Machine-checked correctness and complexity of a Union-Find implementation

Arthur Chaguéraud    François Pottier

inria

December 16, 2015
Let’s begin with a demo...

Proving correctness and termination is not enough!
Verification methodology

We extend the **CFML** logic and tool with **time credits**.

This allows reasoning about the correctness and (amortized) complexity of realistic (imperative, higher-order) OCaml programs.
Separation Logic

Heap predicates:

\[ H : \text{Heap} \rightarrow \text{Prop} \]

Usually, Heap is loc \( \leftrightarrow \) value. The basic predicates are:

\[
\begin{align*}
[\lambda] & \quad \equiv \quad \lambda h. \ h = \emptyset \\
[P] & \quad \equiv \quad \lambda h. \ h = \emptyset \land P \\
H_1 \star H_2 & \quad \equiv \quad \lambda h. \ \exists h_1 h_2. \ h_1 \perp h_2 \land h = h_1 \uplus h_2 \land H_1 h_1 \land H_2 h_2 \\
\exists x. H & \quad \equiv \quad \lambda h. \ \exists x. \ H h \\
l \leftarrow v & \quad \equiv \quad \lambda h. \ h = (l \leftarrow v)
\end{align*}
\]
We wish to introduce a new heap predicate:

\[ n : \text{Heap} \rightarrow \text{Prop} \quad \text{where} \; n \in \mathbb{N} \]

Intended properties:

\[ (n + n’) = n \star n’ \quad \text{and} \quad 0 = [] \]

Intended use:

A time credit is a permission to perform “one step” of computation.
Connecting computation and time credits

Idea:

- Make sure that **every function call consumes one time credit**.
- Provide **no way of creating** a time credit.

Thus,

\[(\text{total number of function calls}) \leq (\text{initial number of credits})\]
Ensuring that every call consumes one credit

The CFML tool inserts a call to pay() at the beginning of every function.

```ocaml
let rec find x =
  pay();
  match !x with
  | Root _ -> x
  | Link y -> let z = find y in x := Link z; z
```

The function pay is fictitious. It is axiomatized:

\[
\text{App pay}(\text{})(1)(\lambda_.[])
\]

This says that \textbf{pay()} \textit{consumes one credit}. 
Contributions

- The first **machine-checked complexity analysis** of Union-Find.
- Not just at an abstract level, but **based on the OCaml code**.
- **Modular.** We establish a specification for clients to rely on.
The Union-Find data structure: OCaml interface

define type elem
val make : unit -> elem
val find : elem -> elem
val union : elem -> elem -> elem
The Union-Find data structure: OCaml implementation

Pointer-based, with path compression and union by rank:

```
let make () = ref (Root 0)

let rec find x = match !x with
  | Root _ -> x
  | Link y ->
    let z = find y in
    x := Link z;
    z

let link x y = if x == y then x else
  match !x, !y with
  | Root rx, Root ry ->
    if rx < ry then begin
      x := Link y;
      y
    end else if rx > ry then begin
      y := Link x;
      x
    end else begin
      y := Link x;
      x := Root (rx+1);
      x
    end
  end

let union x y = link (find x) (find y)
```
Complexity analysis

Tarjan, 1975: the **amortized** cost of union and find is $O(\alpha(N))$.

- where $N$ is a fixed (pre-agreed) bound on the number of elements.


\[
A_0(x) = x + 1
\]

\[
A_{k+1}(x) = A_k^{(x+1)}(x)
= A_k(A_k(...A_k(x)...)) \quad (x + 1 \text{ times})
\]

\[
\alpha(n) = \min\{k \mid A_k(1) \geq n\}
\]

Quasi-constant cost: for all practical purposes, $\alpha(n) \leq 5$. 
Specification of find

**Theorem** \( \text{find} \_\text{spec} : \forall \ N \ D \ R \ x, \ x \in D \rightarrow \)

\[
\text{App } \text{find } x
\]
\[
(UF \ N \ D \ R \star $(alpha \ N + 2))
\]
\[
(\text{fun } r \Rightarrow UF \ N \ D \ R \star \backslash[r = R \ x]).
\]

The **abstract predicate** \( UF \ N \ D \ R \) is the invariant. It asserts that the data structure is well-formed and that we own it.

- \( D \) is the set of all elements, i.e., the domain.
- \( N \) is a bound on the cardinality of the domain.
- \( R \) maps each element of \( D \) to its representative.
Specification of union

**Theorem** union_spec : \( \forall N \text{ D R x y}, x \in \text{D} \rightarrow y \in \text{D} \rightarrow \)

App union x y

(\( \text{UF N D R} \star\$(3 \ast(\text{alpha N})+6))\)

(\( \text{fun z} \Rightarrow \)

\( \text{UF N D (fun w} \Rightarrow \text{If R w} = \text{R x} \lor \text{R w} = \text{R y then z else R w)\)

\( \ast [z = \text{R x} \lor z = \text{R y}]\)).

The amortized cost of union is \(3\alpha(N) + 6\).
Definition of $\Phi$, on paper

$p(x) = \text{parent of } x$

$k(x) = \max\{k \mid K(p(x)) \geq A_k(K(x))\}$ (the level of $x$)

$i(x) = \max\{i \mid K(p(x)) \geq A_{k(x)}^{(i)}(K(x))\}$ (the index of $x$)

$\phi(x) = \alpha(N) \cdot K(x)$

$\phi(x) = (\alpha(N) - k(x)) \cdot K(x) - i(x)$

$\Phi = \sum_{x \in D} \phi(x)$

For some intuition, see Seidel and Sharir (2005).
Definition of $\Phi$, in Coq

Definition $p \ F \ x :=$
  \( \epsilon (\text{fun } y \Rightarrow F \ x \ y) \).

Definition $k \ F \ K \ x :=$
  \( \text{Max (fun } k \Rightarrow K (p \ F \ x) \geq A \ k \ (K \ x)) \).

Definition $i \ F \ K \ x :=$
  \( \text{Max (fun } i \Rightarrow K (p \ F \ x) \geq \text{iter } i (A \ (k \ F \ K \ x)) \ (K \ x)) \).

Definition $\phi \ F \ K \ N \ x :=$
  If \( \text{is\_root } F \ x \lor (K \ x = 0) \)
  then \( \text{alpha } N \ast (K \ x) \)
  else \( \text{alpha } N - k \ F \ K \ x \ast (K \ x) - (i \ F \ K \ x) \).

Definition $\Phi \ D \ F \ K \ N :=$
  Sum $D \ (\phi \ F \ K \ N)$.
Machine-checked amortized complexity analysis

Proving that the invariant is preserved naturally leads to this goal:

\[ \Phi + \text{advertised cost} \geq \Phi' + \text{actual cost} \]

For instance, in the case of \texttt{find}, we must prove:

\[ \text{Phi D F K N} + (\alpha N + 2) \geq \text{Phi D F'} K N + (d + 1) \]

where:

- \( F \) is the graph before the execution of \texttt{find } x,
- \( F' \) is the graph after the execution of \texttt{find } x,
- \( \alpha \) is the length of the path in \( F \) from \( x \) to its root.
A machine-checked proof of correctness and complexity.
Down to the level of the OCaml code.
3000 loc of high-level mathematical analysis.
400 loc of specification and low-level verification.
Future work: write $O(\alpha(n))$ instead of $3\alpha(n) + 6$.

http://gallium.inria.fr/~fpottier/dev/uf/