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)

And our verified implementation turned out to be 7x faster!
Context
Program verification...

Our verification framework: Coq and CFML

OCaml program → CFML generator → (generated) characteristic formulae + (hand written) Specifications and proofs
Example specification, using explicit time credits:

$$\forall g G. \{ \text{IsGraph} \ g \ G \ \ast \ \$(3 \ |\text{edges} \ G| \ + \ 5) \} \ df\lfloor s(g) \ \{ \text{IsGraph} \ g \ G \}$$

Asymptotic complexity specification:

$$\exists (f : \mathbb{Z} \rightarrow \mathbb{Z}).$$

nonnegative $f$ $\wedge$ monotonic $f$ $\wedge$ $f \in O_{\mathbb{Z}}(\lambda m.m)$$
\wedge$$\forall g G. \{ \text{IsGraph} \ g \ G \ \ast \ f(|\text{edges} \ G|) \} \ df\lfloor s(g) \ \{ \text{IsGraph} \ g \ G \}$$
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 (!).
Overview of the library
Implementation

\[ \sim 150 \text{ lines of (terse) hand written OCaml code} \]
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)

\[\forall g \mathcal{G}. \text{IsGraph } g \mathcal{G} \equiv \text{IsGraph } g \mathcal{G} \times [\forall x. x \rightarrow^*_{\mathcal{G}} x] \]
Toplevel specification (functional correctness only)

∀g G v w. let m := |edges G| in
  let n := |vertices G| in
  v, w ∈ vertices G ∧ (v, w) ∉ edges G →
  { IsGraph g G }
  (add_edge_or_detect_cycle g v w)

  λ res. match res with
  { EdgeAdded ⇒ IsGraph g (G + (v, w))
  EdgeCreatesCycle ⇒ [w →^* G v])

IsGraph g G := ∃L M I. IsRawGraph g G L M I * [Inv G L I]
Inv G L I := (∀x. x →^+ G x) ∧ ...
Toplevel specification (correctness and complexity)

∀g G v w. let m := |edges G| in
let n := |vertices G| in
v, w ∈ vertices G ∧ (v, w) ∉ edges G ⊃

\{ IsGraph g G \}

(add_edge_or_detect_cycle g v w)

\{ λ res. match res with
    | EdgeAdded ⇒ IsGraph g (G + (v, w))
    | EdgeCreatesCycle ⇒ [w →* G v] \}

IsGraph g G := ∃L M I. IsRawGraph g G L M I * [Inv G L I] * \$φ(G, L)
Inv G L I := (∀x. x →* G x) ∧ ...
∀g G v w. let m := |edges G| in
  let n := |vertices G| in
  v, w ∈ vertices G ∧ (v, w) ∉ edges G ⟹
  \{ IsGraph g G ∗ $(\psi (m + 1, n) − \psi (m, n)) \}
(\text{add\_edge\_or\_detect\_cycle } g v w)

\lambda \text{res. match res with}
  \{ \text{EdgeAdded ⇒ IsGraph } g (G + (v, w)) \}
  \{ \text{EdgeCreatesCycle ⇒ } [w \rightarrow^*_G v] \}

\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 \rightarrow^*_G x) ∧ \ldots

\psi ∈ O(m \cdot \min(m^{1/2}, n^{2/3}) + n)
Using the specification

```plaintext
let g = create_graph () in
add_vertex g 1;    $(\psi(0, 1) - \psi(0, 0))
...
add_vertex g n;    $(\psi(0, n - 1) - \psi(0, n))
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: $\psi(m, n) - \psi(0, 0)$
```
Using the specification

\[
\begin{align*}
\text{let } g &= \text{create_graph } () \text{ in} \\
\text{add_vertex } g &\ 1; & $(\psi(0, 1) - \psi(0, 0)) \\
\ldots \\
\text{add_vertex } g &\ n; & $(\psi(0, n - 1) - \psi(0, n)) \\
\text{add_edge_or_detect_cycle } g &\ 1\ 2; & $(\psi(1, n) - \psi(0, n)) \\
\text{add_edge_or_detect_cycle } g &\ 2\ 3; & $(\psi(2, n) - \psi(1, n)) \\
\ldots \\
\text{add_edge_or_detect_cycle } g &\ (m-1)\ m; & $(\psi(m, n) - \psi(m - 1, n))
\end{align*}
\]

**Total cost:** $\psi(m, n) - \psi(0, 0) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$
In the rest of this talk

Bender et al.’s algorithm: Key Ideas

Formalization: Data Structure Invariants and Potential

Proof Methodology

Integer Time Credits

Future Work
Bender et al.’s algorithm: Key Ideas
Idea 1: Levels

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

Invariant: $v \rightarrow_G w \implies L(v) \leq L(w)$

Can accelerate the search, but needs to be maintained:
Idea 1: Levels

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

Invariant: $v \to_G w \implies L(v) \leq L(w)$

Can accelerate the search, but needs to be maintained:
Idea 2: Two-way Search

The backward search is:

- restricted at the same level
- bounded by a predetermined number of edges $F$
**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).
Demo!
Formalization: Data Structure Invariants and Potential
∀ g ∇ G v w.
let m, n := |edges G|, |vertices G| in
v, w ∈ vertices G ∧ (v, w) ∉ edges G →

\{ IsGraph g G * $(\psi(m + 1, n) - \psi(m, n)) \} \\
(add_edge_or_detect_cycle g v w)

\{ λ res. match res with \\
    | EdgeAdded ⇒ IsGraph g (G + (v, w)) \\
    | EdgeCreatesCycle ⇒ [w →^* G v] \}
∀g G L M I v w.
let m, n := |edges G|, |vertices G| in
v, w ∈ vertices G ∧ (v, w) ⊈ edges G ⇒

\[
\begin{align*}
\text{IsRawGraph } g G L M I & \ast \text{[Inv G L I]} \ast \$\phi(G, L) \\
& \ast \$(\psi (m + 1, n) − \psi (m, n)) \\
\end{align*}
\]
(add_edge_or_detect_cycle g v w)

λ res. match res with
| EdgeAdded ⇒ let G' := G + (v, w) in ∃L' M' I'.
| IsRawGraph g G' L' M' I' \ast \text{[Inv G' L' I']} \ast \$\phi(G', L')
| EdgeCreatesCycle ⇒ [w \rightarrow^* G v])
Low-level Data Structure

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 vertex \( \rightarrow \mathbb{Z} \)
- \( M \): marks, as a map vertex \( \rightarrow \) mark
- \( I \): horizontal incoming edges, a map vertex \( \rightarrow \) set vertex
Functional Invariant

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

Inv $G\ L\ I :=$

\[
\begin{align*}
\text{acyclicity:} & & \forall x. \ x \xrightarrow{}_G x \\
\text{positive levels:} & & \forall x. \ L(x) \geq 1 \\
\text{pseudo-topological levels:} & & \forall x\ y. \ x \xrightarrow{G} y \implies L(x) \leq L(y) \\
\text{incoming edges:} & & \forall x\ y. \ x \in I(y) \iff x \xrightarrow{G} y \land L(x) = L(y) \\
\text{replete levels:} & & \forall x. \ \text{enough\_edges\_below\_G\ L\ x}
\end{align*}
\]

enough\_edges\_below\_G\ L\ x :=

|coacc\_edges\_at\_level\ G\ L\ k\ x| \geq k \text{ where } k = L(x) - 1

coacc\_edges\_at\_level\ G\ L\ k\ x :=

\{(y, z) \mid y \xrightarrow{G} z \xrightarrow{G} x^* \land L(y) = L(z) = k\}
Potential (informally)

Time Credits are stored in the graph, depending on the current level (lower level = more credits).

Credits are received at each edge insertion, and spent when raising nodes.
\[ \forall g \; G \; L \; M \; I \; v \; w. \]

let \( m, n := |\text{edges } G|, |\text{vertices } G| \) in

\( v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \implies \)

\[
\left\{
\begin{array}{l}
\text{IsRawGraph } g \; G \; L \; M \; I \ast [\text{Inv } G \; L \; I] \ast \$\phi(G, L) \\
\ast $(\psi (m + 1, n) - \psi (m, n))
\end{array}
\right.
\]

(\text{add\_edge\_or\_detect\_cycle } g \; v \; w)

\[
\lambda \text{res. match res with}
\]

\[
\left\{
\begin{array}{l}
| \text{EdgeAdded } \Rightarrow \text{let } G' := G + (v, w) \text{ in } \exists L' \; M' \; I'.
\text{IsRawGraph } g \; G' \; L' \; M' \; I' \ast [\text{Inv } G' \; L' \; I'] \ast \$\phi(G', L')
| \text{EdgeCreatesCycle } \Rightarrow [w \longrightarrow_{G}^* v]
\end{array}
\right.
\]
Potential and Advertised Cost (formally)

Potential of an edge \((u, v)\): \(\max_{m, n} m \cdot n - L(u)\).

\[
\begin{align*}
\phi(G, L) & := C \cdot (\text{net } G \cdot L) \\
\text{net } G \cdot L & := \text{received } m \cdot n - \text{spent } G \cdot L \\
\text{spent } G \cdot L & := \sum_{(u,v) \in \text{edges } G} L(u) \\
\text{received } m \cdot n & := m \cdot (\max_{m, n} m \cdot n + 1) \\
\max_{m, n} m \cdot n & := \min\left(\left\lceil (2m)^{1/2} \right\rceil, \left\lceil \frac{3}{2} n^{2/3} \right\rceil \right) + 1 \\
\psi(m, n) & := C' \cdot (\text{received } m \cdot n + m + n)
\end{align*}
\]

where \(m = |\text{edges } G|\) and \(n = |\text{vertices } G|\).
Proof Methodology
Synthesizing Cost Expressions

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

\[
\phi(G, L) := C \cdot (\text{net } G \ L)
\]
\[
\psi(m, n) := C' \cdot (\text{received } m \ n + m + n)
\]

Similarly in the specification for the backward traversal:

\[
\exists a \ b. \ 0 \leq a \land \forall F \ g \ v \ w \ \ldots.
\]
\[
\{$(a \cdot F + b) \ast \ldots\} \ \text{backward_search } F \ g \ v \ w \ \{$\lambda res. \ \ldots\}$
\]

We want the proof to be independent from their exact expression, and avoid writing it explicitly by hand.
Integer Time Credits
Originally, Time Credits are counted in $\mathbb{N}$:

\[
\begin{align*}
0 & \equiv true \\
(m + n) & \equiv m \star n \\
n & \models true
\end{align*}
\]

Corollary:

\[
n \equiv (n - m) \star m \quad \text{if } m \leq n
\]
Starting with $n$ then paying for operations with costs $m_1, m_2, \ldots, m_k$ produces redundant proof obligations:

\[
\begin{align*}
\text{n} & \text{ pay } m_1 & \Rightarrow n - m_1 \geq 0 \\
(n - m_1) & \text{ pay } m_2 & \Rightarrow n - m_1 - m_2 \geq 0 \\
& \ldots & \\
(n - m_1 - m_2 - \ldots - m_{k-1}) & \text{ pay } m_k & \Rightarrow n - m_1 - m_2 - \ldots - m_k \geq 0
\end{align*}
\]
We work in a variant of SL with credits counted in $\mathbb{Z}$:

$$0 \equiv true$$

$$\underbrace{m + n}_\text{true} \equiv \underbrace{m \ast n}_\text{true}$$

$$\underbrace{n \ast [n \geq 0]}_\text{true}$$

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

$$0 \equiv n \ast (-n)$$

$$n \equiv (n - m) \ast m$$

Negative credits are not affine!
Time Credits in $\mathbb{Z}$ (2)

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

$$n$$

pay $m_1$ $\implies$ no proof obligation

$$(n - m_1)$$

pay $m_2$ $\implies$ no proof obligation

... $$(n - m_1 - \ldots - m_{k-1})$$

pay $m_k$ $\implies$ no proof obligation

discard $$(n - m_1 - \ldots - m_k)$$ $\implies n - m_1 - \ldots - m_k \geq 0$$
With integer time credits, these two specifications are equivalent (using the frame rule):

\[
\{\$n\} \ f \ n \ \{\lambda(). \ emp\}
\]
\[
\{emp\} \ f \ n \ \{\lambda(). \ $(\$n)\}
\]

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

\[
\{emp\} \ \text{collatz\_stopping\_time} \ n \ \{\lambda. \ $(\$i)\}
\]
Interaction with loops

From the proof of the forward traversal:

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

while ... do

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

[pay for operations using $\phi(G, L')$]

... done

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

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

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

```ocaml
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:

@I\[p@x l_1 l_1 u n t\]\I[l_1 u f x t b:\I[p@x :: l_1 q q u n t\I[rs:|l|u interruptible_iter f l t b:if b then I[l] else D D l_1 l_2 :I[l_1 l_2 su]]
Interruptible Iteration

```plaintext
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:

\[
\forall I \ l \ f.
(\forall x \ l'. \ \text{prefix} l' l \implies \{I \ l'\} \ f \ x \ \{\lambda b. \ I \ (x :: l')\}) \implies \{I [] \ast \$|l|\}
i interruptible_iter f l
\{\lambda b. \text{if } b \text{ then } I \ l \text{ else } \exists l'' \ l''. \ I \ l' \ast \$|l''| \ast [l = l' ++ l'']\}
\]
Future Work
Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

\[ (?c) \ast \text{potential} \models cost_1 \ast \ldots \ast cost_n \ast ?F \]

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

\[ (?c) \ast \text{potential} \ast (\neg cost_1) \ast \ldots \ast (\neg cost_n) \models ?F \]

Is this useful?...
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).
https://gitlab.inria.fr/agueneau/incremental-cycles