A Persistent Union-Find Data Structure

Sylvain Conchon and Jean-Christophe Filliâtre

Université Paris Sud – CNRS

The Stone Soup

Sylvain Conchon and Jean-Christophe Filliâtre

Université Paris Sud – CNRS

The Disjoint Sets Problem

goal: a data structure to maintain a partition of $\{0, 1, \ldots, n-1\}$

imperative data structure:

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

The Disjoint Sets Problem

goal: a data structure to maintain a partition of $\{0, 1, \ldots, n-1\}$

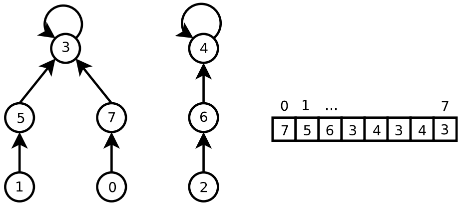
imperative data structure:

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

Optimal Solution: Tarjan's Algorithm

(solution actually due to M. D. McIllroy and R. Morris, and “only” analyzed by Tarjan)

within each class, elements are chained up to the representative



two key ideas:

- **path compression** while performing `find`
- use of **ranks** to balance the `union`s

Persistent Union-Find

the previous solution is not adapted to **backtracking**

- no immediate `undo` operation
- copying the structure would be disastrous

a solution would be a **persistent** data structure:

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

Persistent Union-Find

the previous solution is not adapted to **backtracking**

- no immediate undo operation
- copying the structure would be disastrous

a solution would be a **persistent** data structure:

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

A Naïve Solution

```
module M = Map.Make(struct type t = int let compare = compare end)

type t = int M.t

let create n = M.empty

let find m i =
  let rec lookup i = try lookup (M.find i m) with Not_found → i in
  lookup i

let union m i j =
  let ri = find m i in
  let rj = find m j in
  if ri <> rj then M.add ri rj m else m
```

Our Solution

we propose a solution

- **as efficient** as Tarjan's algorithm
- which is **persistent** (but uses side-effects)
- formally **proved correct**

obtained by combining

- a data structure of **persistent arrays**
- a persistent version of Tarjan's algorithm

Our Solution

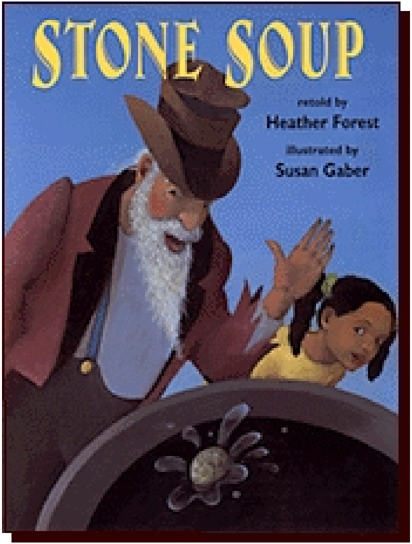
we propose a solution

- **as efficient** as Tarjan's algorithm
- which is **persistent** (but uses side-effects)
- formally **proved correct**

obtained by combining

- a data structure of **persistent arrays**
- a persistent version of Tarjan's algorithm

Our Solution



A Persistent Version of Tarjan's Algorithm

written independently of the structure for persistent arrays:

```
module Make(A : PersistentArray) : PersistentUnionFind
```

assuming

```
module type PersistentArray = sig
  type α t
  val init : int → (int → α) → α t
  val get : α t → int → α
  val set : α t → int → α → α t
end
```

A Persistent Version of Tarjan's Algorithm

the same arrays as in the original solution:

```
type t = { rank: int A.t; mutable link: int A.t }
```

the `mutable` field allows path compression

A Persistent Version of Tarjan's Algorithm

representative lookup with path compression:

```
let rec find_aux a i =
  let ai = A.get a i in
  if ai == i then
    a, i
  else
    let a, r = find_aux a ai in
    let a = A.set a i r in
    a, r
```

the path compression made effective:

```
let find h x =
  let a,rx = find_aux h.link x in h.link <- a; rx
```

A Persistent Version of Tarjan's Algorithm

representative lookup with path compression:

```
let rec find_aux a i =
  let ai = A.get a i in
  if ai == i then
    a, i
  else
    let a, r = find_aux a ai in
    let a = A.set a i r in
    a, r
```

the path compression made effective:

```
let find h x =
  let a,rx = find_aux h.link x in h.link <- a; rx
```

Persistent Arrays

to keep Tarjan's efficiency, we need efficient persistent arrays

solution known since 1978 and due to **Henry Baker**

idea: a persistent array is

- either a usual array
- or a modification of another persistent array

```
type α t = α data ref
and  α data =
  | Arr of α array
  | Diff of int × α × α t
```

Persistent Arrays

to keep Tarjan's efficiency, we need efficient persistent arrays

solution known since 1978 and due to **Henry Baker**

idea: a persistent array is

- either a usual array
- or a modification of another persistent array

```
type α t = α data ref
and  α data =
   | Arr of α array
   | Diff of int × α × α t
```

Baker's Persistent Arrays

```
let a0 = create 7 0

let a1 = set a0 1 7

let a2 = set a1 2 8

let a3 = set a1 2 9
```

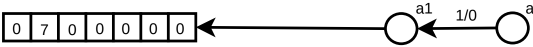
Baker's Persistent Arrays

```
let a0 = create 7 0

let a1 = set a0 1 7

let a2 = set a1 2 8

let a3 = set a1 2 9
```



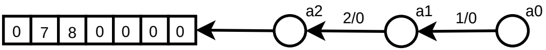
Baker's Persistent Arrays

```
let a0 = create 7 0

let a1 = set a0 1 7

let a2 = set a1 2 8

let a3 = set a1 2 9
```



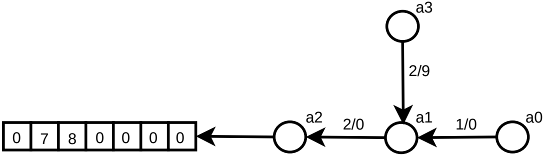
Baker's Persistent Arrays

```
let a0 = create 7 0

let a1 = set a0 1 7

let a2 = set a1 2 8

let a3 = set a1 2 9
```



Baker's Persistent Arrays

these arrays are persistent but inefficient as soon as we access old versions

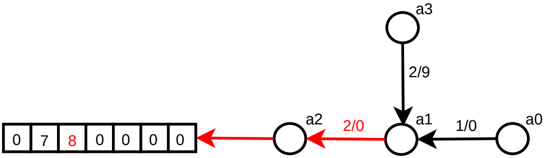
to improve efficiency, Baker introduces **rerooting**

Baker's Persistent Arrays

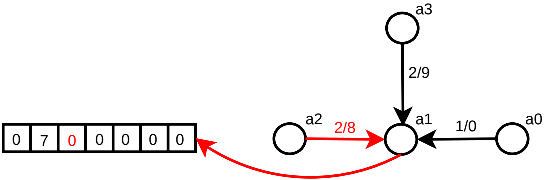
these arrays are persistent but inefficient as soon as we access old versions

to improve efficiency, Baker introduces **rerooting**

Rerooting



if we now access to a1, we first perform a rerooting



Improvement

when doing only **backtracking**, we can even save the pointers reversal (since the nodes are going to be collected by the GC anyway)

back to array a1

Finally...

- we get arrays which are not fully persistent anymore
  $\Rightarrow$ they are only **semi-persistent**

- the final solution is similar to a “stack of *undos*”,
  but which is hidden in the data structure

Finally...

- we get arrays which are not fully persistent anymore
  $\Rightarrow$ they are only **semi-persistent**

- the final solution is similar to a "stack of *undos*", but which is hidden in the data structure

Performances

tests inspired by the use of union-find in the Ergo theorem prover

p = ratio `union` / `find`

N = number of operations between backtracking points

p / N	5% / 2.10^4	5% / 10^5	5% / 5.10^5	10% / 2.10^4	10% / 10^5	10% / 5.10^5	15% / 2.10^4	15% / 10^5	15% / 5.10^5
Tarjan	0.31	2.23	12.50	0.33	2.34	12.90	0.34	2.36	13.20
V1	0.52	3.03	17.10	0.81	4.78	26.80	1.16	6.78	37.90
V2	0.34	2.01	11.70	0.42	2.54	14.90	0.52	3.21	18.70
V3	**0.33**	**1.90**	**11.30**	**0.41**	**2.45**	**14.40**	**0.52**	**3.14**	**17.80**
naïve	0.76	5.28	37.50	1.22	9.14	63.80	40.40	>10mn	>10mn

- V1 = Baker's persistent arrays
- V2 = semi-persistent arrays
- V3 = defunctorized V2

Formal Proof

persistent structure but many **side-effects**

$\Rightarrow$ correctness is not obvious

$\Rightarrow$ formal proof conducted within Coq

Modeling ML References

a type for ML references

```
Parameter pointer : Set.
```

memory = map from pointers to values

```
Module PM.
  Parameter t : Set → Set.

  Parameter find: ∀a, t a → pointer → option a.
  Parameter add : ∀a, t a → pointer → a → t a.
  Parameter new : ∀a, t a → pointer.

  Axiom find_add_eq : ...
  Axiom find_add_neq: ...
  Axiom find_new     : ...
End PM.
```

Modeling ML References

a type for ML references

```
Parameter pointer : Set.
```

memory = map from `pointers` to values

```
Module PM.
  Parameter t : Set → Set.

  Parameter find: ∀a, t a → pointer → option a.
  Parameter add : ∀a, t a → pointer → a → t a.
  Parameter new : ∀a, t a → pointer.

  Axiom find_add_eq : ...
  Axiom find_add_neq: ...
  Axiom find_new     : ...
End PM.
```

Memory Model

the Ocaml datatype

```
type t =
  data ref
and data =
  | Arr of int array
  | Diff of int × int × t
```

is modeled by

```
Inductive data : Set :=
  | Arr : data
  | Diff: Z → Z → pointer → data.

Record mem : Set := { ref : PM.t data; arr : Z→Z }.
```

Modelling the Persistence

`pa_valid` m p = p well-formed persistent array in memory m
`pa_model` m p f = the contents of p is modelled by function f

```
Definition set :
  ∀m:mem, ∀p:pointer, ∀i v:Z,
  pa_valid m p →
  { p':pointer & { m':mem |
    ∀f, pa_model m p f →
        pa_model m' p f ∧ pa_model m' p' (upd f i v) } }.
```

proof size = 140 lines for persistent arrays + 600 for union-find

Modelling the Persistence

pa_valid m p = p well-formed persistent array in memory m
pa_model m p f = the contents of p is modelled by function f

```
Definition set :
  ∀m:mem, ∀p:pointer, ∀i v:Z,
  pa_valid m p →
  { p':pointer & { m':mem |
    ∀f, pa_model m p f →
       pa_model m' p f ∧ pa_model m' p' (upd f i v) } }.
```

proof size = 140 lines for persistent arrays + 600 for union-find

Modelling the Persistence

pa_valid m p = p well-formed persistent array in memory m
pa_model m p f = the contents of p is modelled by function f

```
Definition set :
  ∀m:mem, ∀p:pointer, ∀i v:Z,
  pa_valid m p →
  { p':pointer & { m':mem |
    ∀f, pa_model m p f →
       pa_model m' p f ∧ pa_model m' p' (upd f i v) } }.
```

proof size = 140 lines for persistent arrays + 600 for union-find

Conclusion

a persistent union-find data structure with efficiency similar to Tarjan's one

- significant example of a persistent but non-purely applicative data structure (note that it is not thread-safe)
- example of imperative ML program proved in Coq
- new notion of **semi-persistence** introduced
  $\Rightarrow$ its legal use can be checked (see *Semi-Persistent Data Structures*)