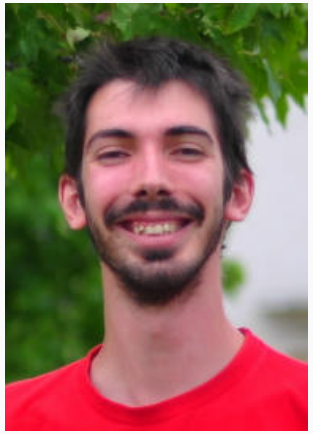
Storable types: free, absorbing, custom

Basile Clément



Gabriel Scherer

2025-01-27

JFLA 2025

Reminder on Store

Store lets you easily add **backtracking** to your mutable data structure. Example:

Reminder on Store

Store lets you easily add **backtracking** to your mutable data structure. Example:

Union-find without Store:

```ocaml
type 'a node = 'a data ref
type 'a data =
| Link of 'a node
| Root of { rank: int; v: 'a }

val make : 'a -> 'a node
val union :
    ('a -> 'a -> 'a) ->
    'a node -> 'a node -> unit
val find : 'a node -> 'a
```

Reminder on Store

Store lets you easily add **backtracking** to your mutable data structure. Example:

Union-find without Store:

```ocaml
type 'a node = 'a data ref
type 'a data =
| Link of 'a node
| Root of { rank: int; v: 'a }

val make : 'a -> 'a node
val union :
    ('a -> 'a -> 'a) ->
    'a node -> 'a node -> unit
val find : 'a node -> 'a
```

Union-find with Store:

```ocaml
type 'a node = 'a data Store.Ref.t *
Store.t
type 'a data =
| Link of 'a node
| Root of { rank: int; v: 'a }

val make : Store.t -> 'a -> 'a node
val union :
    ('a -> 'a -> 'a) ->
    'a node -> 'a node -> unit
val find : 'a node -> 'a
```

Reminder on Store

Store lets you easily add **backtracking** to your mutable data structure. Example:

Union-find without Store:

```ocaml
type 'a node = 'a data ref
type 'a data =
| Link of 'a node
| Root of { rank: int; v: 'a }

val make : 'a -> 'a node
val union :
    ('a -> 'a -> 'a) ->
    'a node -> 'a node -> unit
val find : 'a node -> 'a
```

Union-find with Store:

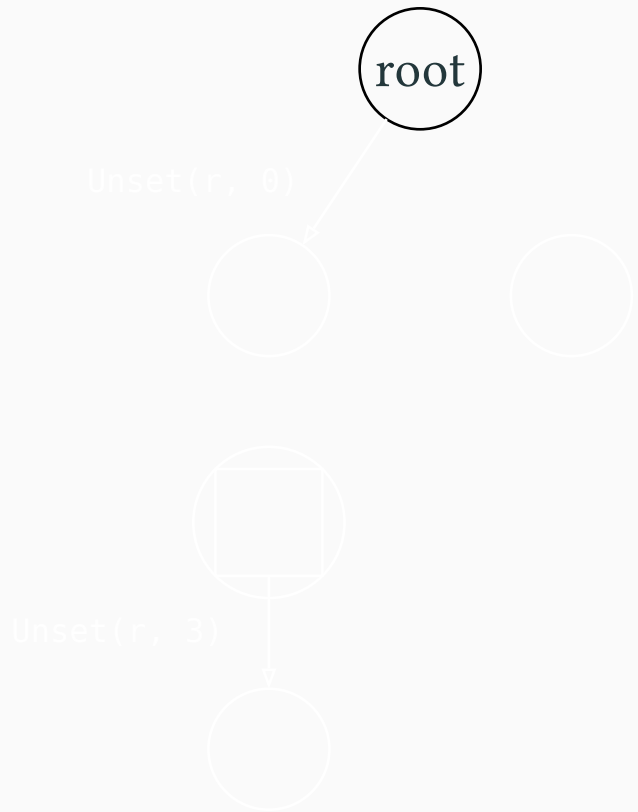
```ocaml
type 'a node = 'a data Store.Ref.t *
Store.t
type 'a data =
| Link of 'a node
| Root of { rank: int; v: 'a }

val make : Store.t -> 'a -> 'a node
val union :
    ('a -> 'a -> 'a) ->
    'a node -> 'a node -> unit
val find : 'a node -> 'a
```

Our use-cases: (1) union-find for GADT equations. (2) Alt-Ergo ?

Demo! (+ parsimony)

```
1  let s = Store.create () in
2  let r = Store.Ref.make s 0 in
3  let sn1 = Store.snapshot s in
4
5  Store.Ref.set s 1;
6  let sn2 = Store.snapshot s in
7  Store.restore s sn1;
8
9  Store.Ref.set s 2;
10 Store.Ref.set s 3; (* record elision *)
11
12 let tr1 = Store.transaction s in
13 Store.Ref.set s 4;
14 Store.rollback tr1;
15
16 let tr2 = Store.transaction s in
17 Store.Ref.set s 4;
18 Store.commit tr2; (* commit compression *)
```



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

$r \mapsto 0$

Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

$r \mapsto 0$

Demo! (+ parsimony)

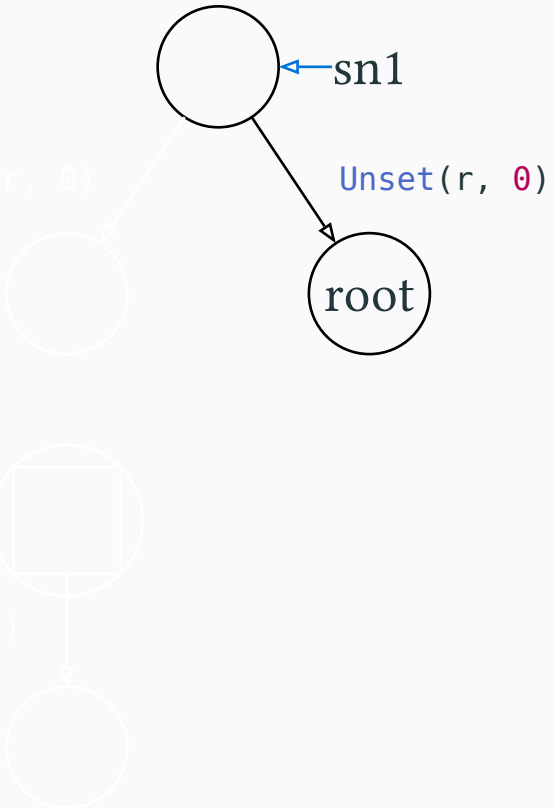
```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

$r \mapsto 0$

Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

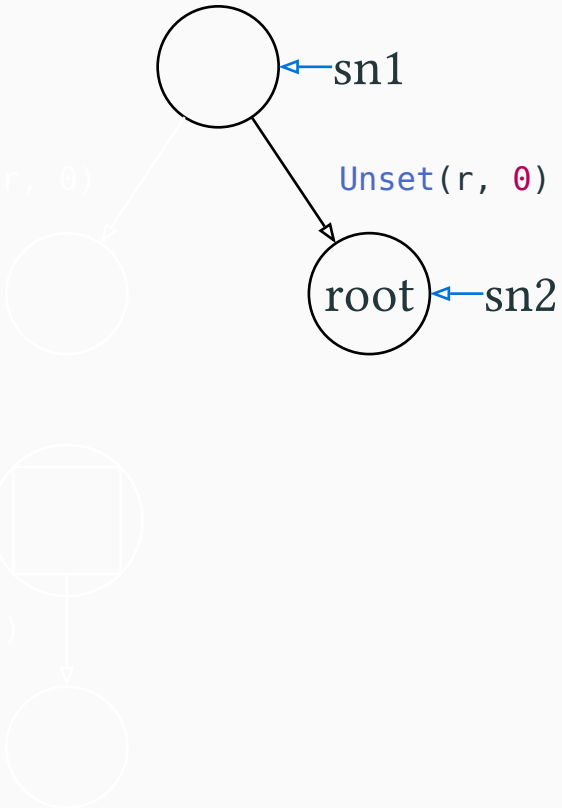
$r \mapsto 1$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

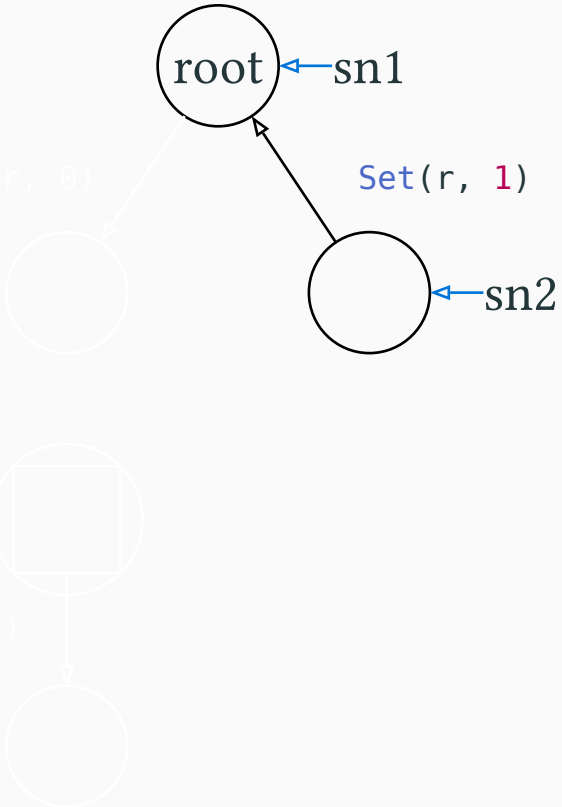
$r \mapsto 1$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

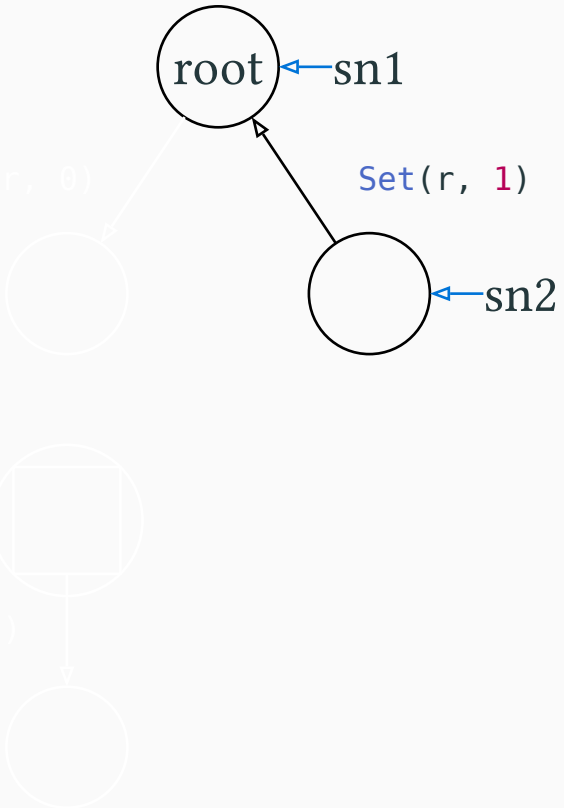
$r \mapsto 0$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

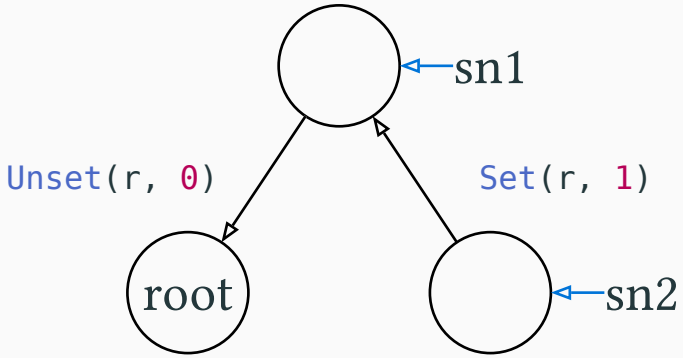
$r \mapsto 0$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

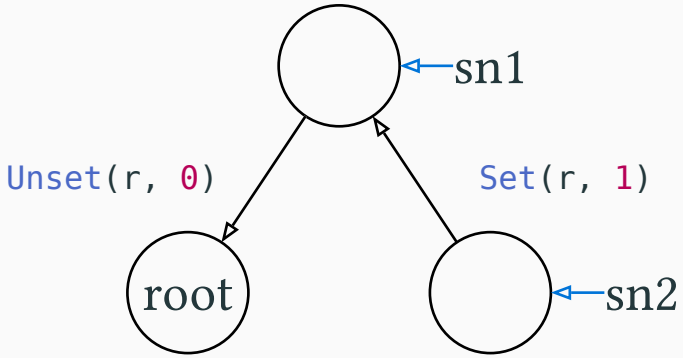
$r \mapsto 2$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

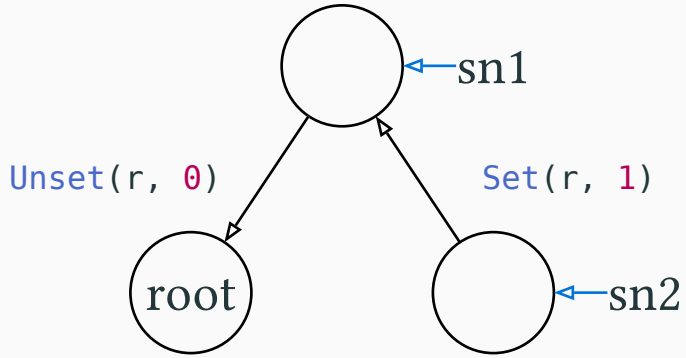
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

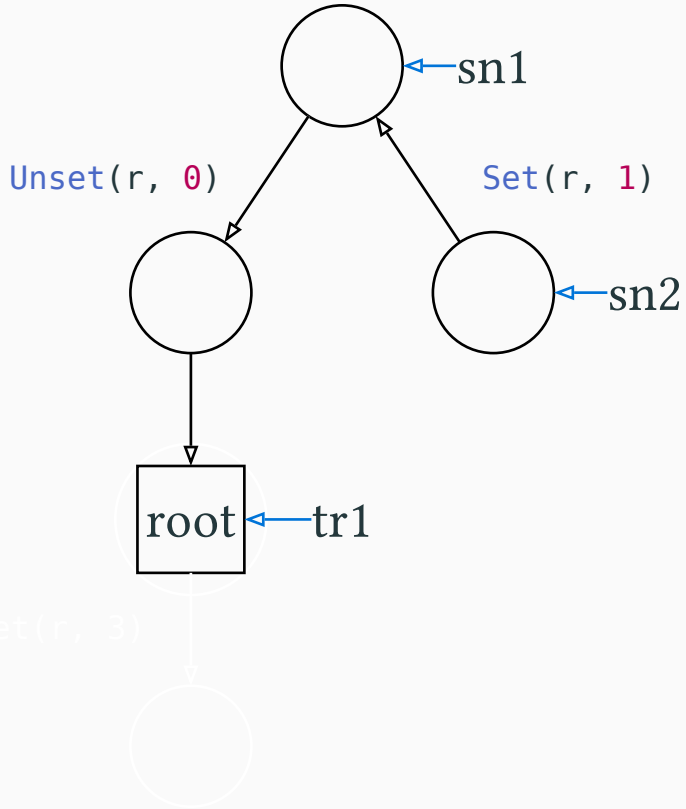
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

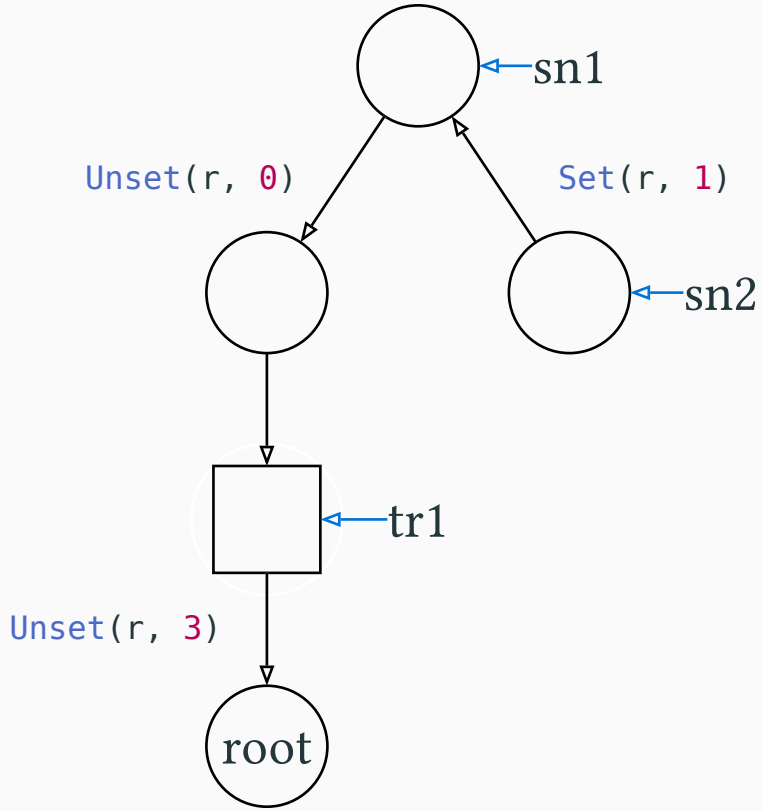
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

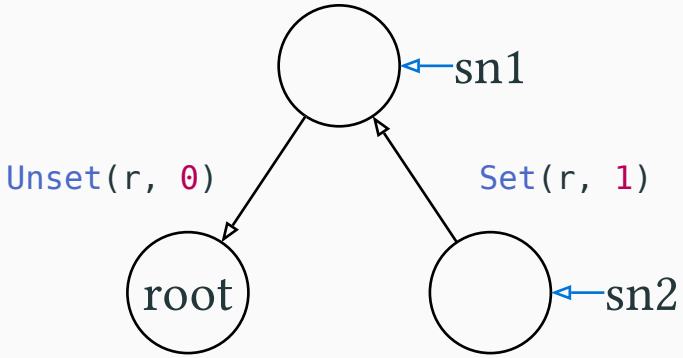
$r \mapsto 4$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

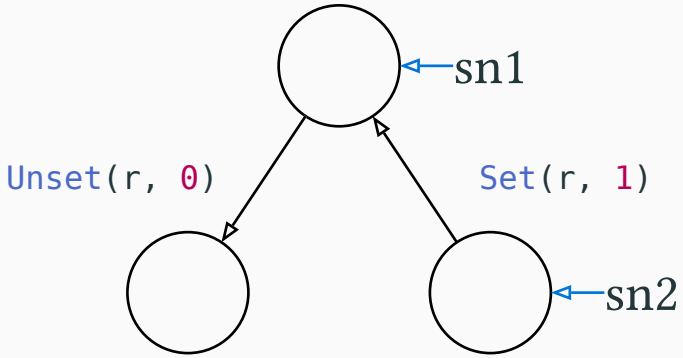
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

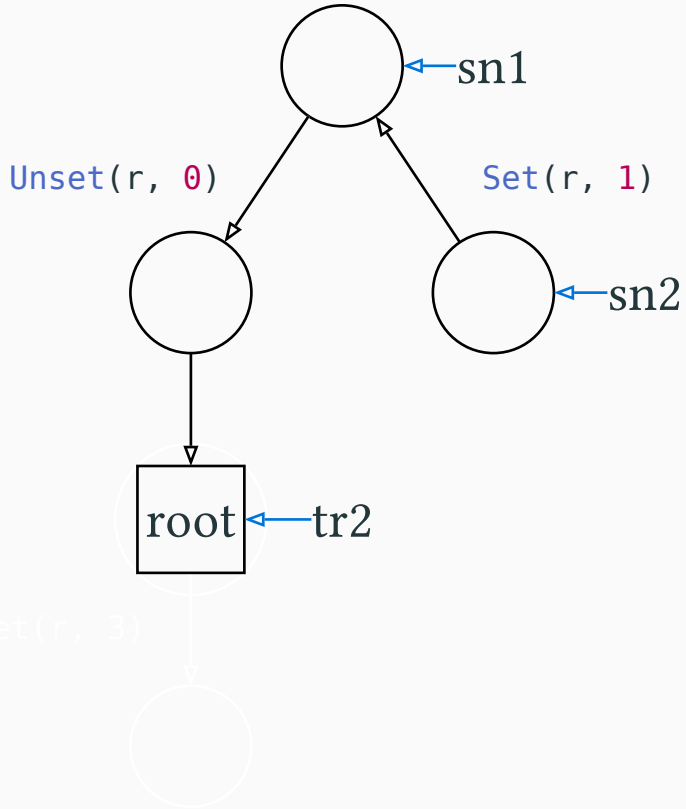
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

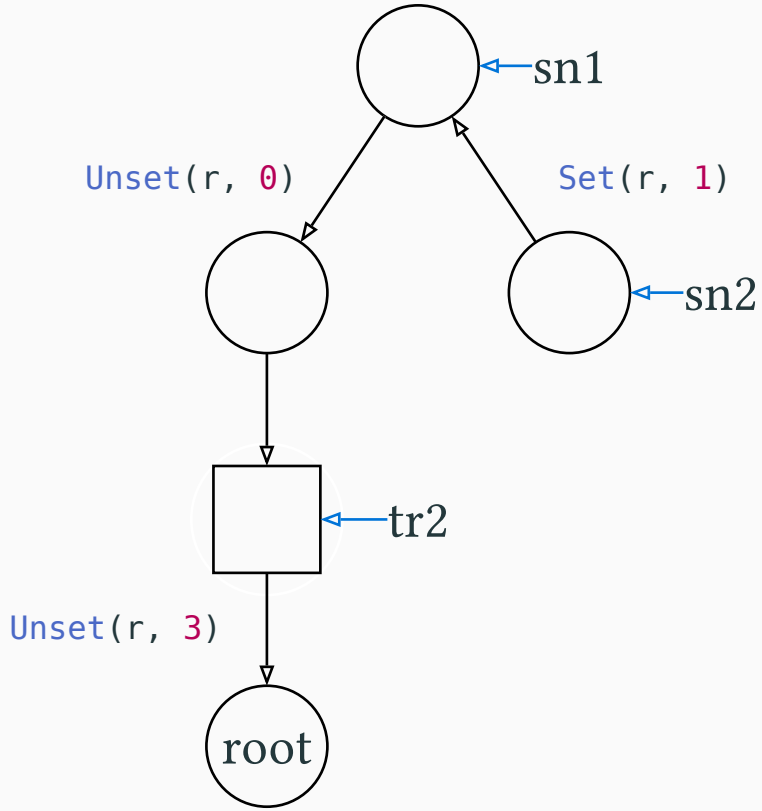
$r \mapsto 3$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

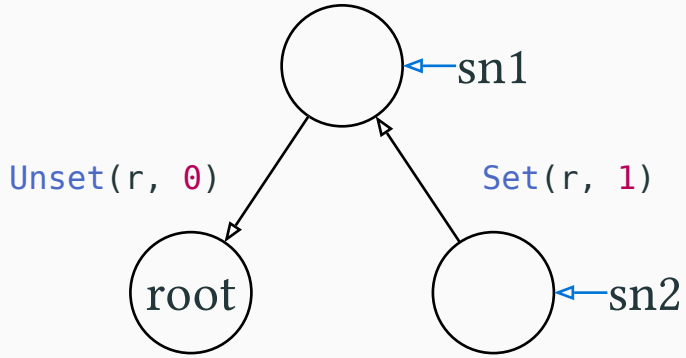
$r \mapsto 4$



Demo! (+ parsimony)

```
1   let s = Store.create () in
2   let r = Store.Ref.make s 0 in
3   let sn1 = Store.snapshot s in
4
5   Store.Ref.set s 1;
6   let sn2 = Store.snapshot s in
7   Store.restore s sn1;
8
9   Store.Ref.set s 2;
10  Store.Ref.set s 3; (* record elision *)
11
12  let tr1 = Store.transaction s in
13  Store.Ref.set s 4;
14  Store.rollback tr1;
15
16  let tr2 = Store.transaction s in
17  Store.Ref.set s 4;
18  Store.commit tr2; (* commit compression *)
```

$r \mapsto 4$



Why the follow-up work?

To add backtracking to a mutable data structure,
simply replace all mutability points with Store references.

Why the follow-up work?

To add backtracking to a mutable data structure,
simply replace all mutability points with Store references.

Why the follow-up work?

To add backtracking to a mutable data structure,
simply replace all mutability points with Store references.

Why the follow-up work?

To add backtracking to a mutable data structure,
simply replace all mutability points with Store references.

Fine for simple data structures whose implementation you own.

What if the implementation is very complex or outside your control?

Storable types

The present work: *storable types* in Store:

- pick an existing data structure
- write some glue code to let Store backtrack over it
- voilà: integrated into your Store-using application

Storable types

The present work: *storable types* in Store:

- pick an existing data structure
- write some glue code to let Store backtrack over it
- voilà: integrated into your Store-using application

What is the interface to describe this backtracking logic?

Storable types

The present work: *storable types* in Store:

- pick an existing data structure
- write some glue code to let Store backtrack over it
- voilà: integrated into your Store-using application

What is the interface to describe this backtracking logic?

We don’t know.

Storable types

The present work: *storable types* in Store:

- pick an existing data structure
- write some glue code to let Store backtrack over it
- voilà: integrated into your Store-using application

What is the interface to describe this backtracking logic?

We don’t know.

We have three interfaces.

Storable types

The present work: *storable types* in Store:

- pick an existing data structure
- write some glue code to let Store backtrack over it
- voilà: integrated into your Store-using application

What is the interface to describe this backtracking logic?

We don't know.

We have three interfaces.

Bonus question: what about record elision?

Operations, anti-operations

Operations update the state.

Anti-operations revert a state update.

For references:

```
type 'a op = Set of 'a       (* new value *)
type 'a antiop = Unset of 'a (* previous value *)
```

For an append-only stack (first attempt):

```
type 'a op = Push of 'a
type 'a antiop = Pop
```

No record elision: we must store several `Pop` nodes.

Composition of anti-operations (example)

For an append-only stack (first attempt):

```
1 type 'a op = Push of 'a
2 type 'a antiop = Pop
```

No record elision: we must record several `Pop` antioperations.

Composition of anti-operations (example)

For an append-only stack (first attempt):

```
type 'a op = Push of 'a
type 'a antiop = Pop
```

No record elision: we must record several `Pop` antioperations.
We could **compose** anti-operations by tweaking the interface.

```
type 'a op = Push of 'a
type 'a antiop = Pop of int
```

Composition of anti-operations: `(Pop m) . (Pop n)` is `Pop (m + n)`.
We can update the last recorded anti-operation by composition.

Composition of anti-operations (example)

For an append-only stack (first attempt):

```
type 'a op = Push of 'a
type 'a antiop = Pop
```

No record elision: we must record several `Pop` antioperations.
We could **compose** anti-operations by tweaking the interface.

```
type 'a op = Push of 'a
type 'a antiop = Pop of int
```

Composition of anti-operations: `(Pop m) . (Pop n)` is `Pop (m + n)`.

We can update the last recorded anti-operation by composition.
A different presentation of anti-operations allows record elision:

```
type 'a op = Push of 'a
type 'a antiop = Truncate of {length:int}
```

Composition of anti-operations (general)

Users come with their data-structure and a set of update operations.

We ask them to design a monoid $(A, \cdot)$
of **antioperations** A with a **composition** operation $(\cdot)$.

We identify three important classes of anti-operations:

1. **free**: $(A, \cdot)$ is a free monoid: record individual nodes
2. **absorbing**: $\forall ab,\ \ a.b = a$
   reference-like, benefit from record elision
3. **custom**: neither free nor absorbing

Composition properties $\Longrightarrow$ performance tradeoffs.

We offer three APIs, in increasing order of complexity.

`Free`: API + example

```
module Free : sig
  val modify :
    store -> 'a -> op:('a -> unit) -> anti:('a -> unit) -> unit
end
```

Example:

```
module RareRef = struct
  type 'a t = 'a ref
  [..]
  let set store r v =
    let setter v = fun r -> r := v in
    Free.modify store r (setter v) (setter !r)
end
```

```
module Absorbing : sig
  type ('a, 'op, 'anti) descr = {
    capture : 'a -> 'anti;
    rollback : 'a -> 'anti -> unit;
    undo : 'a -> 'anti -> 'op;
    redo : 'a -> 'op -> 'anti;
  }

  type ('a, 'op, 'anti) t
  val make : ('a, 'op, 'anti) descr -> 'a -> ('a, 'op, 'anti) t
  val get : ('a, 'op, 'anti) t -> 'a
  val modify : store -> 'a t -> unit
end
```

```
(* internally *) type _ t = Absorbing : { value : 'a;
                  mutable generation : Generation.t;
                      descr : ('a, 'op, 'anti) descr; } -> 'a t
```

`Absorbing`: Example

```
1   module PushQueue = struct
2     type 'a t = ('a Dynarray.t, 'a array, int) Absorbing.t
3     let capture = Dynarray.length
4     let rollback = Dynarray.truncate
5     let undo arr n = [..]
6     let redo arr suff = [..]
7     let descr = { capture; rollback; undo; redo; }
8
9     let create _s =
10      Absorbing.make descr (Dynarray.create ())
11    let push s d v =
12      Absorbing.modify s d;
13      Dynarray.add_last (Absorbing.get d) v
14    let length _s d =
15      Dynarray.length (Absorbing.get d)
16  end
```

Custom: API

```
module Custom : sig
  type ('a, 'op, 'trail) descr = {
    record : 'a -> 'trail;
    append : 'trail -> 'trail -> unit;
    rollback : 'a -> 'trail -> unit;
    undo : 'a -> 'trail -> 'op;
    redo : 'a -> 'op -> 'trail -> unit;
  }

  type ('a, 'op, 'trail) t
  val make : ('a, 'op, 'trail) descr -> 'a -> ('a, 'op, 'trail) t
  val get : ('a, 'op, 'trail) t -> 'a
  val modify : store -> ('a, 'op, 'trail) t -> 'trail
end
```

`Custom`: Example

```
1 module Queue = struct
2   type 'a op = { truncate: int; suffix : 'a Dynarray.t }
3   type 'a trail = { mutable truncate: int; mutable suffix : 'a list }
4   type 'a t = ('a, 'a op, 'a trail) Custom.t
5   [..] (* you don't want to see the rest... *)
6 end
```

Conclusion

What's better than a new API? **Three** new APIs for storable types in Store.

Also: a solid basis for us to add built-in data structures to Store: queues, hashtables, etc.

SMT authors want this. (Basile Clément works on Alt-Ergo at OCamlPro)

Take-away: the performance of recording operations depends on
the algebraic structure of anti-operations.

Full code: https://gitlab.com/basile.clement/store/-/tree/jfla2025+existentials

… soon in a new `Store` release.