Formally verified incremental cycle detection

Armaël Guéneau
with J.-H. Jourdan, A. Charguéraud and F. Pottier
In this talk

The story of a formally verified algorithm...

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

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

... and our implementation turns out to be 7x faster!
Context
Program verification framework: Coq and (extended) CFML

OCaml program

CFML generator

.(generated) characteristic formulae +

.(hand written) Specifications and proofs
Separation Logic with Time Credits

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

Type of assertions, in the model:

- Standard Separation Logic: Heap $\rightarrow$ Prop
- Separation Logic with Time Credits: Heap $\times \mathbb{N} \rightarrow$ Prop
Example specifications using time credits

Complexity specification using explicit time credits:

\[ \forall g \in G. \{ \text{IsGraph} \ g \ G \ \ast \ \$(3 \ \mid \text{edges} \ G \mid \ + \ 5) \} \ \text{dfs}(g) \ \{ \text{IsGraph} \ g \ G \} \]

Asymptotic complexity specification:

\[ \exists (f : \mathbb{Z} \rightarrow \mathbb{Z}). \]
\[ \text{nonnegative} \ f \ \land \ \text{monotonic} \ f \ \land \ f \in O_{\mathbb{Z}}(\lambda m. m) \]
\[ \land \ \forall g \in G. \ \{ \text{IsGraph} \ g \ G \ \ast \ \$ f (\mid \text{edges} \ G \mid) \} \ \text{dfs}(g) \ \{ \text{IsGraph} \ g \ G \} \]
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 (!).
In the rest of this talk

Overview of the library: interface and specification

Bender et al.’s algorithm: Key Ideas

Complexity Analysis

A taste of time credits: forward traversal analysis

Integer Time Credits
Overview of the library: interface and specification
Implementation

~150 lines of (terse) hand written OCaml code
Minimal OCaml interface

module Make (G : Raw_graph) : sig

  val add_vertex : G.graph -> G.vertex -> unit

  type add_edge_result =
    | EdgeAdded
    | EdgeCreatesCycle

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

end
∀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] \}
  \}

∀g G. IsGraph g G ⊢ IsGraph g G * [∀x. x ↠^+_G x]
∀g G v w. let m := |edges G| in
    let n := |vertices G| in
    \( v, w \in \text{vertices } G \land (v, w) \notin \text{edges } G \implies \)

\[
\left\{ \begin{array}{l}
\text{IsGraph } g \ G \\
(\text{add_edge_or_detect_cycle } g \ v \ w)
\end{array} \right. \\
\lambda \text{res. match res with} \\
\text{\{ | EdgeAdded } \Rightarrow \text{IsGraph } g \ G + (v, w) \\
\text{\{ | EdgeCreatesCycle } \Rightarrow [w \overset{*}{\rightarrow}_{G} v] \}
\]

IsGraph g G := \exists L M I. \text{IsRawGraph } g \ G L M I \ast [\text{Inv } G L I]

Inv G L I := (\forall x. x \overset{\rightarrow}{G} x) \land \ldots
Toplevel specification (correctness and complexity)

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

\[
\begin{cases}
\text{IsGraph } g \; G \\
(\text{add\_edge\_or\_detect\_cycle } g \; u \; w)
\end{cases}
\]

\[
\lambda \text{res. match res with}
\begin{cases}
| \text{EdgeAdded} \Rightarrow \text{IsGraph } g \; (G + (u, w)) \\
| \text{EdgeCreatesCycle} \Rightarrow \left[ w \rightarrow^*_G v \right]
\end{cases}
\]

\[
\text{IsGraph } g \; G := \exists L \; M \; I. \; \text{IsRawGraph } g \; G \; L \; M \; I \ast [\text{Inv } G \; L \; I] \ast \$\phi(G, L)
\]
Inv \( G \; L \; I := (\forall x. x \rightarrow^*_G x) \wedge \ldots \)
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 ⇒

\{ \begin{align*}
&\text{IsGraph } g \ G \ast \$(\psi(m + 1, n) − \psi(m, n)) \\
\end{align*} \}

(add_edge_or_detect_cycle g v w)

\begin{align*}
&\lambda \text{res. match res with} \\
&\begin{cases}
&\text{EdgeAdded} \Rightarrow \text{IsGraph } g \ (G + (v, w)) \\
&\text{EdgeCreatesCycle} \Rightarrow [w \rightarrow^{*} v]
\end{cases}
\end{align*}

IsGraph g G := ∃L M I. IsRawGraph g G L M I ∗ [Inv G L I] ∗ $\phi(G, L)$

Inv G L I := (∀x. x →^{+} G x) ∧ …

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

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

let g = create_graph () in
add_vertex g 1;  
\[\psi(0, 1) - \psi(0, 0)\]
...
add_vertex g n;  \[\psi(0, n) - \psi(0, n - 1)\]
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:  \[\psi(m, n) - \psi(0, 0) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)\]
Bender et al.’s algorithm: Key Ideas
Idea 1: Levels

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

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

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 1 (bis): Tradeoff on the number of levels

- Too many levels: the expensive case triggers often, outweights the cheap case
- Too few levels: similar to the naive algorithm, no gain from the cheap case
Idea 2: Two-way Search

The backward search is:

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

The forward search restores the invariant on levels as it goes.
Not explained at this point: when do new levels get created?

Let’s see: Demo!
Main complexity invariant: levels are “replete”

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

Corollary: there is at least \( k \) edges at level \( k \).
Complexity Analysis
∀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]) \}
\[ \forall g \; G \subseteq L \subseteq M \subseteq I \; v, w. \]

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

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

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

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

\[ \lambda \text{res. match res with} \left\{ \begin{array}{l}
| \text{EdgeAdded} \Rightarrow \text{let } G' := G + (v, w) \text{ in } \exists L' \subseteq M' \subseteq I'. \\
\quad \text{IsRawGraph } g \; G' \subseteq L' \subseteq M' \subseteq I' \ast [\text{Inv } G' \subseteq L' \subseteq M' \subseteq I']\ast \$\phi(G', L') \\
| \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow^*_G v])
\end{array} \right\} \]
A sketch of the complexity analysis

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

Good choice for $F$

$\implies$ Main complexity invariant (levels are “replete”)

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

In Bender et al., $F$ depends on the final graph; we give an alternative definition that works on the current graph.
A sketch of the complexity analysis

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

Good choice for $F$

$\implies$ Main complexity invariant (levels are “replete”)

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

In Bender et al., $F$ depends on the final graph; we give an alternative definition that works on the current graph.
A taste of time credits: forward traversal analysis
The graph potential $\phi$

The potential $\phi$ stores Time Credits for edges depending on their current level (lower level = more credits). Credits are received at each edge insertion, and spent when raising nodes.

$$\phi(G, L) := \sum_{(u,v) \in G} (\text{highest_level}_G L - L(u))$$
Forward traversal economics

- Traversing an edge \((u, v)\) costs 1
- Raising \(v\) releases \(\text{card}\{w \mid (v, w) \in G\}\) 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.
\[ \text{out}(v) := \text{card}(\{ w | (v, w) \in G \}) \]

\[ |\text{stack}| := \sum_{v \in \text{stack}} \text{out}(v) \]

```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
```
out(v) := card({w | (v, w) ∈ G})

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

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

$\varphi(G,L)$

$(out(u) + |stack|)$

$(out(v) + |stack|)$
In practice, credit counts involve multiplicative constants:

\[
\phi(G, L) := C \cdot \sum_{(u,v) \in G} (\text{highest\_level } G \cdot L - L(u))
\]

\[
|\text{stack}| := C' \cdot \sum_{v \in \text{stack}} \text{out}(v)
\]

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

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

\[C, C'\text{ and } C''\text{ depend on specifics of the implementation.}\]

We develop tactics to make the proofs independent from their exact expression, and avoid writing it explicitly by hand.
Integer Time Credits
Time Credits and redundant proof obligations

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

\[ 0 \equiv true \]
\[ (m + n) \equiv m \times n \]
\[ n \models true \]

Corollary:

\[ n \equiv (n - m) \times m \quad \text{if} \quad m \leq n \]
Time Credits and redundant proof obligations (2)

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

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

$$0 \equiv true$$
$$m + n \equiv m \times n$$
$$n \times [n \geq 0] \models true$$

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

$$0 \equiv n \times (-n)$$
$$n \equiv (n - m) \times m$$

Negative credits are not affine!
Paying for a sequence of operations produces a single final proof obligation:

\[
\begin{align*}
  & n \\
  & \text{pay } m_1 \quad \leadsto \text{ no proof obligation} \\
  & (n - m_1) \\
  & \text{pay } m_2 \quad \leadsto \text{ no proof obligation} \\
  & \cdots \\
  & (n - m_1 - \ldots - m_{k-1}) \\
  & \text{pay } m_k \quad \leadsto \text{ no proof obligation} \\
  & \text{discard } (n - m_1 - \ldots - m_k) \quad \leadsto n - m_1 - \ldots - m_k \geq 0
\end{align*}
\]
Pre/Post-condition duality

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

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

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

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

From the proof of the forward traversal:

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

```ocaml
@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:

@If:
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:

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

\[
\text{interruptible\_iter} \, f \, l
\}
\]

\[
\{ \lambda b. \, \text{if} \, b \, \text{then} \, I \, l \, \text{else} \, \exists l'''. \, I \, l' \, \ast \, \$|l''| \, \ast \, [l = l' \, \&\& \, l'''] \}
\]
Conclusion
Challenges

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

Code, proofs, paper and demo available at:

https://gitlab.inria.fr/agueneau/incremental-cycles
Idea 3: Policy for raising nodes to a new level

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

Bender et al.’s policy:

- If the backward search from $v$ was not interrupted: raised to $L(v)$
- Otherwise, raised to $L(v) + 1$ (possibly creating a new level).
Idea 4: choice of $F$

Recall: backward search is bounded to visit at most $F$ edges.
The choice of $F$ is crucial to get the correct complexity.

In Bender et al.:

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

In our modified algorithm:

$$F = L(v),$$
in the current graph
(this makes the algorithm truly online).
Low-level Data Structure

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 $G \ L \ I :=$

- **acyclicity:** $\forall x. \ x \rightarrow_{G}^{+} x$
- **positive levels:** $\forall x. \ L(x) \geq 1$
- **pseudo–topological levels:** $\forall x \ y. \ x \rightarrow_{G} y \implies L(x) \leq L(y)$
- **incoming edges:** $\forall x \ y. \ x \in I(y) \iff x \rightarrow_{G} y \land L(x) = L(y)$
- **replete levels:** $\forall x. \ enough\_edges\_below\ G \ L \ x$

**enough\_edges\_below G L x :=**

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

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

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

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

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

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

Specification excerpt for the backward traversal:

\[ \exists a \ b. \ 0 \leq a \land \forall F \ g \ v \ w \ \ldots \]
\[ \{ (a \cdot F + b) \ast \ldots \} \ \text{backward}\_\text{search} \ F \ g \ v \ w \ \{ \lambda res. \ \ldots \} \]
Future Work
Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

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

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

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

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

Credit synthesis produces in the end goals of the form:

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

Where “…” usually:

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