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

Arthur Charguéraud     François Pottier

December 16, 2015

# Message

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:

$$H : \mathsf{Heap} \to \mathsf{Prop}$$

Usually, Heap is loc $\mapsto$ value. The basic predicates are:

$$\begin{array}{lcl}
[\,] & \equiv & \lambda h.\ h = \varnothing \\
[P] & \equiv & \lambda h.\ h = \varnothing \ \wedge \ P \\
H_1 \star H_2 & \equiv & \lambda h.\ \exists h_1 h_2.\ \ h_1 \perp h_2 \ \wedge \ h = h_1 \uplus h_2 \ \wedge \ H_1\, h_1 \ \wedge \ H_2\, h_2 \\
\exists x.\, H & \equiv & \lambda h.\ \exists x.\ H\, h \\
l \hookrightarrow v & \equiv & \lambda h.\ h = (l \mapsto v)
\end{array}$$

# Separation Logic with time credits

We wish to introduce a new heap predicate:

$$\$\, n \quad : \quad \text{Heap} \to \text{Prop} \qquad\qquad \text{where } n \in \mathbb{N}$$

Intended properties:

$$\$(n + n') \;=\; \$\, n \;\star\; \$\, n' \qquad \text{and} \qquad \$\, 0 \;=\; [\,]$$

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,

$$\text{(total number of function calls)} \quad \leqslant \quad \text{(initial number of credits)}$$

# Ensuring that every call consumes one credit

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

```
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:

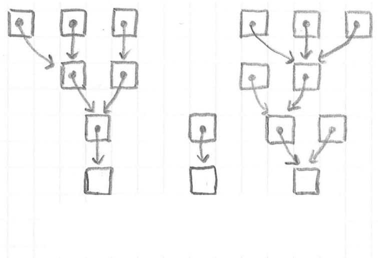
$$\mathsf{App}\;\mathsf{pay}\;()\;(\$\,1)\;(\lambda\_.\,[\,])$$

This says that **pay() 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



```
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:

```
type rank = int

type elem = content ref

and content =
  | Link of elem
  | Root of 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
  | _, _ -> assert false

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.

Streamlined proof in *Introduction to Algorithms*, 3rd ed. (1999).

$$\begin{aligned}
A_0(x) &= x + 1 \\
A_{k+1}(x) &= A_k^{(x+1)}(x) \\
&= A_k(A_k(\ldots A_k(x)\ldots)) \quad (x + 1 \text{ times}) \\
\alpha(n) &= \min\{k \mid A_k(1) \geqslant n\}
\end{aligned}$$

Quasi-constant cost: for all practical purposes, $\alpha(n) \leqslant 5$.

# Specification of `find`

```
Theorem find_spec : ∀N D R x, x ∈ D →
  App find x
    (UF N D R ⋆ $(alpha N + 2))
    (fun r ⇒ UF N D R ⋆ \[r = R x]).
```

The **abstract predicate** $\mathrm{UF}\,N\,D\,R$ 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 : ∀N D R x y, x ∈ D → y ∈ D →
  App union x y
    (UF N D R ⋆ $(3*(alpha N)+6))
    (fun z ⇒
       UF N D (fun w ⇒ If R w = R x ∨ R w = R y then z else R w)
       ⋆ [z = R x ∨ z = R y]).
```

The amortized cost of union is $3\alpha(N) + 6$.

# Definition of $\Phi$, on paper

$$
\begin{array}{lll}
p(x) & = \text{parent of } x & \text{if } x \text{ is not a root} \\
k(x) & = \max\{k \mid K(p(x)) \geqslant A_k(K(x))\} & \text{(the \textbf{level} of } x\text{)} \\
i(x) & = \max\{i \mid K(p(x)) \geqslant A^{(i)}_{k(x)}(K(x))\} & \text{(the \textbf{index} of } x\text{)} \\
\phi(x) & = \alpha(N) \cdot K(x) & \text{if } x \text{ is a root or has rank 0} \\
\phi(x) & = (\alpha(N) - k(x)) \cdot K(x) - i(x) & \text{otherwise} \\
\Phi & = \sum_{x \in D} \phi(x) &
\end{array}
$$

For some intuition, see Seidel and Sharir (2005).

# Definition of $\Phi$, in Coq

```
Definition p F x :=
  epsilon (fun y ⇒ F x y).
Definition k F K x :=
  Max (fun k ⇒ K (p F x) ⩾ A k (K x)).
Definition i F K x :=
  Max (fun i ⇒ K (p F x) ⩾ iter i (A (k F K x)) (K x)).
Definition phi F K N x :=
  If (is_root F x) ∨(K x = 0)
    then (alpha N) ∗ (K x)
    else (alpha N − k F K x) ∗ (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:

$$\Phi + \text{advertised cost} \;\geqslant\; \Phi' + \text{actual cost}$$

For instance, in the case of `find`, we must prove:

$$\texttt{Phi D F K N} + (\texttt{alpha N} + 2) \geqslant \texttt{Phi D F' K N} + (\texttt{d} + 1)$$

where:

- `F` is the graph before the execution of `find x`,
- `F'` is the graph after the execution of `find x`,
- `d` is the length of the path in `F` from `x` to its root.

# Summary

- 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/`