Formally verified incremental cycle detection

Armaël Guéneau

with J.-H. Jourdan, A. Charguéraud and F. Pottier
Can we formally verify the *functional correctness*...
Can we formally verify the *functional correctness* and *asymptotic complexity*...
Can we formally verify the *functional correctness* and *asymptotic complexity* of *non-trivial* algorithms...
Can we formally verify the *functional correctness* and *asymptotic complexity* of *non-trivial* algorithms with respect to concrete source code?
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;
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

```ocaml
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)

\[\text{INITGRAPH}\]
\[
\{\text{emp}\} \text{ init\_graph()} \{\lambda g. \text{IsGraph } g \emptyset\}
\]

\[\text{ACYCLICITY}\]
\[
\forall g G. \text{IsGraph } g G \models \text{IsGraph } g G * [\forall x. x \rightarrow G x]
\]
AddVertex

∀g G v. v ∉ vertices G ⇒

{IsGraph g G ⊗ IsNewVertex v}

(add_vertex g v)

{λ(). IsGraph g (G + v) }
**ADD EDGE**

∀g G v w. let m := |edges G| in

let n := |vertices G| in

v, w ∈ vertices G ∧ (v, w) ∉ edges G ⇒

\{ \text{IsGraph } g G \}

(\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] \}
Separation Logic with Time Credits:

- $n$ asserts the ownership of $n$ time credits
- $n$ is a Separation Logic assertion, like $p \leftrightarrow 3$
- Each function call (or loop iteration) consumes 1
- $(n + m) \equiv n \star m$
- Credits are not duplicable: $1 \not\rightarrow 1 \star 1$
- Specifications are of the form:

$$\{ \text{IsGraph } g \ G \star \$(3 \lvert \text{edges } G \rvert + 5) \} \ \text{dfs } g \ \{ \text{IsGraph } g \ G \}$$
Toplevel specification (correctness and complexity) (1)

**ADDEdge**

\[ \forall g \in G \forall v w. \quad \text{let } m := |\text{edges } G| \text{ in} \]

\[ \text{let } n := |\text{vertices } G| \text{ in} \]

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

\[ \{ \text{IsGraph } g \in G \star $( \ldots ) \} \]

(add_edge_or_detect_cycle \( g \) \( v \) \( w \))

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

\[ \{ \text{EdgeAdded } \implies \text{IsGraph } g \in (G + (v, w)) \} \]

\[ \{ \text{EdgeCreatesCycle } \implies [w \longrightarrow^* G v] \} \]
**Toplevel specification (correctness and complexity) (1)**

**ADDEDGE**

\[ \forall g \in G \, v \, w. \quad \text{let } m := |\text{edges } G| \text{ in} \]

\[ \text{let } n := |\text{vertices } G| \text{ in} \]

\[ v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \quad \Rightarrow \]

\[ \begin{cases} 
\text{IsGraph } g \, G \quad \star \quad \$ (\psi (m + 1, n) - \psi (m, n)) \\
(\text{add_edge_or_detect_cycle } g \, v \, w) 
\end{cases} \]

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

\[ \begin{cases} 
| \text{EdgeAdded} \Rightarrow \text{IsGraph } g \,(G + (v, w)) \\
| \text{EdgeCreatesCycle} \Rightarrow [w \rightarrow^*_G v] 
\end{cases} \]

**Complexity**

\[ \psi \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n) \land \text{nonnegative } \psi \land \text{monotonic } \psi \]
**AddVertex**

\[ \forall g \; G \; v. \; \text{let } m := |\text{edges } G| \; \text{ in} \]

\[ \text{let } n := |\text{vertices } G| \; \text{ in} \]

\[ v \not\in \text{vertices } G \implies \]

\[ \begin{cases} 
\text{IsGraph } g \; G \star \text{IsNewVertex } v \star \\
\psi(m, n + 1) - \psi(m, n) \\
(\text{add_vertex } g \; v) \\
\lambda(). \text{IsGraph } g \; (G + v) 
\end{cases} \]
**Toplevel specification (correctness and complexity) (3)**

**InitGraph**

\[ \exists k. \{ k \} \text{ init\_graph() } \{ \lambda g. \text{IsGraph } g \varnothing \} \]

**Acyclicity**

\[ \forall g G. \text{IsGraph } g G \models \text{IsGraph } g G \star [\forall x. x \rightarrow \overset{G}{\rightarrow} x] \]

**DisposeGraph**

\[ \forall g G. \text{IsGraph } g G \models \text{emp} \]
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; \( \psi(0, 1) - \psi(0, 0) \)
...
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

```ocaml
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

\[ \forall g \in G \forall v, w. \]
\[
\text{let } m, n := |\text{edges } G|, |\text{vertices } G| \text{ in }
\]
\[
v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \implies
\]
\[
\{ \text{IsGraph } g \in G \ast (\psi(m + 1, n) - \psi(m, n)) \}
\]
\[
(\text{add\_edge\_or\_detect\_cycle } g \in G \in v, w)
\]
\[
\lambda \text{res. match res with}
\]
\[
\{ \begin{align*}
| \text{EdgeAdded} & \implies \text{IsGraph } g\ (G + (v, w)) \\, \text{\{IsRawGraph } g\ GLMI \implies \text{Inv } GLI \colon \psi (G, L) \equiv \text{Inv } GLI: x.x \mapsto G x q ^ & \end{align*}
\}
\]
\[
| \text{EdgeCreatesCycle} & \implies [w \mapsto _G^* v]
\}
\]
IsGraph’s hidden potential

∀g G v w.
let m, n := |edges G| , |vertices G| in
v, w ∈ vertices G ∧ (v, w) ∉ edges G →
\{ IsGraph g G × $(ψ (m + 1, n) − ψ (m, n)) \} 
(add_edge_or_detect_cycle g v w)

λ res. match res with
\{ | EdgeAdded ⇒ IsGraph g (G + (v, w)) 
| EdgeCreatesCycle ⇒ [w ñ ñ 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 → ñ G x) ∧ …
IsGraph’s hidden potential

\[ \forall g \ G \ L \ M \ I \ v \ w. \]

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

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

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

\((\text{add_edge_or_detect_cycle } g \ v \ w)\)

\(\lambda \text{res. match res with}\)

\[ \begin{array}{l}
| \text{EdgeAdded } \Rightarrow \text{let } G' := G + (v, w) \text{ in } \exists L' M' I'. \\
\quad \text{IsRawGraph } g \ G' \ L' M' I' \ast [\text{Inv } G' \ L' I'] \ast \psi(G', L') \\
| \text{EdgeCreatesCycle } \Rightarrow [w \longrightarrow^*_{G} v]
\end{array} \]
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 $\phi$.

Adding the new edge:
increases the potential $\phi$, 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:

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

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 $v \rightarrow w$, 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 $\psi$ (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 $\phi$ 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 $\phi$, 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.

$$\phi((G + (v, w)), L) - \phi(G, L) = C \cdot (\sqrt{m + 1} - L(v))$$

$$\leq C \cdot \sqrt{m + 1}$$

$$\in O(\psi(m + 1, n) - \psi(m, n))$$

QED.
Verification Techniques and Methodology
We define $\phi$ and $\psi$ as follows:

\[
\phi(G, L) := C \cdot \sum_{(v, w) \in G} (\sqrt{m} - L(v)) \\
\psi(m, n) := C' \cdot (m \sqrt{m} + m + n + 1)
\]

...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:

\[ \exists C. \forall g w l. \{ (C \cdot (\ldots)) \star \ldots \} \text{ update_levels } g w l \{ \ldots \} \]

- Defer choosing a value for \( C \);
- Cost synthesis yields obligations of the form “\( C \geq \text{cost}_\text{foo} + \text{cost}_\text{bar} + \ldots \)”: defer them;
- Automatically deduce a suitable value for \( C \).

Then, \( \phi \) is defined using “the” \( C \) from the specification.
Time Credits in \( \mathbb{Z} \)

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

\[
0 \equiv \text{emp} \\
\forall m \, n \in \mathbb{N}. \quad (m + n) \equiv (m \star n) \\
\forall n \in \mathbb{N}. \quad n \not|\not\exists \text{emp}
\]

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

\[
0 \equiv \text{emp} \\
\forall m \, n \in \mathbb{Z}. \quad (m + n) \equiv (m \star n) \\
\forall n \in \mathbb{Z}. \quad n \star [n \geq 0] \not|\not\exists \text{emp}
\]
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...”

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

“...as long as any debt incurred is eventually paid off.”

$$\forall n \in \mathbb{Z}. \; \$n \star [n \geq 0] \Vdash \text{emp}$$
Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

```ocaml
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)

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

∀l. let k := “index of the first 0 in l” in
{$(k + 1)} walk l {λl'. [suffix l' l]}
Time Credits in \( \mathbb{Z} \) enable simpler specifications (& invariants)

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

\[ \forall l. \{ l \to (l| + 1) \} \text{ walk } l \{ \lambda l'. l'| * [\text{suffix } l' l] \} \]
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
```

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

\[
\forall l. \{\text{emp}\} \text{ walk } l \{\lambda l'. \$(|l'| - |l| - 1) \star [\text{suffix } l' l]\}
\]
Time Credits in $\mathbb{Z}$ enable simpler specifications (& invariants)

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

$\forall l. \{ |l| + 1 \} \quad \text{walk} \; l \quad \{ \lambda l'. |l'| \ast [\text{suffix} \; l' \; l] \}$

$\forall l. \{ \text{emp} \} \quad \text{walk} \; l \quad \{ \lambda l'. |l'| - |l| - 1 \ast [\text{suffix} \; l' \; l] \}$

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.

⇒ 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

OCaml program

CFML generator

(generated) characteristic formulae

+ (hand written) Specifications and proofs
Example specifications using time credits

Complexity specification using explicit time credits:

\[ \forall g \in G. \{ \text{IsGraph } g \in G \} \Rightarrow (3|\text{edges } G| + 5) \} \text{dfs}(g) \{ \text{IsGraph } g \in G \} \]

Asymptotic complexity specification:

\[ \exists (f : \mathbb{Z} \rightarrow \mathbb{Z}). \]
\[ f \in O_{\mathbb{Z}}(\lambda m. m) \]
\[ \land \forall g \in G. \{ \text{IsGraph } g \in G \Rightarrow f(|\text{edges } G|) \} \text{dfs}(g) \{ \text{IsGraph } g \in G \} \]
Idea 1: Levels

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

Invariant: \( v \xrightarrow{G} w \implies L(v) \leq L(w) \)

Levels can accelerate the search, but need to be maintained:
Idea 1: Levels

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

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

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, outweighing 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 $\text{card}\{w \mid (v, w) \in G\}$ 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.
$$out(v) := \text{card}(\{w \mid (v,w) \in G\})$$

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

```ml
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
```
\[ \text{out}(v) := \text{card}(\{w \mid (v, w) \in G\}) \]
\[ |\text{stack}| := \sum_{v \in \text{stack}} \text{out}(v) \]

Let recursively define \( \varphi(G, L) \) as:

\[ \varphi(G, L) = (\text{out}(u) + |\text{stack}|) \]

\[ \text{let rec visit_forward } g \text{ new_level visited stack } = \]
\[ \text{match stack with } \]
\[ | [] \rightarrow () \]
\[ | u :: \text{stack} \rightarrow \]

\[ \text{let stack } = \text{List.fold_left (fun stack } v \rightarrow \]
\[ \ldots \text{ | } \text{stack} \]
\[ \text{set_level } g \text{ } v \text{ new_level}; \]
\[ v :: \text{stack} \]

\[ \) stack (get_outgoing } g \text{ } u) \text{ in } \]
\[ \text{visit_forward } g \text{ new_level visited stack} \]
Proof methodology, in practice

In practice, credit counts involve multiplicative constants:

\[
\phi(G, L) := C \cdot \sum_{(u,v) \in G} (\text{highest}_\text{level} \ G \ L - L(u)) \\
|\text{stack}| := C' \cdot \sum_{v \in \text{stack}} \text{out}(v)
\]

\[
\exists C''. \ 0 \leq C'' \land \forall g \ \text{nl vs stack} \ldots.
\]

\[
\{C'' \ast |\text{stack}| \ast \ldots\} \text{ visit}_\text{forward} g \ \text{nl vs stack} \{\lambda(). \ldots\}
\]

\(C, C'\) 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.
Time Credits in $\mathbb{N}$ and redundant proof obligations

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

\[
\begin{align*}
& n \\
& \quad \text{pay } m_1 \quad \implies n - m_1 \geq 0 \\
&(n - m_1) \\
& \quad \text{pay } m_2 \quad \implies n - m_1 - m_2 \geq 0 \\
& \quad \vdots \\
&(n - m_1 - m_2 - \ldots - m_{k-1}) \\
& \quad \text{pay } m_k \quad \implies n - m_1 - m_2 - \ldots - m_k \geq 0
\end{align*}
\]
Time Credits in $\mathbb{Z}$ eliminate redundant proof obligations

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$$

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):

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

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

\[
\{\ emp\} \ \text{collatz\_stopping\_time} \ n \ \{\lambda i. \ ($(-i))\}
\]
From the proof of the forward traversal:

```plaintext
// $\phi(G, L) \star [\text{Inv } G \ L \ I]
List.fold_left ... (fun ... ->
  // $\exists L'. \phi(G, L')
  [\text{extract credits from } \phi(G, L')] 
  ...
)
// $\phi(G, L'') \star [\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.
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
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

∀a i A. 0 ≤ i ≤ |A| ⇒
{a ↦ Array A} walk a i {λj. a ↦ Array A ⋆ $(i − j) ⋆ [i < j ≤ |A|]}
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

\(\forall a \ i \ A. \ 0 \leq i \leq |A| \implies \{a \leadsto Array \ A\} \ \text{walk} \ a \ i \ \{\lambda j. \ a \leadsto Array \ A \star $(i-j) \star [i < j \leq |A|]\}\)

\(\forall a \ i \ A. \ 0 \leq i \leq |A| \implies \{a \leadsto Array \ A \star $(|A| - i)\} \text{walk} \ a \ i \ \{\lambda j. \ a \leadsto Array \ A \star $(|A| - j) \star [i < j \leq |A|]\}\)
Interruptible Iteration

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

\[
\forall I \ l \ f.
(\forall x \ l'. \prefix l' \ l \implies \{ I \ l' \} \ f \ x \ \{ \lambda b. I \ (x :: l') \}) \implies
\{ I \ [] \star \$|l|\}
\]

\[
\text{interruptible_iter} \ f \ l
\]

\[
\{ \lambda b. \text{if } b \text{ then } I \ l \text{ else } \exists l'' \ l'. \ l' \star \$|l''| \star \l = l' ++ l'' \}\}
```
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).
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 \( \text{vertex} \rightarrow \mathbb{Z} \)
- \( M \): marks, as a map \( \text{vertex} \rightarrow \text{mark} \)
- \( I \): horizontal incoming edges, a map \( \text{vertex} \rightarrow \text{set vertex} \)
Functional Invariant

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

Inv $G \ L \ I :=$

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

enough\_edges\_below\_G\_L\_x :=

\[
|\text{coacc\_edges\_at\_level\_G\_L\_k\_x}| \geq k \quad \text{where } k = L(x) - 1
\]

coacc\_edges\_at\_level\_G\_L\_k\_x :=

\[
\{ (y, z) \mid y \rightarrow G z \rightarrow^{*} G x \land L(y) = L(z) = k \}
\]
Potential and Advertised Cost (formally)

Potential of an edge \((u, v)\): \(\max_{\text{level}} mn - L(u)\).

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

where \(m = |\text{edges } G|\) and \(n = |\text{vertices } G|\).
Proof methodology

Specification excerpt for the backward traversal:

$$\exists a b. \ 0 \leq a \land \forall F \ g \ v \ w \ \ldots \ \{ (a \cdot F + b) \ast \ldots \} \ \text{backward_search} \ F \ g \ v \ w \ \{ \lambda res. \ \ldots \}$$
Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

\[(?c) \star \mathit{potential} \vdash cost_1 \star \ldots \star cost_n \star ?F\]

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

\[(?c) \star \mathit{potential} \star (-cost_1) \star \ldots \star (-cost_n) \vdash ?F\]

Is this useful?...
Automation for processing synthesized cost expressions

Credit synthesis produces in the end goals of the form:

\[ \exists f. \ldots f \ldots \]
\[ \exists a \ b. \ldots a \ldots b \ldots \]

Where “…” usually:

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