Program Proofs in Hybrid Separation Logic

Armaël Guéneau & Jules Villard

Imperial College of London & ENS Lyon

4th September 2014

INTRODUCTION

General field of study: imperative programs verification.

- We want to prove specifications, as Hoare triples: $\{P\}$ c $\{Q\}$
- We are also interested in *memory safety*

An existing framework: Separation logic

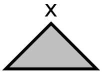
- Assertions P, Q describe memory heaps
- An additional inference rule for triples
- Proving a specification requires memory safety

OUTLINE OF THIS TALK

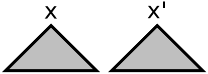
- Let's play with separation logic: a motivational example
- Introducing hybrid separation logic
- Can we do nicer proofs of our example using it?
- Can we automate these proofs?

A motivational example: `copytree`

THE copytree EXAMPLE [REY02]




```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;

  tree* l' = copytree(x->l);
  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  return x';
}
```

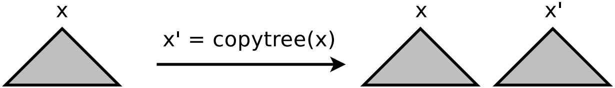



```
struct tree {
  int val;
  tree* l;
  tree* r;
};
```

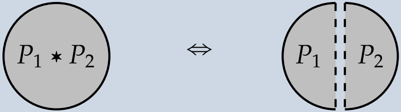
What specification for copytree?

THE `copytree` EXAMPLE [REY02]

A FIRST SPECIFICATION



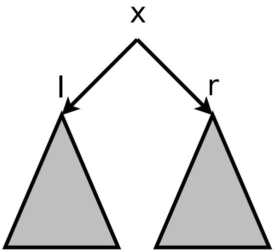
Separating conjunction



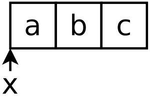
$\{\text{tree } x\}$ `x' = copytree(x)` $\{\text{tree } x \star \text{tree } x'\}$

THE copytree EXAMPLE [REY02]

A FIRST SPECIFICATION



$x \mapsto a, b, c$:



emp: the empty heap

$$\text{tree } x \equiv (\exists l, r : x \mapsto val, l, r \star \text{tree } l \star \text{tree } r) \\ \vee (x = 0 \wedge \texttt{emp})$$

```
// {tree x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;


  tree* l' = copytree(x->l);



  tree* r' = copytree(x->r);


  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;

  return x';
}
// {tree x * tree x'}
```

```
// {tree x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// {x ↦ val, l, r * tree l * tree r}

  tree* l' = copytree(x->l);

// {x ↦ val, l, r * tree l * tree r * tree l'}

  tree* r' = copytree(x->r);


  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;

  return x';
}
// {tree x * tree x'}
```

$$\text{Frame}\ \frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{P \star R\}\ \mathsf{c}\ \{Q \star R\}}$$

```
// {tree x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// {x ↦ val,l,r ∗ tree l ∗ tree r}
// ↯ {tree l}
  tree* l' = copytree(x->l);
// ↯ {tree l ∗ tree l'}
// {x ↦ val,l,r ∗ tree l ∗ tree r ∗ tree l'}

  tree* r' = copytree(x->r);


  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;

  return x';
}
// {tree x ∗ tree x'}
```

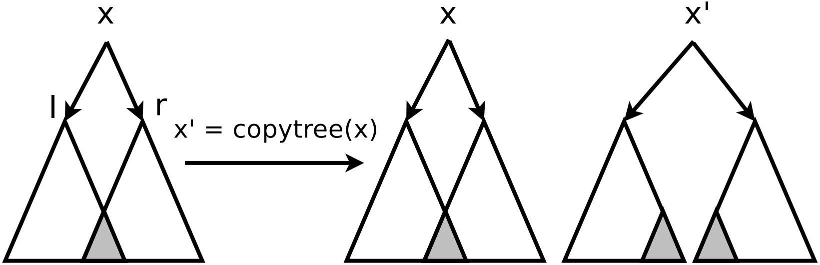
$$\text{Frame}\ \frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{P \ast R\}\ \mathsf{c}\ \{Q \ast R\}}$$

```
// {tree x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// {x ↦ val, l, r * tree l * tree r}
// ↯ {tree l}
  tree* l' = copytree(x->l);
// ↯ {tree l * tree l'}
// {x ↦ val, l, r * tree l * tree r * tree l'}

  tree* r' = copytree(x->r);
// {x ↦ val, l, r * tree l * tree r * tree l' * tree r'}

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;

  return x';
}
// {tree x * tree x'}
```

```
// {tree x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// {x ↦ val, l, r ∗ tree l ∗ tree r}
// ↯ {tree l}
  tree* l' = copytree(x->l);
// ↯ {tree l ∗ tree l'}
// {x ↦ val, l, r ∗ tree l ∗ tree r ∗ tree l'}

  tree* r' = copytree(x->r);
// {x ↦ val, l, r ∗ tree l ∗ tree r ∗ tree l' ∗ tree r'}

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  x'->val = x->val;
// {x ↦ val, l, r ∗ tree l ∗ tree r ∗ x' ↦ val, l', r' ∗ tree l' ∗ tree r'}
  return x';
}
// {tree x ∗ tree x'}
```

THE `copytree` EXAMPLE [REY02]

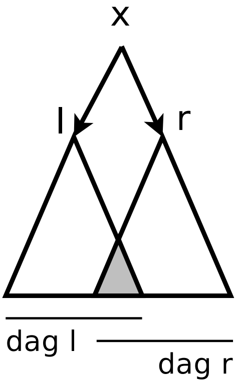
In fact, `copytree` also works on dags (directed acyclic graphs).



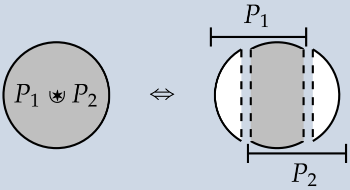
$$\mathrm{dag}\, x \quad \equiv \quad \exists l, r : x \mapsto val, l, r \star (\mathrm{dag}\, l \; \textbf{??} \; \mathrm{dag}\, r)$$
$$\vee (x = 0 \wedge \mathtt{emp})$$

THE copytree EXAMPLE [REY02]

TALKING ABOUT OVERLAPPING HEAPS

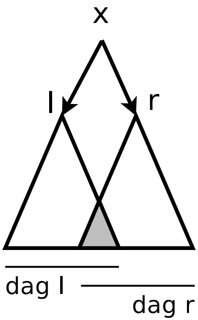


Overlapping conjunction



THE copytree EXAMPLE [REY02]

WHAT DEFINITION OF dag x?



$$\begin{aligned} \operatorname{dag} x \quad \equiv \quad & \exists l, r : x \mapsto val, l, r \star (\operatorname{dag} l \circledast \operatorname{dag} r) \\ & \vee (x = 0 \wedge \mathsf{emp}) \end{aligned}$$

THE copytree EXAMPLE [REY02]

$\{\text{dag } x\}$ `x' = copytree(x)` $\{\text{dag } x \star \text{tree } x'\}$

We cannot prove this specification.

```
// {x ↦ val, l, r ⋆ (dag l ⊛ dag r)}
tree* l' = copytree(x->l);
// {x ↦ val, l, r ⋆ (dag l ⊛ dag r) ⋆ tree l'}
```

THE copytree EXAMPLE [REY02]

$\{\mathsf{dag}\ x\}$ `x' = copytree(x)` $\{\mathsf{dag}\ x \star \mathsf{tree}\ x'\}$

We cannot prove this specification.

// $\{x \mapsto val, l, r \star (\mathsf{dag}\ l \circledast \mathsf{dag}\ r)\}$
// ↯ $\{\mathsf{dag}\ l\}$
```
tree* l' = copytree(x->l);
```
// ↯ $\{\mathsf{dag}\ l \star \mathsf{tree}\ l'\}$
// $\{x \mapsto val, l, r \star (\mathsf{dag}\ l \circledast \mathsf{dag}\ r) \star \mathsf{tree}\ l'\}$

THE `copytree` EXAMPLE [REY02]

Solution idea from Reynolds [Rey02]: use an *assertion variable* that implicitly quantifies over properties on the heap.

$$\{p \land \mathrm{dag}\ \tau\ x\}\ \texttt{x'} \leftarrow \texttt{copytree(x)}\ \{p \star \mathrm{tree}\ \tau\ x\}$$

- Has a taste of second-order logic
- Overkill?

THE copytree EXAMPLE [REY02]

Solution from Hobor & Villard [HobVill13]:

- Very precise dag predicate (parametrized by a mathematical view of the dag)
- Prove functional correctness
- Ramification instead of frame rule + heavy semantic proofs

THE copytree EXAMPLE

Automated reasoning requires a much simpler reasoning.

To talk about preserving parts of the heap, we can use **labels**! (think **heap variables**)

HYBRID SEPARATION LOGIC:

SEPARATION LOGIC + LABELS

INTRODUCING THE HYBRID SEPARATION LOGIC

Separation logic: defines the interpretation of $\wedge$, $\vee$, $\Rightarrow$, $\star$, $-\!\!\star$, ...

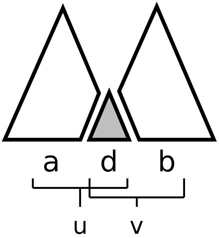
Hybrid separation logic: separation logic
+ ℓ (*heap variables* or *labels*)
+ $@_\ell A$ (@-modality)
+ $\exists$-quantifiers on labels

ρ: valuation: Labels $\rightharpoonup$ Heaps

$$
\begin{array}{lcl}
h \models_\rho \ell & \Leftrightarrow & h = \rho(\ell) \\
h \models_\rho @_\ell A & \Leftrightarrow & \rho(\ell) \models_\rho A \\
h \models_\rho \exists \ell : A & \Leftrightarrow & \text{exists } h_\ell \text{ heap st. } h \models_{\rho[\ell \rightarrow h_\ell]} A
\end{array}
$$

EXAMPLE: CHARACTERIZATION OF $\circledast$

$$P \circledast Q \Leftrightarrow \exists d, a, b, u, v : (a \star d \star b) \wedge @_u(a \star d) \wedge @_v(d \star b) \wedge @_u P \wedge @_v Q$$



REMARK: ∃ ARE PRENEX

In practice, ∃-quantifiers are prenex.

$$\text{Exists}\ \frac{\{P\}\ \mathsf{c}\ \{Q\}}{\{\exists x.P\}\ \mathsf{c}\ \{\exists x.Q\}}$$

We never need to explicitely manipulate formulæ with ∃.

INTUITIVE REMARKS

- *Propagation lemma*

$$\frac{\{A \wedge @_\ell P\}\ \mathsf{c}\ \{B\}}{\{A \wedge @_\ell P\}\ \mathsf{c}\ \{B \wedge @_\ell P\}}$$

- $u \wedge @_u P \Rightarrow P$

- $P \wedge @_u P \not\Rightarrow u \wedge @_u P$

- $\ell_1 \ast \ldots \ast \ell_n \wedge @_u(\ell_1 \ast \ldots \ast \ell_n) \Rightarrow u \wedge @_u(\ell_1 \ast \ldots \ast \ell_n)$

PROVING PROGRAM SPECIFICATIONS USING HYBRID SEPARATION LOGIC

BACK TO copytree

$$\begin{aligned} \text{dag}\, x \quad &\equiv \quad \exists l, r : x \mapsto val, l, r \star (\text{dag}\, l \circledast \text{dag}\, r) \\ &\quad\quad \vee ((x = 0) \wedge \texttt{emp}) \end{aligned}$$

$$\begin{aligned} \text{tree}\, x \quad &\equiv \quad \exists l, r : x \mapsto val, l, r \star \text{tree}\, l \star \text{tree}\, r \\ &\quad\quad \vee ((x = 0) \wedge \texttt{emp}) \end{aligned}$$

$$\{\ell \wedge @_\ell \text{dag}\, x\}\ \texttt{x'} \ \leftarrow\ \texttt{copytree(x)}\ \{\ell \star \text{tree}\, x'\}$$

This specification is provable.

```
// {ℓ ∧ @ℓdag x}
tree* copytree(tree* x) {
  if (x == NULL)
    return x;




  tree* l' = copytree(x->l);

  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  return x';
}
// {ℓ ∗ tree x'}
```

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
```

$$// \left\{ \begin{array}{ll} \ell & \wedge @_\ell(x \mapsto val, l, r \ast d_\ell \ast d \ast d_r) \\ & \wedge @_u(d_\ell \ast d) \wedge @_u \mathsf{dag}\ l \\ & \wedge @_v(d \ast d_r) \wedge @_v \mathsf{dag}\ r \end{array} \right\}$$

```
  tree* l' = copytree(x->l);

  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  return x';
}
```

$// \ \{\ell \ast \mathsf{tree}\ x'\}$

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
```

$$// \left\{ \begin{array}{ll} \ell & \wedge @_\ell(x \mapsto val, l, r * d_\ell * d * d_r) \\ & \wedge @_u(d_\ell * d) \wedge @_u \mathsf{dag}\ l \\ & \wedge @_v(d * d_r) \wedge @_v \mathsf{dag}\ r \end{array} \right\}$$

```
  tree* l' = copytree(x->l);
```

$// \ \{\ell * \mathsf{tree}\ l' \wedge \ldots\}$

```
  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  return x';
}
```

$// \ \{\ell * \mathsf{tree}\ x'\}$

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
```

// $\left\{ \begin{array}{l} \ell \quad \wedge @_\ell(x \mapsto val, l, r \ast d_\ell \ast d \ast d_r) \\ \quad\;\; \wedge @_u(d_\ell \ast d) \wedge @_u \mathsf{dag}\ l \\ \quad\;\; \wedge @_v(d \ast d_r) \wedge @_v \mathsf{dag}\ r \end{array} \right\}$

```
  tree* l' = copytree(x->l);
```

// $\{\ell \ast \mathsf{tree}\ l' \wedge \ldots\}$

```
  tree* r' = copytree(x->r);

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
  return x';
}
```

// $\{\ell \ast \mathsf{tree}\ x'\}$

	spatial part	pure facts (propagate)
	ℓ	$\wedge @_l(x \mapsto val, l, r * d_\ell * d * d_r)$ $\wedge @_u(d_\ell * d) \wedge @_u \mathsf{dag}\ l$ $\wedge @_v(d * d_r) \wedge @_v \mathsf{dag}\ r$
$\Rightarrow$	$x \mapsto val, l, r * d_\ell * d * d_r$	
Frame	$\lightning\ d_\ell * d$	
$\Rightarrow$	u	$\wedge @_u \mathsf{dag}\ l$
	`l’ = copytree(x->l)`	
	$\lightning\ u * \mathtt{tree}\ l'$	
	$x \mapsto val, l, r * u * \mathtt{tree}\ l' * d_r$	
$\Rightarrow$	$x \mapsto val, l, r * d_\ell * d * \mathtt{tree}\ l' * d_r$	
$\Rightarrow$	$\ell * \mathtt{tree}\ l'$	

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
```

// $\left\{ \begin{array}{ll} \ell & \wedge @_\ell(x \mapsto val, l, r * d_\ell * d * d_r) \\ & \wedge @_u(d_\ell * d) \wedge @_u \mathsf{dag}\ l \\ & \wedge @_v(d * d_r) \wedge @_v \mathsf{dag}\ r \end{array} \right\}$

```
  tree* l' = copytree(x->l);
```

// $\{\ell * \mathsf{tree}\ l' \wedge \ldots\}$

```
  tree* r' = copytree(x->r);


  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';

  return x';
}
```

// $\{\ell * \mathsf{tree}\ x'\}$

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
```

$$// \left\{ \begin{array}{ll} \ell & \wedge @_\ell(x \mapsto val, l, r * d_\ell * d * d_r) \\ & \wedge @_u(d_\ell * d) \wedge @_u \mathsf{dag}\ l \\ & \wedge @_v(d * d_r) \wedge @_v \mathsf{dag}\ r \end{array} \right\}$$

```
  tree* l' = copytree(x->l);
```

// $\{\ell * \mathsf{tree}\ l' \wedge \ldots\}$

```
  tree* r' = copytree(x->r);
```

// $\{\ell * \mathsf{tree}\ l' * \mathsf{tree}\ r' \wedge \ldots\}$

```
  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';

  return x';
}
```

// $\{\ell * \mathsf{tree}\ x'\}$

```
tree* copytree(tree* x) {
  if (x == NULL)
    return x;
// { ℓ  ∧@ℓ(x ↦ val, l, r ∗ dℓ ∗ d ∗ dr)
//      ∧@u(dℓ ∗ d) ∧ @u dag l
//      ∧@v(d ∗ dr) ∧ @v dag r }
  tree* l' = copytree(x->l);
// {ℓ ∗ tree l' ∧ ...}
  tree* r' = copytree(x->r);
// {ℓ ∗ tree l' ∗ tree r' ∧ ...}

  tree* x' = malloc(sizeof(tree));
  x'->l = l';
  x'->r = r';
// {ℓ ∗ tree l' ∗ tree r' ∗ x' ↦ val, l', r' ∧ ...}
  return x';
}
// {ℓ ∗ tree x'}
```

To wrap up

We're happy: we have a formal framework (Hybrid separation logic) that gives us a nicer proof than the state of the art.

However, `copytree` is a bit simple: it doesn't modify anything.

More programs on dags

- `mark`: mark every node of a dag → changes the content
- `spanning`: compute the spanning tree of a dag → changes the shape

TOWARDS AUTOMATIC REASONING ON HYBRID FORMULÆ

INTEGRATION IN llStar

We have a working automatic proof of copytree in llStar.

HANDLING HYBRID FORMULÆ IN TOOLS

We have a canonical form for hybrid formulæ and the associated algorithm.

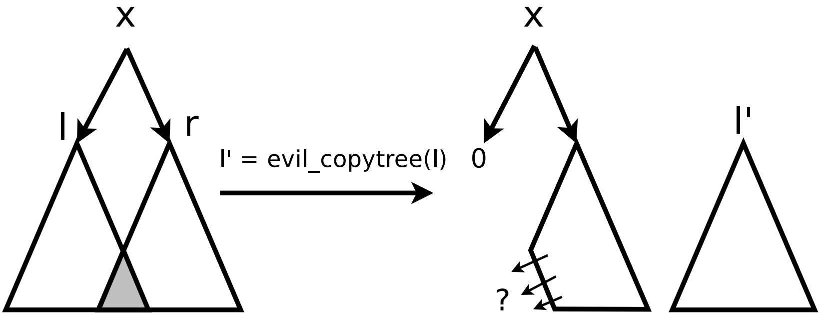
CONCLUSION

- Hybrid separation logic: nicer and simpler proofs on some examples
- Proofs that just look like folding/unfolding @s: easy automatic proving?

THE `copytree` EXAMPLE [REY02]

(Counter) example:

$$\{\text{dag } x\}\ \texttt{x' = evil\_copytree(x)}\ \{\text{dag } x \star \text{tree } x'\}$$



$$\{x \mapsto val, l, r \star (\text{dag } l \circledast \text{dag } r)\}$$

```
l' = evil_copytree(l)
```

$$\{x \mapsto val, l, r \star (\text{dag } l \circledast \text{dag } r) \star \text{tree } l'\}$$

Some more examples

- $\{\ell\}$ `skip` $\{\ell\}$ is valid.
- $\{\ell \wedge @_\ell(y \mapsto v)\}$ `x = *y` $\{\ell\}$ (modifies the stack, not the heap)
- $\{\ell\}$ `*x = 3` $\{\ell\}$ is not
- $\{\ell \wedge @_\ell(x \mapsto 4)\}$ `*x = 3` $\{\ell\}$ is not

mark

`mark` marks the nodes it goes through.

- It does not preserve the heap
- However it preserves its shape

```
void mark(dag* x) {
  if (x == NULL)
    return;

  mark(x->l);
  mark(x->r);
  x->val = 1;
}
```

mark

"Same region" operator on labels

$h \models_\rho \ell \sim \ell'$ **iff** $\rho(\ell)$ and $\rho(\ell')$ cover the same heap region

(i.e. $\mathrm{Dom}(\rho(\ell)) = \mathrm{Dom}(\rho(\ell'))$)

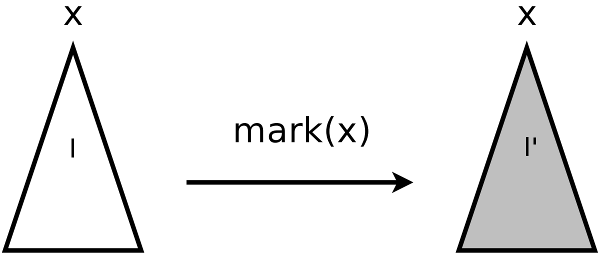
Eg,

$$
\begin{array}{lcl}
@_\ell(x \mapsto 3) \wedge @_{\ell'}(x \mapsto 4) & \Rightarrow & \ell \sim \ell' \\
@_\ell(x \mapsto 3 \ast y \mapsto 4) \wedge @_{\ell'}(x \mapsto 3) & \Rightarrow & \ell \not\sim \ell' \\
 & & \ell \sim \ell \\
@_u a \ast b \wedge @_v a' \ast b' & \Rightarrow & a \sim a' \wedge b \sim b' \Rightarrow u \sim v
\end{array}
$$

mark

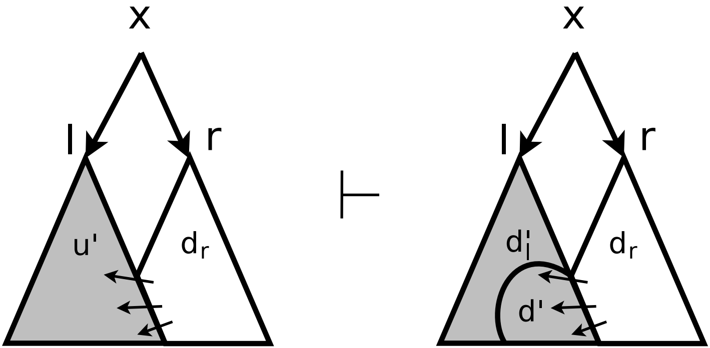
We can now write a specification for mark:

$$\{\ell \wedge @_\ell \text{dag } x\} \texttt{ mark(x) } \{\exists \ell' : \ell' \wedge @_{\ell'} \text{dag } x \wedge \ell \sim \ell'\}$$



mark

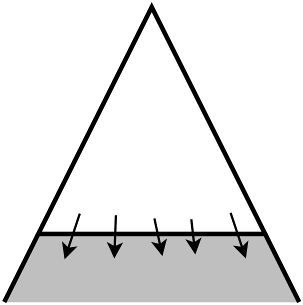
Preserving region: the minimum needed for the induction to hold.



spanning

Computes the spaning tree of a dag by removing superfluous edges.

- Input: a dag of unmarked nodes, which may point to marked nodes.



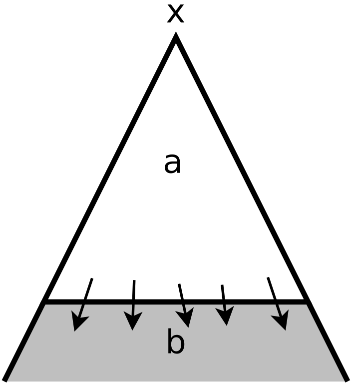
- Preserve the marked part; mark and compute the spanning tree of the unmarked one.

spanning

```
void spanning(dag* x) {
  if (x == NULL)
    return;

  x->val = 1;

  if (x->l && !x->l->val)
    spanning(l);
  else
    x->l = NULL;

  if (x->r && !x->r->val)
    spanning(r);
  else
    x->r = NULL;
}
```

spanning

Idea: parametrize the inductive predicate by labels



$$\mathrm{dag}(x, a, b)$$

spanning

Specification:

$$\{\mathrm{dag}(x, a, b)\}\ \texttt{spanning(x)}\ \{\exists a' : a' \star b \wedge @_{a'}\mathrm{mtree}(x) \wedge a' \sim a\}$$