# Formally verified incremental cycle detection

Armaël Guéneau

with J.-H. Jourdan, A. Charguéraud and F. Pottier

# In this talk

The story of a formally verified algorithm...

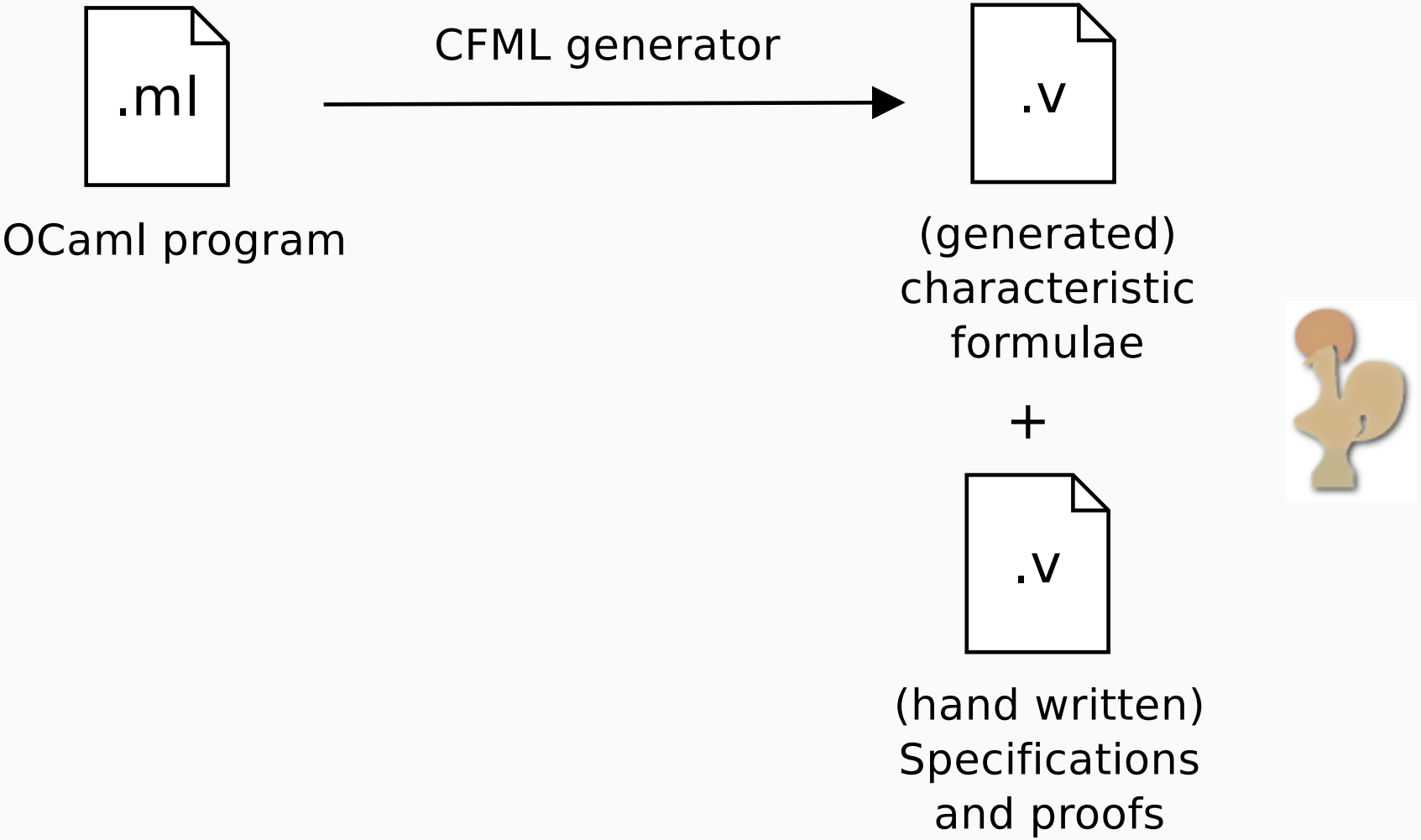
- initially motivated by Coq's implementation...
  (universe constraints)
- ...which ended up integrated into the Dune build system.
  (dependencies between build actions)

We verify at the same time *correctness and complexity*...

... and our implementation turns out to be 7x faster!

# Context

# Program verification framework: Coq and (extended) CFML



# Separation Logic with Time Credits

- Each function call (or loop iteration) consumes $\$1$
- $\$n$ asserts the ownership of $n$ time credits
- $\$(n+m) = \$n * \$m$
- Credits are not duplicable: $\$1 \not\Longrightarrow \$1 * \$1$
- Enables amortized analysis

Type of assertions, in the model:

- Standard Separation Logic: Heap $\to$ Prop
- Separation Logic with Time Credits: Heap $\times \mathbb{N} \to$ Prop

# Example specifications using time credits

Complexity specification using explicit time credits:

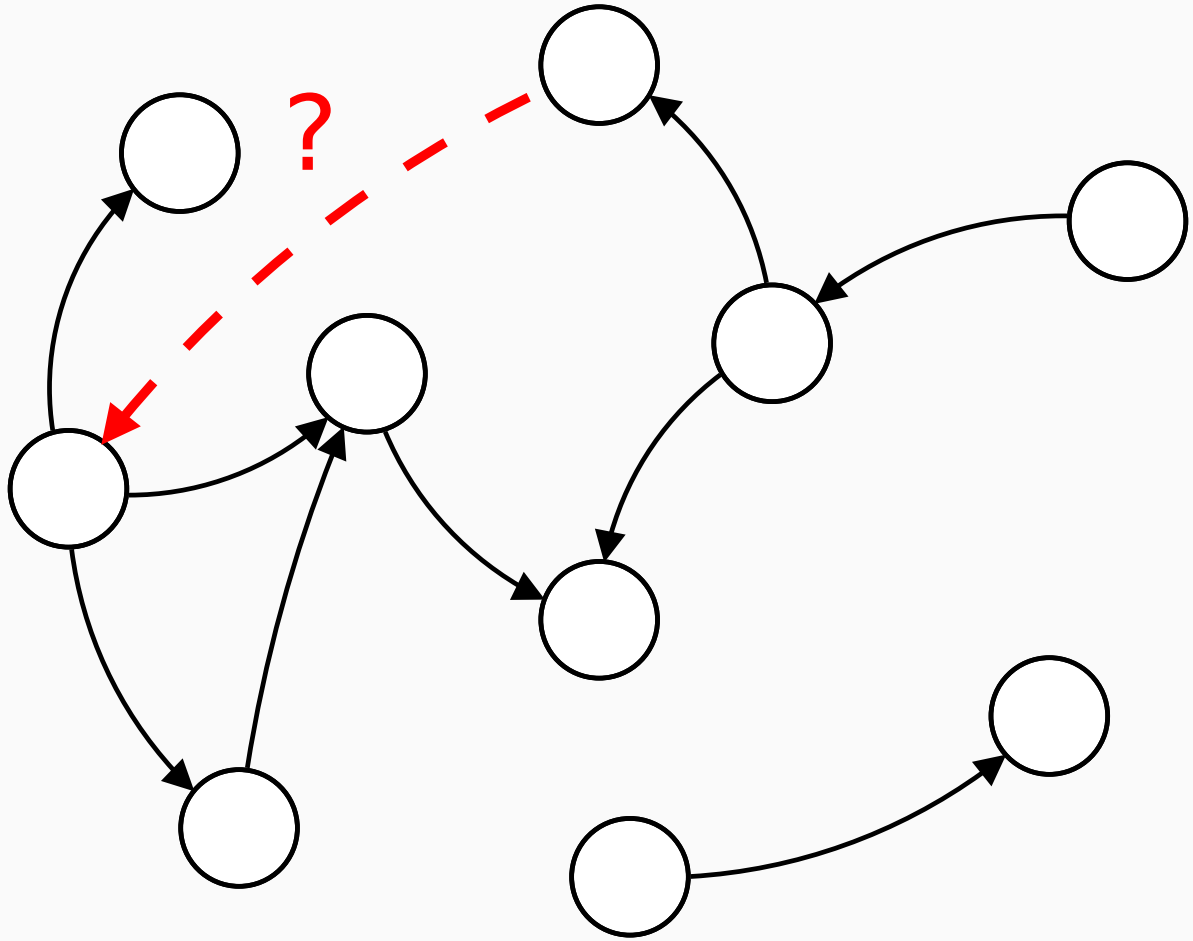
$$\forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\ G * \$(3\,|\mathrm{edges}\ G| + 5)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}$$

Asymptotic complexity specification:

$$\begin{aligned}&\exists(f : \mathbb{Z} \to \mathbb{Z}).\\ &\qquad \mathrm{nonnegative}\ f\ \ \wedge\ \ \mathrm{monotonic}\ f\ \ \wedge\ \ f \in O_{\mathbb{Z}}(\lambda m.m)\\ &\quad \wedge\ \forall\, g\, G.\ \ \{\, \mathrm{IsGraph}\, g\ G * \$\, f(|\mathrm{edges}\ G|)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}\end{aligned}$$

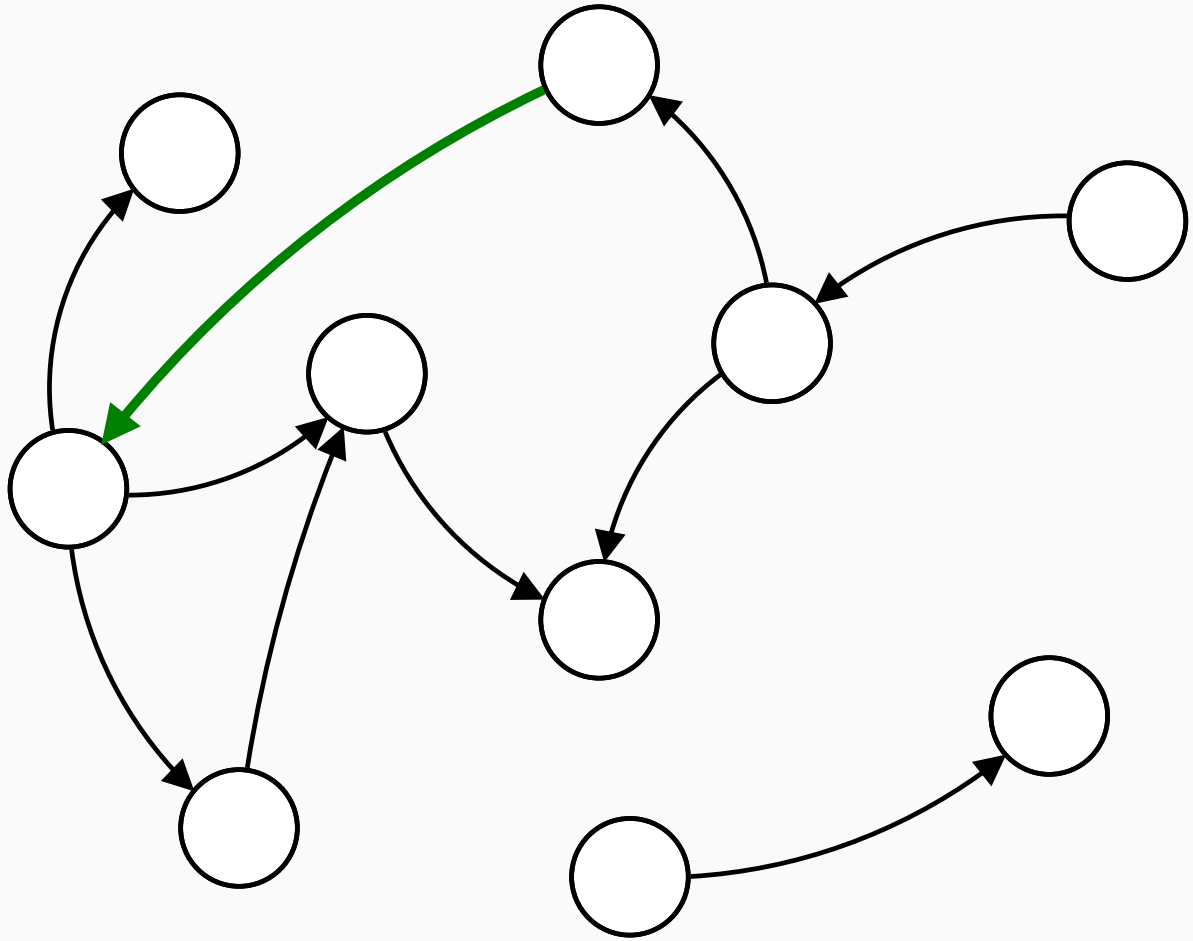
# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



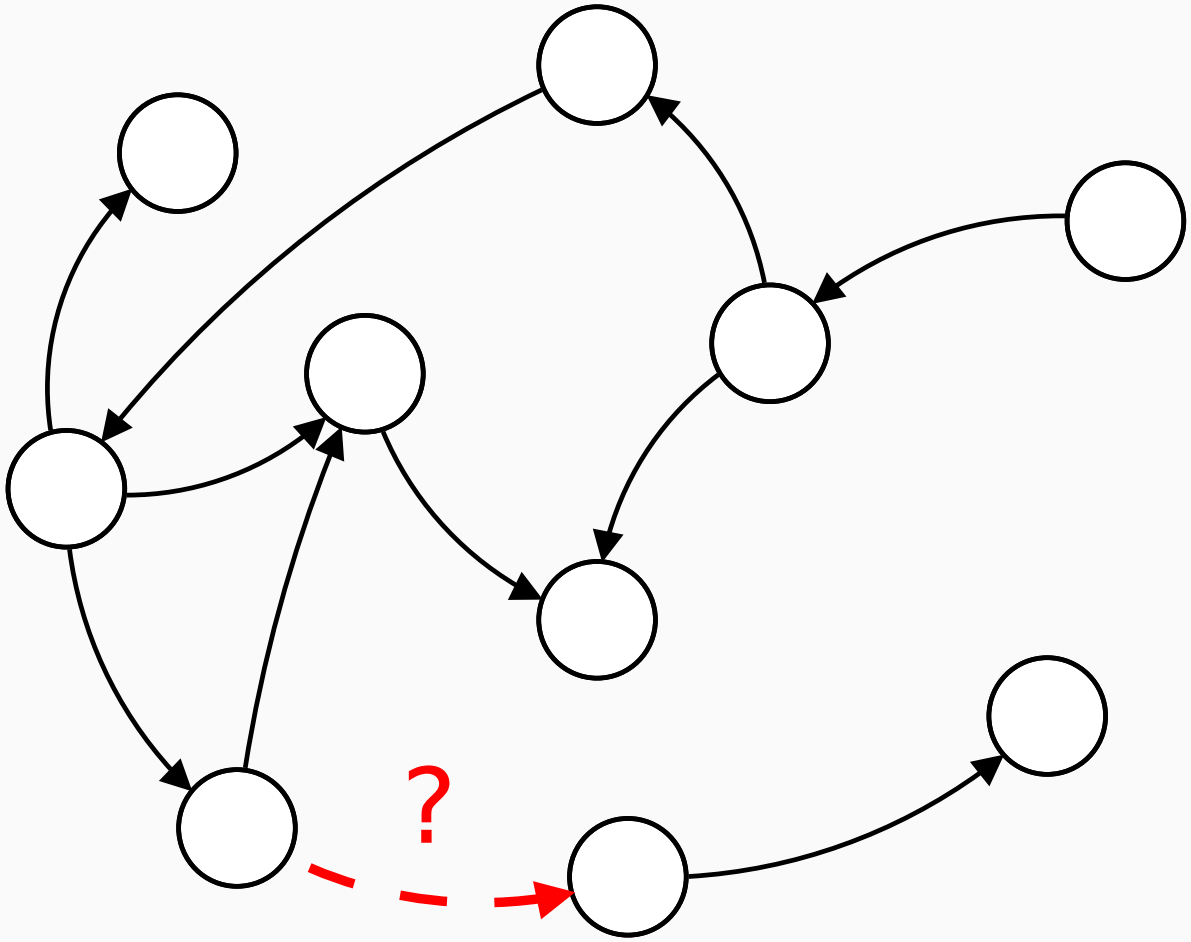
# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



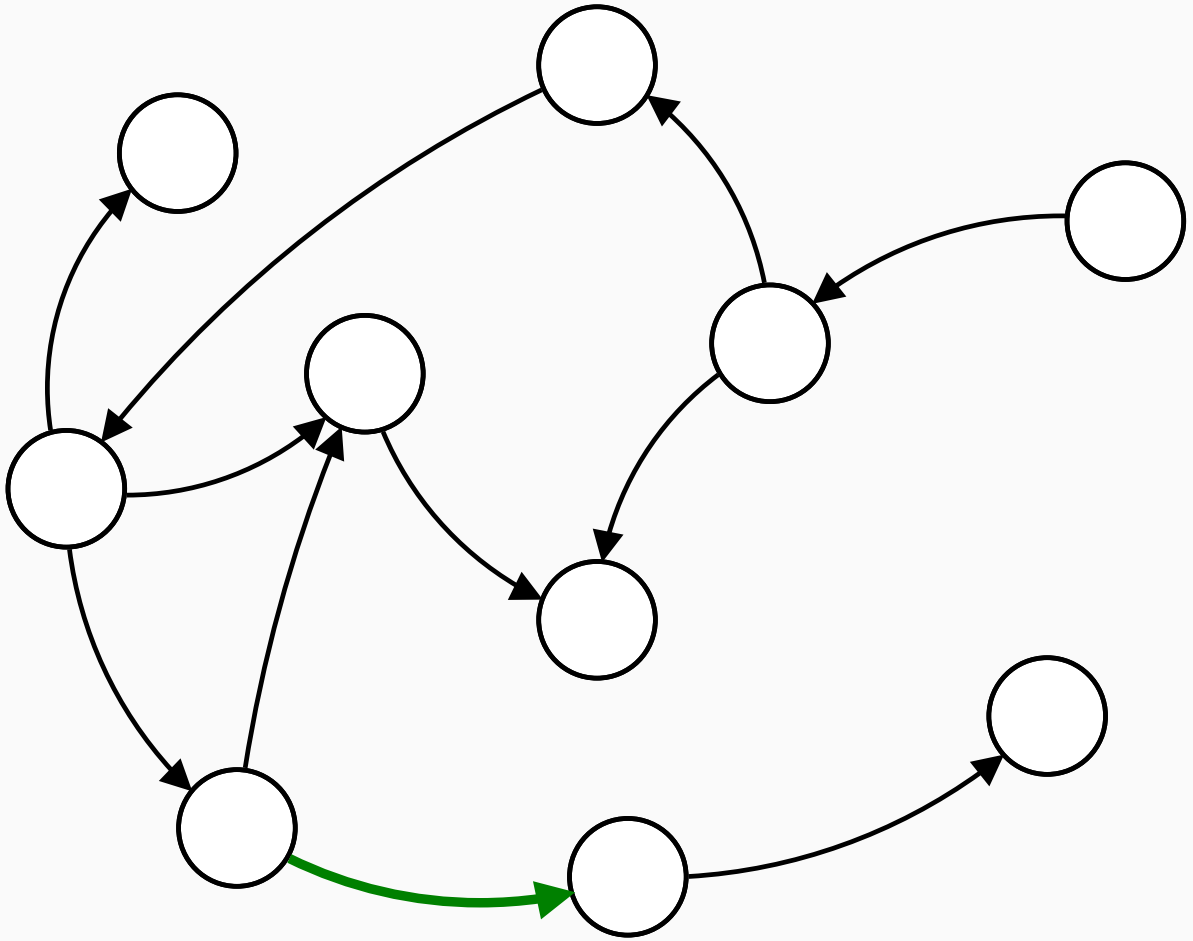
# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



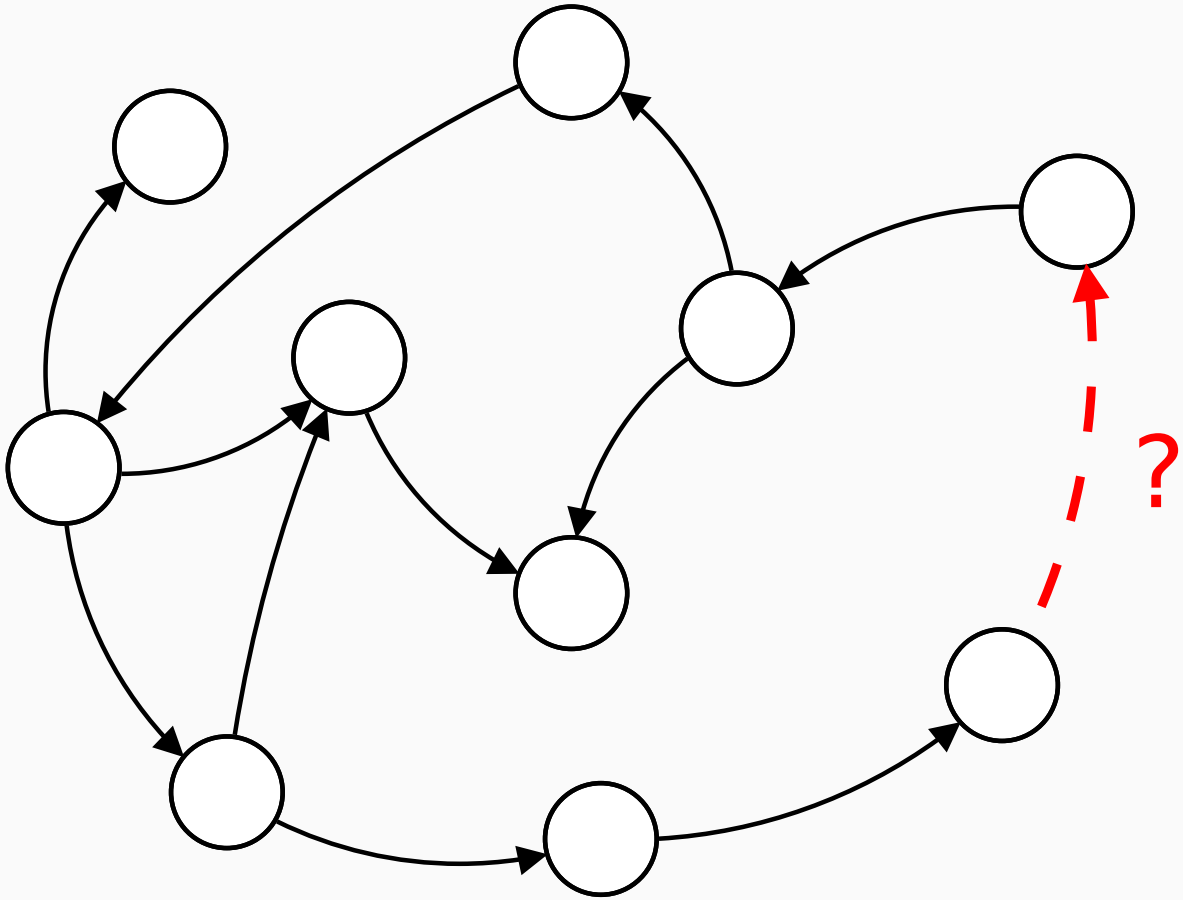
# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



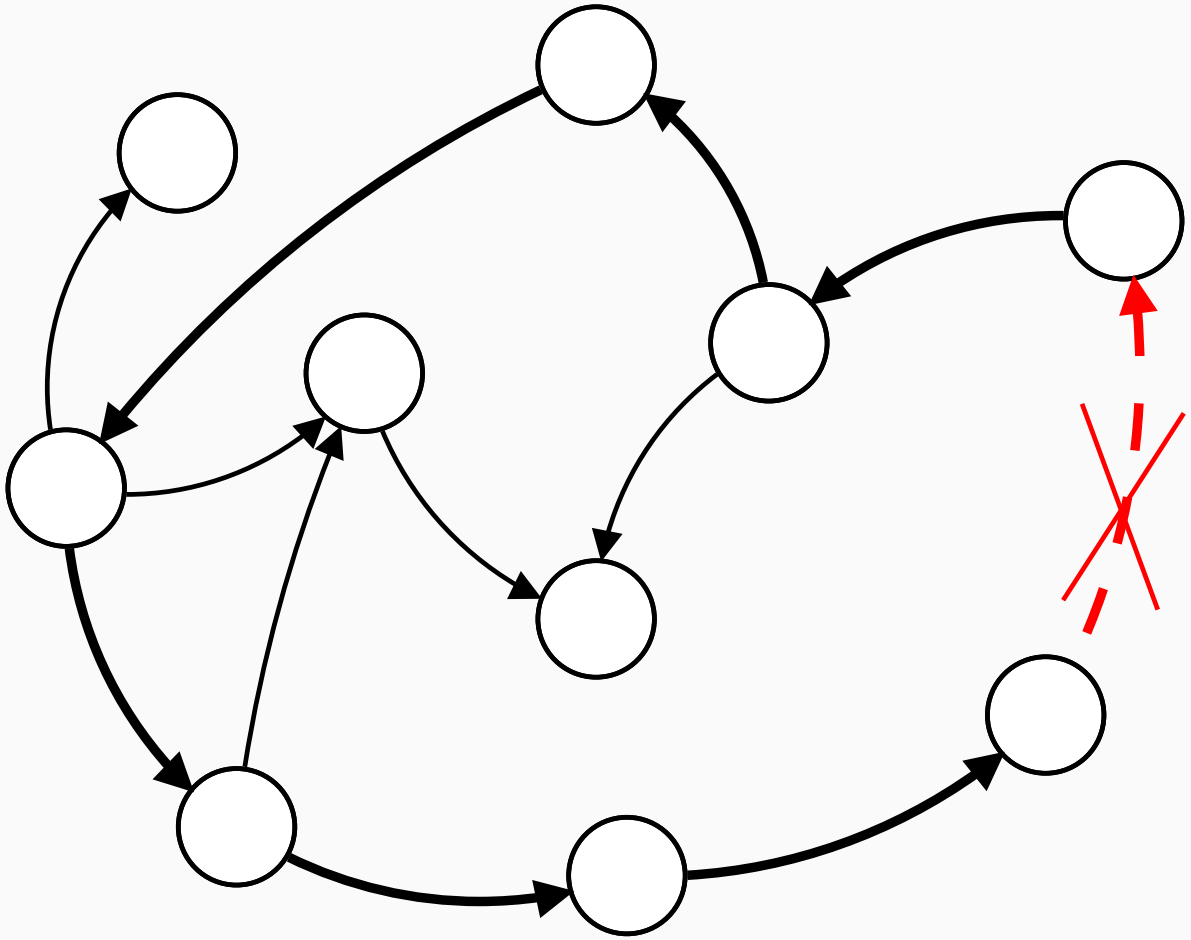
# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



# Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



Naive algorithm: traverse the graph at each step.

Each arc insertion costs $O(m)$.

Coq and Dune implement a state-of-the-art algorithm by Bender, Fineman, Gilbert and Tarjan (2016).

It runs in $O(m \cdot \min(m^{1/2}, n^{2/3}))$ for $m$ arc insertions.

In particular, in a sparse graph, $O(\sqrt{m})$ amortized for each insertion.

# Contributions

- A simple yet crucial improvement to make Bender et al.'s algorithm truly online;
- An OCaml implementation as a standalone library;
- A machine-checked proof of both its functional correctness and amortized asymptotic complexity;
- Time credits that are counted in $\mathbb{Z}$ (instead of $\mathbb{N}$): this leads to significantly fewer proof obligations (!).

# In the rest of this talk

Overview of the library: interface and specification

Bender et al.'s algorithm: Key Ideas

Complexity Analysis

A taste of time credits: forward traversal analysis

Integer Time Credits

# Overview of the library: interface and specification

# Implementation

~150 lines of (terse) hand written OCaml code

# Minimal OCaml interface

```
module Make (G : Raw_graph) : sig
  val add_vertex :
    G.graph -> G.vertex -> unit

  type add_edge_result =
    | EdgeAdded
    | EdgeCreatesCycle

  val add_edge_or_detect_cycle :
    G.graph -> G.vertex -> G.vertex ->
    add_edge_result
end
```

# Toplevel specification (functional correctness only)

$$
\begin{array}{l}
\forall g\, G\, v\, w.\ \text{let } m := |\text{edges } G| \text{ in} \\
\qquad \text{let } n := |\text{vertices } G| \text{ in} \\
\qquad v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
\qquad \left\{ \begin{array}{l} \text{IsGraph } g\ G \end{array} \right\} \\
\qquad (\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
\qquad \left\{ \begin{array}{l} \lambda \text{res. match res with} \\ \quad | \text{ EdgeAdded} \Rightarrow \text{IsGraph } g\ (G + (v, w)) \\ \quad | \text{ EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v]) \end{array} \right\}
\end{array}
$$

---

$$\forall g\, G.\ \text{IsGraph } g\ G \ \Vdash\ \text{IsGraph } g\ G * [\forall x.\ x \longrightarrow\!\!\!\!\!+\ {}_G^+\ x]$$

# Toplevel specification (functional correctness only)

$$
\begin{aligned}
&\forall g\, G\, v\, w.\ \text{let } m := |\text{edges } G| \text{ in} \\
&\quad \text{let } n := |\text{vertices } G| \text{ in} \\
&\quad v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
&\quad \left\{ \ \text{IsGraph}\, g\ G \ \right\} \\
&\quad (\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
&\quad \left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad \mid \text{EdgeAdded} \Rightarrow \text{IsGraph}\, g\ (G + (v, w)) \\ \quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v]) \end{array} \right\}
\end{aligned}
$$

---

$$
\begin{aligned}
&\text{IsGraph}\, g\ G\ :=\ \exists L\, M\, I.\ \text{IsRawGraph}\, g\ G\ L\ M\ I * [\text{Inv}\ G\ L\ I] \\
&\text{Inv}\ G\ L\ I\ :=\ (\forall x.\ x \longrightarrow\!\!\!\!\!+\ {}^{+}_{G}\ x)\ \wedge\ \ldots
\end{aligned}
$$

# Toplevel specification (correctness and complexity)

$$
\begin{array}{l}
\forall g\, G\, v\, w.\ \text{let } m := |\text{edges } G| \text{ in} \\
\qquad \text{let } n := |\text{vertices } G| \text{ in} \\
\qquad v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
\qquad \left\{ \ \text{IsGraph}\, g\ G \ \right\} \\
\qquad (\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
\qquad \left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad \mid \text{EdgeAdded} \Rightarrow \text{IsGraph}\, g\ (G + (v, w)) \\ \quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v]) \end{array} \right\}
\end{array}
$$

---

$$
\begin{array}{l}
\text{IsGraph}\, g\ G\ :=\ \exists L\, M\, I.\ \text{IsRawGraph}\, g\ G\ L\ M\ I * [\text{Inv}\ G\ L\ I] * \$\phi(G, L) \\
\text{Inv}\ G\ L\ I\ :=\ (\forall x.\ x \longrightarrow\!\!\!\!\!+\ {}^{+}_{G}\ x)\ \wedge\ \ldots
\end{array}
$$

---

# Toplevel specification (correctness and complexity)

$$
\begin{array}{l}
\forall g\, G\, v\, w.\ \text{let } m := |\text{edges } G| \text{ in} \\
\qquad \text{let } n := |\text{vertices } G| \text{ in} \\
\qquad v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
\qquad \left\{ \ \text{IsGraph}\, g\ G * \$(\psi\,(m+1, n) - \psi\,(m, n)) \ \right\} \\
\qquad (\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
\qquad \left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad \mid \text{EdgeAdded} \Rightarrow \text{IsGraph}\, g\ (G + (v, w)) \\ \quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v]) \end{array} \right\}
\end{array}
$$

---

$$
\begin{array}{l}
\text{IsGraph}\, g\ G\ :=\ \exists L\, M\, I.\ \text{IsRawGraph}\, g\ G\ L\ M\ I * [\text{Inv}\ G\ L\ I] * \$\phi(G, L) \\
\text{Inv}\ G\ L\ I\ :=\ (\forall x.\ x \longrightarrow\!\!\!\!\!+\ {}^{+}_{G}\ x)\ \wedge\ \ldots
\end{array}
$$

---

$$\psi \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$$

# Using the specification

```
let g = create_graph () in
add_vertex g 1;                          $(ψ(0,1) − ψ(0,0))
...
add_vertex g n;                          $(ψ(0,n) − ψ(0,n−1))
add_edge_or_detect_cycle g 1 2;          $(ψ(1,n) − ψ(0,n))
add_edge_or_detect_cycle g 2 3;          $(ψ(2,n) − ψ(1,n))
...
add_edge_or_detect_cycle g (m-1) m;      $(ψ(m,n) − ψ(m−1,n))
```

Total cost: $\psi(m,n) - \psi(0,0)$

# Using the specification

```
let g = create_graph () in
add_vertex g 1;                          $(ψ(0,1) − ψ(0,0))
...
add_vertex g n;                          $(ψ(0,n) − ψ(0,n−1))
add_edge_or_detect_cycle g 1 2;          $(ψ(1,n) − ψ(0,n))
add_edge_or_detect_cycle g 2 3;          $(ψ(2,n) − ψ(1,n))
...
add_edge_or_detect_cycle g (m-1) m;      $(ψ(m,n) − ψ(m−1,n))
```

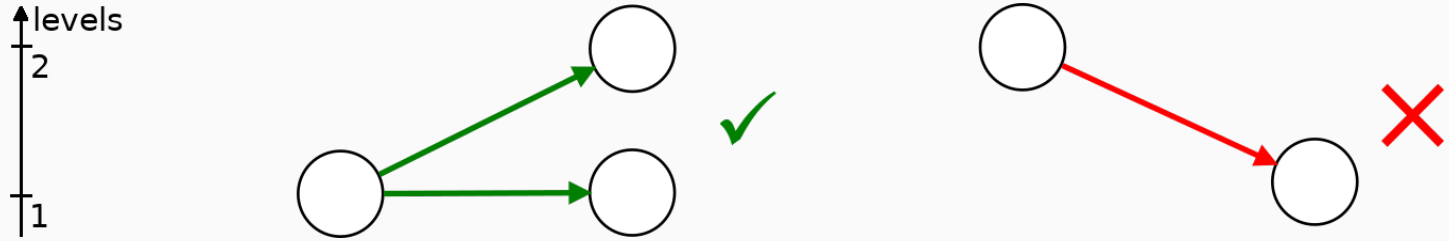
Total cost: $\psi(m,n) - \psi(0,0) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$

# Bender et al.'s algorithm: Key Ideas

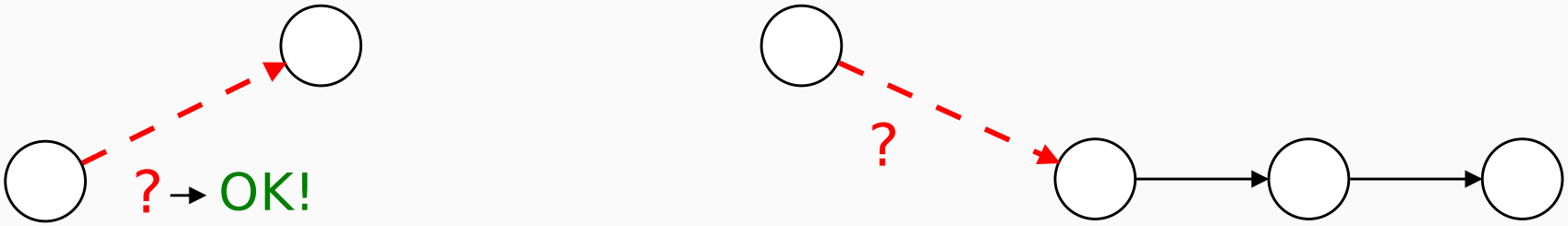
# Idea 1: Levels

Each vertex $v$ is given a level $L(v)$.

Invariant: $v \longrightarrow_G w \implies L(v) \leqslant L(w)$



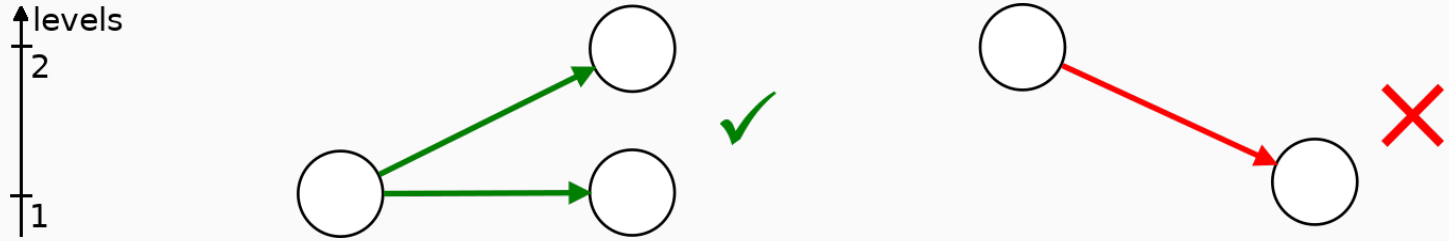

Can accelerate the search, but needs to be maintained:



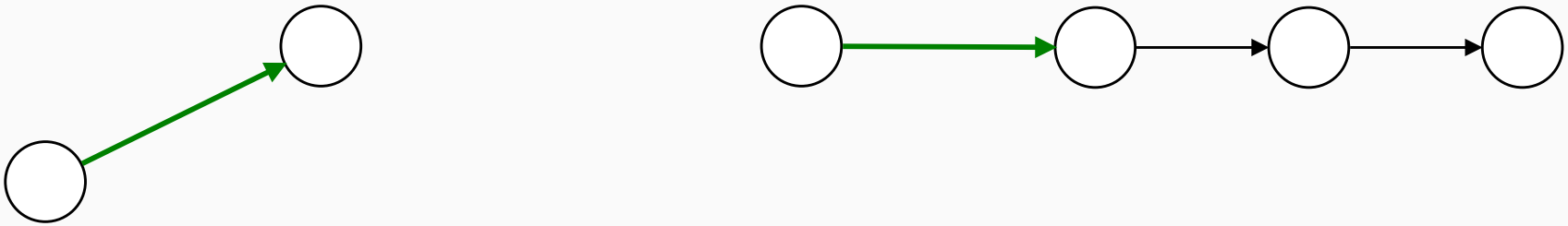
# Idea 1: Levels

Each vertex $v$ is given a level $L(v)$.

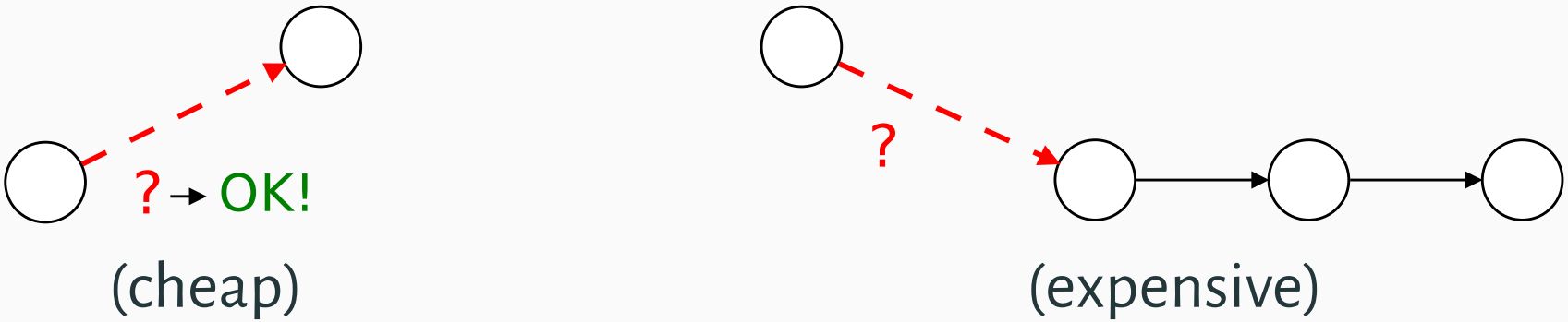
Invariant: $v \longrightarrow_G w \implies L(v) \leqslant L(w)$




Can accelerate the search, but needs to be maintained:

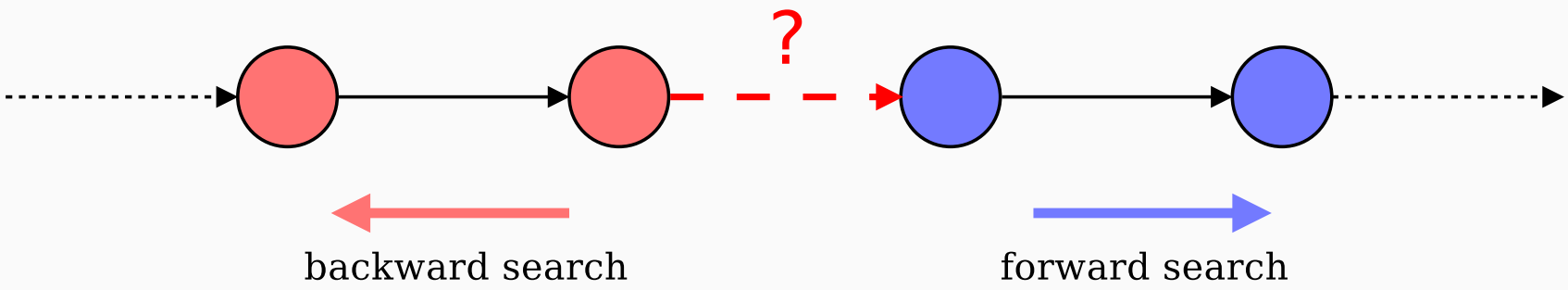


# Idea 1 (bis): Tradeoff on the number of levels




- Too many levels: the expensive case triggers often, outweights the cheap case
- Too few levels: similar to the naive algorithm, no gain from the cheap case

# Idea 2: Two-way Search




The backward search is:

- *restricted* at the same level
- *bounded* by a predetermined number of edges $F$

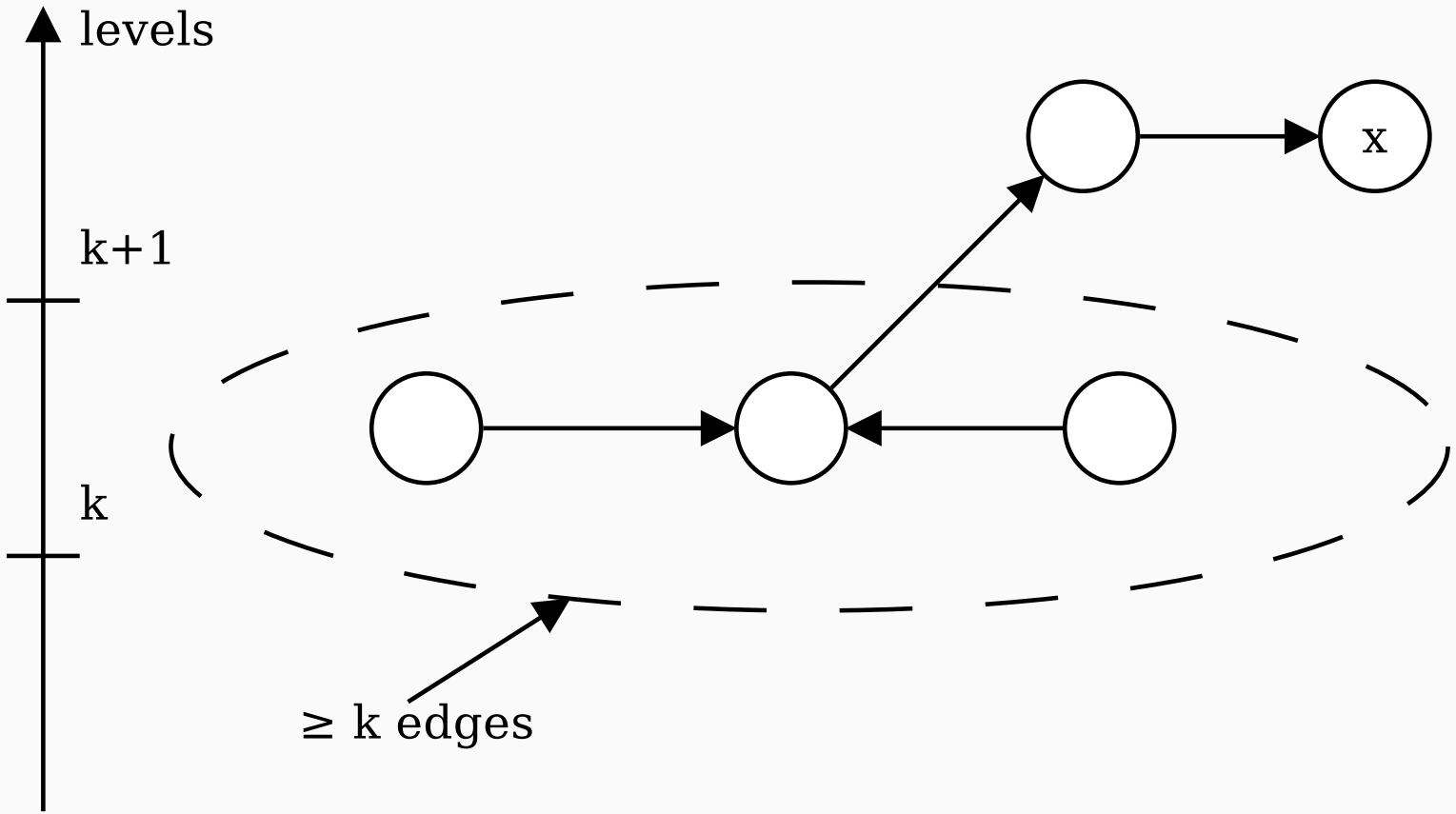
The forward search restores the invariant on levels as it goes.

Not explained at this point: when do new levels get created?

Let's see: Demo!

# Main complexity invariant: levels are “replete”

For every node $x$ at level $k+1$ there is at least $k$ co-accessible edges at level $k$.




Corollary: there is at least $k$ edges at level $k$.

# Complexity Analysis

$$
\begin{array}{l}
\forall g\, G\, v\, w. \\
\text{let } m, n := |\text{edges } G|\,, |\text{vertices } G| \text{ in} \\
v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
\left\{ \ \text{IsGraph}\, g\ G * \$(\psi\,(m+1, n) - \psi\,(m, n)) \ \right\} \\
(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
\left\{ \begin{array}{l}
\lambda\, \text{res. match res with} \\
\quad \mid \text{EdgeAdded} \Rightarrow \text{IsGraph}\, g\ (G + (v, w)) \\
\quad \mid \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v])
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\forall g\, G\, L\, M\, I\, v\, w. \\
\text{let } m, n := |\text{edges } G|\,, |\text{vertices } G| \text{ in} \\
v, w \in \text{vertices } G \ \wedge\ (v, w) \notin \text{edges } G \implies \\
\left\{ \begin{array}{l} \text{IsRawGraph}\, g\, G\, L\, M\, I * [\text{Inv}\, G\, L\, I] * \$\phi(G, L) \\ * \$(\psi\,(m+1, n) - \psi\,(m, n)) \end{array} \right\} \\
(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w) \\
\left\{ \begin{array}{l} \lambda\, \text{res. match res with} \\ \quad | \text{ EdgeAdded} \Rightarrow \text{let } G' := G + (v, w) \text{ in } \exists L'\, M'\, I'. \\ \qquad \text{IsRawGraph}\, g\, G'\, L'\, M'\, I' * [\text{Inv}\, G'\, L'\, I'] * \$\phi(G', L') \\ \quad | \text{ EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v]) \end{array} \right\}
\end{array}
$$

# A sketch of the complexity analysis

- Backward traversal (bounded by $F$): $O(F)$
- Forward traversal: $O(1)$ amortized (!) using $\phi$
- Adding the new edge: $O(1)$
- Potential for the new edge: $O(\psi(m+1,n)-\psi(m,n))$

Good choice for $F$

$\Longrightarrow$ Main complexity invariant (levels are "replete")

$\Longrightarrow$ $F \in O(\psi(m+1,n)-\psi(m,n)) \wedge \psi \in O(m \cdot \min(m^{1/2}, n^{2/3})+n)$

In Bender et al., $F$ depends on the *final* graph; we give an alternative definition that works on the *current* graph.

# A sketch of the complexity analysis

- Backward traversal (bounded by $F$): $O(F)$
- Forward traversal: $O(1)$ amortized (!) using $\phi$
- Adding the new edge: $O(1)$
- Potential for the new edge: $O(\psi(m+1,n) - \psi(m,n))$

Good choice for $F$
$\implies$ Main complexity invariant (levels are "replete")
$\implies$ $F \in O(\psi(m+1,n) - \psi(m,n)) \wedge \psi \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$

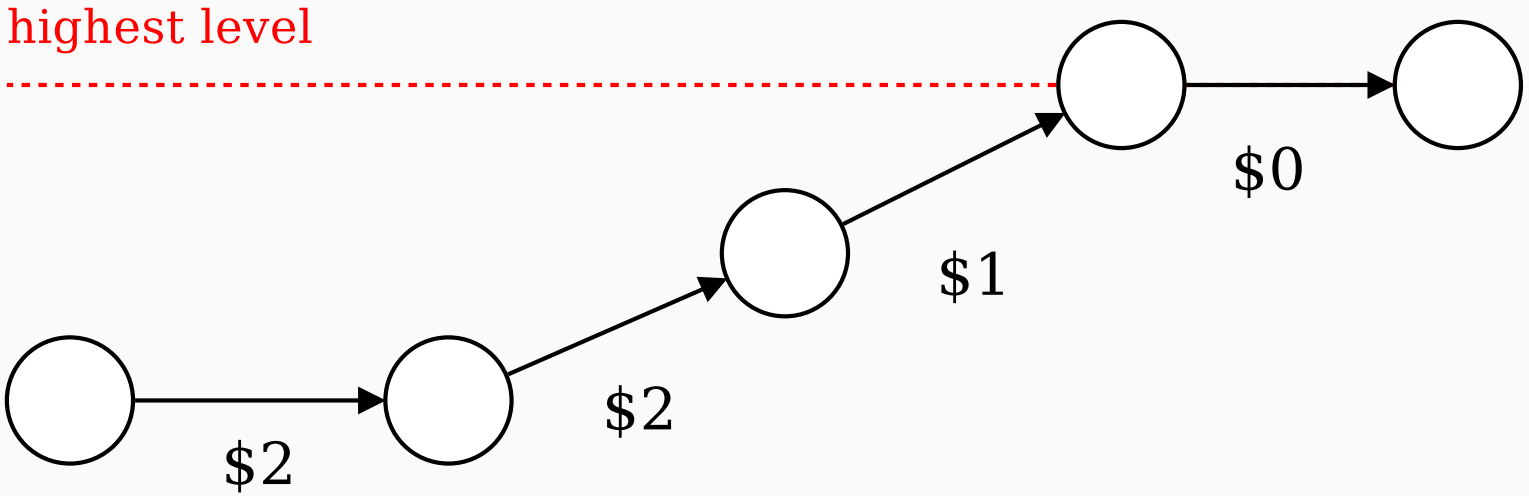
In Bender et al., $F$ depends on the *final* graph; we give an alternative definition that works on the *current* graph.

# A taste of time credits: forward traversal analysis

# The graph potential $\phi$

The potential $\phi$ stores Time Credits for edges depending on their current level (lower level = more credits).

Credits are received at each edge insertion, and spent when *raising* nodes.




$$\phi(G, L) \;:=\; \sum_{(u,v)\in G} (\text{highest\_level}\ G\ L - L(u))$$

# Forward traversal economics

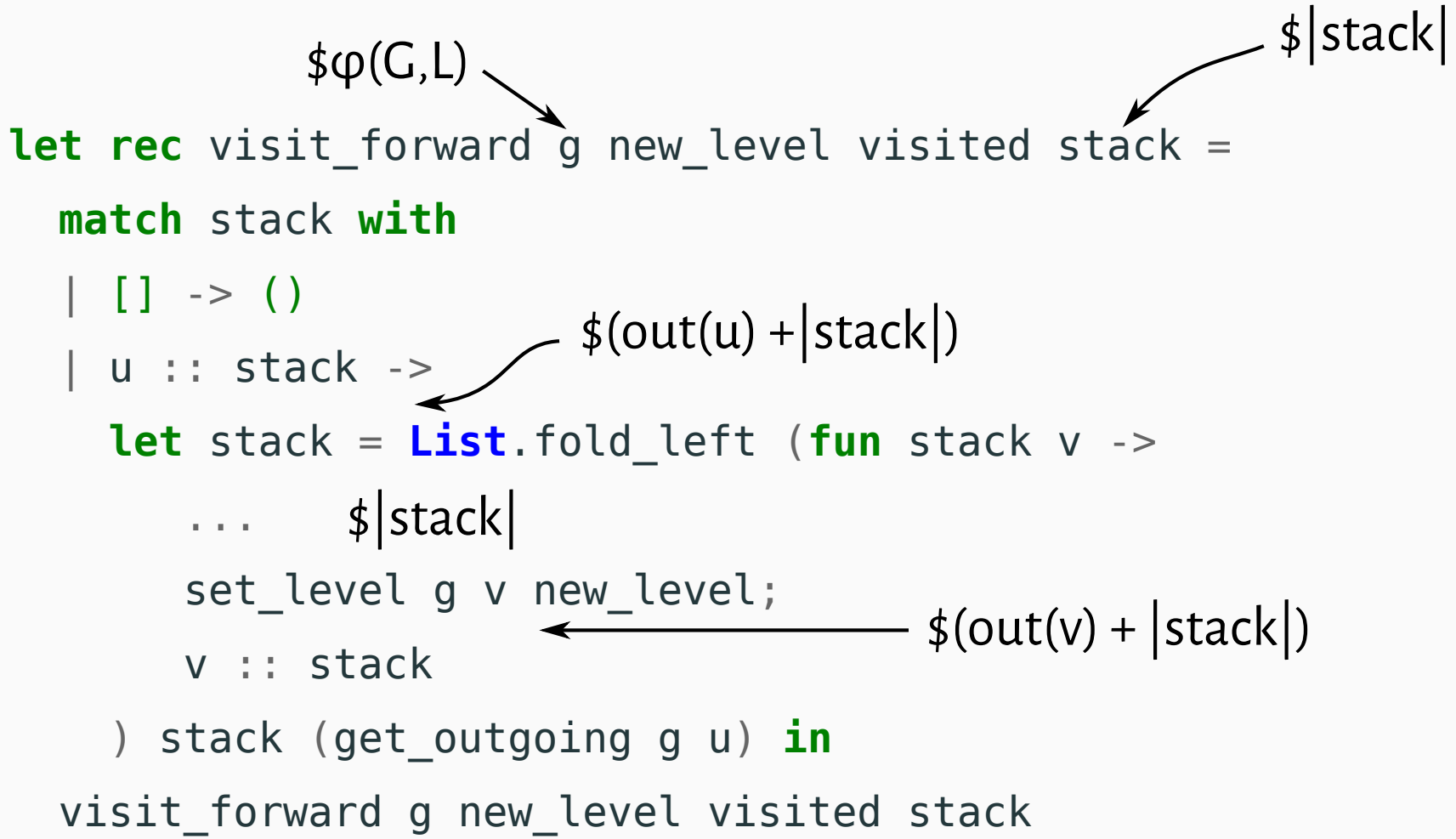
- Traversing an edge $(u, v)$ costs $1$
- Raising $v$ releases $\mathrm{card}(\{w \mid (v, w) \in G\})$ from $\phi$
  (this pays for exploring all the successors of $v$)
- The *stack* holds credits for the next edges to explore

The traversal stack contains credits representing the "working capital" of the traversal.

$$\mathrm{out}(v) := \mathrm{card}(\{w \mid (v, w) \in G\})$$
$$|stack| := \sum_{v \in stack} \mathrm{out}(v)$$

```
let rec visit_forward g new_level visited stack =
  match stack with
  | [] -> ()
  | u :: stack ->
    let stack = List.fold_left (fun stack v ->
        ...
        set_level g v new_level;
        v :: stack
    ) stack (get_outgoing g u) in
  visit_forward g new_level visited stack
```

$$\mathrm{out}(v) := \mathrm{card}(\{w \mid (v,w) \in G\})$$
$$|stack| := \sum_{v \in stack} \mathrm{out}(v)$$



# Proof methodology, in practice

In practice, credit counts involve multiplicative constants:

$$\begin{array}{lcl} \phi(G, L) & := & C \cdot \sum_{(u,v)\in G} (\text{highest\_level}\ G\ L - L(u)) \\ |stack| & := & C' \cdot \sum_{v \in stack} \text{out}(v) \end{array}$$

$$\exists C''.\ 0 \leqslant C'' \ \wedge\ \forall g\ nl\ vs\ stack\ \ldots.$$
$$\{\$C'' * \$|stack| * \ldots\}\ \texttt{visit\_forward}\ g\ nl\ vs\ stack\ \{\lambda().\ \ldots\}$$

$C$, $C'$ and $C''$ depend on specifics of the implementation.

We develop tactics to make the proofs independent from their exact expression, and avoid writing it explicitly by hand.

# Integer Time Credits

# Time Credits and redundant proof obligations

Originally, Time Credits are counted in $\mathbb{N}$:

$$\begin{aligned} \$0 &\equiv true \\ \$(m+n) &\equiv \$m * \$n \\ \$n &\Vdash true \end{aligned}$$

Corollary:

$$\$n \equiv \$(n-m) * \$m \qquad \textbf{if } m \leqslant n$$

# Time Credits and redundant proof obligations (2)

Starting with $\$n$ then paying for operations with costs $m_1$, $m_2$, ..., $m_k$ produces redundant proof obligations:

$$
\begin{array}{lll}
\$n & & \\
\qquad \text{pay } \$m_1 & \rightsquigarrow & n - m_1 \geqslant 0 \\
\$(n - m_1) & & \\
\qquad \text{pay } \$m_2 & \rightsquigarrow & n - m_1 - m_2 \geqslant 0 \\
\ldots & & \\
\$(n - m_1 - m_2 - \ldots - m_{k-1}) & & \\
\qquad \text{pay } \$m_k & \rightsquigarrow & n - m_1 - m_2 - \ldots - m_k \geqslant 0
\end{array}
$$

# Time Credits in $\mathbb{Z}$

We work in a variant of SL with credits counted in $\mathbb{Z}$:

$$\begin{aligned} \$0 &\equiv true \\ \$(m+n) &\equiv \$m * \$n \\ \$n * [n \geqslant 0] &\Rrightarrow true \end{aligned}$$

Corollaries (for any $n, m \in \mathbb{Z}$!):

$$\begin{aligned} \$0 &\equiv \$n * \$(-n) \\ \$n &\equiv \$(n-m) * \$m \end{aligned}$$

Negative credits *are not affine!*

# Time Credits in $\mathbb{Z}$ (2)

Paying for a sequence of operations produces a single final proof obligation:

$$
\begin{array}{lll}
\$n & & \\
\qquad \text{pay } \$m_1 & \rightsquigarrow & \text{no proof obligation} \\
\$(n - m_1) & & \\
\qquad \text{pay } \$m_2 & \rightsquigarrow & \text{no proof obligation} \\
\ldots & & \\
\$(n - m_1 - \ldots - m_{k-1}) & & \\
\qquad \text{pay } \$m_k & \rightsquigarrow & \text{no proof obligation} \\
\qquad \text{discard } \$(n - m_1 - \ldots - m_k) & \rightsquigarrow & n - m_1 - \ldots - m_k \geqslant 0
\end{array}
$$

# Pre/Post-condition duality

With integer time credits, these two specifications are equivalent (using the frame rule):

$$\{\$n\}\ \texttt{f}\ n\ \{\lambda().\ emp\}$$
$$\{emp\}\ \texttt{f}\ n\ \{\lambda().\ \$(-n)\}$$

Bonus: returning negative credits allow the complexity to depend on the result of the function! Example:

$$\{emp\}\ \texttt{collatz\_stopping\_time}\ n\ \{\lambda i.\ \$(-i)\}$$

# Interaction with loops

From the proof of the forward traversal:

$$// \ \$\phi(G, L) * [\text{Inv } G\ L\ I]$$

```
List.fold_left ... (fun ... ->
```

$$\quad // \ \exists L'.\ \$\phi(G, L')$$

$$\quad [\text{extract credits from } \$\phi(\mathrm{G}, \mathrm{L}')]$$

$$\quad \ldots$$

```
)
```

$$// \ \$\phi(G, L'') * [\text{Inv } G\ L''\ I'']$$

(Difficult) Lemma: $\forall G\ L\ I.\ \text{Inv } G\ L\ I \implies \phi(G, L) \geqslant 0$

Time Credits in $\mathbb{N}$ would require a nontrivial strengthening of the loop invariant.

# Interruptible Iteration

```
let rec interruptible_iter f l =
  match l with
  | [] -> true
  | x :: l' -> f x && interruptible_iter f l'
```

# Interruptible Iteration

```
let rec interruptible_iter f l =
  match l with
  | [] -> true
  | x :: l' -> f x && interruptible_iter f l'
```

Integer time credits allow for an intuitive specification:

$$
\begin{array}{l}
\forall I\ l\ f. \\
\quad (\forall x\ l'.\ \mathrm{prefix}\ l'\ l \implies \{I\ l'\}\ f\ x\ \{\lambda b.\ I\ (x :: l')\}) \implies \\
\quad \{I\ [] * \$|l|\} \\
\quad \texttt{interruptible\_iter}\ f\ l \\
\quad \{\lambda b.\ \mathrm{if}\ b\ \mathrm{then}\ I\ l\ \mathrm{else}\ \exists l'\ l''.\ I\ l' * \$|l''| * [l = l' +\!\!+ l'']\}
\end{array}
$$

# Conclusion

# Challenges

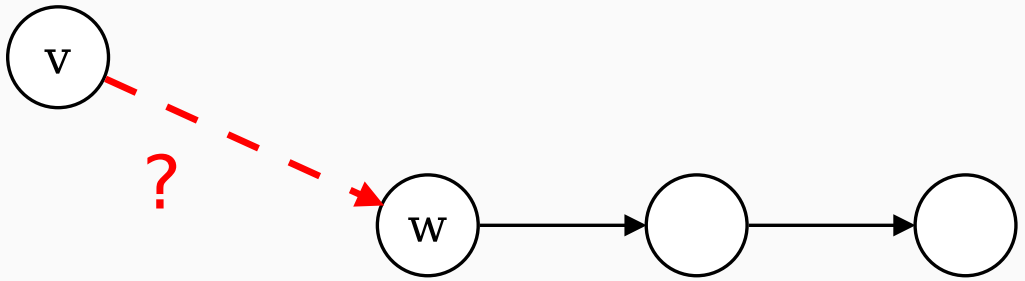
- Understanding the algorithm (!)
- (Re)inventing the complexity invariants
- Designing robust and generic invariants for (interruptible) graph traversals
- Designing Coq tactics for interactive reasoning using integer time credits

Thank you!

Code, proofs, paper and demo available at:

`https://gitlab.inria.fr/agueneau/incremental-cycles`

# Idea 3: Policy for raising nodes to a new level




$w$ and its descendants need to be raised to $L(v)$ or higher.

Bender et al.'s policy:

- If the backward search from $v$ was not interrupted: raised to $L(v)$
- Otherwise, raised to $L(v) + 1$ (possibly creating a new level).

# Idea 4: choice of $F$

Recall: backward search is bounded to visit at most $F$ edges.

The choice of $F$ is crucial to get the correct complexity.

In Bender et al.:

$F = \min(m^{1/2}, n^{2/3})$, for $m$ and $n$ of the *final* graph
(hard to know in practice).

In our modified algorithm:

$F = L(v)$, in the *current* graph
(this makes the algorithm truly online).

# Low-level Data Structure

$\mathrm{IsRawGraph}\ g\ G\ L\ M\ I$: a SL predicate that asserts the ownership of a data structure at address $g$, with logical model $G$, $L$, $M$, $I$.

- $G$: a mathematical graph
- $L$: levels, as a map $\mathrm{vertex} \to \mathbb{Z}$
- $M$: marks, as a map $\mathrm{vertex} \to \mathrm{mark}$
- $I$: horizontal incoming edges, a map $\mathrm{vertex} \to \mathrm{set\ vertex}$

# Functional Invariant

$\mathrm{Inv}\ G\ L\ I$: a pure proposition that relates $G$ with $L$ and $I$.

$$\mathrm{Inv}\ G\ L\ I\ :=$$

$$\begin{cases} \textit{acyclicity}: & \forall x.\quad x \longrightarrow\!\!\!\!\!+\ {}^{+}_{G}\ x \\ \textit{positive levels}: & \forall x.\quad L(x) \geqslant 1 \\ \textit{pseudo–topological levels}: & \forall x\,y.\ x \longrightarrow_G y \implies L(x) \leqslant L(y) \\ \textit{incoming edges}: & \forall x\,y.\ x \in I(y) \iff x \longrightarrow_G y \ \wedge\ L(x) = L(y) \\ \textit{replete levels}: & \forall x.\quad \mathrm{enough\_edges\_below}\ G\ L\ x \end{cases}$$

$$\mathrm{enough\_edges\_below}\ G\ L\ x\ :=$$

$$\quad |\mathrm{coacc\_edges\_at\_level}\ G\ L\ k\ x| \geqslant k \ \text{ where } k = L(x) - 1$$

$$\mathrm{coacc\_edges\_at\_level}\ G\ L\ k\ x\ :=$$

$$\quad \{\,(y,z)\ \mid\ y \longrightarrow_G z \longrightarrow^{*}_{G} x\ \wedge\ L(y) = L(z) = k\,\}$$

# Potential and Advertised Cost (formally)

Potential of an edge $(u, v)$: $\text{max\_level}\ m\ n - L(u)$.

$$
\left.
\begin{aligned}
\phi(G, L) &:= C \cdot (\text{net}\ G\ L) \\
\text{net}\ G\ L &:= \text{received}\ m\ n - \text{spent}\ G\ L
\end{aligned}
\right\}
\text{where } m = |\text{edges } G| \text{ and } n = |\text{vertices } G|
$$

$$
\begin{aligned}
\text{spent}\ G\ L &:= \sum_{(u,v) \in \text{edges}\ G} L(u) \\
\text{received}\ m\ n &:= m \cdot (\text{max\_level}\ m\ n + 1) \\
\text{max\_level}\ m\ n &:= \min(\lceil (2m)^{1/2} \rceil, \lfloor (\tfrac{3}{2} n)^{2/3} \rfloor) + 1
\end{aligned}
$$

---

$$
\psi(m, n) := C' \cdot (\text{received}\ m\ n + m + n)
$$

# Proof methodology

Specification excerpt for the backward traversal:

$$\exists a\, b.\ 0 \leqslant a \ \wedge\ \forall F\ g\ v\ w\ \ldots.$$
$$\{\$(a \cdot F + b) * \ldots\}\ \texttt{backward\_search}\ F\ g\ v\ w\ \{\lambda res.\ \ldots\}$$

# Future Work

# Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

$$\$(?c) * \$\text{potential} \;\Vdash\; \$cost_1 * \ldots * \$cost_n * ?F$$

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

$$\$(?c) * \$\text{potential} * \$(-cost_1) * \ldots * \$(-cost_n) \;\Vdash\; ?F$$

Is this useful?...

# Automation for processing synthesized cost expressions

Credit synthesis produces in the end goals of the form:

$$\exists f. \quad \ldots f \ldots$$
$$\exists a\, b. \quad \ldots a \ldots b \ldots$$

Where "…" usually:

- are complex expressions unwieldy to handle manually;
- contain symbolic expressions (abstract cost functions or constants).