

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...

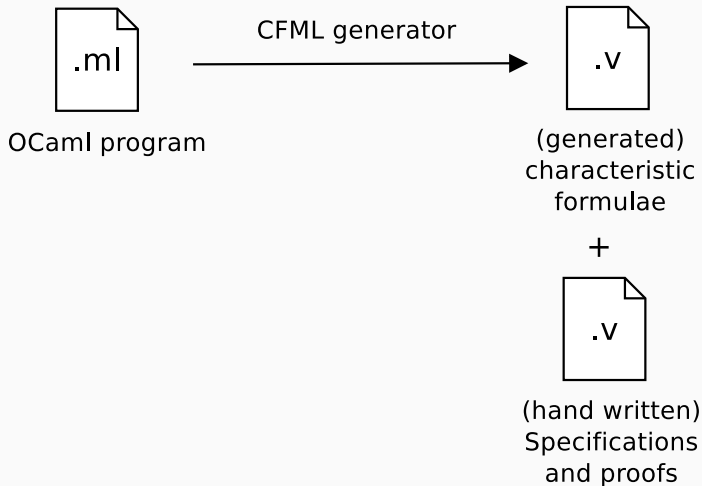
- initially motivated by Coq's implementation...
(universe constraints)
- ...which ended up integrated into the Dune build system.
(dependencies between build actions)

And our verified implementation turned out to be 7x faster!

Context

Program verification...

Our verification framework: Coq and CFML



...using Separation Logic with Time Credits

Example specification, using explicit time credits:

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

Asymptotic complexity specification:

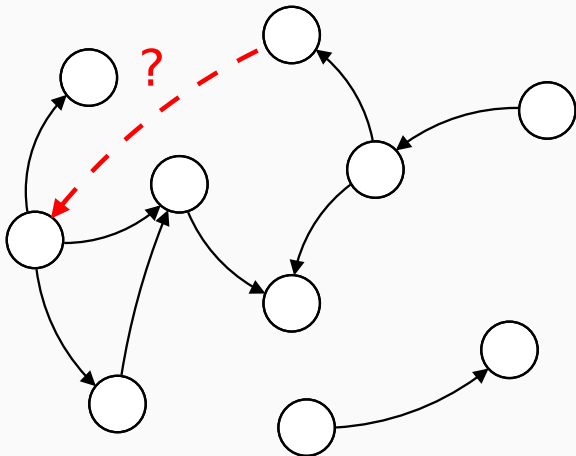
$$\exists (f : \mathbb{Z} \rightarrow \mathbb{Z}).$$

$$\text{nonnegative } f \wedge \text{monotonic } f \wedge f \in O_{\mathbb{Z}}(\lambda m.m)$$

$$\wedge \forall g G. \{ \text{IsGraph } g \ G * \$ f(|\text{edges } G|) \} \text{dfs}(g) \{ \text{IsGraph } g \ G \}$$

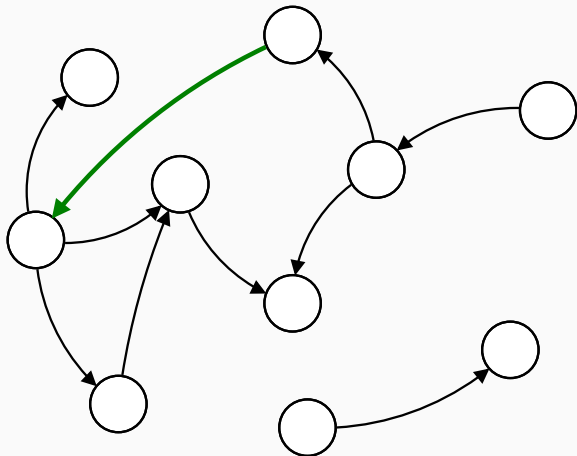
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



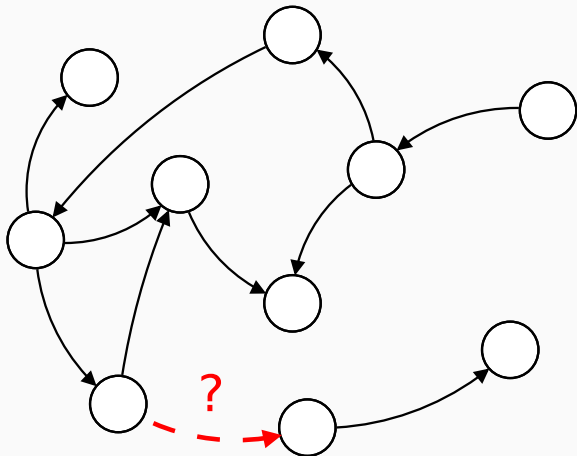
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



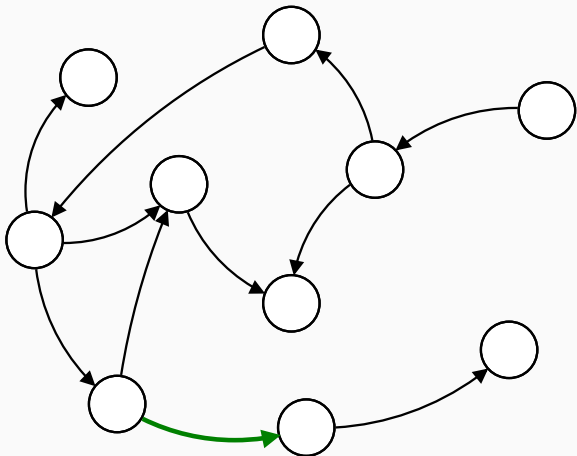
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



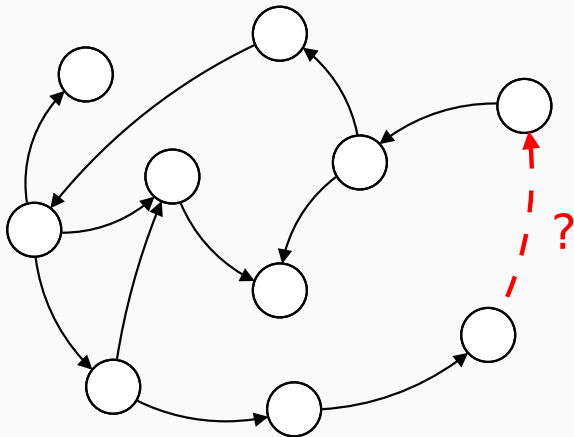
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



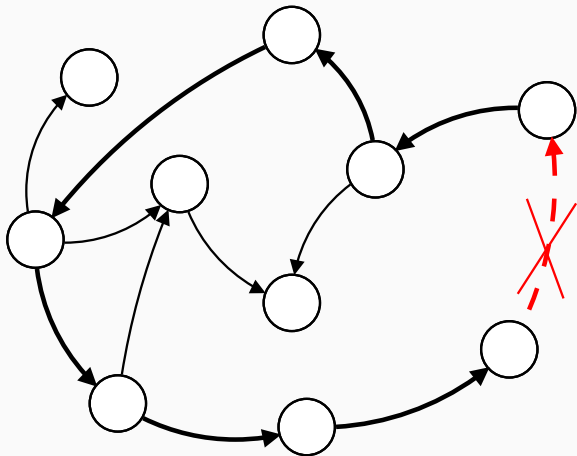
Incremental cycle detection

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

~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)

$$\begin{array}{l} \forall g G v w. \text{ let } m := |\text{edges } G| \text{ in} \\ \quad \text{let } n := |\text{vertices } G| \text{ in} \\ \quad v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ \quad \left\{ \begin{array}{l} \text{IsGraph } g G \\ \\ \text{(add_edge_or_detect_cycle } g v w) \\ \\ \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \xrightarrow{*}_G v] \end{array} \right\} \end{array}$$

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

Toplevel specification (functional correctness only)

$$\begin{aligned} & \forall g G v w. \text{ let } m := |\text{edges } G| \text{ in} \\ & \quad \text{let } n := |\text{vertices } G| \text{ in} \\ & \quad v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ & \quad \left\{ \begin{array}{l} \text{IsGraph } g G \\ \text{(add_edge_or_detect_cycle } g v w) \end{array} \right\} \\ & \quad \left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \xrightarrow{*}_G v] \end{array} \right\} \end{aligned}$$

$$\text{IsGraph } g G := \exists L M I. \text{IsRawGraph } g G L M I * [\text{Inv } G L I]$$
$$\text{Inv } G L I := (\forall x. x \dashrightarrow_G^+ x) \wedge \dots$$

Toplevel specification (correctness and complexity)

$$\begin{aligned} & \forall g G v w. \text{ let } m := |\text{edges } G| \text{ in} \\ & \quad \text{let } n := |\text{vertices } G| \text{ in} \\ & \quad v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ & \quad \left\{ \begin{array}{l} \text{IsGraph } g G \\ \text{(add_edge_or_detect_cycle } g v w) \end{array} \right\} \\ & \quad \left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \xrightarrow{*}_G v] \end{array} \right\} \end{aligned}$$

$$\text{IsGraph } g G := \exists L M I. \text{ IsRawGraph } g G L M I * [\text{Inv } G L I] * \$\phi(G, L)$$
$$\text{Inv } G L I := (\forall x. x \dashrightarrow_G^+ x) \wedge \dots$$

Toplevel specification (correctness and complexity)

$\forall g G v w$. let $m := |\text{edges } G|$ in
let $n := |\text{vertices } G|$ in
 $v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$

$$\left\{ \begin{array}{l} \text{IsGraph } g G * \psi(m+1, n) - \psi(m, n) \\ (\text{add_edge_or_detect_cycle } g v w) \\ \left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \Rightarrow \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \Rightarrow [w \xrightarrow*_G v] \end{array} \right\} \end{array} \right\}$$

$\text{IsGraph } g G := \exists L M I. \text{IsRawGraph } g G L M I * [\text{Inv } G L I] * \phi(G, L)$

$\text{Inv } G L I := (\forall x. x \xrightarrow+_G x) \wedge \dots$

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

Using the specification

<code>let g = create_graph () in</code>	
<code>add_vertex g 1;</code>	$\$(\psi(0, 1) - \psi(0, 0))$
<code>...</code>	
<code>add_vertex g n;</code>	$\$(\psi(0, n - 1) - \psi(0, n))$
<code>add_edge_or_detect_cycle g 1 2;</code>	$\$(\psi(1, n) - \psi(0, n))$
<code>add_edge_or_detect_cycle g 2 3;</code>	$\$(\psi(2, n) - \psi(1, n))$
<code>...</code>	
<code>add_edge_or_detect_cycle g (m-1) m;</code>	$\$(\psi(m, n) - \psi(m - 1, n))$

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

Using the specification

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

In the rest of this talk

Bender et al.'s algorithm: Key Ideas

Formalization: Data Structure Invariants and Potential

Proof Methodology

Integer Time Credits

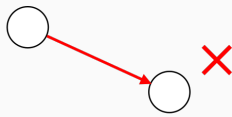
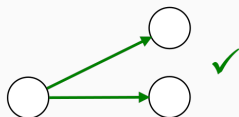
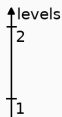
Future Work

Bender et al.'s algorithm: Key Ideas

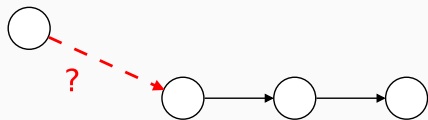
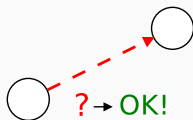
Idea 1: Levels

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

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



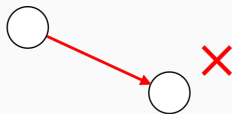
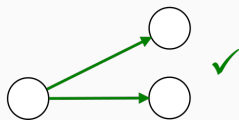
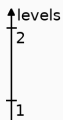
Can accelerate the search, but needs to be maintained:



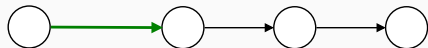
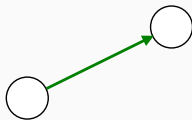
Idea 1: Levels

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

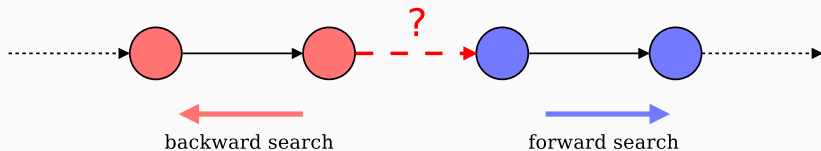
Invariant: $v \rightarrow_G w \implies L(v) \leq L(w)$



Can accelerate the search, but needs to be maintained:



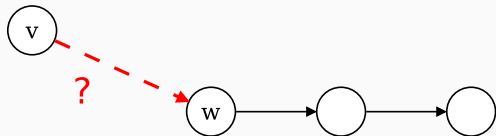
Idea 2: Two-way Search



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

$\forall g G v w.$

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

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

$\left\{ \text{IsGraph } g G * \$(\psi(m+1, n) - \psi(m, n)) \right\}$

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

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \longrightarrow_G^* v] \end{array} \right\}$

$\forall g G L M I v w.$

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

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

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

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

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ | \text{EdgeAdded} \implies \text{let } G' := G + (v, w) \text{ in } \exists L' M' I'. \\ \quad \text{IsRawGraph } g G' L' M' I' * [\text{Inv } G' L' I'] * \$\phi(G', L') \\ | \text{EdgeCreatesCycle} \implies [w \longrightarrow_G^* v] \end{array} \right\}$

Low-level Data Structure

$\text{IsRawGraph } g \ G \ L \ M \ I$: 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

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

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

$$\left\{ \begin{array}{ll} \text{acyclicity:} & \forall x. \quad x \not\rightarrow_G^+ x \\ \text{positive levels:} & \forall x. \quad L(x) \geq 1 \\ \text{pseudo-topological levels:} & \forall x y. \quad x \rightarrow_G y \implies L(x) \leq L(y) \\ \text{incoming edges:} & \forall x y. \quad x \in I(y) \iff x \rightarrow_G y \wedge L(x) = L(y) \\ \text{replete levels:} & \forall x. \quad \text{enough_edges_below } G L x \end{array} \right.$$

$\text{enough_edges_below } G L x :=$

$$|\text{coacc_edges_at_level } G L k x| \geq k \quad \text{where } k = L(x) - 1$$

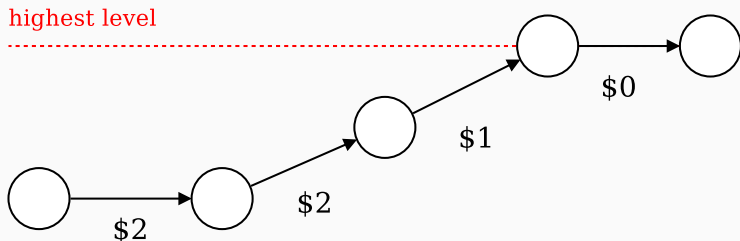
$\text{coacc_edges_at_level } G L k x :=$

$$\{ (y, z) \mid y \rightarrow_G z \rightarrow_G^* x \wedge L(y) = L(z) = k \}$$

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

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

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

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

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ | \text{EdgeAdded} \implies \text{let } G' := G + (v, w) \text{ in } \exists L' M' I'. \\ \quad \text{IsRawGraph } g G' L' M' I' * [\text{Inv } G' L' I'] * \$\phi(G', L') \\ | \text{EdgeCreatesCycle} \implies [w \longrightarrow_G^* v] \end{array} \right\}$

Potential and Advertised Cost (formally)

Potential of an edge (u, v) : $\max_level\ m\ n - L(u)$.

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

$$\text{spent } G\ L := \sum_{(u,v) \in \text{edges } G} L(u)$$

$$\text{received } m\ n := m \cdot (\max_level\ m\ n + 1)$$

$$\max_level\ m\ n := \min(\lceil (2m)^{1/2} \rceil, \lfloor (\frac{3}{2}n)^{2/3} \rfloor) + 1$$

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

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:

$$\exists a \ b. 0 \leq a \wedge \forall F \ g \ v \ w \ \dots$$

$$\{ \$ (a \cdot F + b) * \dots \} \text{ backward_search } F \ g \ v \ w \ \{ \lambda \text{res. } \dots \}$$

We want the proof to be independent from their exact expression, and avoid writing it explicitly by hand.

Demo

Integer Time Credits

Time Credits and redundant proof obligations

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

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

Corollary:

$$\$n \equiv \$(n - m) * \$m \quad \mathbf{if} \ m \leq n$$

Time Credits and redundant proof obligations (2)

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

$\$n$

pay $\$m_1$

$$\rightsquigarrow n - m_1 \geq 0$$

$\$(n - m_1)$

pay $\$m_2$

$$\rightsquigarrow n - m_1 - m_2 \geq 0$$

...

$\$(n - m_1 - m_2 - \dots - m_{k-1})$

pay $\$m_k$

$$\rightsquigarrow n - m_1 - m_2 - \dots - m_k \geq 0$$

Time Credits in \mathbb{Z}

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

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

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:

$\$n$

pay $\$m_1$

\rightsquigarrow no proof obligation

$\$(n - m_1)$

pay $\$m_2$

\rightsquigarrow no proof obligation

...

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

pay $\$m_k$

\rightsquigarrow no proof obligation

discard $\$(n - m_1 - \dots - m_k)$

$\rightsquigarrow n - m_1 - \dots - m_k \geq 0$

Pre/Post-condition duality

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

$$\{\$n\} \text{ f } n \{ \lambda(). \text{emp} \}$$

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

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

$$\{\text{emp}\} \text{ collatz_stopping_time } n \{ \lambda i. \$(-i) \}$$

Interaction with loops

From the proof of the forward traversal:

```
//  $\phi(G, L) * [\text{Inv } G \ L \ I]$ 
while ... do
  //  $\exists L'. \phi(G, L')$ 
  [pay for operations using  $\phi(G, L')$ ]
  ...
done
//  $\phi(G, L'') * [\text{Inv } G \ L'' \ I'']$ 
```

(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

```
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:

$$\forall I l f.$$
$$(\forall x l'. \text{prefix } l' l \implies \{I l'\} f x \{\lambda b. I (x :: l')\}) \implies$$
$$\{I [] * \$|l|\}$$
$$\text{interruptible_iter } f l$$
$$\{\lambda b. \text{if } b \text{ then } I l \text{ else } \exists l' l''. I l' * \$|l''| * [l = l' ++ l'']\}$$

Future Work

Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

$$\$(?c) * \$potential \Vdash \$cost_1 * \dots * \$cost_n * ?F$$

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

$$\$(?c) * \$potential * \$(-cost_1) * \dots * \$(-cost_n) \Vdash ?F$$

Is this useful?...

Automation for processing synthesized cost expressions

Credit synthesis produces in the end goals of the form:

$$\begin{aligned} &\exists f. \quad \dots f \dots \\ &\exists a b. \quad \dots a \dots b \dots \end{aligned}$$

Where “...” usually:

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

<https://gitlab.inria.fr/agueneau/incremental-cycles>