

# Finding and Fixing Bugs in Liquid Haskell

---

Anish Tondwalkar

# Overview

- Motivation
  - Liquid Haskell
  - Fault Localization
  - Fault Localization Evaluation
  - Predicate Discovery
  - Predicate Discovery Evaluation
  - Conclusion
-

*Motivation:*

Bugs are Bad

# Bugs are Bad

- Bugs are Expensive
  - Modifying code, correcting defects, and evolving code account for as much as 90% of the total cost of software projects.



The screenshot shows the Google Application Security website. The header features the Google logo and the text "Application Security". Below the header is a navigation bar with links for Home, Learning, Reward Programs, Hall of Fame, and Research. Underneath the navigation bar, there are links for Google VRP, Patch Rewards, Research Grants, Chrome Rewards, and Android Rewards. The main content area is titled "Chrome Reward Program Rules" and contains the following text: "The Chrome Reward Program was launched in January 2010 to help reward the contributions of security make Chrome and Chrome OS more secure. Through this program we provide monetary awards and pu Chrome project." Below this text is a section titled "Scope of program" with the text: "Any security bug in Chrome or Chrome OS may be considered. It's that simple!"

Google Application Security

Home Learning **Reward Programs** Hall of Fame Research

Google VRP Patch Rewards Research Grants **Chrome Rewards** Android Rewards

## Chrome Reward Program Rules

The Chrome Reward Program was launched in January 2010 to help reward the contributions of security make Chrome and Chrome OS more secure. Through this program we provide monetary awards and pu Chrome project.

### Scope of program

Any security bug in Chrome or Chrome OS may be considered. It's that simple!

# Bugs are Bad

- Bugs are Expensive
- Bugs are Dangerous

**WIRED**

GEAR SCIENCE ENTERTAINMENT

ADVERTISEMENT

SCIENCE : DISCOVERIES 

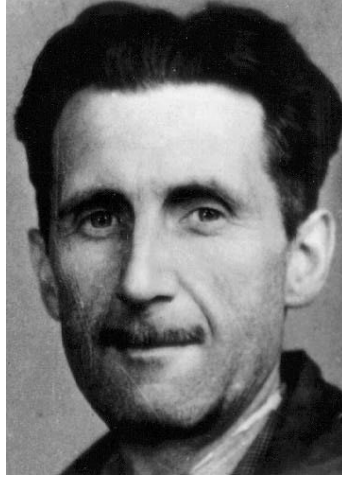
## Sunk by Windows NT

07.24.98

While Microsoft continues to trumpet the success of its NT operating systems, the US Navy is having second thoughts about putting



# Programming Languages to the Rescue!



George Orwell (1984)

*"We shall make thoughtcrime literally impossible, because there will be no words to express it."*

*Presence of bugs*  
...is half the problem

# Verification, Model Checking, and Type Systems

We can check the correctness of programs against formal specifications, which is information we can encode in Liquid Haskell type signatures.

- We still have to *find* the bug after we know it exists
- What about missing parts of the specification?



# Contributions

- We still have to *find* the bug after we know it exists
  
- What about missing parts of the specification?

# Contributions

- We still have to *find* the bug after we know it exists

A **fault localization** algorithm searches for a minimal unsatisfiable constraint set, whose constraints map to likely locations of the bug in the implementation.

- What about missing parts of the specification?

# Contributions

- We still have to *find* the bug after we know it exists

A **fault localization** algorithm searches for a minimal unsatisfiable constraint set, whose constraints map to likely locations of the bug in the implementation.

- What about missing parts of the specification?

A **predicate discovery** algorithm uses *disjunctive interpolation* to automatically expand the abstract domain by inferring predicate templates that serve as the refinement types of program expressions.



# Liquid Haskell

# Well-typed (Haskell) programs can go very wrong!

Divide-by-zero

Keys missing in Maps

Pattern-match failures

Non-termination

Functional Correctness / Assertions...

Solution:

Refinement Types

# Simple Refinement Types

Refinement Types = **Types** + **Predicates**

# Refinement Types

## Types

```
b := Int | Bool | ... -- primitives
    | a, b, c      -- variables
```

## Refinements

```
t := {x:b | p}      -- refined base
    | x:t -> t      -- refined function
```



# Predicates

## Quantifier-Free Logic of Uninterpreted Functions & Linear Arithmetic

`e := x, y, z, ...`                    `-- variables`  
`| 0, 1, 2, ...`                    `-- constants`  
`| e + e | c * e | ...`                `-- arithmetic`  
`| f e1 ... en`                    `-- uninterpreted function`

### Uninterpreted Functions

$$\forall \bar{x}, \bar{y}. \bar{x} = \bar{y} \Rightarrow f(x) = f(y)$$

# Predicates

## Quantifier-Free Logic of Uninterpreted Functions & Linear Arithmetic

Given a **Verification Condition** (VC)

$$p_1 \Rightarrow p_2$$

SMT solvers can **decide if VC is Valid** ("always true")

## An Example

```
1  sums :: k:Int -> { v:Int | k <= v }
2  sums 0 = 0
3  sums k = k + (sum (k-1))
```

# Refinement Type Solving

# Refinement Type Solving

$0 \leq v, k \leq v$

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0    = 0
6       | otherwise = let s = go (k-1) in s + k
```

# Refinement Type Solving

$0 \leq v, k \leq v$

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0     = 0
6       | otherwise = let s = go (k-1) in s + k
```

Instantiate every element with a conjunction of the abstract domain

$$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$$

# Refinement Type Solving

$0 \leq v, k \leq v$

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0    = 0
6       | otherwise = let s = go (k-1) in s + k
```

$$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$$
$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$$
$$\emptyset \vdash \text{Int} <: \kappa_k$$
$$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$$

# Refinement Type Solving

k	$0 \leq v$	$k \leq v$
s	$0 \leq v$	$k \leq v$

Instantiate every element with a conjunction of the abstract domain

$$k : \kappa_k, k \leq 0 \vdash \{\nu = 0\} <: \kappa_s$$


$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{\nu = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{\nu \geq k\}$$



# Refinement Type Solving



k	<del><math>0 \leq v</math></del>	$k \leq v$
s	$0 \leq v$	$k \leq v$

**true**  $\Rightarrow$   $0 \leq v$  


$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$

$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$

$\emptyset \vdash \text{Int} <: \kappa_k$

$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$

# Refinement Type Solving



k	<del><math>0 \leq v</math></del>	<del><math>k \leq v</math></del>
s	$0 \leq v$	$k \leq v$

**true**  $\Rightarrow$   $k \leq v$  

$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$

$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$

$\emptyset \vdash \text{Int} <: \kappa_k$

$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$

# Refinement Type Solving

k	<del><math>0 \leq v</math></del>	<del><math>k \leq v</math></del>
s	$0 \leq v$	$k \leq v$

▲

$$k < 0 \wedge v = 0 \Rightarrow 0 \leq v \quad \checkmark$$

$$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$$

# Refinement Type Solving

k	<del><math>0 \leq v</math></del>	<del><math>k \leq v</math></del>
s	$0 \leq v$	$k \leq v$

▲

$$k < 0 \wedge v = 0 \Rightarrow k \leq v \quad \checkmark$$

$$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$$

# Refinement Type Solving

k	<del><math>0 \leq v</math></del>	<del><math>k \leq v</math></del>
s	$0 \leq v$	$k \leq v$

$$0 \leq k \wedge k \leq v \Rightarrow k \leq v$$



$$k : \kappa_k, k \leq 0 \vdash \{v = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{v = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$\blacktriangleright k : \kappa_k \vdash \kappa_s <: \{v \geq k\}$$

# Contributions

- We still have to *find* the bug after we know it exists

Our **fault localization** algorithm searches for a minimal unsatisfiable constraint set, whose constraints map to likely locations of the bug in the implementation.

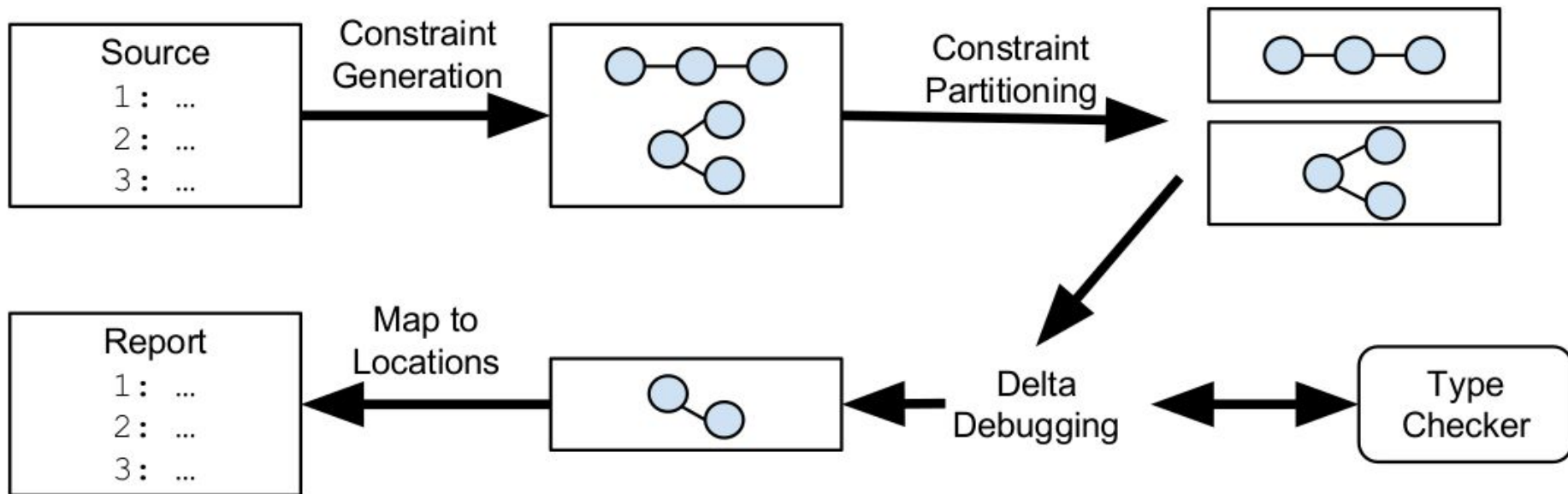
- What about missing parts of the specification?

Our **predicate discovery** algorithm uses *disjunctive interpolation* to automatically expand the abstract domain by inferring predicate templates that serve as the refinement types of program expressions.

# Fault Localization

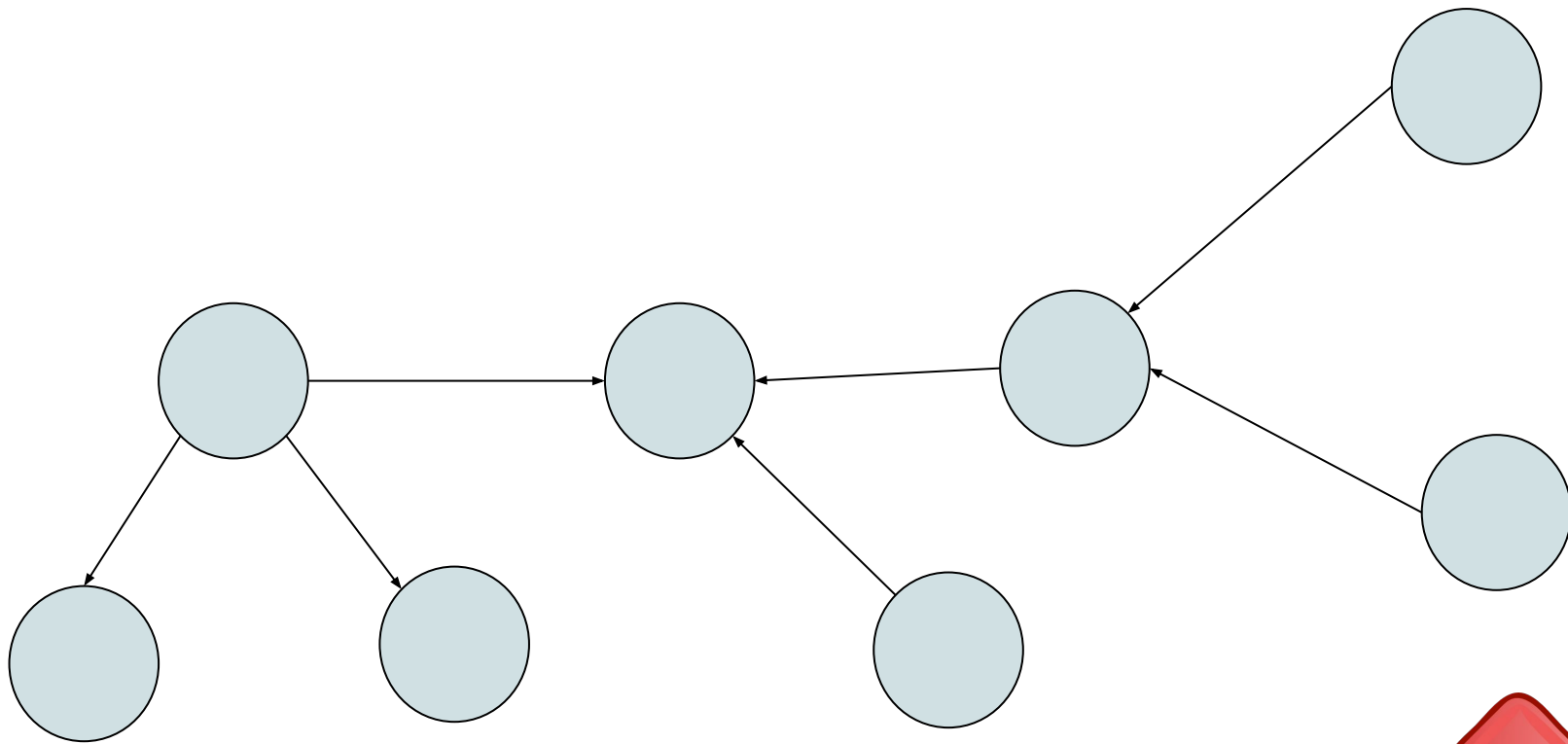
---

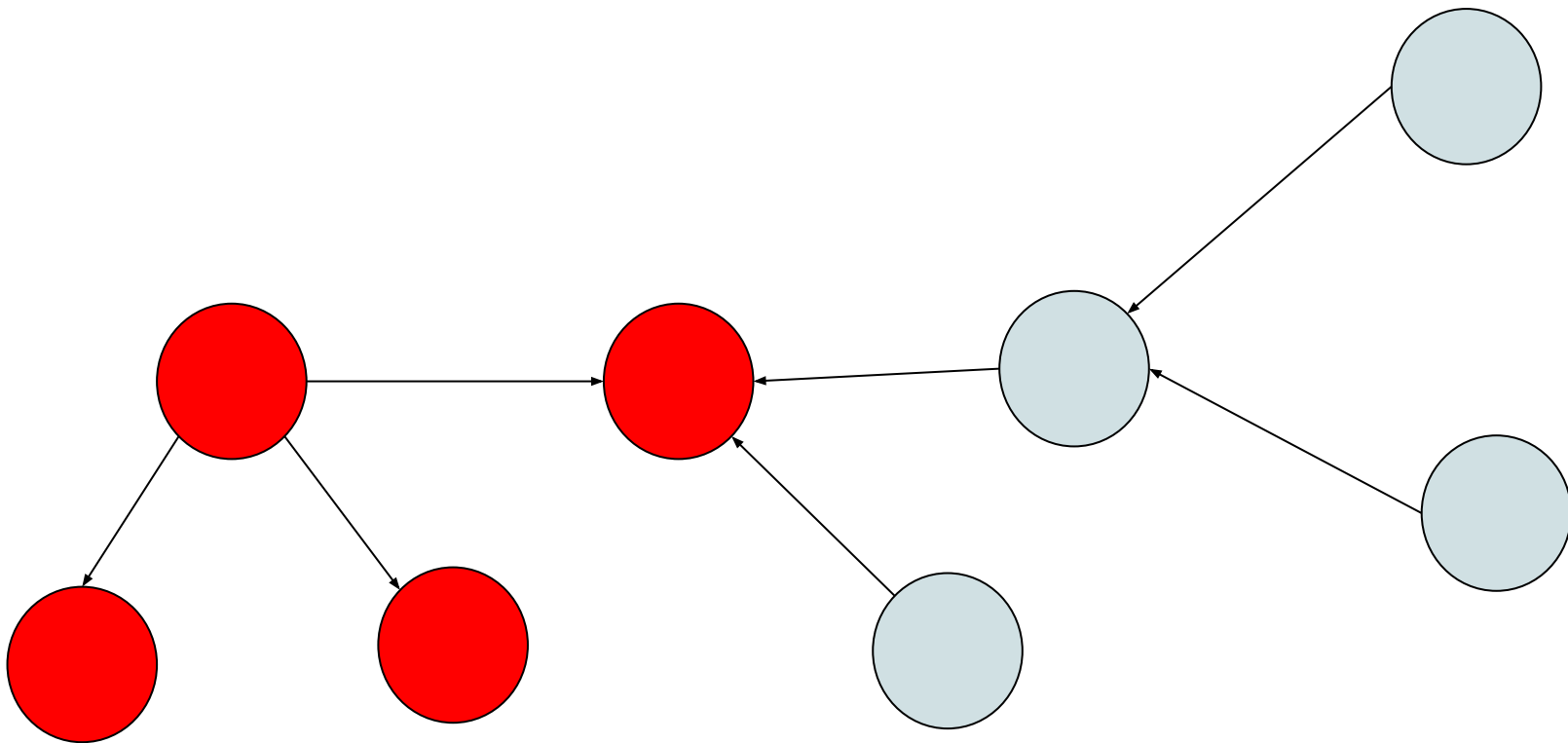
# Fault Localization Overview

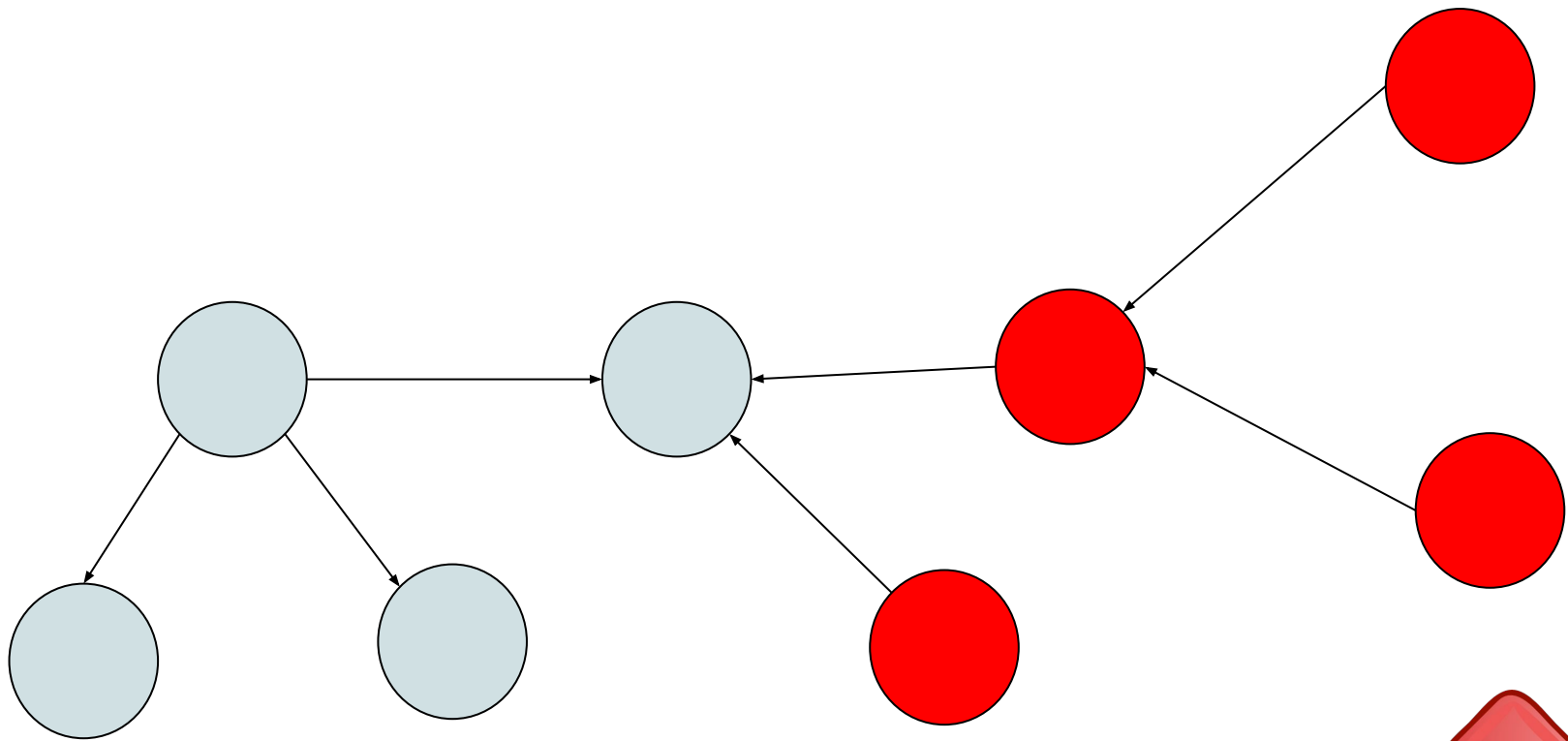


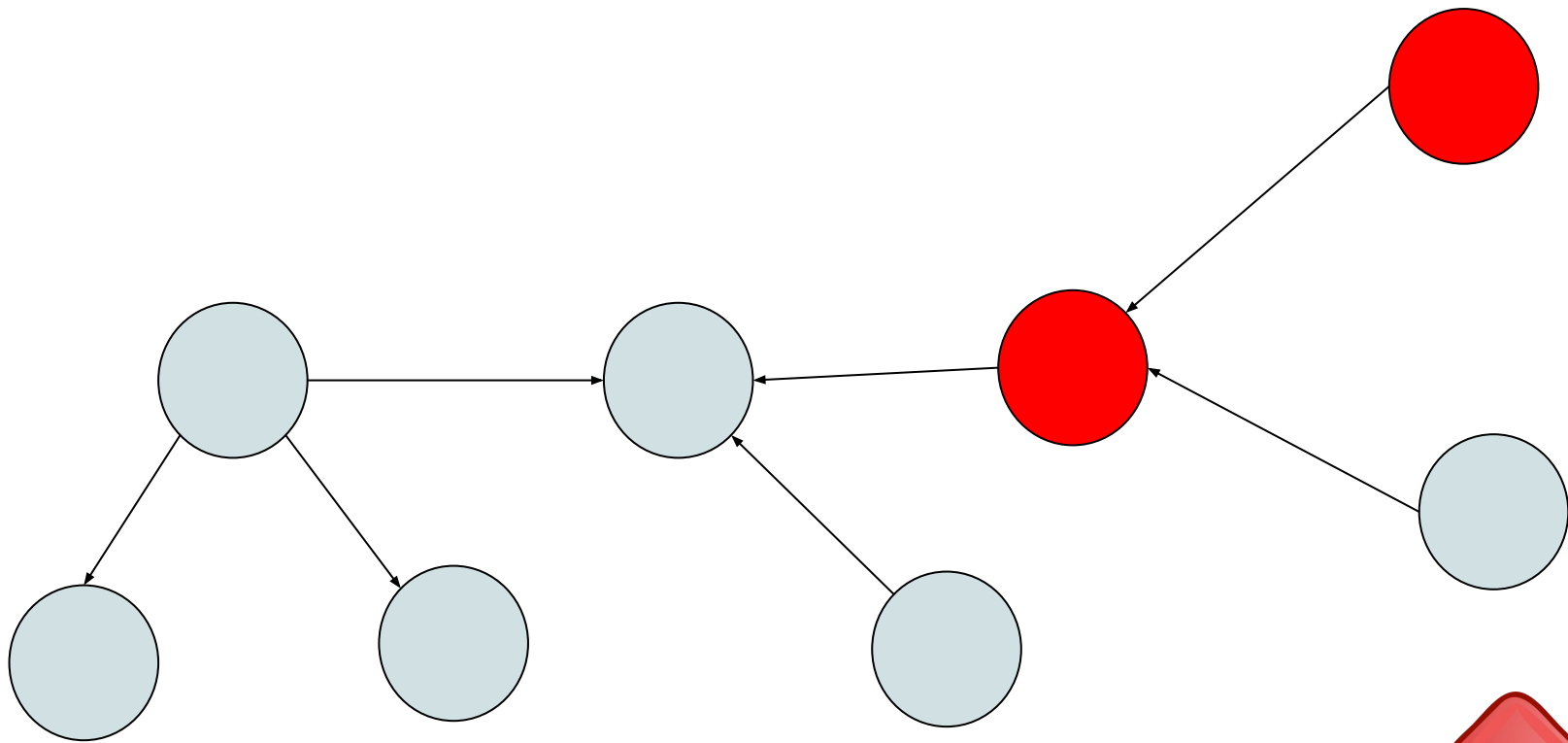


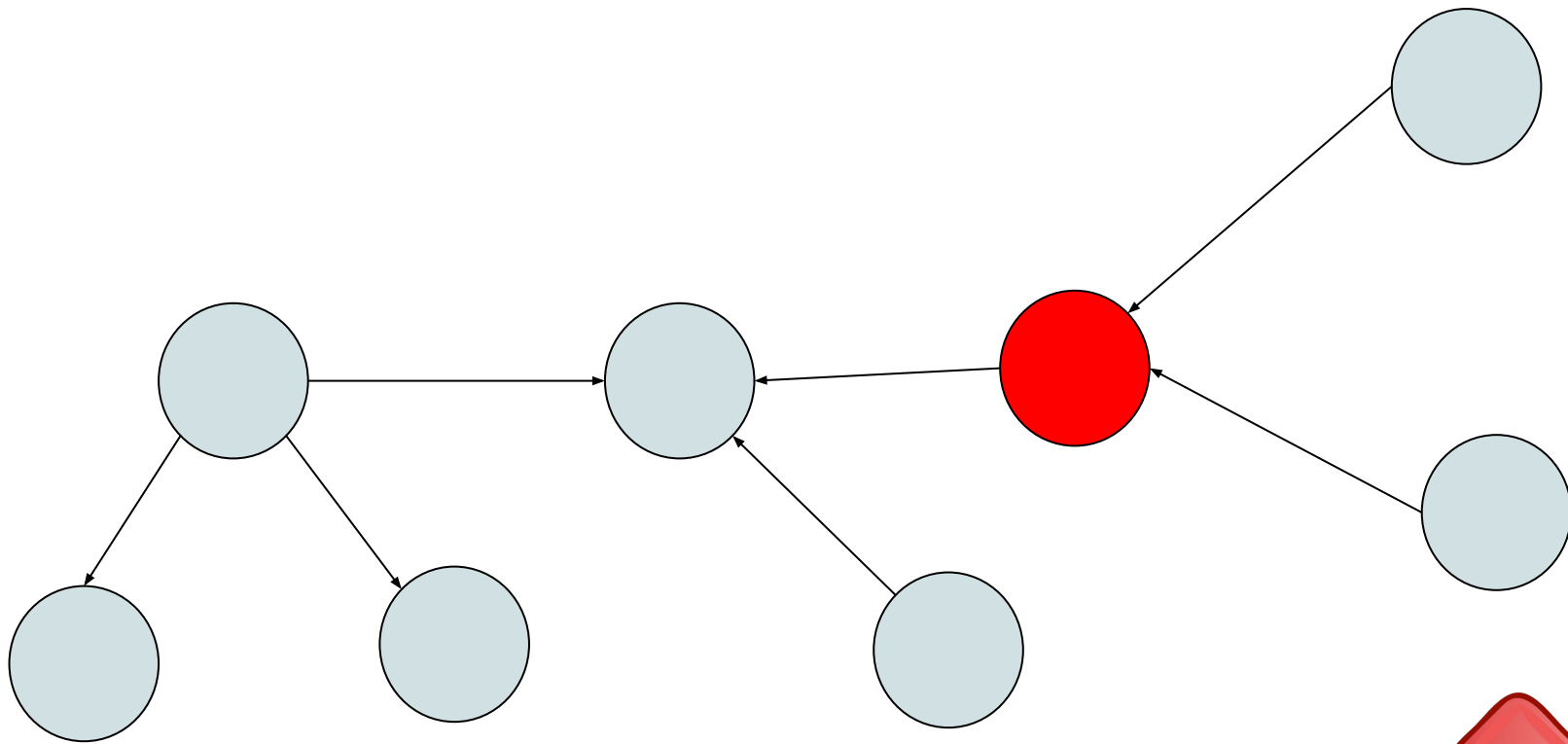
# Constraint Minimization

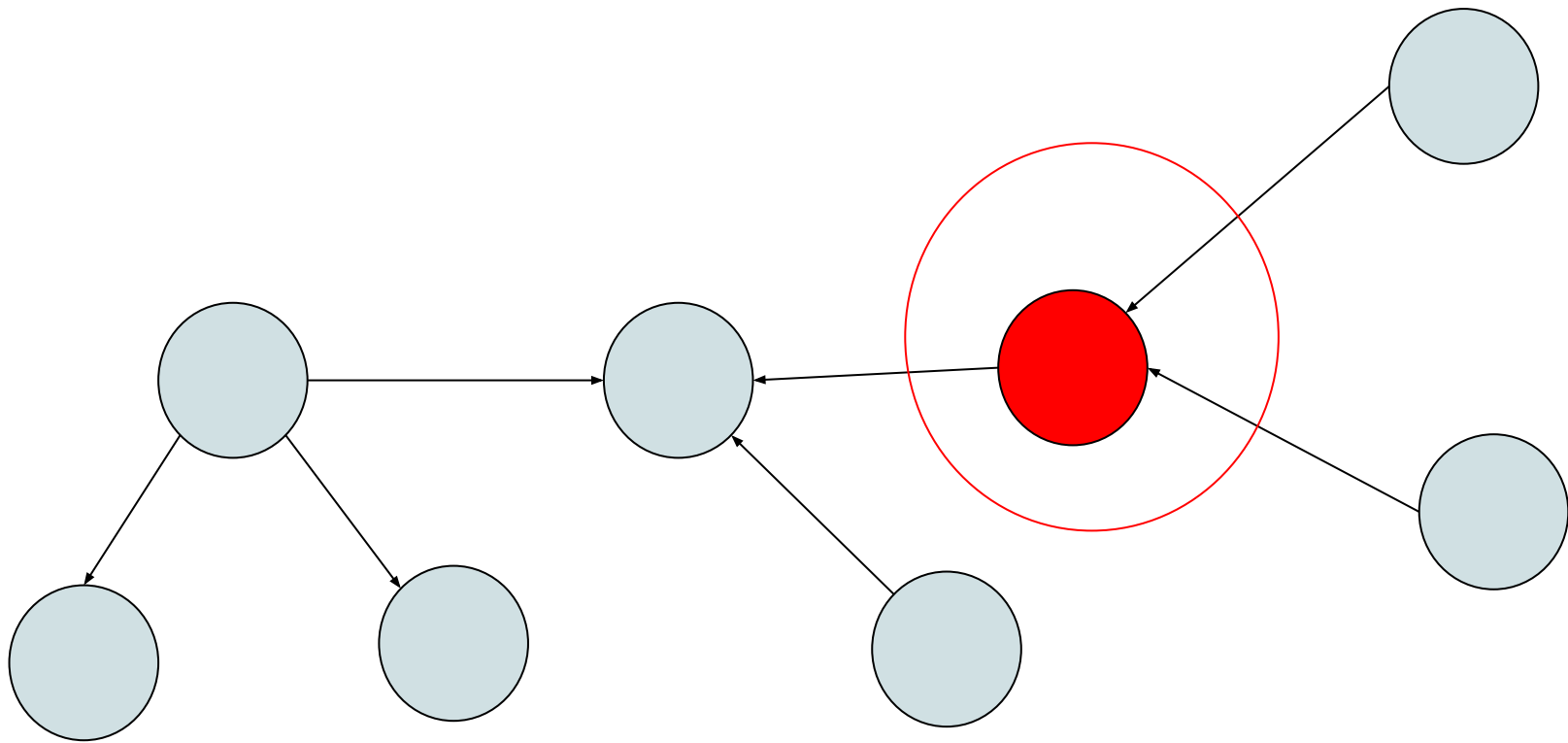






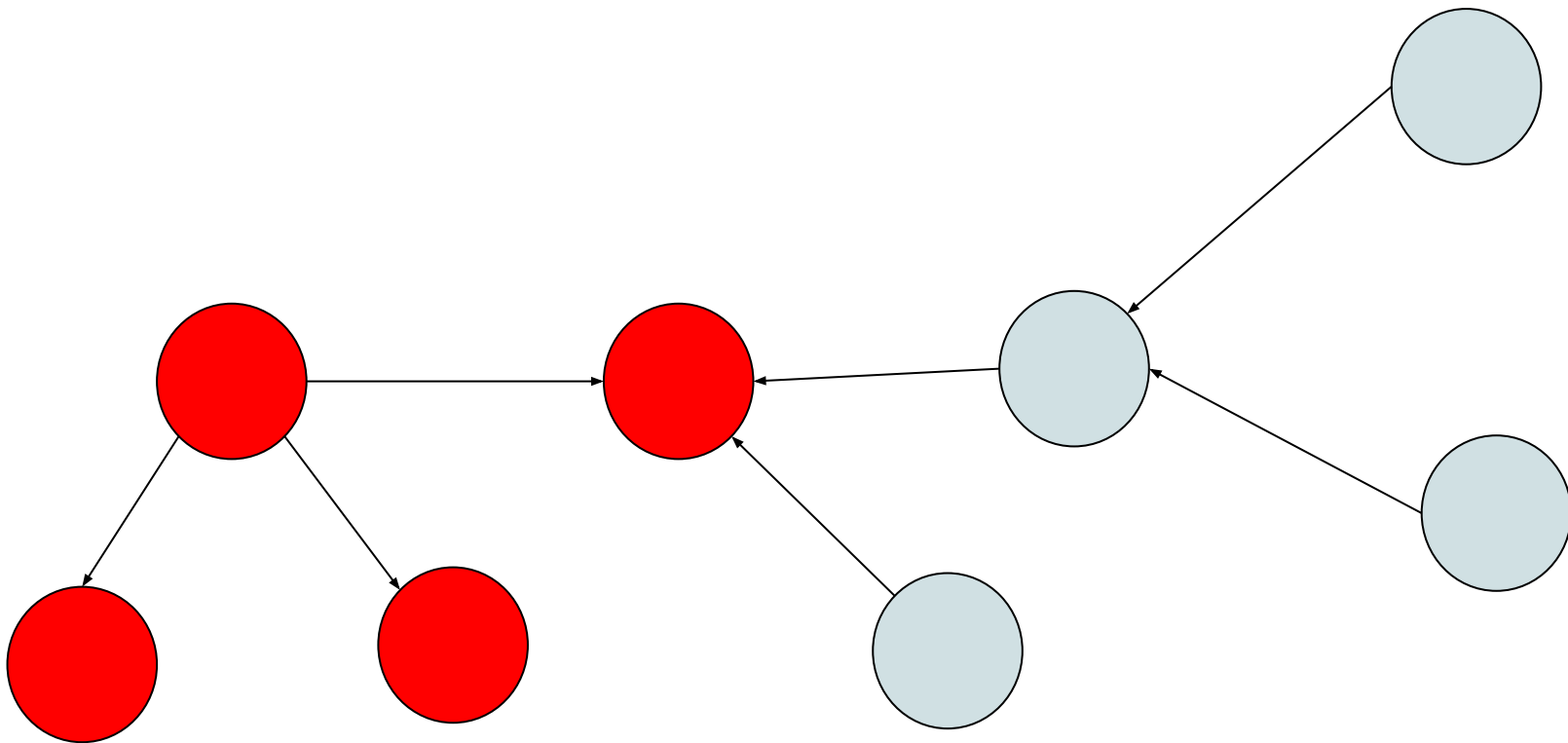


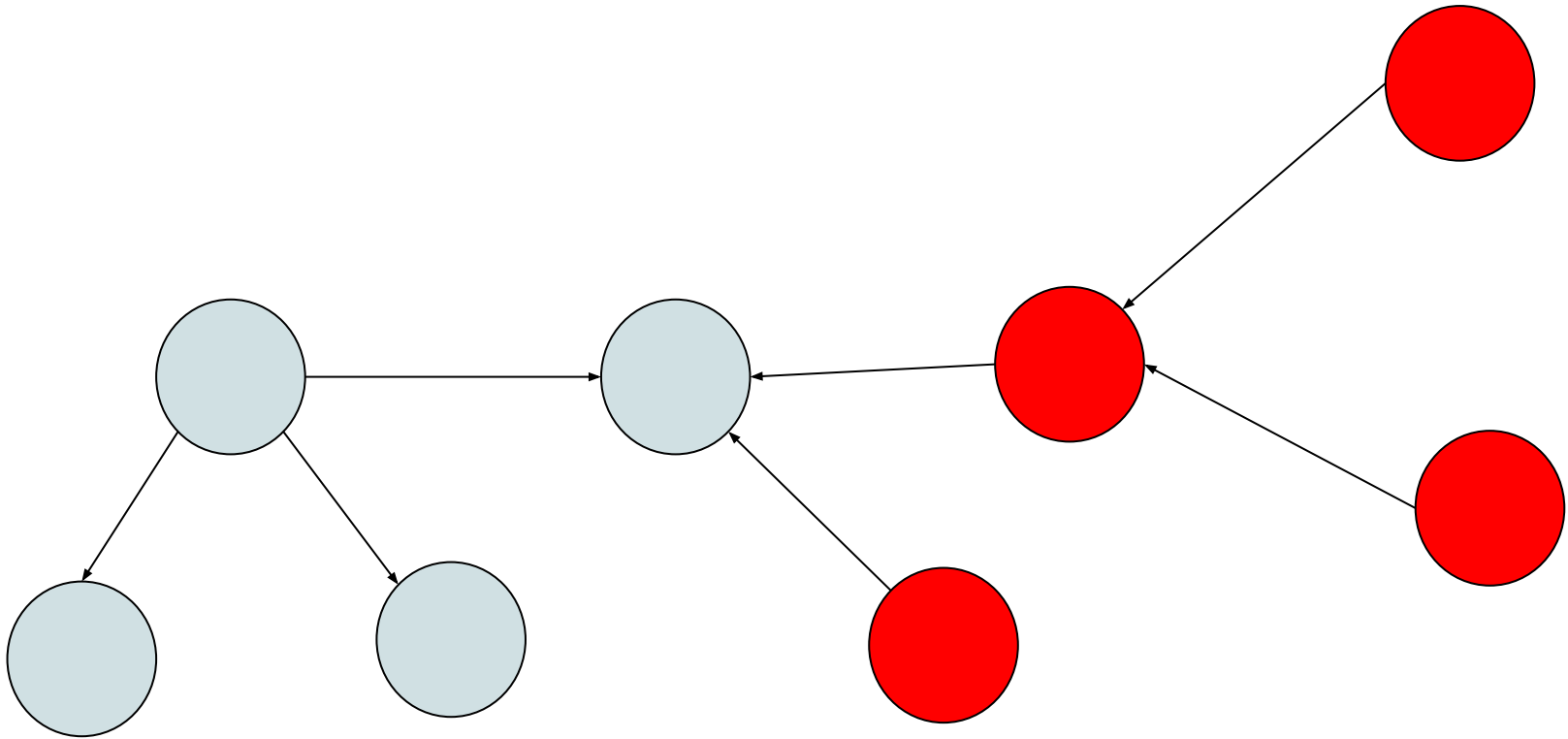


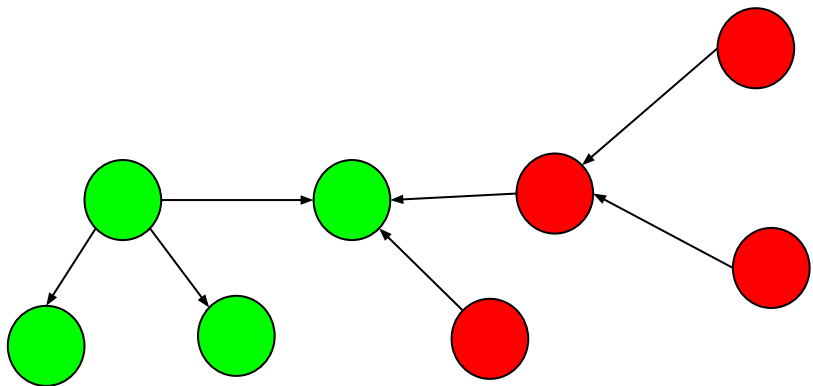


What if neither half fails?

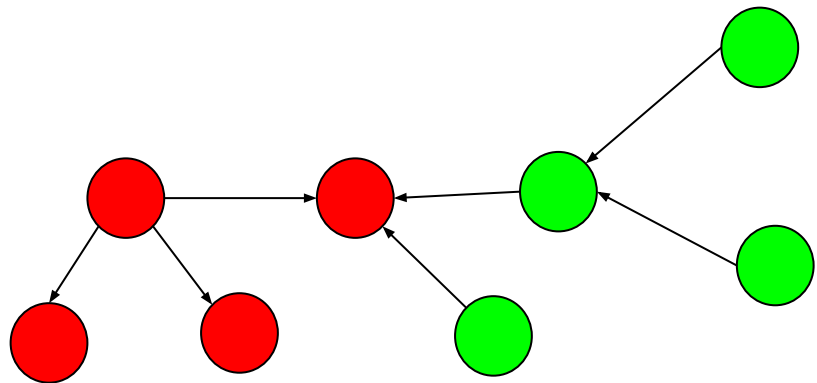


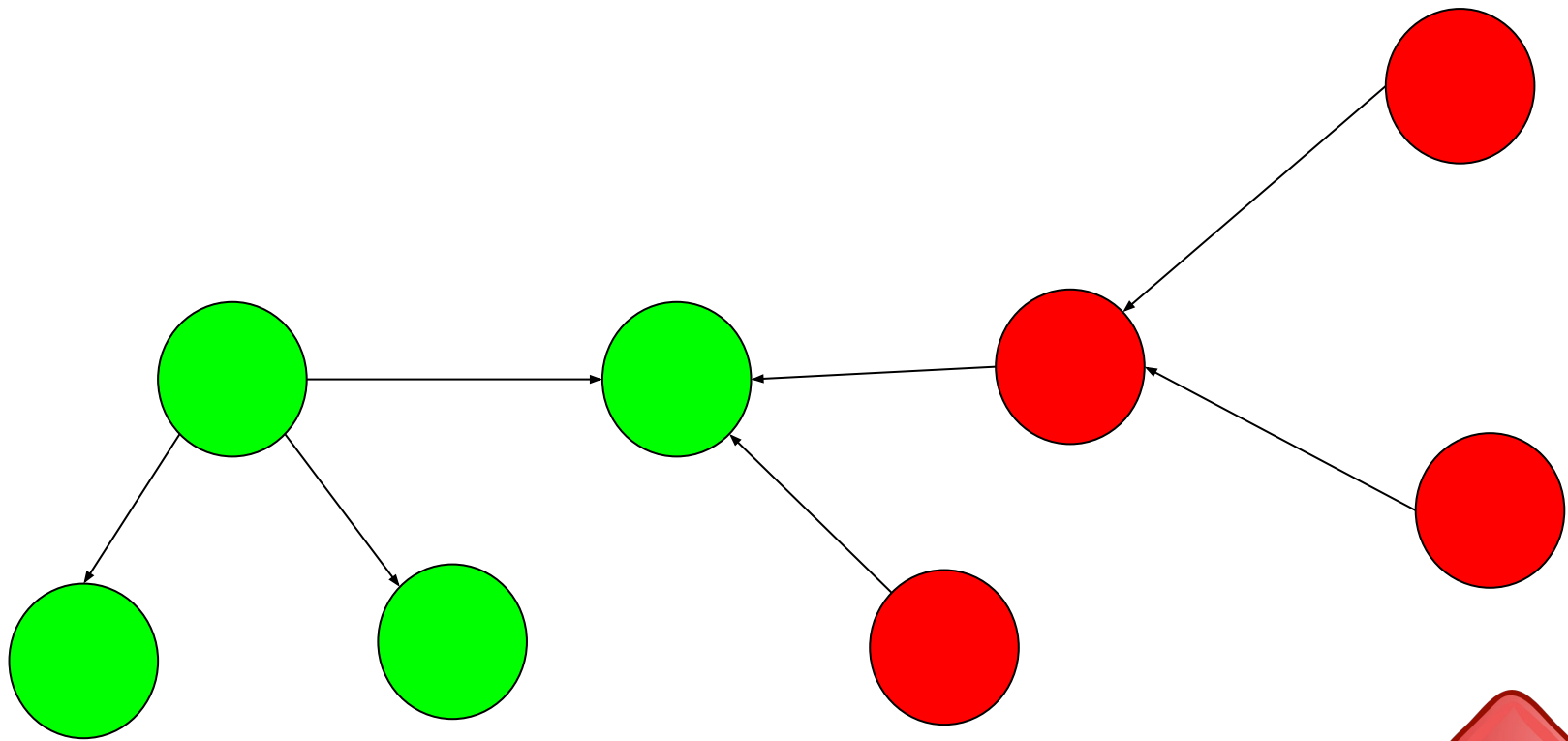


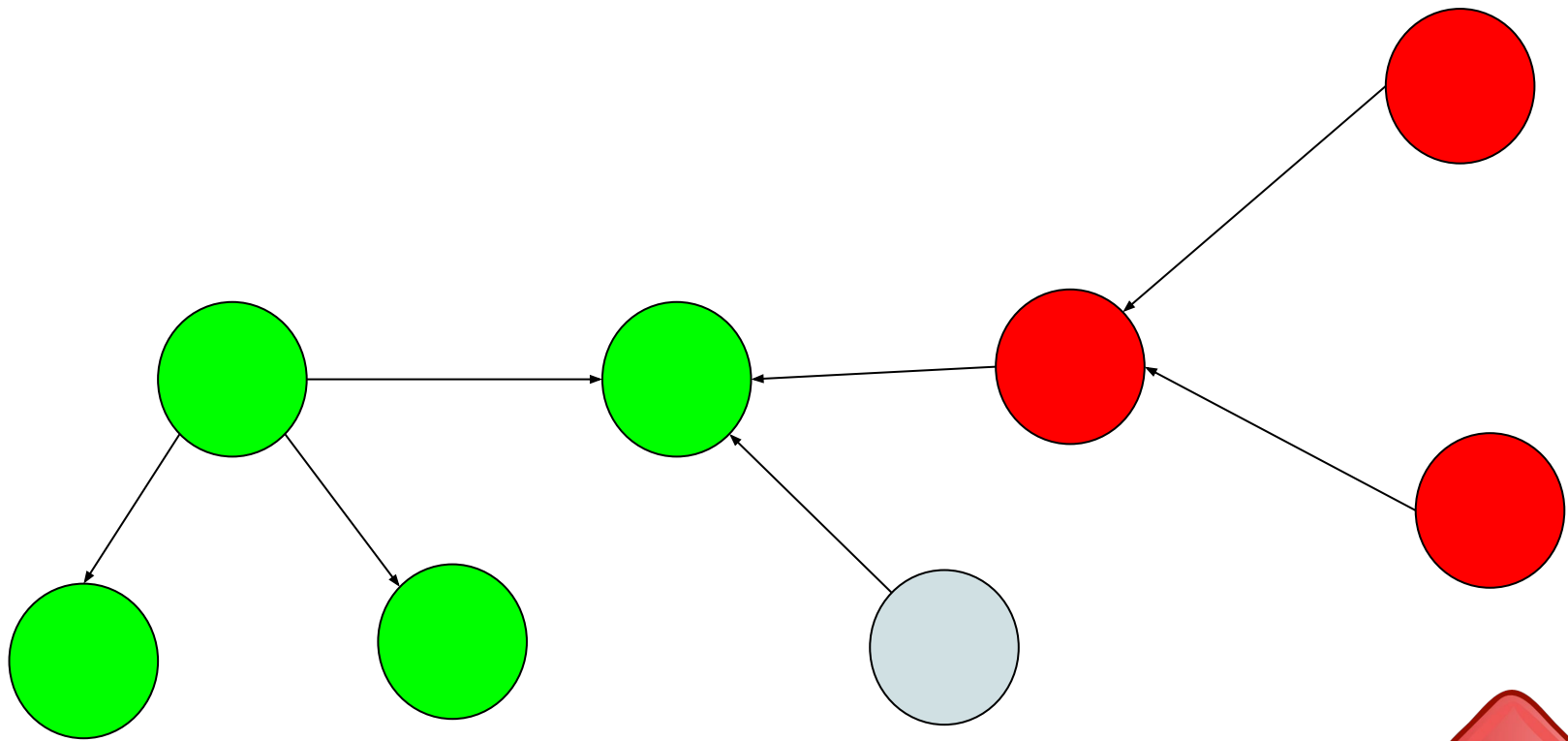


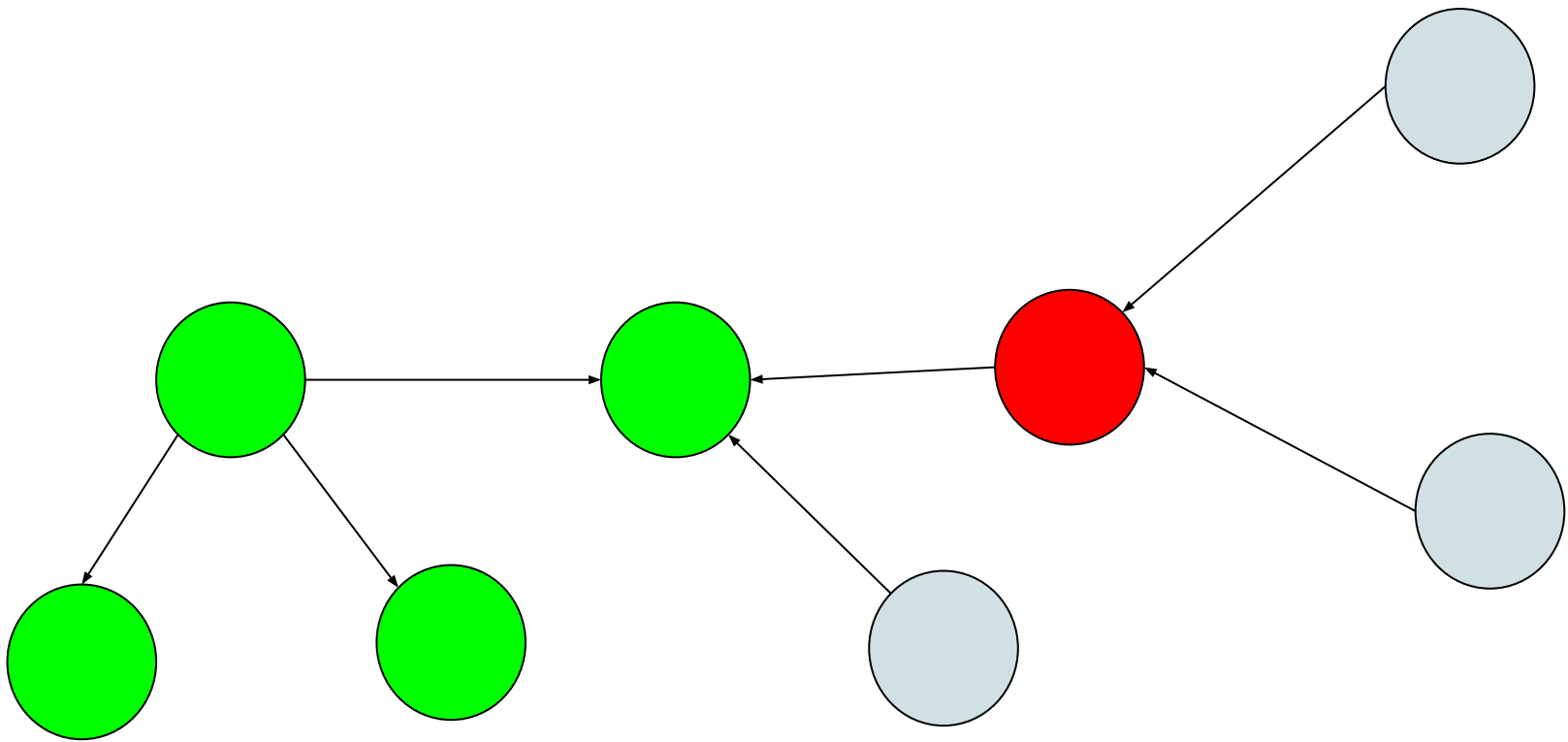


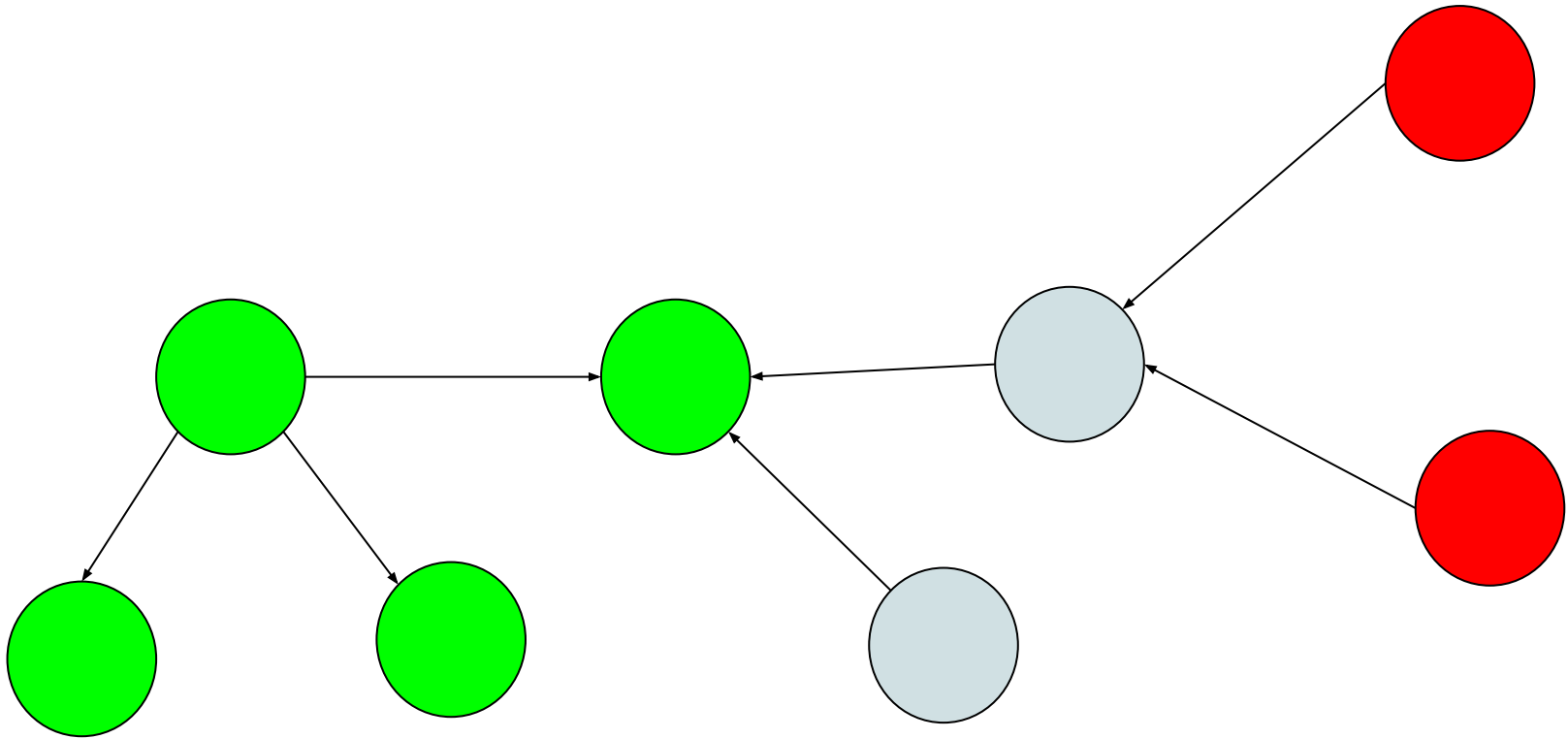
U

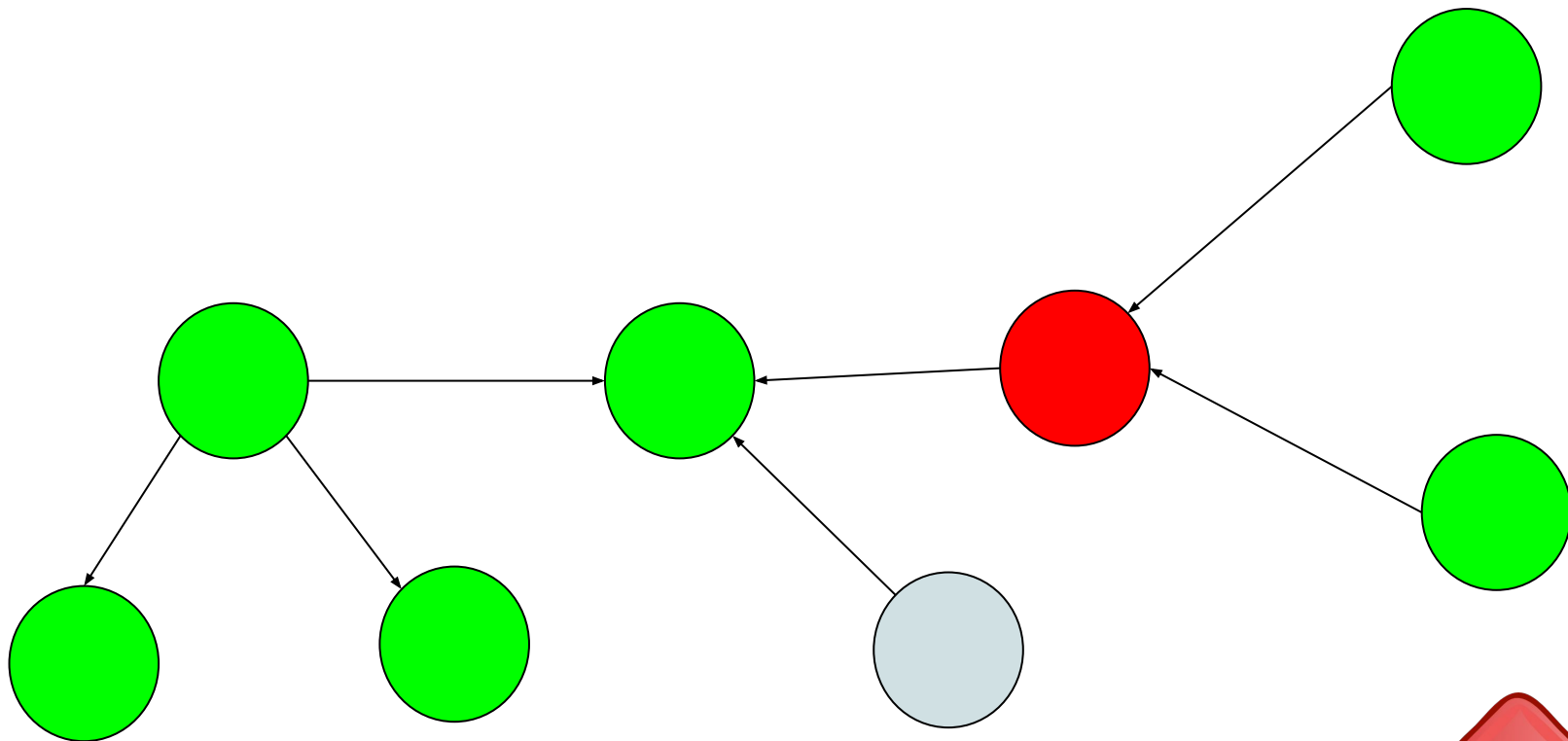




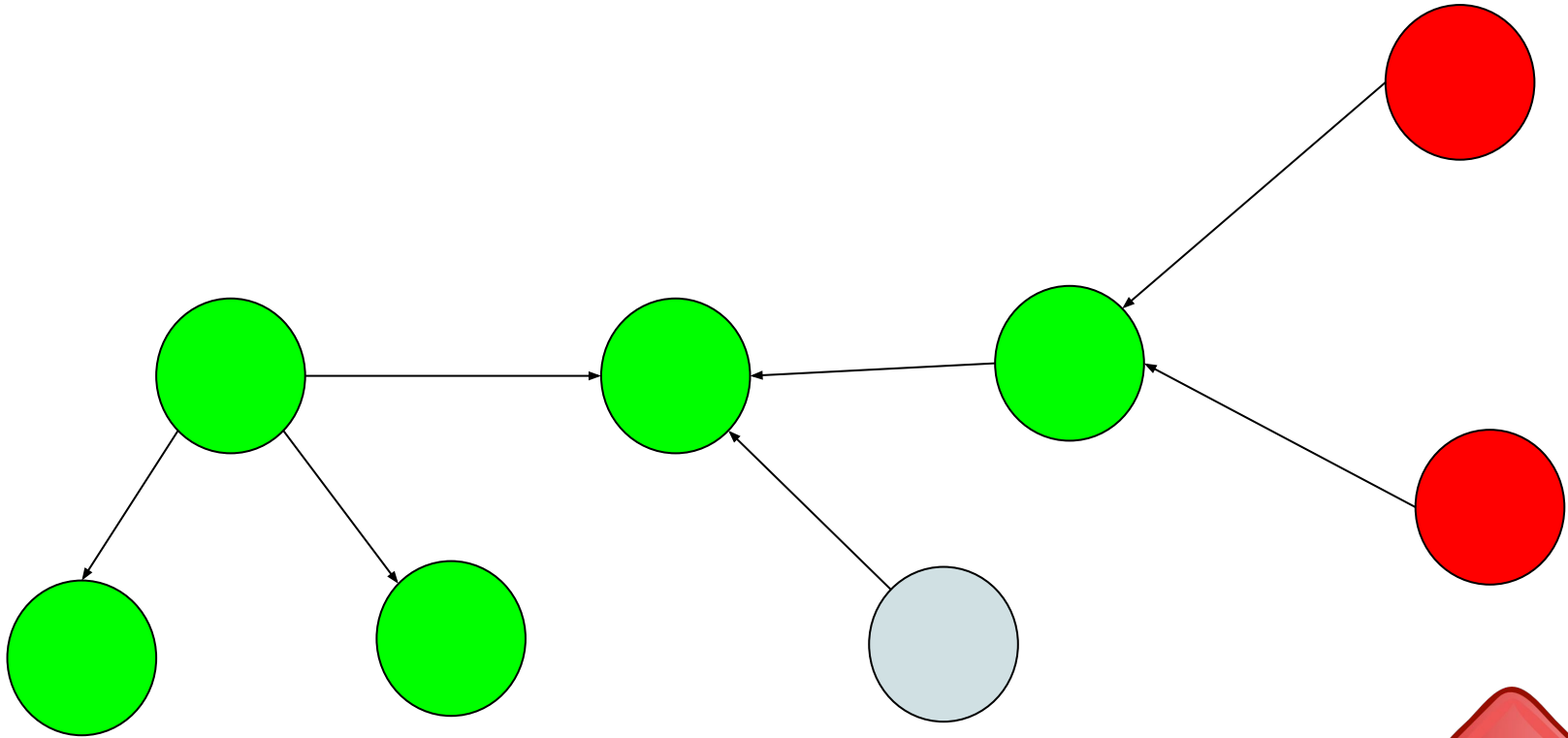


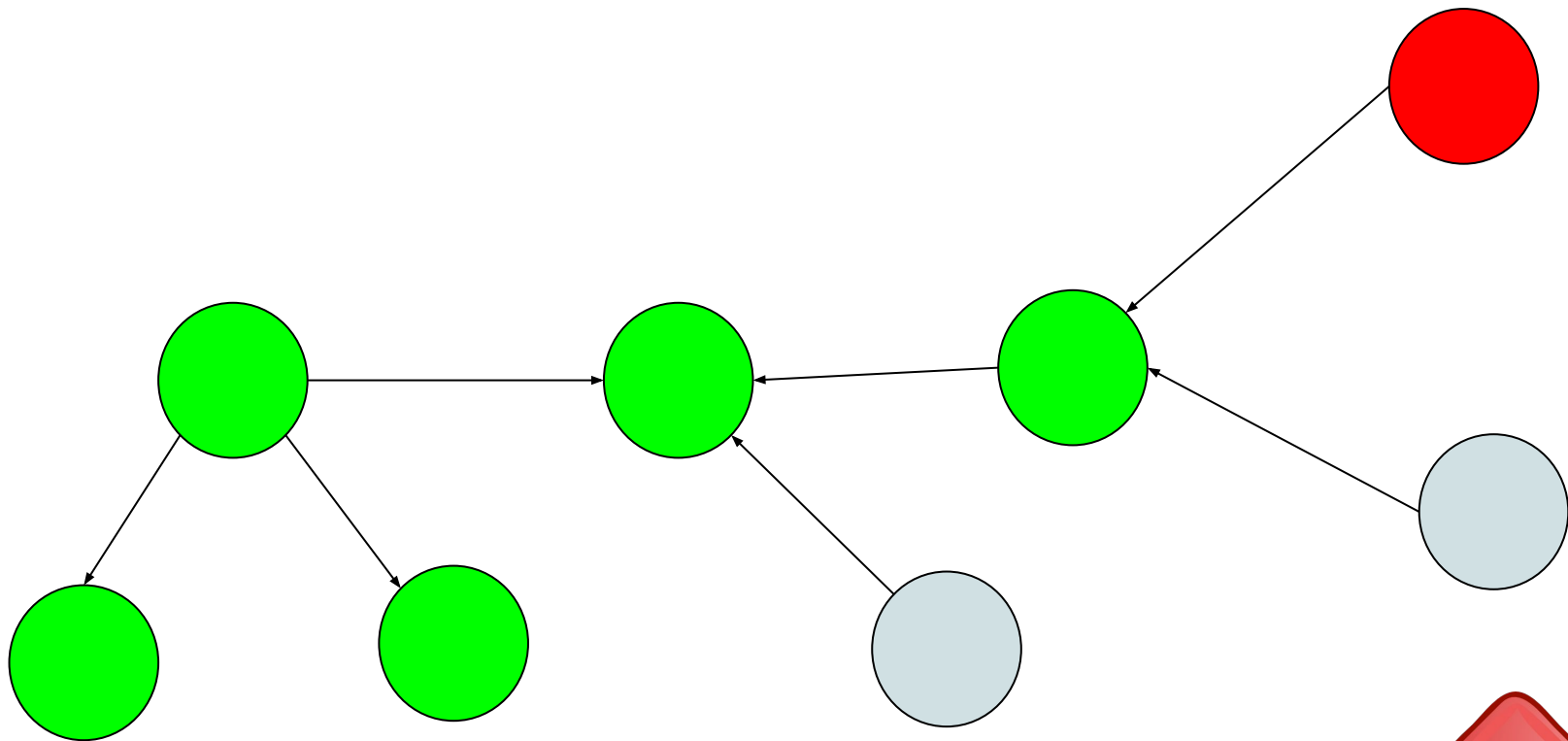


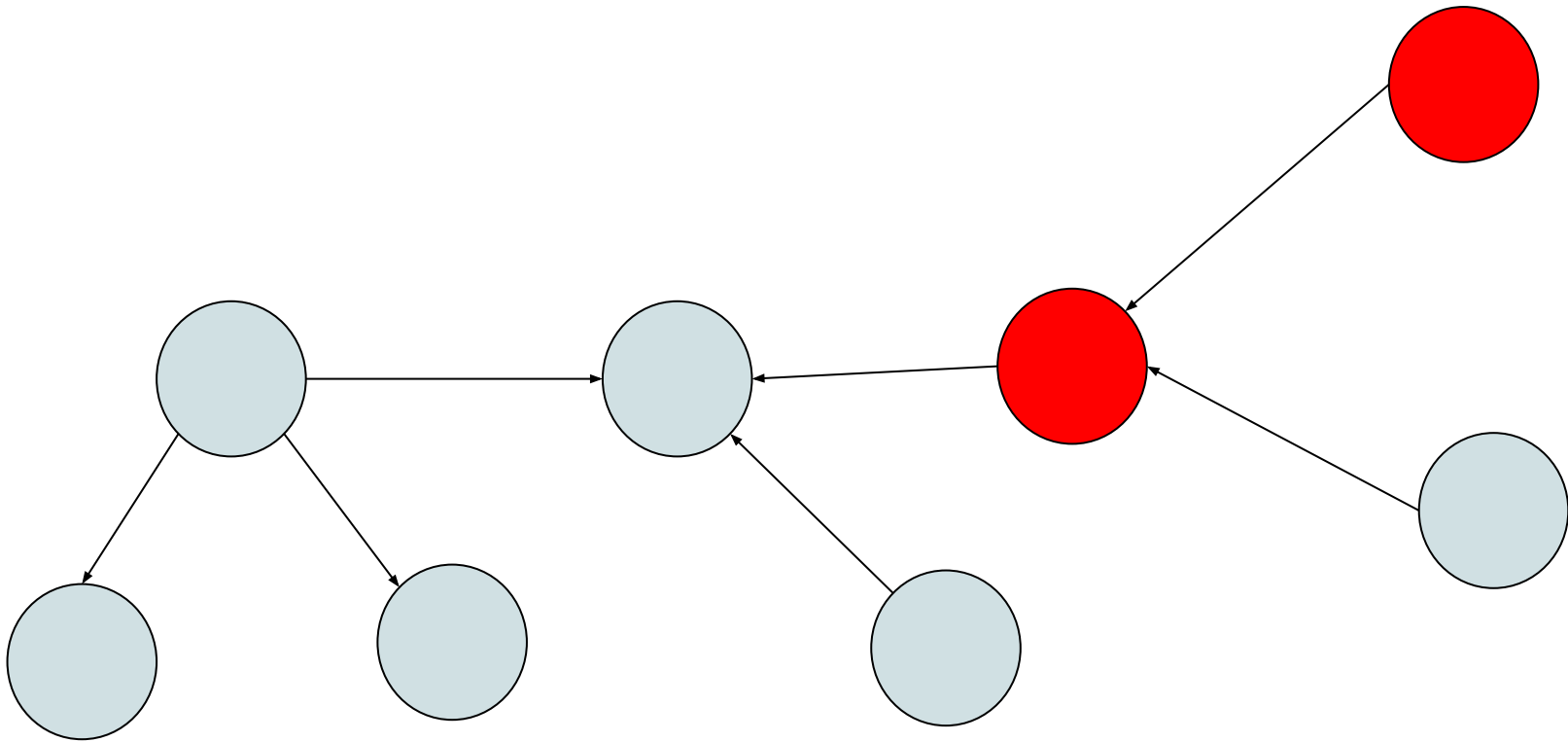


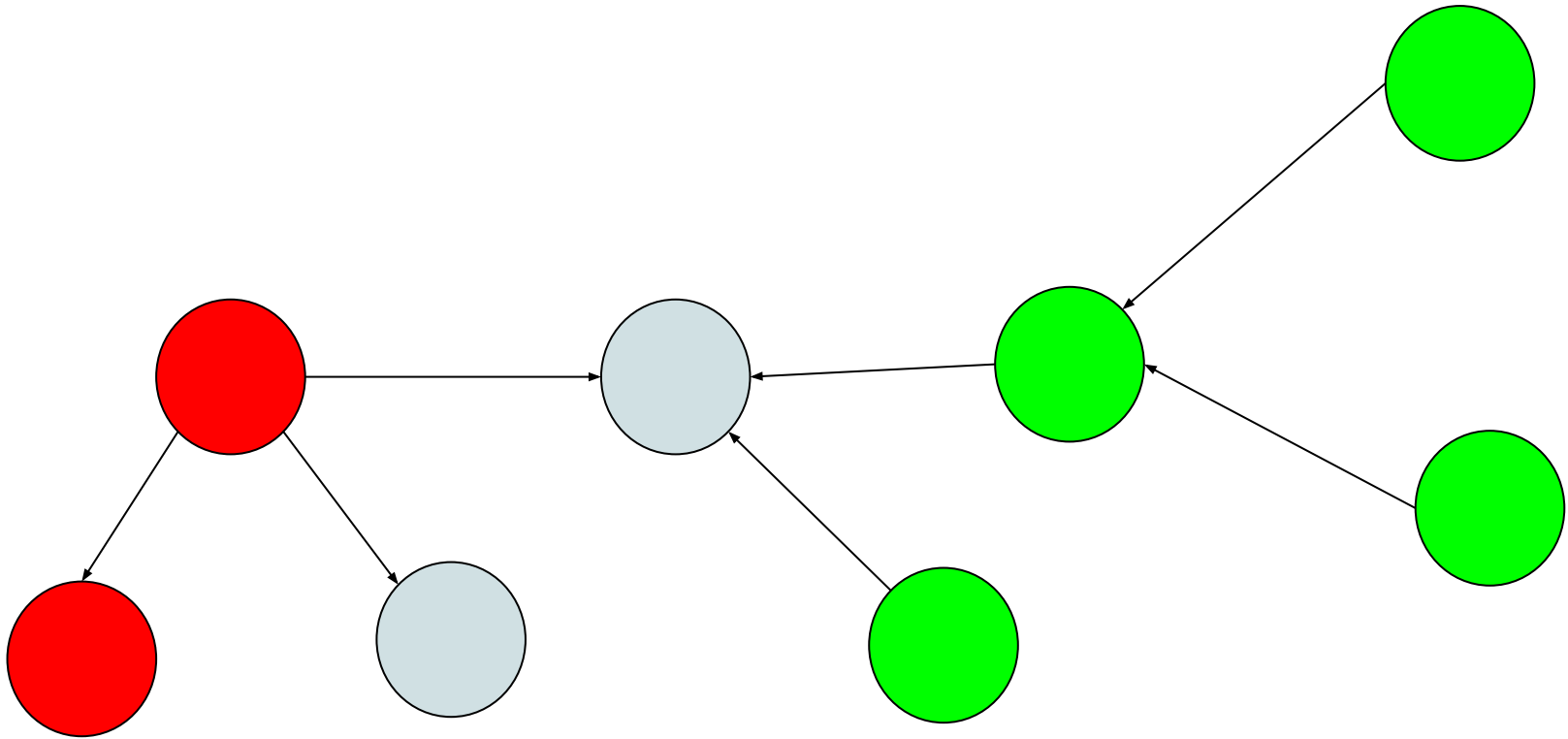


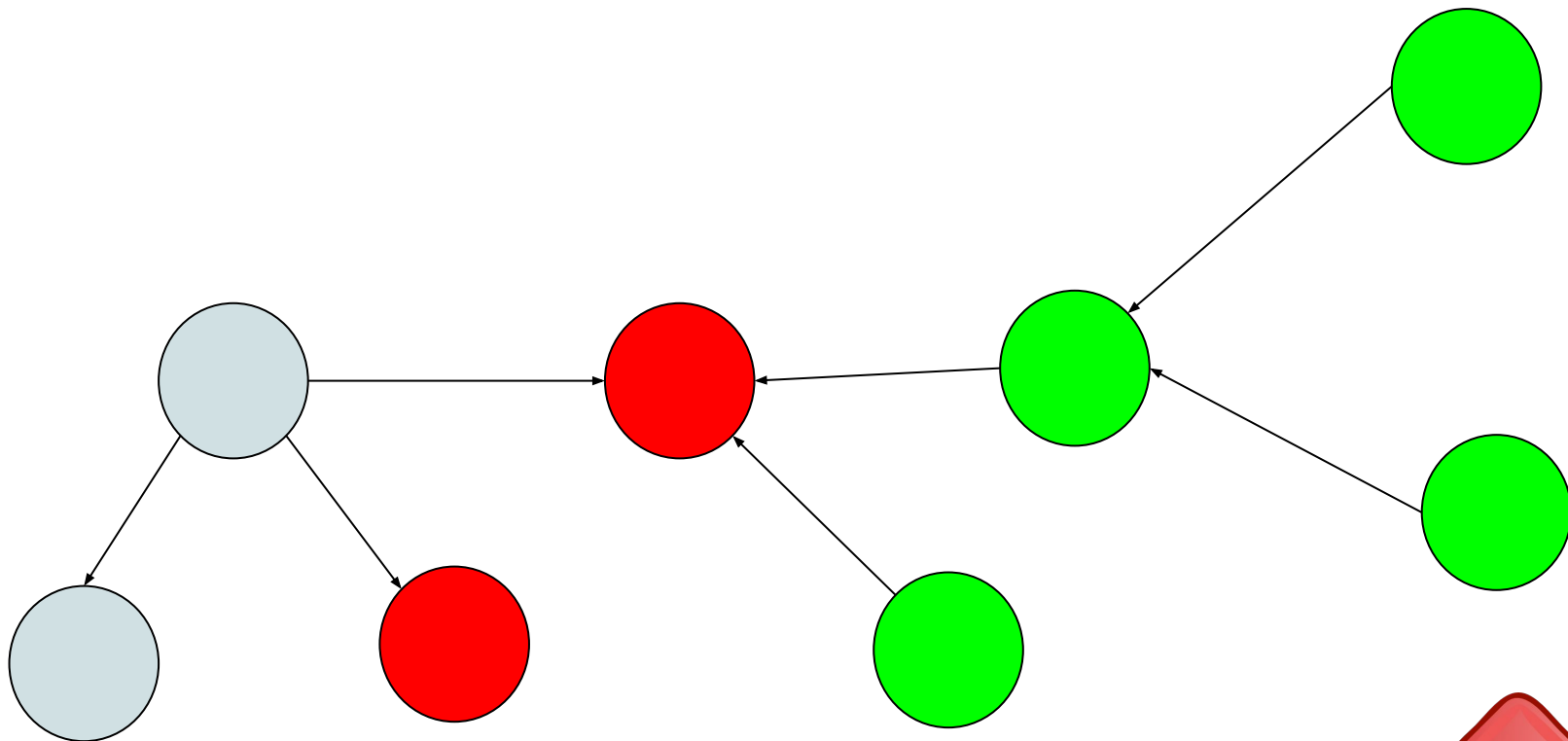


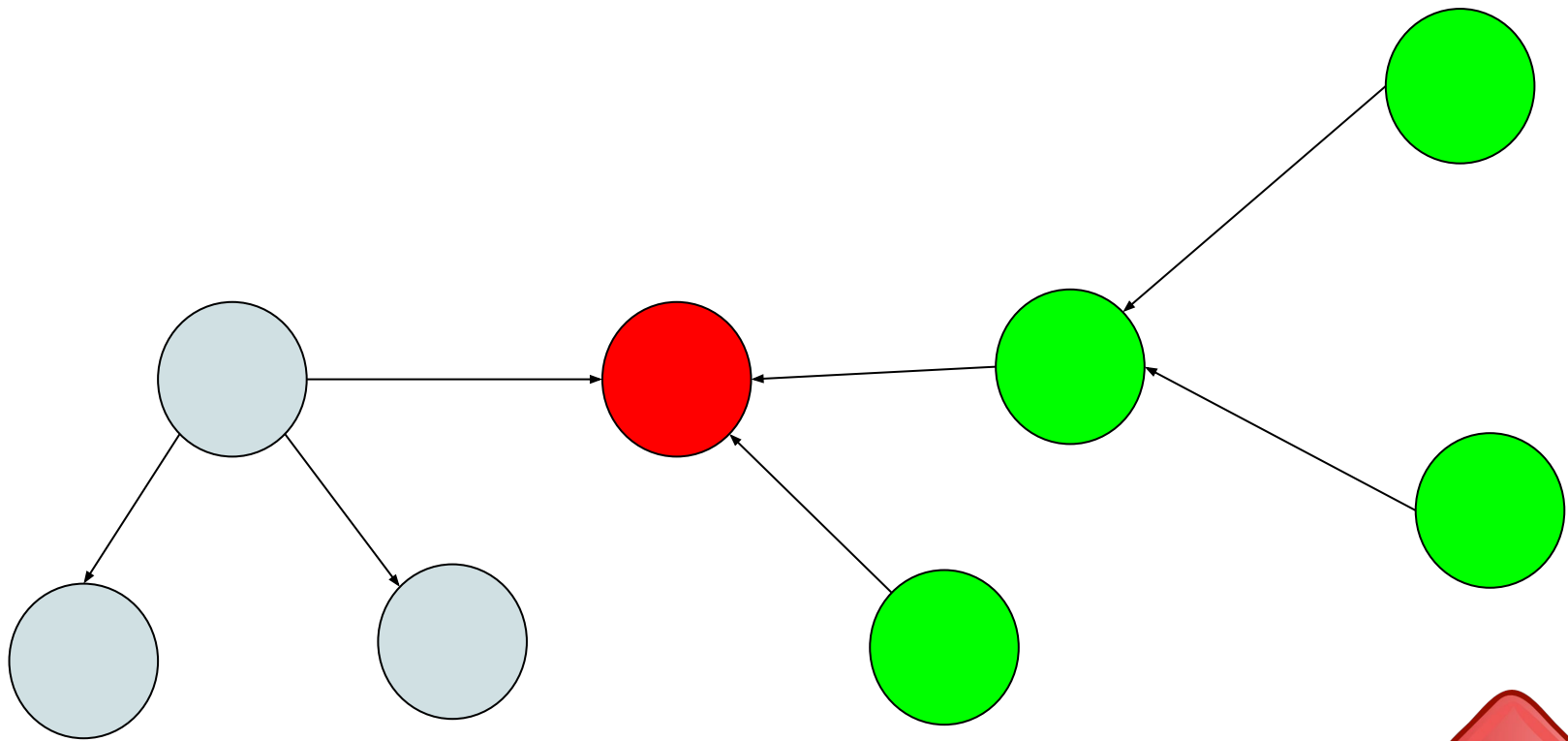


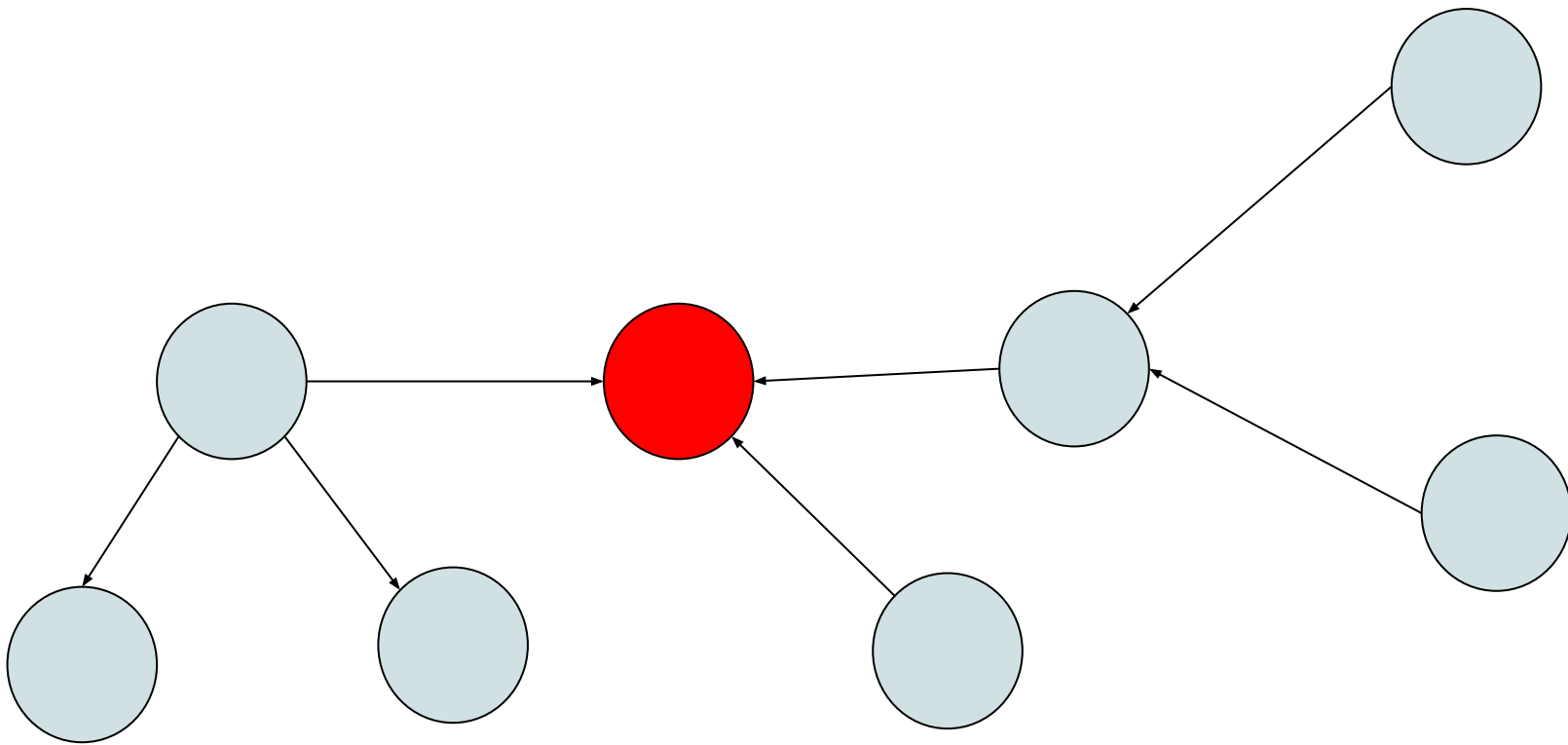


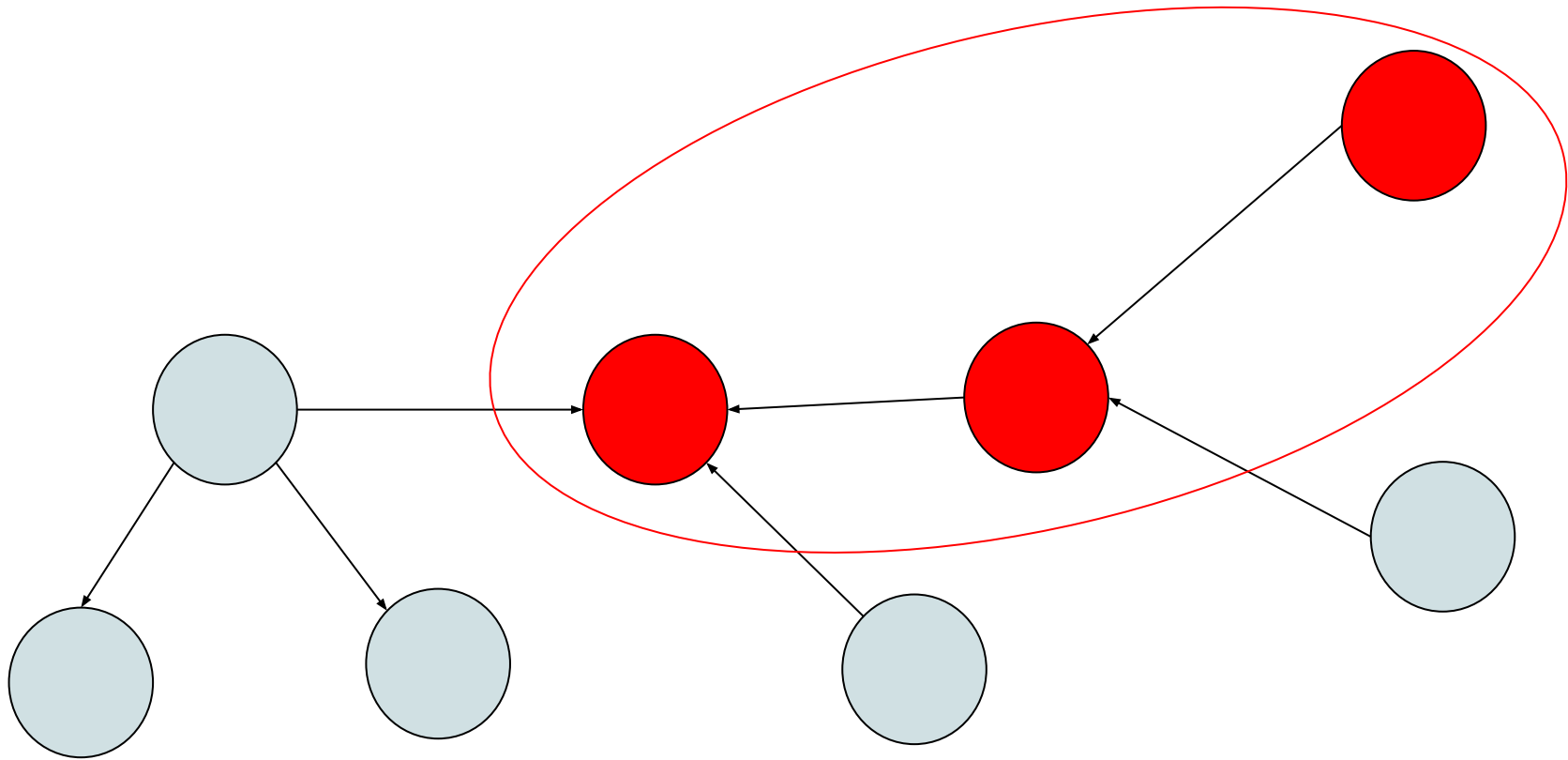














# Fault Localization Evaluation

---

# Fault Localization Accuracy

- Every location reported by the algorithm that is in the ground truth set is a true positive.
- Every location reported by the algorithm that is not is a false positive.
  
- We measure false positives relative to some tolerance level  $n$ : if an algorithm returns fewer than  $n$  spurious bug locations for a file, then the algorithm is not counted as having a false positive for that file.

# Fault Localization Accuracy

	Ground Truth	True Positives	False Positives (t=0)	False Positives (t=1)	False Positives (t=2)
<code>minimizeWCC</code>	seeded	18	25	14	9
Vanilla	seeded	8	20	7	1
<code>minimizeWCC</code>	expert	21	24	13	9
Vanilla	expert	10	19	6	1

# Fault Localization Efficiency

Algorithm	Min Time	Max Time	Avg Time
<code>minimizeWCC</code>	0.004 s	43.64 s	3.73 s

Runtimes do not necessarily correlate with program sizes: processed the largest benchmark, red black trees, at around 6 seconds.

These numbers indicate reasonable scaling for larger programs: the time taken relates to the relevant partitions of the constraint graph, not the program overall.

# Predicate Discovery

---

# Craig's Interpolation Theorem

- If  $A, B$  mutually inconsistent, i.e.,  $A \wedge B$  is unsatisfiable
- Exists an interpolant,  $I$
- such that  $\models A \rightarrow I$  and that  $I$  and  $B$  are mutually inconsistent
- where  $\text{atoms}(I) \subseteq \text{atoms}(A) \cap \text{atoms}(B)$ .

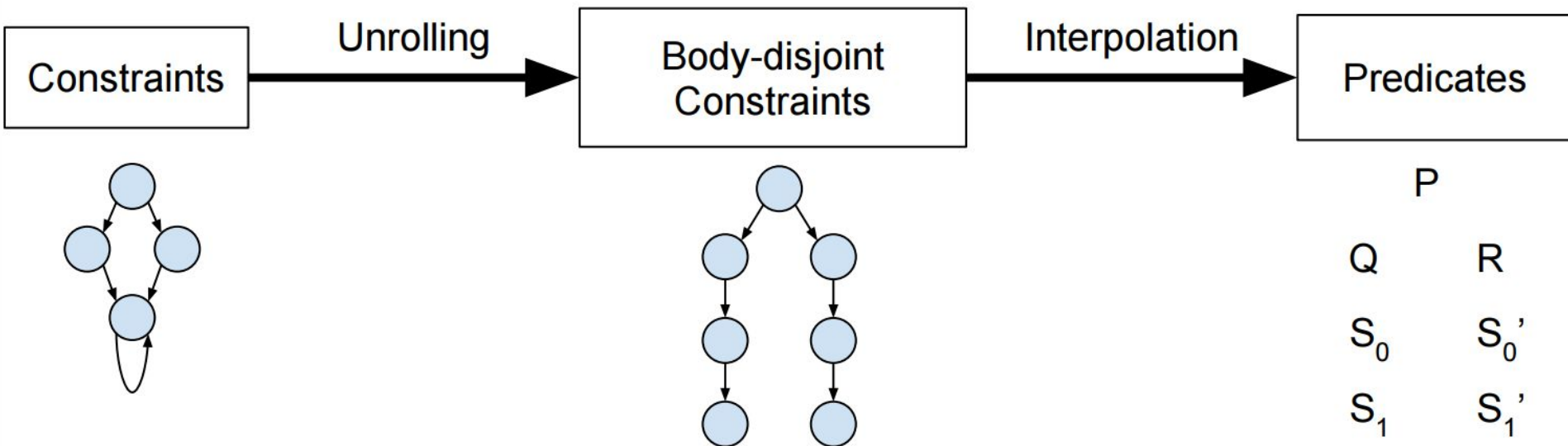
# Disjunctive Interpolation

- If  $A, B$  mutually inconsistent, i.e.,  $A \wedge B$  is unsatisfiable
- Exists an interpolant,  $I$
- such that  $\models A \rightarrow I$  and that  $I$  and  $B$  are mutually inconsistent
- where  $\text{atoms}(I) \subseteq \text{atoms}(A) \cap \text{atoms}(B)$ .

Rummer et al. generalize:

- Instead of pairs of formulas, one unsatisfiable formula
- Any number of labelled subformulas

# Predicate Discovery Overview





## Triangular numbers, again

```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0     = 0
6       | otherwise = let s = go (k-1) in s + k
```

# Constraints

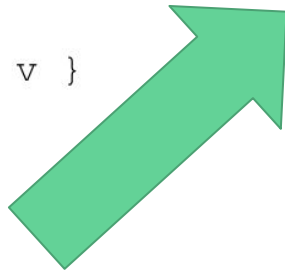
# Constraint Generation

$$k : \kappa_k, k \leq 0 \vdash \{\nu = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{\nu = s + k\} <: \kappa_s$$

$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{\nu \geq k\}$$

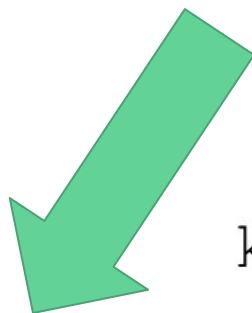


```
1 sum :: k:Int -> { v:Int | k <= v }
2 sum = go
3   where
4     go k
5       | k <= 0    = 0
6       | otherwise = let s = go (k-1) in s + k
```

# Horn Constraints

$$k : \kappa_k, k \leq 0 \vdash \{\nu = 0\} <: \kappa_s$$

$$k : \kappa_k, \neg(k \leq 0), s : \kappa_s[k/k - 1] \vdash \{\nu = s + k\} <: \kappa_s$$



$$\emptyset \vdash \text{Int} <: \kappa_k$$

$$k : \kappa_k \vdash \kappa_s <: \{\nu \geq k\}$$

$$\kappa_k(k) \wedge k \leq 0 \wedge \nu = 0 \rightarrow \kappa_s(\nu)$$

$$\kappa_k(k) \wedge k > 0 \wedge \kappa_s(s)[k/k - 1] \wedge \nu = s + k \rightarrow \kappa_s(\nu)$$

$$\text{true} \rightarrow \kappa_k(\nu)$$

$$\kappa_s(\nu) \rightarrow \nu \geq k$$

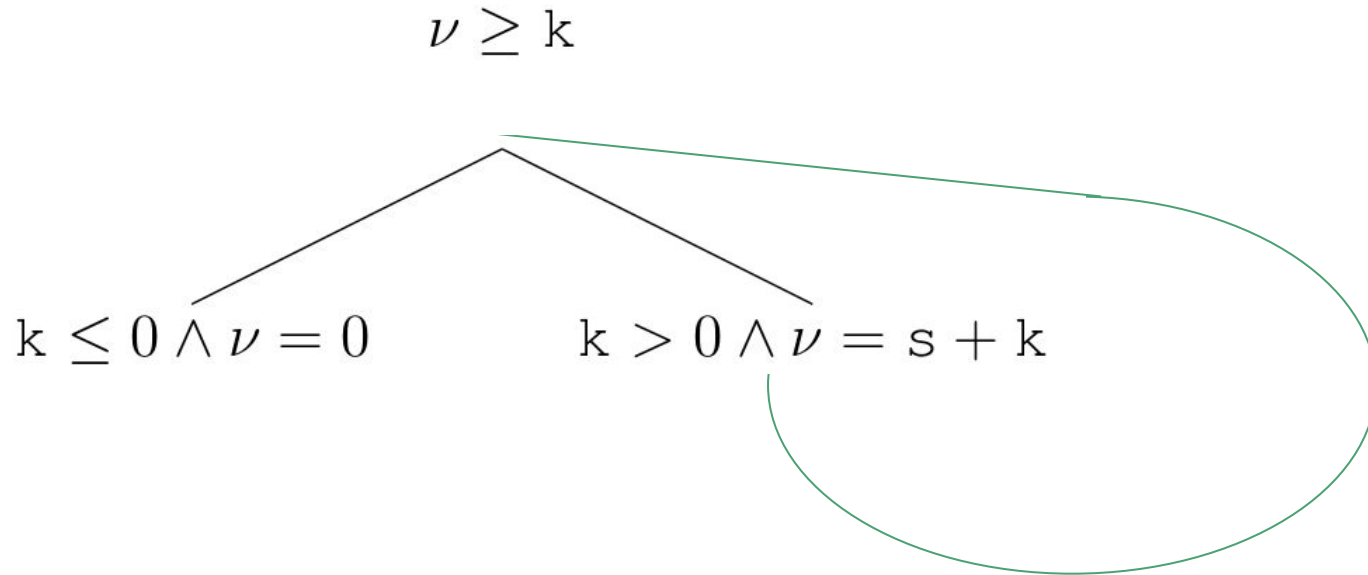
# Horn Constraints

$$\kappa_k(k) \wedge k \leq 0 \wedge \nu = 0 \rightarrow \kappa_s(\nu)$$

$$\kappa_k(k) \wedge k > 0 \wedge \kappa_s(s)[k/k - 1] \wedge \nu = s + k \rightarrow \kappa_s(\nu)$$

$$true \rightarrow \kappa_k(\nu)$$

$$\kappa_s(\nu) \rightarrow \nu \geq k$$



# Horn Constraints

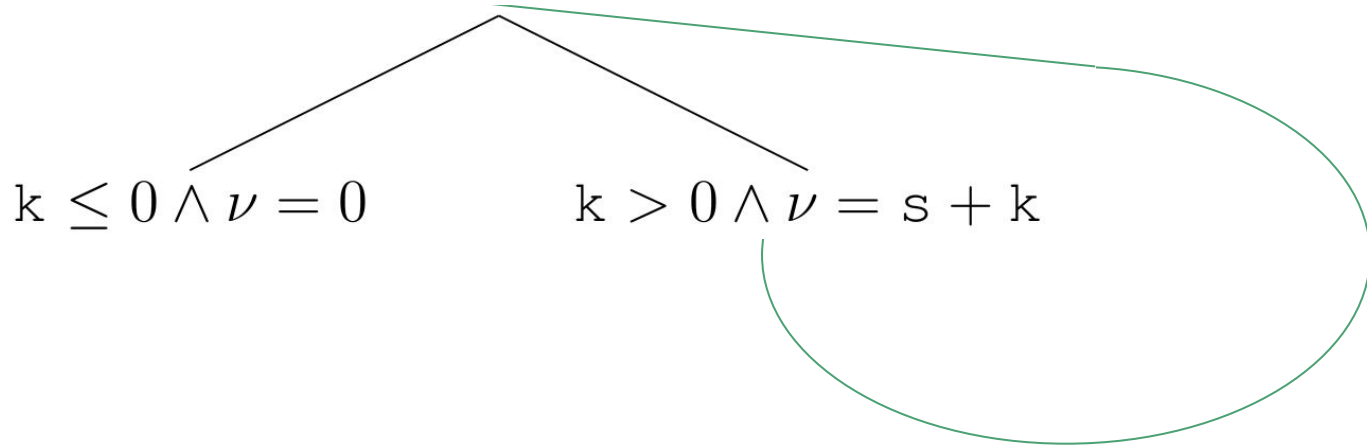
$$\kappa_k(k) \wedge k \leq 0 \wedge \nu = 0 \rightarrow \kappa_s(\nu)$$

$$\kappa_k(k) \wedge k > 0 \wedge \kappa_s(s)[k/k - 1] \wedge \nu = s + k \rightarrow \kappa_s(\nu)$$

$$true \rightarrow \kappa_k(\nu)$$

$$\kappa_s(\nu) \rightarrow \nu \geq k$$

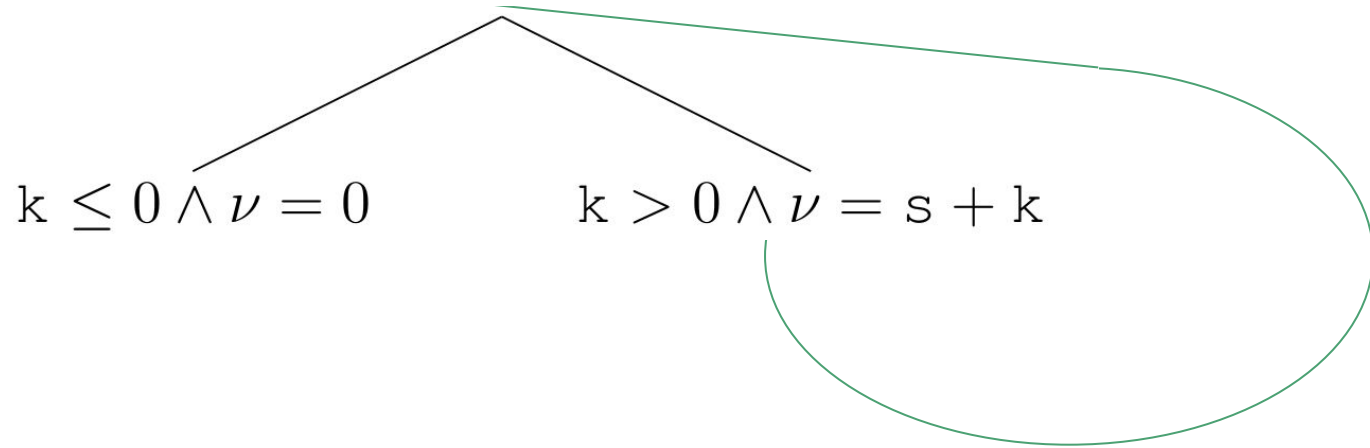
$$\nu < k = \neg(\nu \geq k)$$



# Unrolling Constraints

# Unrolling Constraints

$$\nu < k$$





# Unrolling Constraints

$$\nu < k$$

$$k \leq 0 \wedge \nu = 0$$

$$k > 0 \wedge \nu = s + k$$

$$k - 1 \leq 0$$

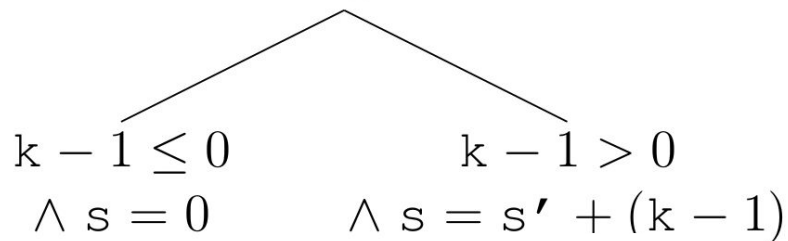
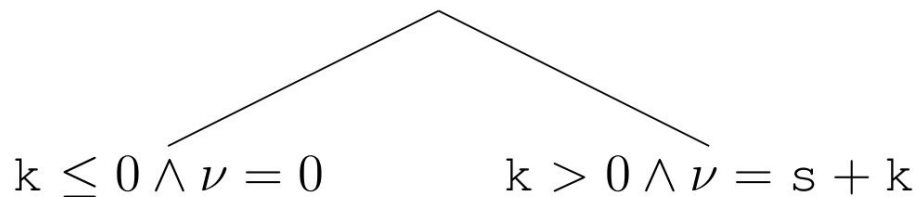
$$\wedge s = 0$$

$$k - 1 > 0$$

$$\wedge s = s' + (k - 1)$$

# Unrolling Constraints

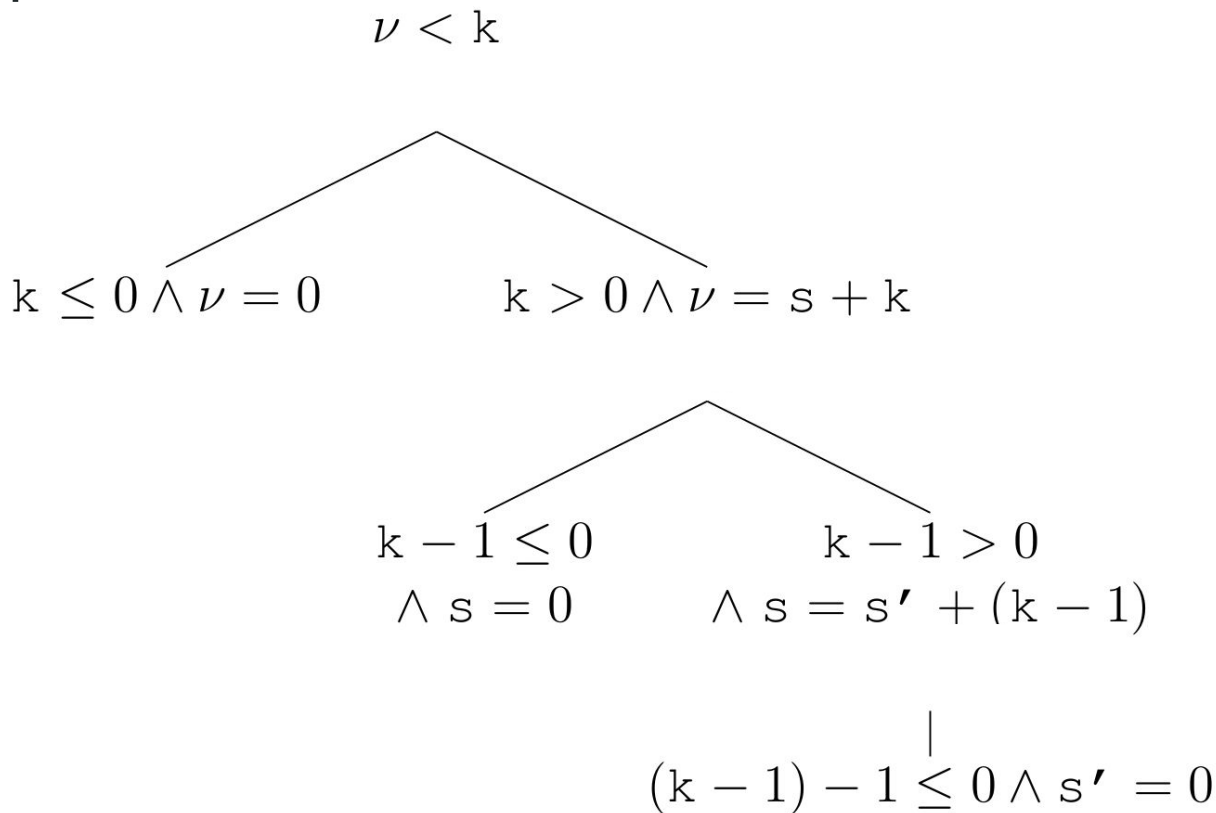
$$\nu < k$$



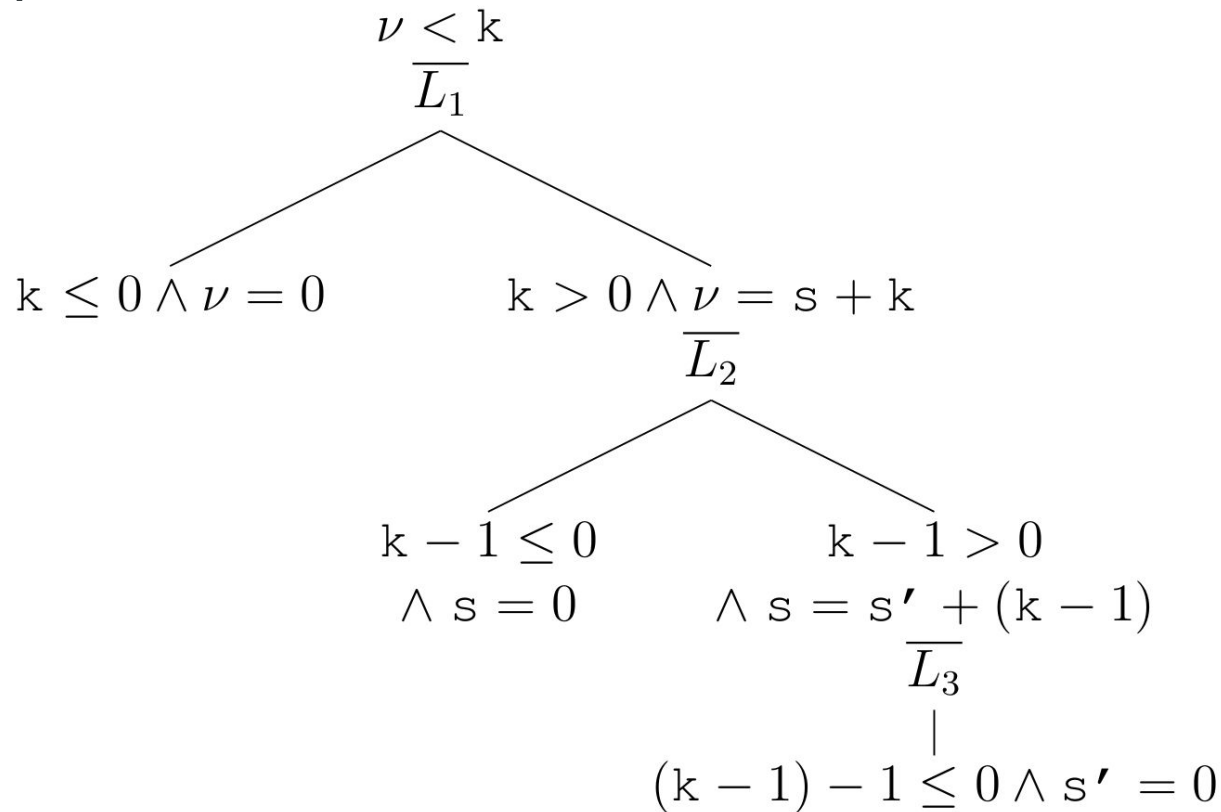
$$(k - 1) - 1 \stackrel{|}{\leq} 0 \wedge s' = 0$$

# Interpolation

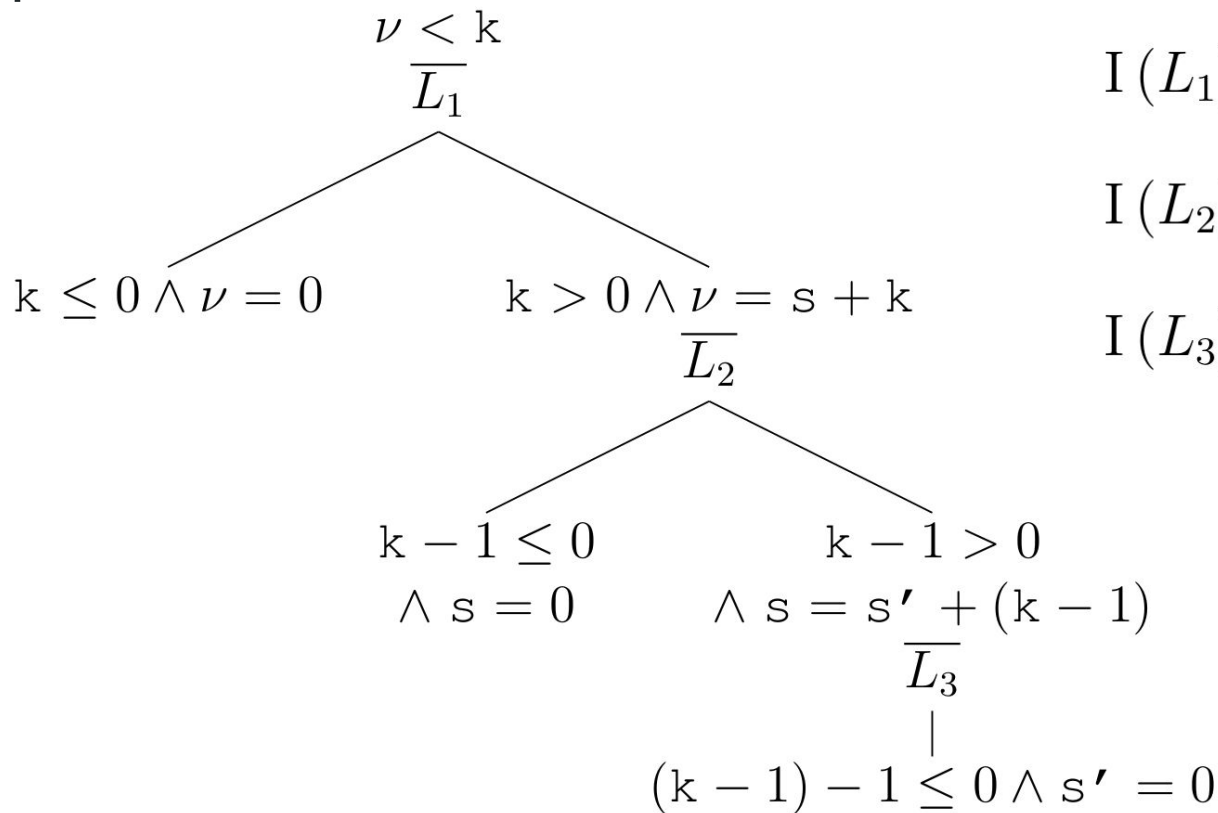
# Interpolation



# Interpolation



# Interpolation



$$I(L_1) = \nu \geq k \wedge k \leq 0$$

$$I(L_2) = s \geq k - 1$$

$$I(L_3) = s' = 0$$

# Predicate Discovery Evaluation

---

# Predicate Discovery on k-bounded Programs

- Considered 21 microbenchmarks.
- All 21 successful
  - including instances where Liquid Haskell alone would not have been able to verify the program.
- Median runtime of 0.1 seconds and a maximum of 7.1 seconds.



# Predicate Discovery in General

- 23 microbenchmarks
  - programs that were not k-bounded for any
  - programs that were k-bounded, but for which we only permitted our algorithm  $i < k$  unrolling
  - Unroll depth of 2
- In 22 of 23 cases, enough predicates to prove correctness
- The one failing case involved a constraint that was not k-bounded for any k in an implementation of merge sort; in this case the domain expansion was missing only one qualifier (out of  $\sim 100$ ).
- Our algorithm took a median of 0.1 seconds and a maximum of 7.2seconds.

# Conclusion

---

# Conclusion

Our fault localization algorithm produces minimal false positives (almost half of which are a single spurious location) and is efficient enough to be used at compile-time. It is much more effective at fault localization than the Liquid Haskell type checker, localizing twice as many bugs overall and finding six times more “hard” bugs than the type checker.

Our predicate discovery algorithm is correct by construction on  $k$ -bounded instances, finding annotations that admit program verification. Together, our two algorithms significantly reduce the barrier to entry for using refinement types systems.

Questions?

# Extra Slides

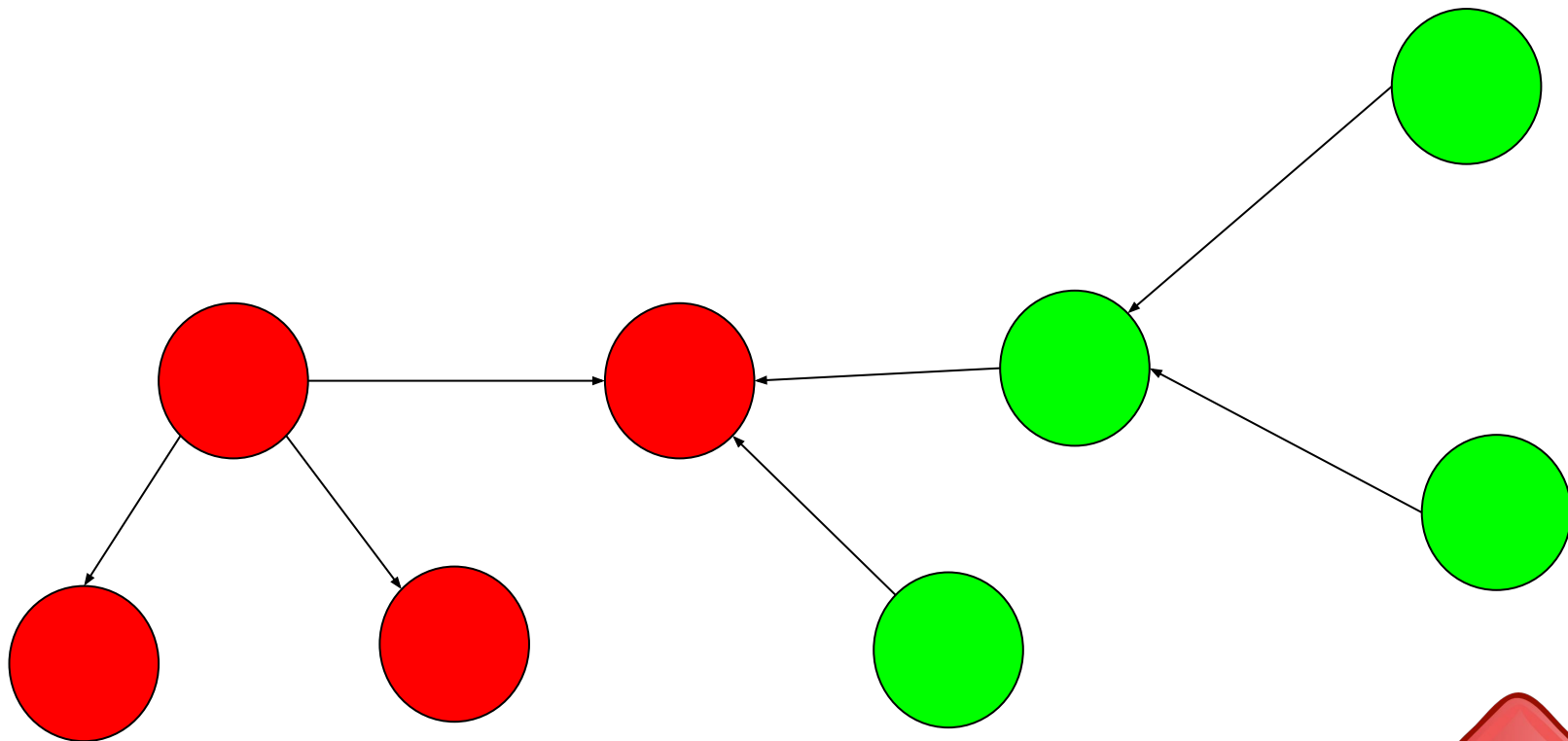
---

# Key Insights

1. A bug captured at the type checking level can be seen as an inconsistency.
2. The locations reported to the user should be minimal to prevent implicating spurious program locations as faults.
3. Minimal explanations implicate relevant locations.

## Central Insight

We can use the structure of type constraints to better localize faults and to fix incomplete specifications





## Related Work

The SEMINAL tool by Lerner et al. [20] uses the OCaml type checker as an oracle in a search procedure to find well-typed programs that are syntactically similar to an input program that fails to type check, which are then used to construct helpful error messages. Our fault localization algorithm likewise uses the type checker as an oracle, but works on the set of constraints generated from the input program instead. Since the space of constraints is much smaller than the space of possible edits for a program, our algorithm can be more efficient than the SEMINAL tool without sacrificing the ability to find bugs.

Pavlinovic et al. [24] reduce fault localization into an instance of the MaxSAT problem by generating a set of assertions from the input program that are weighted by a ranking measure provided by the compiler. An SMT solver is used to find a minimum set of error clauses that can be mapped back to possible bug locations.

# Thesis

We can use the **structure of type constraints** to better localize faults and to fix incomplete specifications

# Thesis

We can use the structure of type constraints to better **localize faults** and to fix incomplete specifications

# Thesis

We can use the structure of type constraints to better localize faults and to **fix** incomplete **specifications**

# Bugs are Bad

- Bugs are Expensive
  - Modifying code, correcting defects, and evolving code account for as much as 90% of the total cost of software projects.



# Bugs are Bad

- Bugs are Expensive
  - Modifying code, correcting defects, and evolving code account for as much as 90% of the total cost of software projects.



# Predicates

## Quantifier-Free Logic of Uninterpreted Functions & Linear Arithmetic

```
e := x, y, z, ...           -- variables
   | 0, 1, 2, ...          -- constants
   | e + e | c * e | ...    -- arithmetic
   | f e1 ... en           -- uninterpreted function
```

# Find the bug

This code doesn't compile. At a glance, can you find why?

(actual code written by a student)

```
1 randomElt :: Vector a -> Rand StdGen (Maybe a)
2 randomElt vector = case (V.length vector) of
3     0 -> liftM (\_ -> Nothing) (getRandom)
4     1 -> liftM (vector !?) (getRandomR (0, (1-1)))
```



# Find the "bug"

This code doesn't compile. At a glance, can you find why?

(actual code written by a student)

```
1 randomElt :: Vector a -> Rand StdGen (Maybe a)
2 randomElt vector = case (V.length vector) of
3   0 -> liftM (\_ -> Nothing) (getRandom :: Int)
4   1 -> liftM (vector !?) (getRandomR (0, (1-1)))
```