A Fully Automated Verification of Union-Find

Martin Clochard, Jean-Christophe Filliâtre,
Simão Melo de Sousa, Mário Pereira

ProofInUse meeting

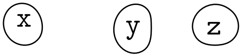
AdaCore, November 23, 2018

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

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

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

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

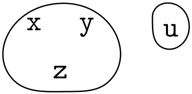


```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

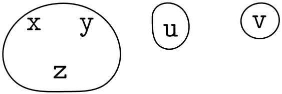
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```

union-find API

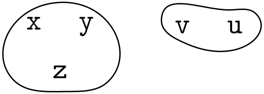
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



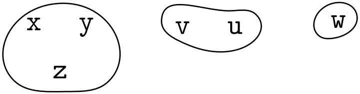
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



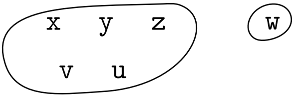
```
type elem
val make : unit -> elem
val union: elem -> elem -> unit
val find : elem -> elem
val same : elem -> elem -> bool
```



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



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



```
type elem
```

```
type uf = {
  mutable dom: set elem;
  mutable rep: elem -> elem;
}
```

```
val ghost create () : uf
val make  (ghost uf: uf) ()           : elem
val union (ghost uf: uf) (x y: elem) : unit
val find  (ghost uf: uf) (x  : elem) : elem
val same  (ghost uf: uf) (x y: elem) : bool
```

```
type elem
```

```
type uf = {
  mutable dom: set elem;
  mutable rep: elem -> elem;
}
invariant { forall x. mem x dom ->
            mem (rep x) dom && rep (rep x) = rep x }
```

```
val ghost create () : uf
  ensures { result.dom = empty }
```

```
val make (ghost uf: uf) () : elem
  writes  { uf.dom, uf.rep }
  ensures { not (mem result (old uf.dom)) }
  ensures { uf.dom = add result (old uf.dom) }
  ensures { uf.rep = (old uf.rep)[result <- result] }
```

```
val find (ghost uf: uf) (x: elem) : elem
  requires { mem x uf.dom }
  ensures  { result = uf.rep x }
```

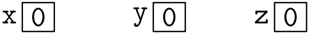
```
val union (ghost uf: uf) (x y: elem) : ghost elem
  requires { mem x uf.dom }
  requires { mem y uf.dom }
  writes   { uf.rep }
  ensures  { result = old (uf.rep x) ||
             result = old (uf.rep y) }
  ensures  { forall z. mem z uf.dom ->
    uf.rep z = if old (uf.rep z = uf.rep x ||
                       uf.rep z = uf.rep y)
               then result
               else old (uf.rep z) }
```

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

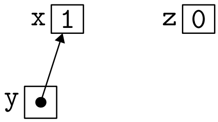
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```

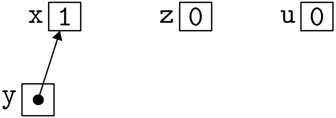
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



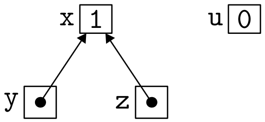
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



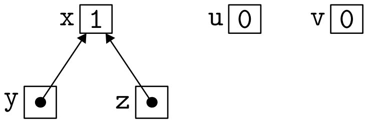
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



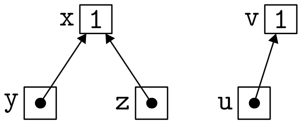
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



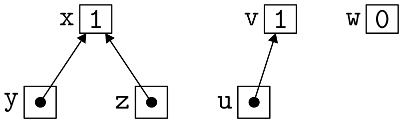
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



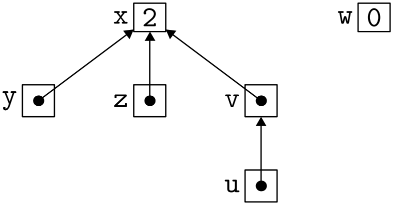
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



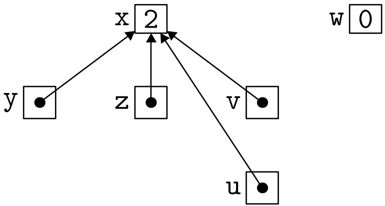
```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



```
type elem =
  content ref

and content =
  | Link of elem
  | Root of int
```



let's verify this with Why3

too complex for Why3's type checker; let's model the heap

```
type loc
```

```
type elem =
  content ref
and content =
  | Link of elem
  | Root of int
```

```
type elem =
  loc
type content =
| Link loc
| Root Peano.t
```

```
type heap = {
  ghost mutable
    refs: loc -> option content;
}
```

```
predicate allocated (h: heap) (x: loc) =
  h.refs x <> None

val alloc_ref (ghost h: heap) (v: content) : loc
  writes  { h.refs }
  ensures { (old h).refs result = None }
  ensures { h.refs = (old h.refs)[result <- Some v] }

val get_ref (ghost h: heap) (l: loc) : content
  requires { allocated h l }
  ensures  { Some result = h.refs[l] }

val set_ref (ghost h: heap) (l: loc) (c: content) : unit
  requires { allocated h l }
  writes   { h.refs }
  ensures  { h.refs = (old h.refs)[l <- Some c] }

val (==) (x y: loc) : bool
  ensures { result <-> x=y }
```

```
type uf = {
          heap: heap;
  mutable dom : set elem;
  mutable rep : elem -> elem;
}
...
invariant { forall x. mem x dom <-> allocated heap x }
```

```
type uf =
  ...
invariant { forall x. match heap.refs x with
      | Some (Link y) -> x <> y /\ allocated heap y /\
                         rep x = rep y
      | Some (Root _) -> rep x = x
      | None -> true end }
invariant { forall x. mem x dom ->
      match heap.refs (rep x) with
      | Some (Root _) -> true
      | _             -> false end }
```

it would be very tempting to introduce an inductive notion of path

```
inductive path (h: heap) (x y: elem) =
| Path0: forall x y k.
         h.refs x = Some (Root k) ->
         path h x x
| Path1: forall x y z.
         h.refs x = Some (Link y) ->
         path h y z -> path h x z
```

this way, we would have `path heap x (rep x)` as an invariant and this would ensure the termination of `find`

but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

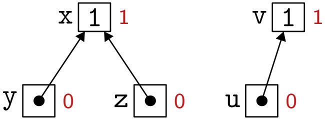
- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

$\texttt{maxd} = 1$

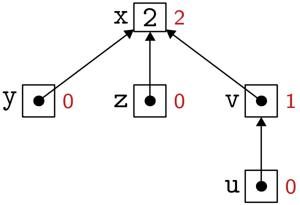


but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

maxd $= 2$

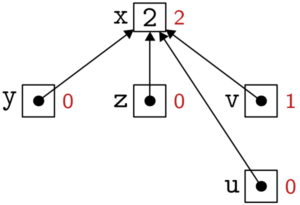


but this is a bad idea, as each assignment in the heap requires you to re-establish all paths (some unchanged, some shortened, etc.)

instead, we assign

- a distance to each node, increasing along `Link`
- a maximum distance for the whole union-find structure

$\texttt{maxd} = 2$



```
type uf =
  ...
  mutable dst : elem -> int;
  mutable maxd: int;
}
...
invariant { forall x. match heap.refs x with
      | Some (Link y) -> ... /\ dst x < dst y
      | ... }
invariant { 0 <= maxd }
invariant { forall x. mem x dom -> dst x <= maxd }
```

the verification is fully automated
(using Alt-Ergo 2.2.0 and CVC4 1.6)

in particular, there is

- no lemma
- no assertion
- no interactive proof

Why3 extraction mechanism

1. removes ghost code
2. maps some Why3 symbols to OCaml symbols

here

- type `Peano.t` is mapped to OCaml's type `int`
- our custom mini-heap is mapped to OCaml's references

we write a small custom driver file `uf.drv` for our model

```
module uf.UnionFind
  syntax type loc       "content ref"
  syntax val  (==)      "%1 == %2"
  syntax val  alloc_ref "ref %1"
  syntax val  get_ref   "!%1"
  syntax val  set_ref   "%1 := %2"
end
```

and then extract OCaml code as follows:

```
why3 extract -D ocaml64 -D uf.drv -L .
  uf.UnionFind -o uf.ml
```

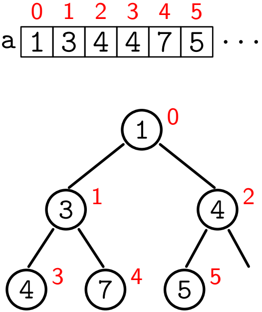
Charguéraud & Pottier did a Coq proof of a similar OCaml code, using CFML [ITP 2015, JAR 2017]

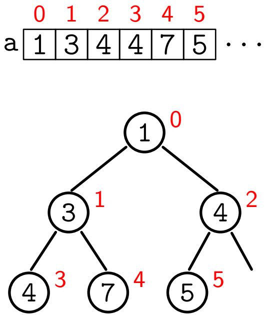
- includes a proof of complexity!
- maps OCaml's type `int` to Coq's type `Z` (unsound)
- more than 4k lines

1. modeling the heap can be easy
   - can be local
   - incurs a small TCB

2. avoid recursive/inductive definitions for better automation

   two other examples:
   - heap stored in an array
   - inverting a permutation in-place


0 1 2 3 4 5
a 1 3 4 4 7 5 ···
1 0
3 1
4 2
4 3
7 4
5 5



it would be tempting to introduce trees

but a universal, local invariant

$$\forall i.\ \mathtt{a}[i] \le \mathtt{a}[2i+1], \mathtt{a}[2i+2]$$

is all you need

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	0	1	5	2

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	0	1	5	-1

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

4	3	-6	1	5	-1

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	5	-1

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	-1	-1

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	-1	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	-6	-1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	-6	-1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	-6	1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	-4	5	1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

-3	3	5	1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

2	3	5	1	0	4

Algorithm I in TAOCP [Sec. 1.3.3, page 176]

2	3	5	1	0	4

again it would tempting to introduce paths, orbits, cycles, etc.

but again a universal, local invariant suffices