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

Arthur Charguéraud    François Pottier

September 8, 2015
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:

```ocaml
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
        rx := Link y;
        y
      end else if rx > ry then begin
        ry := Link x;
        x
      end else begin
        y := Link x;
        x
      end
  end

let union x y = link (find x) (find y)
```
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.


\[
A_0(x) = x + 1 \\
A_{k+1}(x) = A_k^{(x+1)}(x) \\
\quad = A_k(A_k(...A_k(x)...)) \quad (x + 1 \text{ times}) \\
\alpha(n) = \min\{k \mid A_k(1) \geq n\}
\]

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

Space of the related work:

- Verification that ignores complexity.
- Verification that includes complexity:
  - Proof only at an abstract mathematical level.
  - Proof that goes down to the level of the source code:
    - with emphasis on automation (e.g., the RAML project);
    - with emphasis on expressiveness (Atkey; this work).
Specification

Separation Logic with time credits

Union-Find: invariants

Conclusion
Specification of find

Theorem find_spec : \( \forall N \, D \, R \, x, \, x \in D \to \)

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

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 : \( \forall N D R x y, x \in D \rightarrow y \in D \rightarrow \)

App union x y

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

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

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

- Reasoning with \( O \)'s is ongoing work.
- Asserting that the worst-case cost is \( O(\log N) \) would require non-storable time credits.
Specification of make

**Theorem** `make_spec`: \( \forall N \; D \; R, \; \text{card} \; D < N \rightarrow \)

\[
\text{App make } \text{tt}
\]

\[
(\text{UF} \; N \; D \; R \; \star \; \$1)
\]

\[
(\text{fun} \; x \Rightarrow \text{UF} \; N \; (D \cup \{x\}) \; R \; \star \; \lbrack x \notin D \rbrack \; \star \; \lbrack R \; x = x \rbrack).
\]

The cost of make is \( O(1) \).

At most \( N \) elements can be created.
Specifying the ghost operations

**Theorem UF_create:** $\forall N$, 

$$[] \triangleright (UF N \not\in id).$$

**Theorem UF_properties:** $\forall N \in D, \in R, UF N D R \triangleright UF N D R \star$

$[(\text{card} D \leq N) \land$

$\forall x, (R (R x) = R x) \land$

$(x \in D \rightarrow R x \in D) \land$

$(x \notin D \rightarrow R x = x)].$

*UF_create* initializes an empty Union-Find data structure. It can be thought of as a ghost operation. $N$ is fixed at this moment.

*UF_properties* reveals a few properties of $D, N$ and $R$. 
Specification

Separation Logic with time credits

Union-Find: invariants

Conclusion
Heap predicates:

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

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

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

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

Intended properties:

\[ (n + n') = n \ast n' \quad \text{and} \quad 0 = [] \]

Intended use:

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

We change Heap to \((\text{loc} \mapsto \text{value}) \times \mathbb{N}\).

A heap is a (partial) memory paired with a (partial) number of credits.

The predicate \(\$ n\) means that we own (exactly) \(n\) credits:

\[
\$ n \equiv \lambda(m, c). \; m = \emptyset \land c = n
\]

Separating conjunction distributes the credits among the two sides:

\[
(m_1, c_1) \uplus (m_2, c_2) \equiv (m_1 \uplus m_2, c_1 + c_2)
\]
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,

\[(\text{total \#function calls}) \leq (\text{initial \#credits})\]

This, we prove (on paper).
Connecting computation and time credits

This is a formal statement of the previous claim.

Theorem (Soundness of characteristic formulae with time credits)

\[ \forall m, c. \quad \left\{ \begin{array}{l} [t] H Q \\ H (m, c) \end{array} \right\} \Rightarrow \exists n v m' c' m''. \left\{ \begin{array}{l} t/m \downarrow^n v/m' \oplus m'' \\ n \leq c - c' \\ Q v (m', c') \end{array} \right\} \]
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:

```
App pay () ($1) (λ_. [])
```

This says that pay() consumes one credit.
Connecting computation and time credits

Hypotheses:
- No loops in the source code. (Translate them to recursive functions.)
- The compiler turns a function into \textbf{machine code with no loop}.
- A machine instruction executes in constant time.

Thus,

\[
\begin{align*}
\text{(total \#instructions executed)} & = O(\text{total \#function calls}) \\
\text{(total execution time)} & = O(\text{total \#function calls}) \\
\text{(total execution time)} & = O(\text{initial \#credits})
\end{align*}
\]

This, we do not prove.
(It would require modeling the compiler and the machine.)
Expressive power

An assertion $n$ can appear in a precondition, a postcondition, a data structure invariant, etc.

That is, time credits can be **passed** from caller to callee (and back), and can be **stored** for later use.

This allows **amortized time complexity** analysis.
Specification

Separation Logic with time credits

Union-Find: invariants

Conclusion
Invariant #1: math

**Definition Inv N D F K R :=**

- confined \( D F \)
- functional \( F \)
- \((\forall x, \text{path} \ F \ x \ (R \ x) \land \text{is_root} \ F \ (R \ x)) \land (\text{finite} \ D) \land (\text{card} \ D \leq \ N) \land (\forall x, x \notin D \rightarrow K \ x = 0) \land (\forall x \ y, F \ x \ y \rightarrow K \ x < K \ y) \land (\forall r, \text{is_root} \ F \ r \rightarrow 2^{(K \ r)} \leq \text{card} (\text{descendants} \ F \ r))\).

The relation \( F \) is the graph (i.e., the disjoint set forest).

\( K \) maps every element to its rank.

\( D, N, R \) are as before.
Invariant #2: memory

CFML describes a region as GroupRef $M$, where the partial map $M$ maps a memory location to the content of the corresponding memory cell.
Invariant #3: connecting math and memory

We must express the connection between $M$ and our $D, N, R, F, K$.

**Definition** \( \text{Mem D F K M} := \)

\[
\begin{align*}
&\quad (\text{dom } M = D) \\
&\quad \land (\forall x, x \in D \rightarrow \\
&\quad \quad \text{match } M[x] \text{ with} \\
&\quad \quad | \text{Link } y \Rightarrow F x y \\
&\quad \quad | \text{Root } k \Rightarrow \text{is_root } F x \land k = K x \\
&\quad \quad \text{end}).
\end{align*}
\]

$M$ contains less information than $D, N, R, F, K$. E.g.,

- $N$ is ghost state;
- the rank $K(x)$ of a non-root node $x$ is ghost state.
Invariant #4: potential

At every time, we store $\Phi$ time credits. ($\Phi$ is defined in a few slides.) $\Phi$ depends on $D, F, K, N$, so the Coq invariant is $\Phi (D F K N)$. 
Invariants #1-#4 together

The **abstract predicate** that appears in the public specification:

**Definition** \( \text{UF} \ N \ D \ R := \exists F \ K \ M, \)

\[ \begin{align*}
&[\text{Inv} \ N \ D \ F \ K \ R ] \ast \\
&(\text{GroupRef} \ M) \ast \\
&[\text{Mem} \ D \ F \ K \ M ] \ast \\
&$(\text{Phi} \ D \ F \ K \ N)$.
\]
Definition of $\Phi$, on paper

\[ p(x) = \text{parent of } x \]
\[ k(x) = \max\{k \mid K(p(x)) \geq A_k(K(x))\} \quad \text{(the level of } x) \]
\[ i(x) = \max\{i \mid K(p(x)) \geq A_{k(x)}^i(K(x))\} \quad \text{(the index of } x) \]
\[ \phi(x) = \alpha(N) \cdot K(x) \quad \text{if } x \text{ is a root or has rank 0} \]
\[ \phi(x) = (\alpha(N) - k(x)) \cdot K(x) - i(x) \quad \text{otherwise} \]
\[ \Phi = \sum_{x \in D} \phi(x) \]

Don’t ask... 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 $k \ F \ K \ x :=$
    Max $(\text{fun } k \Rightarrow K (p \ F \ x) \geq A \ k (K \ x))$.

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

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

Definition $\Phi \ D \ F \ K \ N :=$
    Sum $D (\phi \ F \ K \ N)$.

Non-constructive operators: epsilon, Max, If, Sum. Convenient!
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:

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

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.
Specification

Separation Logic with time credits

Union-Find: invariants

Conclusion
Summary

- A machine-checked proof of correctness and complexity.
- Down to the level of the OCaml code.
- 3Kloc of high-level mathematical analysis.
- 0.4Kloc of specification and low-level verification.

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

- Establish a **local bound** of $\alpha(n)$ instead of $\alpha(N)$ where $N$ is fixed.
  - Follow Alstrup et al. (2014).
- Introduce **$O$ notation** and write $O(\alpha(n))$ instead of $3\alpha(n) + 6$.
- Attach a **datum** to every root. Offer a few more operations.
- Develop a **verified OCaml library** of basic algorithms and data structures (with Filliâtre and others).
Appendix
The CFML approach

(** UnionFind.ml **)  (** UnionFind_ml.v **)  (** UnionFind_proof.v **)  

let rec find x =  
  ...  

Axiom find_c : Func.  
Axiom find_cf : \forall x H Q,  
  (...) \rightarrow App find x H Q.  

Theorem find_spec : \forall x \in D,  
  App find x (...) (...).  
Proof.  
  intros. apply find_cf.  
  ...  
  Qed.
Characteristic formulae

The characteristic formula of a term $t$, written $\llbracket t \rrbracket$, is a predicate such that:

$$\forall HQ. \llbracket t \rrbracket H Q \Rightarrow \{H\} t \{Q\}$$

In any state satisfying $H$, $t$ terminates on $v$, in a state satisfying $Q v$.

Example definition:

$$\llbracket t_1 ; t_2 \rrbracket \equiv \lambda HQ. \exists H'. \llbracket t_1 \rrbracket H (\lambda_. H') \land \llbracket t_2 \rrbracket H' Q$$

Characteristic formulae: sound and complete, follow the structure of the code (compositional and linear-sized), and support the frame rule.
Characteristic formula generation

\[
[v] = \lambda HQ. \ H > Q \ v \\
[t_1 ; t_2] = \lambda HQ. \ \exists Q'. \ [t_1] \ H \ Q' \land [t_2] \ (Q' \ tt) \ Q \\
[\text{let } x = t_1 \text{ in } t_2] = \lambda HQ. \ \exists Q'. \ [t_1] \ H \ Q' \land \forall x. \ [t_2] \ (Q' \ x) \ Q \\
[f \ v] = \lambda HQ. \ \text{App } f \ v \ H \ Q \\
[\text{let } f = \lambda x. \ t_1 \text{ in } t_2] = \lambda HQ. \ \forall f. \ P \Rightarrow [t_2] \ H \ Q
\]

where \( P = (\forall x H' Q'. \ [t_1] \ H' Q' \Rightarrow \text{App } f \ x \ H' Q') \)

App has type:

\( \forall A B. \ \text{Func} \rightarrow A \rightarrow (\text{Heap} \rightarrow \text{Prop}) \rightarrow (B \rightarrow \text{Heap} \rightarrow \text{Hprop}) \rightarrow \text{Prop.} \)
Other amortized analyses using CFML with credits

Resizable arrays
- push and pop at back in $O(1)$.

Random-access lists
- push and pop at head in $O(1)$, get and set in $O(\log n)$.

Bootstrapped chunked sequence
- push and pop at the two ends in $O(1)$, split and join in $O(B \log_B n)$. 