = $x'$ ∧ $x' \in δ_c$ ∧
$\forall z \in δ$. $\forall \pi'$. $\pi' \notin$ promise($z$) ∧ $\pi' \neq$ thread($x$)

ic($z$, $z'$, δ, $δ_c$) ≜ $z'$ is a node in $δ_c$ and the children of $z$ are also copied

$Q^{\bullet}$($x$, $x'$, δ, $δ_c$) ∧ Inv(δ, $δ_c$) ⇒ all nodes in δ are copied to nodes in $δ_c$

# 5. Spatial Objects

# 5. Spatial Objects

- A concrete implementation of the data structures in the heap

# 5. Spatial Objects

- A concrete implementation of the data structures in the heap
  - e.g. a pair of heap-represented dags:

$$\text{icdag}(\delta, \delta_c) \triangleq d \Rightarrow \delta, \delta_c \ast \text{dag}(\delta) \ast \text{dag}(\delta_c)$$

# 5. Spatial Objects

- A concrete implementation of the data structures in the heap
  - e.g. a pair of heap-represented dags:

$$\mathrm{icdag}(\delta, \delta_c) \triangleq d \Rightarrow \delta, \delta_c * \mathrm{dag}(\delta) * \mathrm{dag}(\delta_c)$$

Tracking the abstract state of the dags:
recorded in the *ghost heap; not "baked in"* to model

# 5. Spatial Objects

- A concrete implementation of the data structures in the heap
  - e.g. a pair of heap-represented dags:

$$\mathrm{icdag}(\delta, \delta_c) \triangleq d \Rightarrow \delta, \delta_c \circledast \mathrm{dag}(\delta) \circledast \mathrm{dag}(\delta_c)$$

Tracking the abstract state of the dags:
recorded in the *ghost heap; not "baked in"* to model

  - each dag(δ) implemented as a collection of nodes:

$$\mathrm{dag}(\delta) \triangleq \mathop{\circledast}_{x \in \delta} \mathrm{node}(x, \delta)$$

# 5. Spatial Objects

- A concrete implementation of the data structures in the heap
  - e.g. a pair of heap-represented dags:

$$\text{icdag}(\delta, \delta_c) \triangleq d \Rightarrow \delta, \delta_c \ast \text{dag}(\delta) \ast \text{dag}(\delta_c)$$

Tracking the abstract state of the dags:
recorded in the *ghost heap; not "baked in"* to model

  - each dag(δ) implemented as a collection of nodes:

$$\text{dag}(\delta) \triangleq \mathop{\circledast}_{x \in \delta} \text{node}(x, \delta)$$

$$\text{node}(x, (V, E, L)) \triangleq \exists l, r, x', P, \pi.\ E(x) = l, r \wedge L(x) = x', \pi, P \wedge$$
$$x \mapsto x', l, r \ast x \Rightarrow \pi, P$$

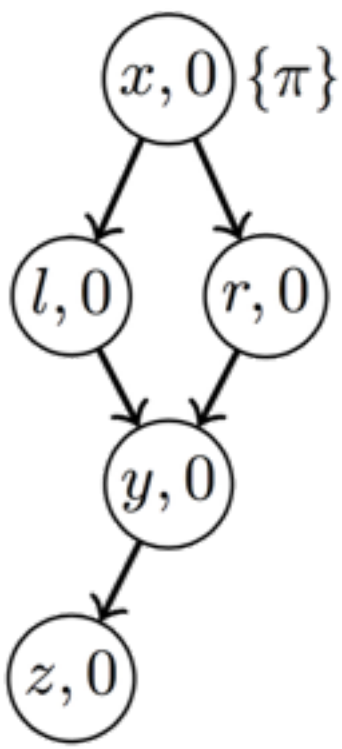
# 6. Spatial Actions

# 6. Spatial Actions

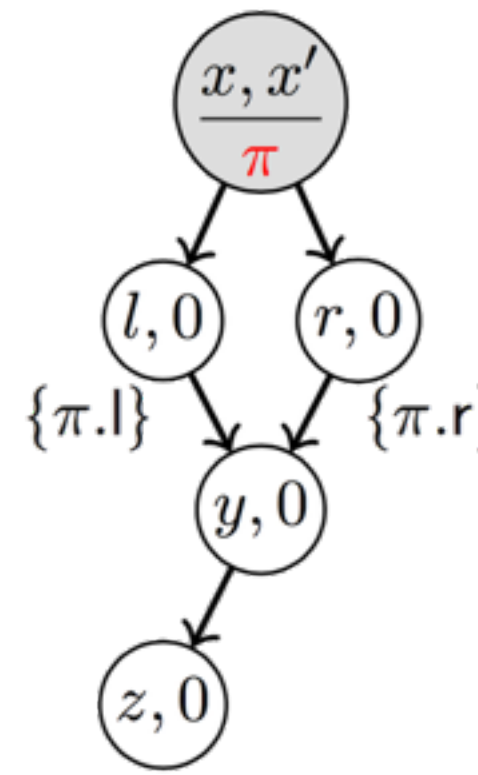
- An implementation of thread actions (on spatial objects)
  - atomic blocks as well as ghost actions

# 6. Spatial Actions

- An implementation of thread actions (on spatial objects)
  - atomic blocks as well as ghost actions



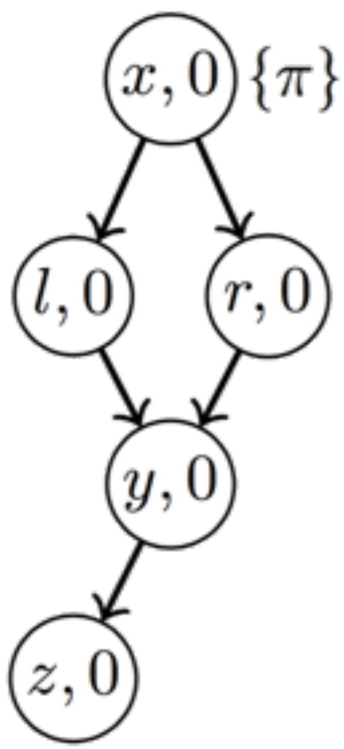

```
struct
copy_dag(
  struct
  if
  x' =
  b = <CAS(x->c, 0, x')>;
  if
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>

  } else {

  }
}
```




$(\delta, \delta_c)$ $\quad \overset{A^{\pi}}{\rightsquigarrow} \quad$ $(\delta', \delta'_c)$

# 6. Spatial Actions

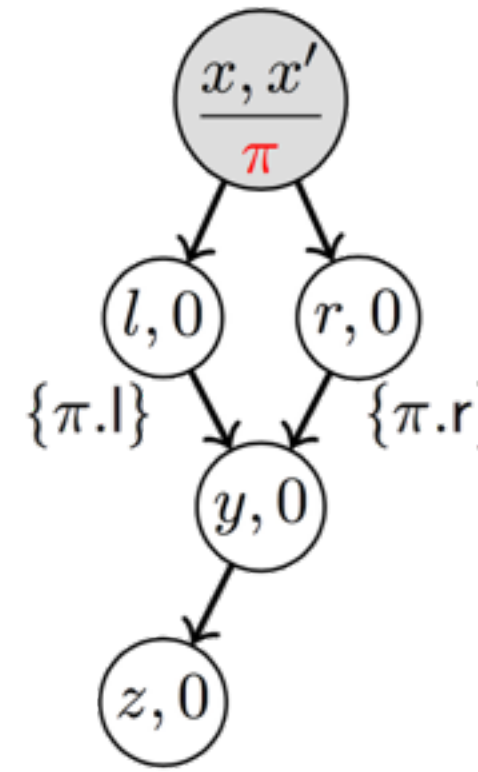
- An implementation of thread actions (on spatial objects)
  - atomic blocks as well as ghost actions
  - Lifting of mathematical actions $A^{\pi}$ to spatial ones $[A^{\pi}]$




($\delta$, $\delta_c$)

```
b = <CAS(x->c, 0, x')>;
```

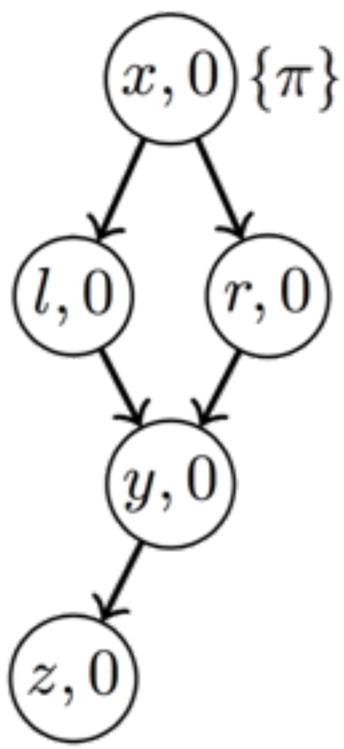
$\overset{A^{\pi}}{\rightsquigarrow}$



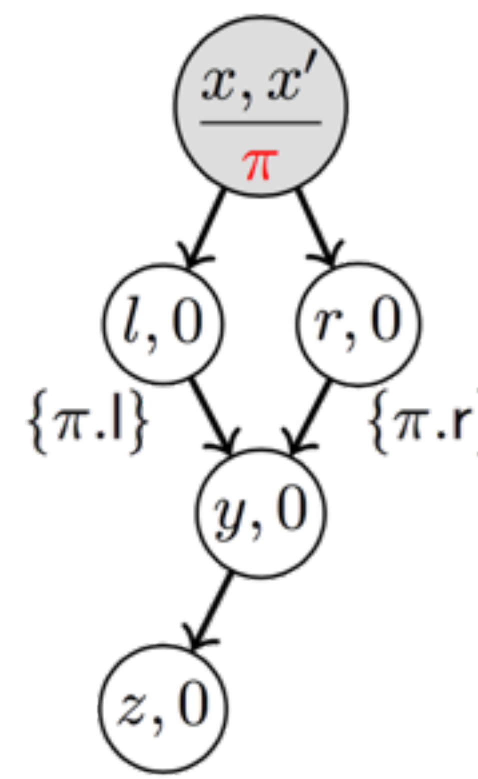

($\delta'$, $\delta'_c$)

# 6. Spatial Actions

- An implementation of thread actions (on spatial objects)
  - atomic blocks as well as ghost actions
  - Lifting of mathematical actions $A^\pi$ to spatial ones $[A^\pi]$




```
struct
copy_dag(
  struct
  if
  x' =
  b = <CAS(x->c, 0, x')>;
  if
    l = x->l; r = x->r;
    ll = copy_dag(l)  ||  rr = copy_dag(r)
    <x'->l = ll>; <x'->r = rr>

  } else {

  }
}
```




(δ, δc) $\overset{A^\pi}{\rightsquigarrow}$ (δ', δ'c)

icdag(δ, δc) $\overset{[A^\pi]}{\rightsquigarrow}$ icdag(δ', δ'c)

# 7. Spatial Specification

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0)$$

# 7. Spatial Specification

- Recall
  - the mathematical invariant Inv($\delta$, $\delta_c$)
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

Pre($x$, $\pi$, $\delta_0$)

the *original* source dag (before the top-most call to `copy_dag`)

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\text{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$\text{Pre}(x, \pi, \delta_0)$

$\pi$: the copying thread

$\delta_0$: the *original* source dag (before the top-most call to `copy_dag`)

# 7. Spatial Specification

- Recall
  - the mathematical invariant Inv(δ, $\delta_c$)
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

Pre($x$, $\pi$, $\delta_0$)

- $\delta_0$: the *original* source dag (before the top-most call to `copy_dag`)
- $\pi$: the copying thread
- $x$: the root node

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $\mathrm{P}^{\pi}(x, \delta)$ and $\mathrm{Q}^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge \mathrm{P}^{\pi}(x, \delta))}^{I}$$

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))}\ I$$

the current source dag

evolved from the original dag:
same vertices and edges
the labels may have changed

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))}_{I}$$

spatial part

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \underbrace{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))}_{I}$$

pure (mathematical) part: $(\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))$

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \land (\delta_0 \cong \delta \land \mathrm{Inv}(\delta, \delta_c) \land P^{\pi}(x, \delta))}^{I}$$

the spatial actions allowed on dags

$$I \triangleq \{\ \pi : \mathrm{icdag}(\delta, \delta_c) \rightsquigarrow \mathrm{icdag}(\delta', \delta'_c)$$

where $\mathrm{icdag}(\delta, \delta_c)\ [A^{\pi}]\ \mathrm{icdag}(\delta', \delta'_c)$

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))}^{\,I}$$

the local permissions for thread $\pi$ and all its descendants

$$\underline{\pi} \triangleq \mathop{\Large *}_{\pi' \in \{\pi' \mid \pi' \sqsubseteq \pi\}} \pi'$$

$$I \triangleq \{\pi : \mathrm{icdag}(\delta, \delta_c) \rightsquigarrow \mathrm{icdag}(\delta', \delta'_c)$$

where $\mathrm{icdag}(\delta, \delta_c)\ [A^{\pi}]\ \mathrm{icdag}(\delta', \delta'_c)$

required local permission

# 7. Spatial Specification

- Recall
  - the mathematical invariant $\mathrm{Inv}(\delta, \delta_c)$
  - and the mathematical pre- and postconditions, $P^{\pi}(x, \delta)$ and $Q^{\pi}(x, x', \delta, \delta_c)$
- The spatial precondition is

$$\mathrm{Pre}(x, \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge P^{\pi}(x, \delta))}_{I}$$

$$\underline{\pi} \triangleq \mathop{\circledast}_{\pi' \in \{\pi' \mid \pi' \sqsubseteq \pi\}} \pi'$$

$$I \triangleq \{\ \pi : \mathrm{icdag}(\delta, \delta_c) \rightsquigarrow \mathrm{icdag}(\delta', \delta'_c)$$
$$\text{where} \quad \mathrm{icdag}(\delta, \delta_c)\ [A^{\pi}]\ \mathrm{icdag}(\delta', \delta'_c)$$

- The spatial postcondition is

$$\mathrm{Post}(x, x', \pi, \delta_0) \triangleq \underline{\pi} * \boxed{\exists \delta, \delta_c.\ \mathrm{icdag}(\delta, \delta_c) \wedge (\delta_0 \cong \delta \wedge \mathrm{Inv}(\delta, \delta_c) \wedge Q^{\pi}(x, x', \delta, \delta_c))}_{I}$$

# Verifying `copy_dag(x)`

```
struct node {struct node *c, *l, *r};
{Pre(x, π, δ)}
copy_dag(struct node *x) {struct node *l, *r, *ll, *rr, *y; bool b;
{π* • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π(x, δ1))⟩_I }
  if(!x){ return 0; }
{π* • ret≐0 • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2))⟩_I }
  y = malloc(sizeof(struct node));
{π* • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π(x, δ1))⟩_I • y ↦ 0,0,0 • y ⇒ π,∅ }
  <if(x->c){ b = false;        //Perform the action A^5_π
{π* • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, δ1^c(x), δ1, δ2) ∧ δ1^c(x) ≠ 0)⟩_I
 • y ↦ 0,−,− • y ⇒ π,∅ • b≐0 }
  }else{ x->c = y; b = true;   //Perform the action A^1_π
{π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
   (x≠y ⇒ π≠δ1^s(y)) ∧ ∃l,r. δ1(x)=(y, π, −, l, r) ∧ y∉δ2 ∧ P^π.l(l, δ1) ∧ P^π.r(r, δ1))⟩_I • b≐1 }
  }>
  if(b){ l = x->l; r = x->r;
{π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
   (x≠y ⇒ π≠δ1^s(y)) ∧ δ1(x)=(y, π, −, l, r) ∧ y∉δ2 ∧ P^π.l(l, δ1) ∧ P^π.r(r, δ1))⟩_I }
{π • ⟨∃δ1,δ2. icdag(δ1,δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
   (x≠y ⇒ π≠δ1^s(y)) ∧ δ1(x)=(y,−,π,l,r) ∧ y∉δ2)⟩_I
 • π.l* • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π.l(l, δ1))⟩_I
 • π.r* • ⟨∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π.r(r, δ1))⟩_I }
{π • ⟨∃δ1,δ2. icdag(δ1,δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y)
   ∧ (x≠y ⇒ π≠δ1^s(y)) ∧ δ1(x)=(y,−,π,l,r) ∧ y∉δ2)⟩_I • Pre(l, π.l, δ) • Pre(r, π.r, δ) }

        {Pre(l, π.l, δ)}          ||       {Pre(r, π.r, δ)}
        ll = copy_dag(l)          ||       rr = copy_dag(r)
        {Post(l, ll, π.l, δ)}     ||       {Post(r, rr, π.r, δ)}

{π • ⟨∃δ1,δ2. icdag(δ1,δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y)
   ∧ (x≠y ⇒ π≠δ1^s(y)) ∧ δ1(x)=(y,−,π,l,r) ∧ y∉δ2)⟩_I • Post(l, ll, π.l, δ) • Post(r, rr, π.r, δ) }
{π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1,δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
   (x≠y ⇒ π≠δ1^s(y)) ∧ δ1(x)=(y,−,π,l,r) ∧ y∉δ2 ∧ Q^π.l(l, ll, δ1, δ2) ∧ Q^π.r(r, rr, δ1, δ2))⟩_I }
   <y->l = ll>; <y->r = rr>;  //Perform A^2_π, A^3_π and A^4_π in order.
{π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, y, δ1, δ2))⟩_I }
   return y;  {π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2))⟩_I }
  }else{
{π* • ⟨∃δ1,δ2. icdag(δ1,δ2) • (δ≐δ1 ∧ Inv(δ1,δ2) ∧ Q^π(x, δ1^c(x), δ1, δ2) ∧ δ1^c(x) ≠ 0)⟩_I • y ↦ 0,−,− • y ⇒ π,∅ }
   free(y, sizeof(struct node)) ; return x->c;
{π* • ⟨∃δ1,δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2))⟩_I }
} }  {Post(x, ret, π, δ)}
```

Changes reflected in the pure (mathematical) part as highlighted

# Verifying `copy_dag(x)`

```
struct node {struct node *c, *l, *r};
{Pre(x, π, δ)}
copy_dag(struct node *x) {struct node *l, *r, *ll, *rr, *y; bool b;
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π(x, δ1)))_I }
  if(!x){ return 0; }
{π* • ret≐0 • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2)))_I }
  y = malloc(sizeof(struct node));
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^π(x, δ1)))_I • y↦0,0,0 • y⇒π,∅ }
  <if(x->c){ b = false;      //Perform the action A^5_π
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, δ1^c(x), δ1, δ2) ∧ δ1^c(x) ≠ 0))_I
 • y↦0,−,− • y⇒π,∅ • b≐0 }
  }else{ x->c = y; b = true;      //Perform the action A^1_π
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
  (x≠y ⇒ π≠δ1^c(y)) ∧ ∃l, r. δ1(x)=(y, π, −, l, r) ∧ y∈̇δ2 ∧ P^{π.l}(l, δ1) ∧ P^{π.r}(r, δ1)))_I • b≐1 }
  }>
  if(b){ l = x->l; r = x->r;
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
  (x≠y ⇒ π≠δ1^c(y)) ∧ δ1(x)=(y, π, −, l, r) ∧ y∈̇δ2 ∧ P^{π.l}(l, δ1) ∧ P^{π.r}(r, δ1)))_I }
{π • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
  (x≠y ⇒ π≠δ1^c(y)) ∧ δ1(x)=(y, −, π, l, r) ∧ y∈̇δ2))_I
 • π.l* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^{π.l}(l, δ1)))_I
 • π.r* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ P^{π.r}(r, δ1)))_I }
{π • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y)
  ∧ (x≠y ⇒ π≠δ1^c(y)) ∧ δ1(x)=(y, −, π, l, r) ∧ y∈̇δ2))_I • Pre(l, π.l, δ) • Pre(r, π.r, δ) }
        {Pre(l, π.l, δ)}        ||   {Pre(r, π.r, δ)}
        ll = copy_dag(l)        ||   rr = copy_dag(r)
        {Post(l, ll, π.l, δ)}   ||   {Post(r, rr, π.r, δ)}
{π • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y)
  ∧ (x≠y ⇒ π≠δ1^c(y)) ∧ δ1(x)=(y, −, π, l, r) ∧ y∈̇δ2))_I • Post(l, ll, π.l, δ) • Post(r, rr, π.r, δ) }
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ ∀y∈δ1. π∉δ1^p(y) ∧
  (x≠y ⇒ π≠δ1^c(y)) ∧ δ1(x)=(y, −, π, l, r) ∧ y∈̇δ2 ∧ Q^{π.l}(l, ll, δ1, δ2) ∧ Q^{π.r}(r, rr, δ1, δ2)))_I }
   <y->l = ll>; <y->r = rr>;  //Perform A^2_π, A^3_π and A^4_π in order.
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, y, δ1, δ2)))_I }
   return y; {π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2)))_I }
  }else{
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, δ1^c(x), δ1, δ2) ∧ δ1^c(x) ≠ 0))_I • y↦0,−,− • y⇒π,∅ }
   free(y, sizeof(struct node)) ; return x->c;
{π* • (∃δ1, δ2. icdag(δ1, δ2) • (δ≐δ1 ∧ Inv(δ1, δ2) ∧ Q^π(x, ret, δ1, δ2)))_I }
} } {Post(x, ret, π, δ)}
```

Changes reflected in the pure (mathematical) part as highlighted

The spatial part *appears* unchanged as highlighted

# Verifying `copy_dag(x)`


Changes reflected in the pure (mathematical) part as highlighted

The spatial part *appears* unchanged as highlighted

# Conclusions

- ☑ Verified 4 concurrent fine-grained graph algorithms
  - ☑ Copying dags (directed acyclic graphs)
  - ☑ Speculative variant of Dijkstra's shortest path
  - ☑ Computing the spanning tree of a graph
  - ☑ Marking a graph
- ☑ Presented a common proof pattern for graph algorithms
  - ☑ *Abstract mathematical* graphs for Functional correctness
  - ☑ *Concrete Spatial (heap-represented)* graphs for memory safety
  - ☑ Combined reasoning for full proof
  - ☑ Inspired by existing logics where this pattern is "baked-in" to the model
  - ☑ "Baking-in" is unnecessary; demonstrated by CoLoSL reasoning

# Conclusions

- ☑ Verified 4 concurrent fine-grained graph algorithms
  - ☑ Copying dags (directed acyclic graphs)
  - ☑ Speculative variant of Dijkstra's shortest path
  - ☑ Computing the spanning tree of a graph
  - ☑ Marking a graph
- ☑ Presented a common proof pattern for graph algorithms
  - ☑ *Abstract mathematical* graphs for Functional correctness
  - ☑ *Concrete Spatial (heap-represented)* graphs for memory safety
  - ☑ Combined reasoning for full proof
  - ☑ Inspired by existing logics where this pattern is "baked-in" to the model
  - ☑ "Baking-in" is unnecessary; demonstrated by CoLoSL reasoning

**Thank you for listening!**

# Speculative Concurrent Shortest Path

```
parallel_dijkstra((int[][] a, int[] c, int size, src) {
  bitarray work[size], done[size];
  for (i=0; i<size; i++){
    c[i] = a[src][i]; work[i] = 1; done[i] = 0;  //initialisation
  }; c[src] = 0;
  dijkstra(a,c,size,work,done) || ... || dijkstra(a,c,size,work,done)
}

dijkstra(int[][] a, int[] c, int size, bitarray work, done){
  i = 0;
  while(done != 2^size-1){
    b = <CAS(work[i], 1, 0)>;
    if(b){ cost = c[i];
      for(j=0; j<size; j++){ newcost = cost + a[i][j]; b = true;
        do{ oldcost = c[j];
          if(newcost < oldcost){
            b = <CAS(work[j], 1, 0)>;
            if(b){ b = <CAS(c[j], oldcost, newcost)>; <work[j] = 1>; }
            else { b = <CAS(done[j], 1, 0)>;
              if(b){ b = <CAS(c[j], oldcost, newcost)>;
                if(b){ < work[j] = 1 > } else { < done[j] = 1 > }
          } } }
        } while(!b)
      } < done[i] = 1 >;
    } i = (i+1) mod size;
} }
```