Formally verified incremental cycle detection

Armaël Guéneau

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

Formally Verified Algorithms

Can we formally verify the *functional correctness*...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*

of *non-trivial* algorithms...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*

of *non-trivial* algorithms

with respect to concrete source code?

Previous work: time credits (1)

Previous work: interactive proofs in Separation Logic with *Time Credits*, using Coq and the CFML library.

Charguéraud and Pottier (2017) verify Tarjan's Union-Find.

- Manual accounting of credits: "union costs $4\alpha(n) + 12$";
- Challenging mathematical analysis but fairly short code;

Previous work: time credits (2)

Guéneau, Charguéraud and Pottier (2018) formalize the O notation and advertise for *asymptotic* complexity specifications, e.g. "union costs $f(n)$ where $f \in O(\alpha(n))$".

- Required for specifications to be modular;
- Proofs use a semi-automated *cost synthesis* mechanism;
- However, only small illustrative examples are presented.

Question: does this approach scale?

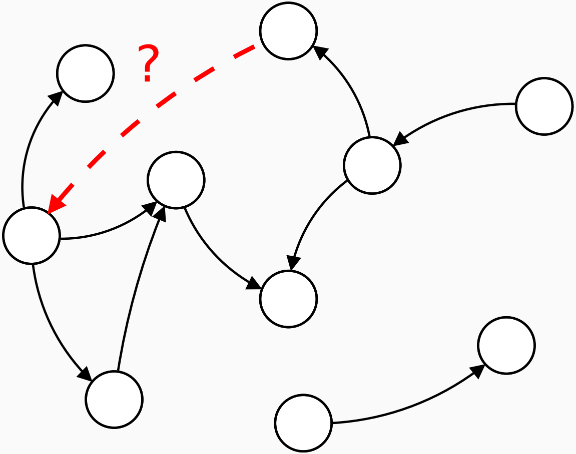
In this talk

Verification of a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).

- non-trivial implementation (200 lines of OCaml code)
- subtle complexity analysis
- used in Coq (universe constraints) and Dune (build dependencies)

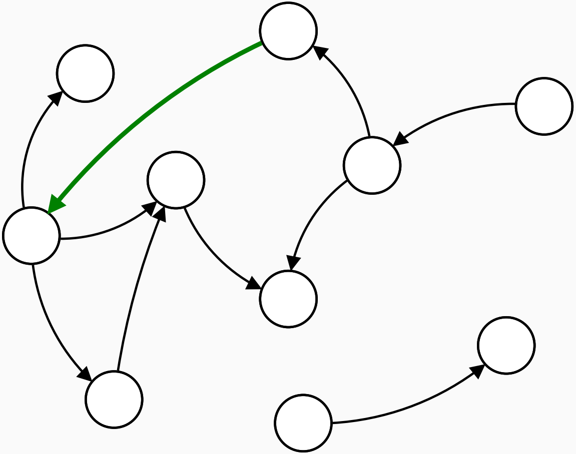
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



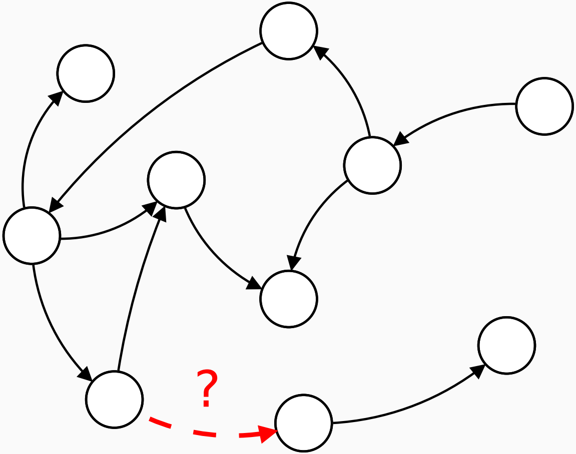
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



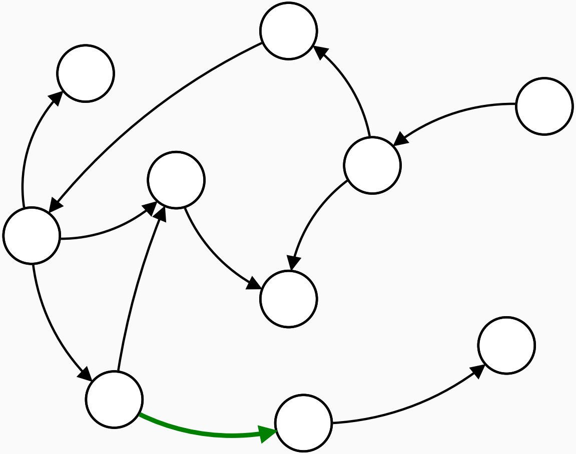
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



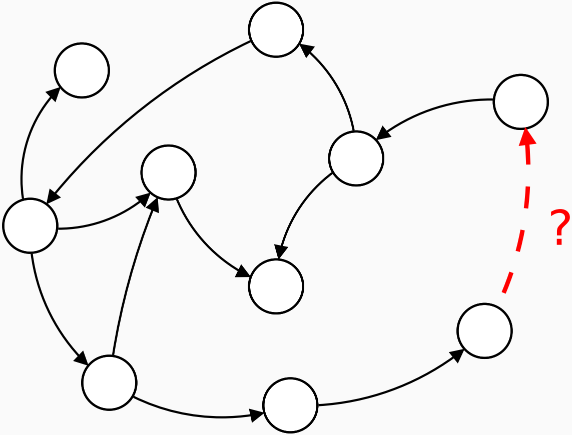
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



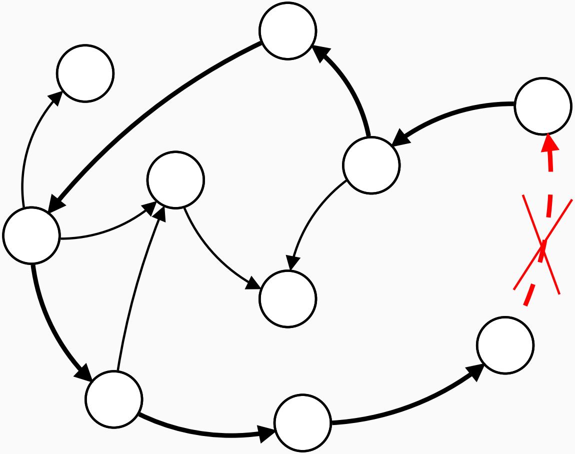
Incremental cycle detection

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.
Inserting m arcs costs $O(m^2)$.

Using Bender et al.'s algorithm, inserting m arcs in a graph with n vertices costs:

- $O(m\sqrt{m})$ for sparse graphs;
- $O(mn^{2/3})$ for dense graphs.

In the general case: $O(m \cdot \min(m^{1/2}, n^{2/3}))$.

⚠ Specifies the cost of a *sequence* of operations, not the cost of a *single* operation.

Contributions

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

Overview

Overview of the library: interface and specification

Complexity Analysis

Verification Techniques and Methodology

Overview of the library: interface and specification

Minimal OCaml interface

```
val init_graph : unit -> graph

val add_vertex : graph -> vertex -> unit

type add_edge_result =
  | EdgeAdded
  | EdgeCreatesCycle

val add_edge_or_detect_cycle :
  graph -> vertex -> vertex ->
  add_edge_result
```

Bender et al.'s algorithm in action

Demo

Toplevel specification (functional correctness only) (1)

$\textsc{InitGraph}$

$\{\mathrm{emp}\}$ `init_graph()` $\{\lambda g.\ \mathrm{IsGraph}\, g\ \varnothing\}$

$\textsc{Acyclicity}$

$\forall g\, G.\ \mathrm{IsGraph}\, g\ G \ \Vdash\ \mathrm{IsGraph}\, g\ G \star [\forall x.\ x \not\longrightarrow^{+}_{G} x]$

Toplevel specification (functional correctness only) (2)

ADDVERTEX

$$\forall g\, G\, v. \quad v \notin \text{vertices}\ G \implies$$

$$\{\text{IsGraph}\, g\ G \star \text{IsNewVertex}\, v\}$$

$$(\texttt{add\_vertex}\ g\ v)$$

$$\{\lambda().\ \text{IsGraph}\, g\ (G + v) \qquad\qquad \}$$

Toplevel specification (functional correctness only) (3)

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

AddEdge

Separation Logic with Time Credits:

- $\$n$ asserts the ownership of n time credits
- $\$n$ is a Separation Logic assertion, like $p \hookrightarrow 3$
- Each function call (or loop iteration) consumes $\$1$
- $\$(n + m) \;\equiv\; \$n \;\star\; \$m$
- Credits are not duplicable: $\$1 \;\not\Longrightarrow\; \$1 \star \$1$
- Specifications are of the form:
  $\{\mathrm{IsGraph}\, g\ G \star \$(3|\mathrm{edges}\ G| + 5)\}$ `dfs` g $\{\mathrm{IsGraph}\, g\ G\}$

Toplevel specification (correctness and complexity) (1)

AddEdge

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

Toplevel specification (correctness and complexity) (1)

AddEdge

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

Complexity

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

Toplevel specification (correctness and complexity) (2)

AddVertex

$$\forall g\, G\, v.\quad \begin{aligned}[t] & \text{let } m := |\text{edges } G| \ \text{ in} \\ & \text{let } n := |\text{vertices } G| \ \text{ in} \\ & v \notin \text{vertices } G \implies \\ & \left\{ \begin{array}{l} \text{IsGraph}\, g\ G \star \text{IsNewVertex}\, v \star \\ \$(\psi(m, n+1) - \psi(m, n)) \end{array} \right\} \\ & (\texttt{add\_vertex}\ g\ v) \\ & \{\lambda().\ \text{IsGraph}\, g\ (G + v) \} \end{aligned}$$

Toplevel specification (correctness and complexity) (3)

InitGraph

$\exists k.\ \{\$k\}$ `init_graph()` $\{\lambda g.\ \mathrm{IsGraph}\, g\ \varnothing\}$

Acyclicity

$\forall g\, G.\ \mathrm{IsGraph}\, g\ G \ \Vdash\ \mathrm{IsGraph}\, g\ G \star [\forall x.\ x \not\longrightarrow^{+}_{G} x]$

DisposeGraph

$\forall g\, G.\ \mathrm{IsGraph}\, g\ G \ \Vdash\ \mathrm{emp}$

Analyzing the cost of a sequence of operations

`let g = init_graph () in`	
`add_vertex g 1;`	$\$(\psi(0,1) - \psi(0,0))$
`...`	
`add_vertex g n;`	$\$(\psi(0,n) - \psi(0,n-1))$
`add_edge_or_detect_cycle g 1 2;`	$\$(\psi(1,n) - \psi(0,n))$
`add_edge_or_detect_cycle g 2 3;`	$\$(\psi(2,n) - \psi(1,n))$
`...`	
`add_edge_or_detect_cycle g (m-1) m;`	$\$(\psi(m,n) - \psi(m-1,n))$

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

Analyzing the cost of a sequence of operations

```
let g = init_graph () in
add_vertex g 1;
...
add_vertex g n;
add_edge_or_detect_cycle g 1 2;
add_edge_or_detect_cycle g 2 3;
...
add_edge_or_detect_cycle g (m-1) m;
```

`let g = init_graph () in`	
`add_vertex g 1;`	$\$(\psi(0,1) - \psi(0,0))$
`...`	
`add_vertex g n;`	$\$(\psi(0,n) - \psi(0,n-1))$
`add_edge_or_detect_cycle g 1 2;`	$\$(\psi(1,n) - \psi(0,n))$
`add_edge_or_detect_cycle g 2 3;`	$\$(\psi(2,n) - \psi(1,n))$
`...`	
`add_edge_or_detect_cycle g (m-1) m;`	$\$(\psi(m,n) - \psi(m-1,n))$

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

Analyzing the cost of a sequence of operations

`let g = init_graph () in`	
`add_vertex g 1;`	$\$(\psi(0,1) - \psi(0,0))$
`...`	
`add_vertex g n;`	$\$(\psi(0,n) - \psi(0,n-1))$
`add_edge_or_detect_cycle g 1 2;`	$\$(\psi(1,n) - \psi(0,n))$
`add_edge_or_detect_cycle g 2 3;`	$\$(\psi(2,n) - \psi(1,n))$
`...`	
`add_edge_or_detect_cycle g (m-1) m;`	$\$(\psi(m,n) - \psi(m-1,n))$

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

$\psi(m,n)$: the cost of inserting m edges and n vertices in an empty graph.

Complexity Analysis

IsGraph's hidden potential

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

IsGraph's hidden potential

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

$$
\begin{array}{l}
\text{IsGraph}\, g\ G\ := \ \exists L\, M\, I.\ \text{IsRawGraph}\, g\ G\, L\, M\, I \star [\text{Inv}\ G\ L\ I] \star \$\phi(G, L) \\
\text{Inv}\ G\ L\ I\ := \ (\forall x.\ x \not\longrightarrow^+_G x) \ \wedge\ \ldots
\end{array}
$$

IsGraph's hidden potential

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

An overview of the complexity analysis

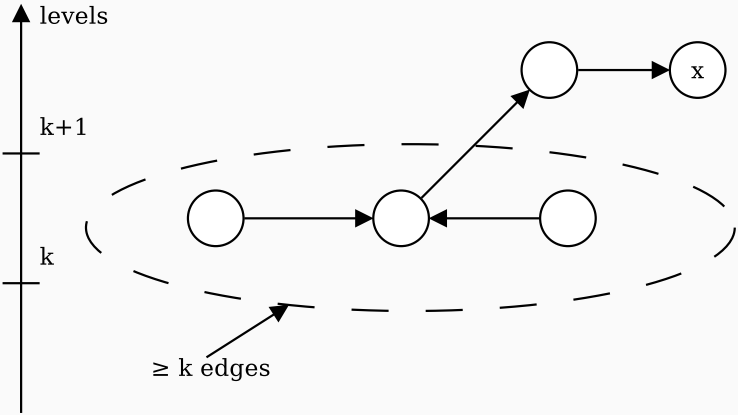
Phase 1 (enumeration of vertices at the current level):
worst-case cost analysis: $O(\psi(m+1,n) - \psi(m,n))$.

Phase 2 (updating levels):
amortized cost analysis: $O(1)$ using the potential ϕ.

Adding the new edge:
increases the potential ϕ, the variation must be
$O(\psi(m+1,n) - \psi(m,n))$.

Main complexity invariant: levels are “replete” (1)

For every node x at level $k+1$ there are at least k edges at level k from which x can be reached.



Main complexity invariant: levels are "replete" (2)

Corollary: there are at least k edges at level k.

This provides a bound on the number of levels:

$$\forall v.\ L(v) \leqslant \sqrt{2m} + 1.$$

I.e. the number of (non-empty) levels is $O(\sqrt{m})$.

(For simplicity we focus on the case of sparse graphs in the rest of the analysis)

Phase 1: worst-case cost $O(\psi(m+1,n) - \psi(m,n))$

When inserting an edge $v \longrightarrow w$, Phase 1 runs in $L(v)$ steps.

Due to the invariant on levels, $L(v)$ is $O(\sqrt{m})$.

$\Rightarrow$ Phase 1 runs in $O(\sqrt{m})$.

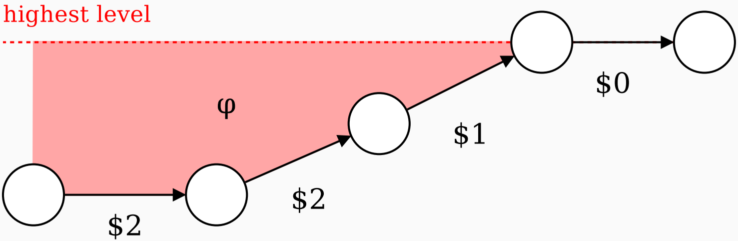
Define ψ (for some constant C', more on that later) as:

$$\psi(m,n) \ := \ C' \cdot (m\sqrt{m} + m + n + 1)$$

From the definition: $\sqrt{m} \in O(\psi(m+1,n) - \psi(m,n))$.

Phase 2: amortized cost $O(1)$

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



$$\phi(G, L) \;:=\; C \cdot \sum_{(v,w)\in G} (\sqrt{m} - L(v))$$

Phase 2 increases the level of edges
$\Rightarrow$ decreases ϕ, releases Time Credits

Edge insertion: potential variation is $O(\psi(m+1,n)-\psi(m,n))$

We need to provide potential for the new edge.

$$\begin{aligned}\phi((G+(v,w)),L)-\phi(G,L) &= C\cdot(\sqrt{m+1}-L(v))\\ &\leqslant C\cdot\sqrt{m+1}\\ &\in O(\psi(m+1,n)-\psi(m,n))\end{aligned}$$

Qed.

Verification Techniques and Methodology

Complexity invariants depend on concrete code

We define ϕ and ψ as follows:

$$\begin{aligned}\phi(G, L) &:= C \cdot \textstyle\sum_{(v,w)\in G}(\sqrt{m} - L(v))\\ \psi(m, n) &:= C' \cdot (m\sqrt{m} + m + n + 1)\end{aligned}$$

...for some constants C and C' *which we must define*.

NB: "$\psi(m, n) := O(m\sqrt{m} + n)$" does not make sense!

C and C' closely depend on details of the implementation.
We *do not want* to write them by hand in the proof!

Robust complexity proofs using abstract constants

The solution relies on our mechanisms for *cost synthesis* and *deferring proof obligations*.

Proof sketch of `update_levels`'s specification:

$$\exists C.\ \forall g\, w\, l.\ \{\$(C \cdot (\ldots)) \star \ldots\}\ \texttt{update\_levels}\ g\ w\ l\ \{\ldots\}$$

- Defer choosing a value for C;
- Cost synthesis yields obligations of the form "$C \geqslant \mathrm{cost\_foo} + \mathrm{cost\_bar} + \ldots$": defer them;
- Automatically deduce a suitable value for C.

Then, ϕ is defined using "the" C from the specification.

Time Credits in $\mathbb{Z}$

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

$$\begin{array}{rrcl} & \$0 & \equiv & \mathrm{emp} \\ \forall m\, n \in \mathbb{N}. & \$(m+n) & \equiv & \$m \star \$n \\ \forall n \in \mathbb{N}. & \$n & \Vdash & \mathrm{emp} \end{array}$$

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

$$\begin{array}{rrcl} & \$0 & \equiv & \mathrm{emp} \\ \forall m\, n \in \mathbb{Z}. & \$(m+n) & \equiv & \$m \star \$n \\ \forall n \in \mathbb{Z}. & \$n \star [n \geqslant 0] & \Vdash & \mathrm{emp} \end{array}$$

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

Time Credits in $\mathbb{Z}$ are more flexible as they allow one to have debts (temporarily!).

As Tarjan puts it: "we can allow borrowing of credits..."

$$\forall n \in \mathbb{Z}.\ \ \mathrm{emp} \ \equiv\ \$n \star \$(-n)$$

"...as long as any debt incurred is eventually paid off."

$$\forall n \in \mathbb{Z}.\ \ \$n \ \star\ [n \geqslant 0] \Vdash\ \mathrm{emp}$$

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

$\forall l.\ \text{let } k := \text{“index of the first 0 in l” in}$

$\{\$(k+1)\}\ \texttt{walk}\ l\ \ \{\lambda l'.\ [\text{suffix } l'\ l]\}$

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

$\forall l.$ let $k :=$ "index of the first 0 in l" in

$\{\$(k+1)\}$ `walk` l $\{\lambda l'. \, [\text{suffix } l' \; l]\}$

$\Rightarrow$ Too complicated: the specification paraphrases the code

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

$$\forall l.\ \{\$(|l| + 1)\}\ \texttt{walk}\ l\ \ \{\lambda l'.\ \$|l'| \star [\mathrm{suffix}\ l'\ l]\}$$

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

$$\forall l.\ \{\$(|l| + 1)\}\ \texttt{walk}\ l\ \ \{\lambda l'.\ \$|l'| \star [\mathrm{suffix}\ l'\ l]\}$$

$$\forall l.\ \{\mathrm{emp}\}\ \texttt{walk}\ l\ \ \{\lambda l'.\ \$(|l'| - |l| - 1) \star [\mathrm{suffix}\ l'\ l]\}$$

Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =
  match l with
  | x :: xs when x <> 0 -> walk xs
  | _ -> l
```

$$\forall l.\ \{\$(|l| + 1)\}\ \ \texttt{walk}\ l\ \ \{\lambda l'.\ \$|l'| \star [\text{suffix}\ l'\ l]\}$$

$$\forall l.\ \{\text{emp}\}\ \ \texttt{walk}\ l\ \ \{\lambda l'.\ \$(|l'| - |l| - 1) \star [\text{suffix}\ l'\ l]\}$$

These two specifications are equivalent.

Summary

- We improve and verify a state-of-the-art algorithm;
- SL with (Possibly Negative) Time Credits is powerful; it allows writing rich and modular specifications;
- Our code is already useful: integrated into Dune, bringing a 7x performance improvement (!);
- Our cost synthesis and deferring mechanisms allow manageable proofs at scale.

More in the paper and my (upcoming) PhD dissertation.

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

Producing the right answer is good.

Producing the right answer is good.

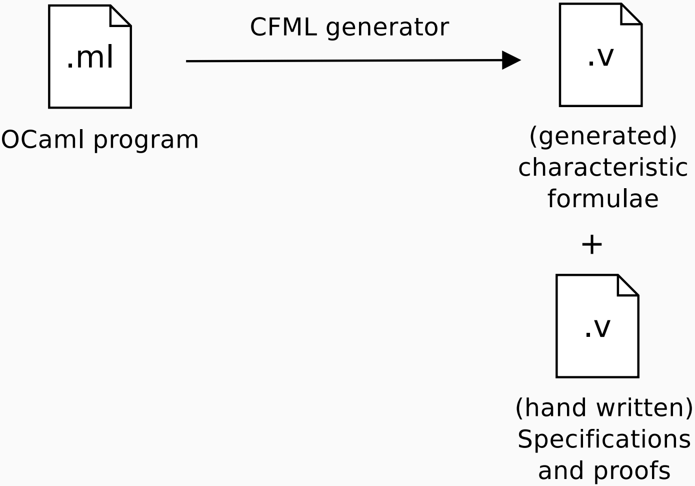
Producing the right answer **at the right time** is better.

Producing the right answer is good.

Producing the right answer **at the right time** is better.

Don’t promise—just **prove** it!

Program verification framework: Coq and CFML



Example specifications using time credits

Complexity specification using explicit time credits:

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

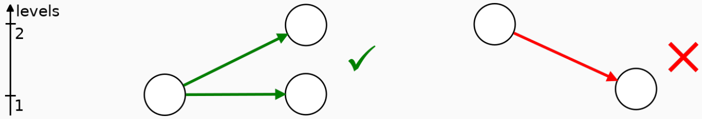
Asymptotic complexity specification:

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

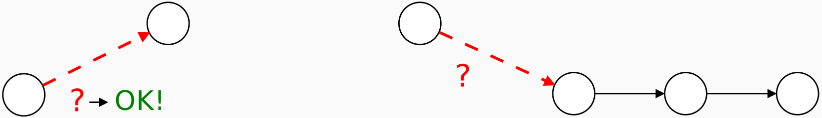
Idea 1: Levels

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

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



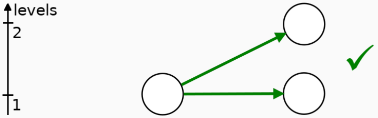
Levels can accelerate the search, but need to be maintained:

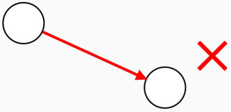


Idea 1: Levels

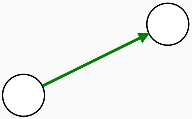
Each vertex v is given a level $L(v)$.

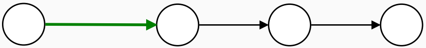
Invariant: $\quad v \longrightarrow_G w \implies L(v) \leqslant L(w)$



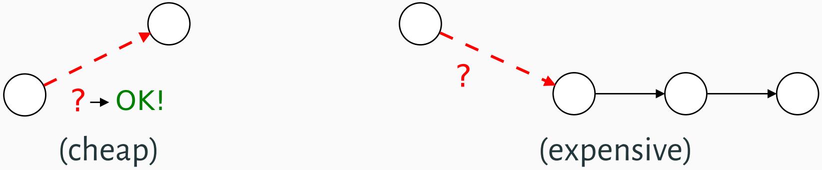


Levels can accelerate the search, but need to be maintained:





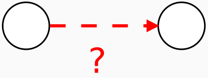
Idea 1 (bis): Tradeoff on the number of levels



- Too many levels: the expensive case triggers often, outweighting the cheap case
- Too few levels: similar to the naive algorithm, insufficient benefit out of the cheap case

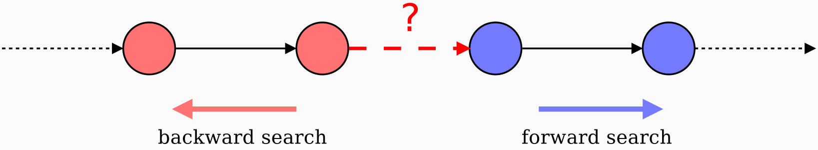
Idea 1 (ter): Tradeoff on the number of levels

Why do we gain anything?



Adding a horizontal edge: the search for a cycle can be restricted *to this level*.

Idea 2: Two-way Search



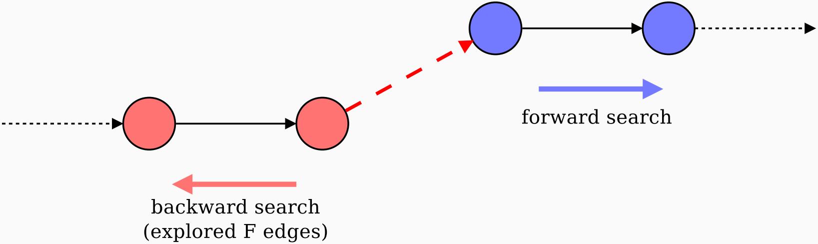
The backward search is:

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

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

Idea 3: when do new levels get created?

If the backward search explores all F edges...



then nodes are moved to a higher level during the forward search.

Forward traversal economics

- Traversing an edge (u, v) costs 1
- Raising v releases $\mathrm{card}(\{w \mid (v, w) \in G\})$ from ϕ
  (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
```

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

$$|stack| \; := \; \textstyle\sum_{v \in stack} \mathrm{out}(v)$$

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

Proof methodology, in practice

In practice, credit counts involve multiplicative constants:

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

$$\begin{array}{l}\exists C''.\ 0 \leqslant C'' \ \wedge\ \forall g\ nl\ vs\ stack\ \ldots. \\ \quad \{\$C'' \star \$|stack| \star \ldots\}\ \texttt{visit\_forward}\ g\ nl\ vs\ stack\ \{\lambda().\ \ldots\}\end{array}$$

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.

Time Credits in $\mathbb{N}$ and redundant proof obligations

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

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

Time Credits in $\mathbb{Z}$ eliminate redundant proof obligations

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

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

This also allows for *simpler* loop invariants and specifications.

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:

$$// \ \$\phi(G, L) \star [\mathrm{Inv}\ G\ L\ I]$$

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

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

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

$$\ldots$$

```
)
```

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

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

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

Walk

```
let rec walk (a: int array) (i: int): int =
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)
  else i+1
```

Walk

```
let rec walk (a: int array) (i: int): int =
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)
  else i+1
```

$$\forall a\, i\, A.\ 0 \leqslant i \leqslant |A| \implies$$
$$\{a \rightsquigarrow \mathrm{Array}\ A\}\ \texttt{walk}\ a\ i\ \{\lambda j.\ a \rightsquigarrow \mathrm{Array}\ A \star \$(i-j) \star [i < j \leqslant |A|]\}$$

Walk

```
let rec walk (a: int array) (i: int): int =
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)
  else i+1
```

$$\forall a\, i\, A.\ 0 \leqslant i \leqslant |A| \implies$$
$$\{a \rightsquigarrow \mathrm{Array}\ A\}\ \texttt{walk}\ a\ i\ \{\lambda j.\ a \rightsquigarrow \mathrm{Array}\ A \star \$(i - j) \star [i < j \leqslant |A|]\}$$

$$\forall a\, i\, A.\ 0 \leqslant i \leqslant |A| \implies$$
$$\{a \rightsquigarrow \mathrm{Array}\ A \star \$(|A| - i)\}$$
$$\texttt{walk}\ a\ i$$
$$\{\lambda j.\ a \rightsquigarrow \mathrm{Array}\ A \star \$(|A| - j) \star [i < j \leqslant |A|]\}$$

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’
```

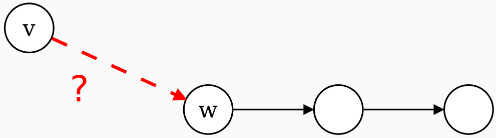
Integer time credits allow for an intuitive specification:

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

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

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.

$$
\mathrm{Inv}\ G\ L\ I \ := \\
\left\{
\begin{array}{lll}
\textit{acyclicity}: & \forall x. & x \not\longrightarrow^{+}_{G} x \\
\textit{positive levels}: & \forall x. & L(x) \geqslant 1 \\
\textit{pseudo–topological levels}: & \forall x\, y. & x \longrightarrow_G y \implies L(x) \leqslant L(y) \\
\textit{incoming edges}: & \forall x\, y. & x \in I(y) \iff x \longrightarrow_G y \ \wedge L(x) = L(y) \\
\textit{replete levels}: & \forall x. & \text{enough\_edges\_below}\ G\ L\ x
\end{array}
\right.
$$

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

$$
\text{coacc\_edges\_at\_level}\ G\ L\ k\ x \ := \\
\quad \{\, (y, z) \mid y \longrightarrow_G z \longrightarrow^{*}_{G} x \ \wedge \ L(y) = L(z) = k \,\}
$$

Potential and Advertised Cost (formally)

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

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

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

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

Proof methodology

Specification excerpt for the backward traversal:

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

Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

$$\$(?c) \star \$\mathrm{potential} \ \Vdash\ \$cost_1 \star \ldots \star \$cost_n \star ?F$$

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

$$\$(?c) \star \$\mathrm{potential} \star \$(-cost_1) \star \ldots \star \$(-cost_n) \ \Vdash\ ?F$$

Is this useful?...

Automation for processing synthesized cost expressions

Credit synthesis produces in the end goals of the form:

$$\begin{array}{ll} \exists f. & \ldots f \ldots \\ \exists a\, b. & \ldots a \ldots b \ldots \end{array}$$

Where "$\ldots$" usually:

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