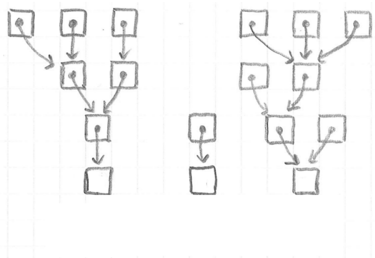
Machine-checked correctness and complexity of a Union-Find implementation

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:

```
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).

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

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

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 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 $3\alpha(N) + 6$.

- 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 : ∀N D R, card D < N →
  App make tt
    (UF N D R ⋆ $1)
    (fun x ⇒ UF N (D ∪ {x}) R ⋆ \[x ∉ D] ⋆ \[R x = x]).
```

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

At most N elements can be created.

Specification of the ghost operations

```
Theorem UF_create : ∀N,
  \[] ▻ (UF N ∅ id).

Theorem UF_properties : ∀N D R, UF N D R ▻ UF N D R ⋆
  [(card D ⩽ N) ∧
   ∀x, (R (R x) = R x) ∧
   (x ∈ D → R x ∈ D) ∧
   (x ∉ D → 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.

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:

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

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.

Model of time credits

We change Heap to (loc $\mapsto$ 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:

$$\$\,n \quad \equiv \quad \lambda(m, c).\ m = \varnothing\ \wedge\ c = n$$

Separating conjunction distributes the credits among the two sides:

$$(m_1, c_1) \uplus (m_2, c_2) \quad \equiv \quad (m_1 \uplus m_2,\ c_1 + c_2)$$

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,

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

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)

$$\forall mc.\ \begin{cases} [\![t]\!]\, H\, Q \\ H\,(m, c) \end{cases} \Rightarrow\ \exists n v m' c' m''.\ \begin{cases} t_{/m} \Downarrow^{n} v_{/m' \uplus m''} \\ n \leqslant c - c' \\ Q\, v\,(m', c') \end{cases}$$

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**.

Connecting computation and time credits

Hypotheses:

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

Thus,

$$\begin{array}{rcl} (\text{total \#instructions executed}) & = & O(\text{total \#function calls}) \\ (\text{total execution time}) & = & O(\text{total \#function calls}) \\ (\text{total execution time}) & = & O(\text{initial \#credits}) \end{array}$$

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.

Invariant #1: math

```
Definition Inv N D F K R :=
  confined D F ∧
  functional F ∧
  (∀ x, path F x (R x) ∧ is_root F (R x)) ∧
  (finite D) ∧
  (card D ⩽ N) ∧
  (∀ x, x ∉ D → K x = 0) ∧
  (∀ x y, F x y → K x < K y) ∧
  (∀ r, is_root F r → 2^(K r) ⩽ card (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 Mem D F K M :=
    (dom M = D)
  ∧ (∀ x, x ∈ D →
       match M[x] with
       | Link y ⇒ F x y
       | Root k ⇒ is_root F x ∧ k = K x
       end).
```

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 Φ time credits. (Φ is defined in a few slides.)

Φ 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 UF N D R := ∃F K M,
  \[ Inv N D F K R ] ⋆
  (GroupRef M) ⋆
  \[ Mem D F K M ] ⋆
  $(Phi D F K N).
```

Definition of Φ, on paper

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

Don't ask... For some intuition, see Seidel and Sharir (2005).

Definition of Φ, 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).
```

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

Machine-checked amortized complexity analysis

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

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

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.

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 **)

let rec find x =
  ...
```

```
(** UnionFind_ml.v **)

Axiom find : Func.

Axiom find_cf : ∀x H Q,
  (...) → App find x H Q.
```

```
(** UnionFind_proof.v **)

Theorem find_spec : ∀x ∈ D,
  App find x (...) (...).
Proof.
  intros. apply find_cf.
  ...
Qed.
```

Characteristic formulae

The characteristic formula of a term t, written $[\![t]\!]$, is a predicate such that:

$$\forall HQ. \quad [\![t]\!]\, H\, Q \;\Rightarrow\; \{H\}\; t\; \{Q\}$$

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

Example definition:

$$[\![t_1\,;\,t_2]\!] \;\equiv\; \lambda HQ.\; \exists H'.\; [\![t_1]\!]\, H\, (\lambda\_.\, H') \;\wedge\; [\![t_2]\!]\, H'\, Q$$

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

Characteristic formula generation

$$
\begin{aligned}
[\![v]\!] &= \lambda H Q.\ \ H \rhd Q\, v \\
[\![t_1 \,;\, t_2]\!] &= \lambda H Q.\ \exists Q'.\ [\![t_1]\!]\, H\, Q' \ \wedge\ [\![t_2]\!]\, (Q'\, tt)\, Q \\
[\![\mathsf{let}\, x = t_1 \,\mathsf{in}\, t_2]\!] &= \lambda H Q.\ \exists Q'.\ [\![t_1]\!]\, H\, Q' \ \wedge\ \forall x.\ [\![t_2]\!]\, (Q'\, x)\, Q \\
[\![f\ v]\!] &= \lambda H Q.\ \mathsf{App}\, f\, v\, H\, Q \\
[\![\mathsf{let}\, f = \lambda x.\, t_1 \,\mathsf{in}\, t_2]\!] &= \lambda H Q.\ \forall f.\, P \ \Rightarrow\ [\![t_2]\!]\, H\, Q \\
\text{where } P &= (\forall x H' Q'.\ [\![t_1]\!]\, H'\, Q' \ \Rightarrow\ \mathsf{App}\, f\, x\, H'\, Q')
\end{aligned}
$$

App has type:

$\forall A\, B.\ \ \mathsf{Func} \to A \to (\mathsf{Heap} \to \mathsf{Prop}) \to (B \to \mathsf{Heap} \to \mathsf{Hprop}) \to \mathsf{Prop}.$

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 $O(B \log_B n)$.