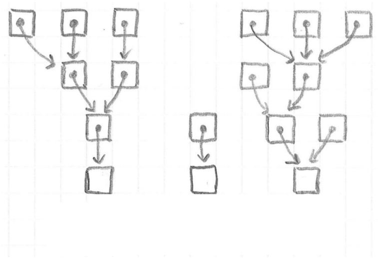
Machine-Checked Verification of the Correctness and Amortized Complexity of an Efficient Union-Find Implementation

Arthur Charguéraud
joint work with François Pottier

Inria

2015/08/25

Union-Find data structure



```
type elem
val make : unit -> elem
val find : elem -> elem
val union : elem -> elem -> elem
```

Implementation

Pointer-based Union-Find, 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)
```

Union-Find analysis

Tarjan (1975): the amortized cost of find is $O(\alpha(n))$.

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

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

→ In this work: the first mechanized complexity analysis of Union-Find.

Following proof from 1999, published in *Introduction to Algorithms*, 3rd ed.

Verification tool

We extend the CFML tool with time credits, to allow for the formalization of amortized complexity analyses for arbitrarily-complex OCaml programs.

Design space:

- Verification ignoring the complexity.
- Verification including the complexity:
    - Proof only at the level of the mathematical abstractions.
    - Proof also connecting to the source code:
        - with emphasis on automation (e.g., RAML project);
        - with emphasis on expressiveness (Atkey and this work).

Contents of the talk

1. Statement of specifications.
2. Separation Logic with time credits.
3. Characteristic formulae with time credits.
4. Invariant and potential for Union-Find.
5. Verification proofs.

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]).
```

- D is the set of all elements, i.e. the domain.
- N is a bound on the cardinality of the domain.
- R maps elements to their corresponding roots.
- "$\mathrm{UF}\ N\ D\ R$" denotes the invariant on the state.

where App has type:

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

Separation Logic

Heap predicates:

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

Core definitions:

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

→ Formalization in Coq following that of Ynot (Chlipala et al, 2009).

Principle of time credits

Time credits:

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

Properties:

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

Principle:

Ensure that every beta-reduction forces the spending of $\$1$.

Time credits are received in preconditions, are spent on function calls. They may be stored in the heap for later retrieval and consumption.

Requires a complexity-preserving compiler:

$$\text{nb machine instructions} = \text{O}(\text{nb beta-reductions})$$

Model of time credits

Without credits:

$$\mathsf{Heap} \quad \equiv \quad (\mathsf{loc} \mapsto \mathsf{value})$$

With credits:

$$\mathsf{Heap} \quad \equiv \quad (\mathsf{loc} \mapsto \mathsf{value}) \times \mathbb{N}$$

Definition of credits:

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

Earlier definitions are lifted to pairs, e.g.:

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

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.

Time credits in characteristic formulae

Goal: ensure that every beta-reduction forces the spending of $\$1$.

Solution: CFML instruments the OCaml code by inserting a call to "pay" at the head of every function or loop body.

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

Axiomatic specification of pay:

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

Soundness

Theorem (Soundness of characteristic formulae with time credits)

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

Union-Find invariant

```
Definition is_root F x := ∀y, ¬ F x y.

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)).
```

- F describes the edges of the underlying graph.
- K gives the rank of every element.
- D describes the domain of the elements.
- R maps elements to their corresponding roots.
- N is a bound on the cardinality of the domain.

Representation predicate

- M maps elements to the contents of the corresponding memory cell.

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

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).
```

Definition of the potential on paper

$$\begin{array}{lcl} p(x) & \equiv & \text{parent of } x \quad \text{(when } x \text{ is not a root)} \\ K(x) & \equiv & \text{rank of } x \\ k(x) & \equiv & \max\{k \mid K(p(x)) \geqslant A_k(K(x))\} \\ i(x) & \equiv & \max\{i \mid K(p(x)) \geqslant A^{(i)}_{k(x)}(K(x))\} \end{array}$$

$$\begin{array}{lcll} \phi(x) & \equiv & \alpha(N) \cdot K(x) & \text{if } x \text{ is a root or has rank 0} \\ \phi(x) & \equiv & (\alpha(N) - k(x)) \cdot K(x) - i(x) & \text{otherwise} \end{array}$$

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

Definition of the potential 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).
```

Amortized analysis

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

In the case of `find`, we prove:

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

where:

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

Verification script

```
Theorem find_spec : ∀N D R, x ∈ D →
  App find x (UF N D R ⋆ $(alpha N + 2)) (fun r ⇒ UF N D R ⋆ \[R x = r]).
Proof.
  asserts S': (∀ d D R F K F' M,
    Inv N D F K R → Mem D F K M → x ∈ D → ipc F x d F' →
    App find x (GroupRef M ⋆ $(d+1))
               (fun r' ⇒ ∃M', GroupRef M' ⋆ \[Mem D F' K M' ∧ r' = R x])).
  { xinduction_heap Wf_nat.lt_wf. apply find_cf.
    intros x d IH. hide IH. introv HI HM Dx HC. credits_split. xpay.
    lets HMD: (Mem_dom HM). unfold elem in ∗.
    xapps∗. forwards∗ HV: (Mem_val HM) x. unfold elem in ∗.
    xmatch; rename H0 into HK; rewrite ← HK in HV.
    (* case root *) (* ... 3 lines not shown ...*)
    (* case link *) (* ... 14 lines not shown ...*)
  }
  xweaken S'. clear S'. simpl. intros x. intros RS LRS KRS.
  introv Dx. unfold UF. xextract as F K M HI HM.
  forwards∗ (d&F'&HC&HP): amortized_cost_of_iterated_path_compression x N.
  change (2%nat) with (1+1)%nat. forwards (H'&E): credits_nat_le_rest HP.
  xchange E. chsimpl. credits_split. xgc H'. xframe ($ Phi D F' K N).
  xapply (>> KRS HI HM Dx HC). change (S d) with (1+d)%nat. chsimpl.
  xok. clears RS. intros z. xextract as M' (HM'&Hz). subst z. intros. hsimpl~.
  constructor; eauto using is_rdsf_bw_ipc, bw_ipc_preserves_RF_agreement.
Qed.
```

Future work

- State bounds using $\alpha(n)$, instead of $\alpha(N)$ with the constraint $n \leqslant N$.
- Improve the degree of inference and automation.
- Introduce the big-O notation, to write $O(\alpha(n))$ instead of $3\,\alpha(n) + 6$.

Conclusion

An integrated verification framework for proving not just the functional correctness but also the asymptotic complexity of a concrete program.

TCB: classic Coq + CFML generator + a complexity-preserving compiler.

Application to the verification of an efficient Union-Find implementation.

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

Thanks!