# Backtracking reference stores

Camille Noûs, Gabriel Scherer

## To cite this version:

Camille Noûs, Gabriel Scherer. Backtracking reference stores. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs, Jan 2023, Praz-sur-Arly, France. pp.190-210. hal-03936704

**HAL Id: hal-03936704**
**https://inria.hal.science/hal-03936704**

Submitted on 12 Jan 2023

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Backtracking reference stores

Camille Noûs[1] and Gabriel Scherer[2]

[1] Laboratoire Cogitamus
[2] INRIA

**Abstract**

François Pottier's unionFind library is parameterized over an underlying store of mutable references, and provides the usual references, transactional reference stores (for rolling back some changes in case of higher-level errors), and persistent reference stores. We extend this library with a new implementation of backtracking reference stores, to get a Union-Find implementation that efficiently supports arbitrary backtracking and also subsumes the transactional interface.

Our backtracking reference stores are not specific to unionFind, they can be used to build arbitrary backtracking data structures. The natural implementation, using a journal to record all writes, provides amortized-constant-time operations with a space overhead linear in the number of store updates. A refined implementation reduces the memory overhead to be linear in the number of store cells updated, and gives performance that match non-backtracking references in practice.

# 1 Introduction

Our own use-case for the present work comes from implementing a type-checker for an explicitly-typed (no inference required) extension of System F with Guarded Algebraic Datatypes (GADTs). With GADTs, datatype constructors may witness equalities between types. For example, matching on the constructor $\mathsf{Int}$ at type $\alpha\ \mathsf{rtty}$ may reveal that the (universally quantified) type variable $\alpha$ is in fact equal to $\mathsf{int}$, that is, introduce the equality $(\alpha = \mathsf{int})$ in the typing context. Then under this context we have to type-check a sub-expression, the right-hand-side of the pattern matching clause, which in particular involves checking many type equalities $(\Gamma \vdash \tau_1 = \tau_2)$ modulo the equations in $\Gamma$. For example, the types $(\alpha \to \mathsf{int})$ and $(\mathsf{int} \to \alpha)$ are distinct in general, but they are equal under the assumption $(\alpha = \mathsf{int})$.

Union-Find is the perfect data structure to efficiently check equalities in presence of equality assumptions. Our type-checker carries a global Union-Find graph, we create nodes for bound type variables, equality assumptions add unification edges in the graph, and equality checking can be implemented efficiently – even in presence of cyclic graphs / equi-recursive types. But adding a *local* equality assumption, as required by GADTs, requires a form of *backtracking*: we add unification edges before checking the sub-expressions of the GADT pattern clause, but these edges should not remain present when type-checking the rest of the term outside this clause.

A simple solution to this problem is to use a *persistent* Union-Find; there is a generic way to implement a persistent version of a mutable data-structure, which is to replace all mutable references by indices into a persistent map. When we extend a persistent Union-Find graph $G$ with a new equality, we get a new Union-Find graph $G'$; we can type-check the clause right-hand-side using $G'$ and then go back to $G$ to check the rest of the type derivation.

```
let rec typeof env = function (* ... *)
  | Use (t, u) ->
      match typeof env t with
      | TyEq (ty1, ty2) ->
        let env = introduce_equalities env ty1 ty2 in
        typeof env u
      | ty -> raise (TypeError (NotAnEquality ty))
```

However, this solution imposes a logarithmic overhead to all accesses and updates, which adds a noticeable constant factor on reasonably-large programs. This is especially frustrating given that GADTs are a relatively rare language feature; the whole type-checker pays for a feature that may not even occur in the program being type-checked.

Another solution is to use a mutable, constant-time-access Union-Find graph, but perform a copy of the graph when entering a GADT equality assumption. This has cost linear in the number of type variables in context, which is unpleasant, but the cost is localized to GADT use sites. This is a reasonable solution that, let's be frank, would probably be fine in practice for our use-case.

```
      match typeof env t with
      | TyEq (ty1, ty2) ->
        let env = { env with store = StoreVector.copy store } in
        introduce_equalities env ty1 ty2;
        typeof env u
```

We want to do better by implementing a Union-Find that supports both constant-time access/update and (amortized) constant-time backtracking.

```
      match typeof env t with
      | TyEq (ty1, ty2) ->
        BtRef.branch env.store;
        let finally () = BtRef.terminate_rollback env.store in
        Fun.protect ~finally @@ fun () ->
        introduce_equalities env ty1 ty2;
        typeof env u
```

## 1.1 A trusted ally: a modular Union-Find library

François Pottier's unionFind library provides an efficient implementation of Union-Find parameterized over a notion of *mutable store*, which allows to choose the implementation of mutable references used in the Union-Find graph. At the time of writing, it provides four implementations of this store interface:

**StoreRef** uses OCaml references directly. This gives the standard UnionFind behavior.

**StoreMap** uses indices into a persistent map. This gives a mutable API that operates on a persistent graph under the hood, providing a constant-time operation to copy (share) the whole graph. Algorithmically this corresponds exactly to the persistent approach we discussed above.

**StoreVector** uses indices into a dynamic array. This gives constant-time access and update, but tracks the set of nodes of the graph (and refers them through indices, a portable indirection) which allows copying the whole graph in linear time. This corresponds exactly to the copying approach we discussed above.

**StoreTransactionalRef** implements a *transactional* reference store, which allow starting a global transaction with the ability to either commit all changes that happened during the transaction or to roll them back. This is important for type-checking algorithms that provide partial operations (typically unification) on top of the Union-Find graph. An operation may perform perform several unifications in sequence, but it should leave the graph unchanged in case the operation fails – typically to print error messages to the user without showing an inconsistent, in-progress state. Transactions cannot be nested.

## 1.2 Our contribution

We implement a *backtracking reference store* that allows to backtrack to previous versions of the store in amortized constant time. A store contains many refereences that are versioned together. There are explicit operations to *branch* the store into a new version, or return a store to its parent version. In particular, if we roll back to a previous version, this rolls back the value of all references in the store that were modified since the last branch. We can instantiate François Pottier's unionFind implementation with our new store, solving our quest for Union-Find with backtracking.

**Implementation** The implementation of our structure is fairly simple, Intuitively the store maintains a *journal*, or *undo log*, of all reference updates, which allows to roll back to previous versions. In fact we have two implementations:

- In our first, simplest implementation, all updates are logged in the journal. Read/write operations are constant-time, backtracking to the parent version is amortized constant-time. But the journal adds a space overhead linear in the number of updates and branches performed. (This is just fine for Union-Find, where the number of updates per node is less than logarithmic. But our store may have other users without such guarantee.)
- We then propose a *record-eliding* optimization that uses timestamps to avoid recording a given reference twice for the same version of the structure. The space overhead is linear in the number of references updated in each version, rather than the number of writes.

**Interface** The building blocks of our implementations subsume two different flavors of interfaces that we can express on top of our work:

- We can provide a semi-persistent interface in the sense of Conchon and Filliâtre (2008).
- We provide a transactional interface, subsuming François Pottier's transactional references, but in addition we support arbitrary nesting of transactions.

**Terminology: *backtracking*, *semi-persistent*, *transactional*** Conchon and Filliâtre (2008) proposes a specific API design for *semi-persistent* structures where backtracking to a previous version is implicit. It happens on-demand when we start again to work on an older version, the first access to this older version implicitly invalidates/backtrack any more recent version. This gives a very declarative programming style, close in spirit to programming with persistent data structurs, hence the name *semi-persistent*.

Our core API is different, more imperative. Terminating a version is explicit; this works better with the transactional interface that gives two different ways to terminate a version (`commit` or `abort`), making the implicit choice awkward. But one can easily build the semi-peersistent API on top of our existing interface – we do it in Section 4.2. We call our API a *backtracking* API, although it is a bit more expressive than typical backtracking APIs that only allow to rollback or abort changes, while we can also commit them – as with transactional interfaces.

A more precise name may be *transactional*, but we already use it in the context of François Pottier's StoreTransactionalRef stores that do not support nested transactions. We could call them *one-transactional*, or call ours *nested-transactional*, but we stick with the more common *backtracking*.

**Generality** We emphasize that while our own use-case is a backtracking Union-Find, our backtracking (and transactional) references are of general interest and could be used to implement basically any backtracking data-structure.

For example, you can rebuild backtracking arrays by simply using arrays of references in a shared backtracking store – at the constant-factor cost of an extra isndirection per array element. This would be silly for backtracking or semi-persistent arrays where specialized implementations are already available, or for structures that are naturally built by composing existing backtracking libraries, in the same way Conchon and Filliâtre (2008) builds semi-persistent hashtables out of semi-persistent dynamic arrays and semi-persistent stacks. But you may be interested in backtracking doubly-linked lists, backtracking quad-trees, backtracking skiplists, etc.

## 1.3 Early challengers: (semi-)persistent dynamic arrays

As we were considering this journey of implementing our own backtracking references, the wizard Jean-Christophe Filliâtre sent us the following remark to test our resolve.

> It is possible to implement a Union-Find graph backed by a dynamic array, so you get a semi-persistent Union-Find by using an existing library of semi-persistent dynamic arrays.

This solution has in fact already been presented in Conchon and Filliâtre (2007), which proposes a persistent Union-Find implementation backed by a dynamic array, but tweaks it slightly to be only semi-persistent in its Section 2.3.3.

Our justification for writing our own code is that array-backed Union-Find graphs – including the use of François Pottier's unionFind instantiated with its StoreVector module we mentioned above – do not play nicely with garbage collection. All nodes of the graph remain alive as long as the graph lives. In contrast, direct representations where the Union-Find graph is realized by pointers in memory preserve the liveliness of individual nodes, which can be collected as soon as they are not used in the program anymore.

This makes a difference in our use-case of using a Union-Find graph on the fly for type-checking: when we traverse our typing derivation, we generate many Union-Find nodes on the fly, each time we check an equality between types. But they are extremely short-lived, we do not use them after we have checked those equalities. We expect those dead fragments of the Union-Find graph to be collected promptly, while they leak when using an array-backed implementation.

Note that transactional or backtracking references also extend the lifetime of values in certain circumstances: to allow rolling back to a previous version we necessarily keep those previous versions alive in memory. But this lifetime-extension ends when the corresponding transaction or version is terminated. In our Union-Find use-case, this means that the garbage-collection behavior of Union-Find nodes is unchanged by these journaling mechanisms.

Finally, (semi-)persistent arrays, in existing OCaml implementations, do not provide the equivalent of our record-eliding optimization – their memory overhead is linear in the total number of writes, instead of the number of distinct positions written. It may be possible to implement it; we suspect that it is non-obvious for semi-persistent arrays and difficult for persistent arrays. If we want record elision, we need to write new code anyway.

**Remark: persistent vs. semi-persistent** Semi-persistent data-structures were introduced in Conchon and Filliâtre (2008) as an optimization of fully-persistent data structures for scenarios where backtracking is required, but not more complex reuse scenarios. We mentioned earlier a naive strategy for persistence adding a logarithmic overhead on all access. But note that persistent data-structures can also be fast thanks to *rerooting* as presented in Conchon and Filliâtre (2007), almost as fast as semi-persistent data structures.

Reusing the existing implementation of a Union-Find graph backed by a persistent array would certainly be fine performance-wise; as with any array-backed approach it prevents garbage collection, and we believe that implementing the record-eliding optimization would be more difficult. Our record-eliding optimization relies on the fact that valid versions of a semi-persistent data structure form a linear structure, so they can be denoted by integers, with a fast check of whether a version is an ancestor of another. On the other hand, fully persistent data structures have a tree of versions, so our approach does not work.

# 2 Specification

In this section we present the set of primitive operations supported by our backtracking store of references, just their specification.

Note: we did not start by writing down a specification, we started by writing an implementation with a bug on nested transactions. We had to work out a clear specification to avoid the bug in the future.

## 2.1 Specifying backtracking data structures

Let us first recall the vocabulary to specify semi-persistent data structures proposed in Conchon and Filliâtre (2008).

In a semi-persistent implementation, there are several *versions* of the same mutable structures, in our case a store, that is, a set of references. Each version of the data-structure has a corresponding state, independent of the other versions.

There is a *current* (most recent) version.

We can *branch* a new version from the current version: it is a child of the current version and becomes the new current version. This new version starts in exactly the same state as its parent version – the previous current version.

We can *terminate* the current version, if it has a parent version. This parent version becomes the new current version. (Note: Conchon and Filliâtre (2008) does not discuss termination explicitly, so we are deviating slightly here.)

Remark that versions form a linear path, not a tree. We would have a tree if we could branch from any version, not just the current one, in particular we could have a version with several children. This would correspond to a fully persistent implementation.

There is a *root version* that is the current version when the store is just created, and has no parent version.

## 2.2 Backtracking store of references

Our backtracking data-structure is a *store* of *references*. We follow François Pottier's design for stores of references (modulo simplifications for presentation). A store is a set of references and each references belongs to exactly one store.

### Usual store operations

```
type store
val new_store : unit -> store

type 'a rref
val make: store -> 'a -> 'a rref
val get:  store -> 'a rref -> 'a
val set:  store -> 'a rref -> 'a -> unit
```

The *state* of a (usual) store of references is just a mapping from each reference to a value of its content type; `get` and `set` operations modify the state as expected.

With our backtracking stores, a store has several versions, and the logical state of each version is exactly this mapping from each reference in the store to a value.

`make` adds a new reference to the store. Note that this operation is independent from the current version: the state of all versions of the store now has a mapping for this reference.

`get` and `set` operate on the current version of the store, as expected.

### Creating new versions

```
val branch : store -> unit
val terminate_nodiff : store -> unit
```

`branch` has the usual specification for a semi-persistent data structure: it creates a new version, child of the current version, which becomes the new current version, and initially has the same state as the previous current version.

We provide a `terminate_nodiff` function to terminate the current version; its parent version becomes the new parent version. For reasons that will become apparent in the next paragraph, this function assumes that the state of the current version is equal to the state of its parent. (It fails if the current version is the root version.)

### Transactions

```
val commit : store -> unit
val rollback : store -> unit
```

Those functions provide the necessary primitives to implement François Pottier's transactional interface on top of our backtracking store – and a bit more. They require the current version to have a parent version, and fail if the current version is the root version.

`commit` changes the state of the parent version to become identical to the state of the current version.

`rollback` changes the state of the current version to become identical to the state of the parent version.

Those operations do not terminate the current version, but one can call `terminate_nodiff` after them – it expects the current and parent version to be in the same state, which they both enforce.

Note: it would be natural to provide a higher-level interface that offers only `terminate_commit` and `terminate_rollback`, hiding the harder-to-use `terminate_nodiff` function.

**Performance** We expect a *fast path* behavior of `get` and `set` when no version has been branched (see below). In that case, `set` should write the content of the reference without any extra book-keeping, so that the performance is very close to standard references. Backtracking must be a *low-cost* abstraction, that only adds noticeable overhead when it is actually used. (This fast path behavior is also present in the StoreTransactionalRef implementation of François Pottier, so it is important that we support it to subsume that implementation.)

# 3 Implementation(s)

Our references are implemented as record with a mutable field. (Our record-eliding implementation will get an extra metadata field, discussed in the relevant section.)

```
type 'a rref = { mutable current: 'a; }
```

We maintain the invariant that the *global state* of a store, the content of the `current` field of each reference, is equal to the logical state of the current version of the store. This guarantees in particular that `get` can be implemented with a single read of a mutable field.

The general idea of our implementations is that the store maintains a journal of writes to its references, so that those writes can be rolled back when backtracking.

To undo a write/set, we need the reference that was written and the value of the reference before the write:

```
type undo_action =
  | Set : 'a rref * 'a -> undo_action
```

Note that if we supported other types of backtracking mutable objects in the store, for example arrays, we could add more undo actions (for an array write we would store the array, the index and the old value at this index).

We went through three different implementations, from the less to the more sophisticated. They differ by the data-structure used to store undo actions, and whether we record an undo action for all writes or only some of them.

## 3.1 Stack of stacks

In our first implementation, the difference between a version and its parent version is stored as a stack of undo actions. Our versioned store is just a mutable list of such differences from the current version to the root version.

```
type diff = undo_action Stack.t
type store = diff list ref
```

When the store is empty, the current version is the root version. The root version has no parent, so it does not store a diff. When the store is of the form `d::ds`, `d` is a diff from the current version to its parent, and `ds` a list of diffs from the parent version to the root version. In other words, a store keeps the whole journal of all writes, but segmented in separate stacks, one for each parent-child relationship between versions.

```
let new_store () = ref []
let make (_s : store) (v : 'a) : 'a rref = { current = v; }
let get (_s : store) (x : 'a rref) : 'a = x.current
```

**Set** The specification of `set` says that it operates on the state of the current version of the store. In our representation, we maintain the invariant that the state of the current version is equal to the global state of the references, so `set` must update this global state. We also maintain the invariant that the store diff list relates the state of the current version to the state of the root version; the current version changes, so `set` must also update the *current diff*, the diff between the current version and its parent.

```
let set (s : store) (x : 'a rref) (v : 'a) : unit =
  begin (* Update the diff list. *)
    match !s with
      (* If the diff list is empty, the current version is the root version
         which has no undo log, so there is nothing to do. *)
    | [] -> ()
    | current_diff :: _rest ->
      (* Add the write to the diff of the current version *)
      Stack.push (Set (x, x.current)) current_diff;
  end;
  (* Update the global state. *)
  x.current <- v
```

In the case where the current version is the root version, our representation does not maintain an undo log for the root version so we do not record the write. The root version has no parent so commit or rollback cannot be called, so we cannot observe the presence of an undo log.

### Branch and terminate

```
let branch s =
  s := Stack.create () :: !s
let terminate_nodiff s =
  match !s with
  | [] -> invalid_arg "terminate_nodiff_version: root version cannot be terminated"
  | diff :: rest ->
    assert (Stack.is_empty diff);
    s := rest
```

Note: checking that diff is empty is slightly stronger than our specification, which allows non-empty diffs of updates that cancel each other. Our implementation enforces the more intentional specification that terminate_nodiff is always called immediately after branch,commit or rollback.

**Rollback and commit** rollback mutates the current state to become equal to the parent state, while commit mutates the parent state to become equal to the current state.

```
let rollback s =
  match !s with
  | [] ->
    invalid_arg "rollback: the root version cannot be rolled back"
  | current_diff :: _ ->
    (* Current state:
         $\left\{ \begin{array}{c}\textit{current version}\\ \textit{in state A}\end{array} \xrightarrow{\textit{current diff}} \begin{array}{c}\textit{parent version}\\ \textit{in state B}\end{array} \right\}$
    *)
    while not (Stack.is_empty current_diff) do
      match Stack.pop current_diff with
      | Set (x, old) ->
        x.current <- old
    done;
    (* Final state:
         $\left\{ \begin{array}{c}\textit{current version}\\ \textit{in state B}\end{array} \xrightarrow{\emptyset} \begin{array}{c}\textit{parent version}\\ \textit{in state B}\end{array} \right\}$
    *)
```

`commit` relies on an auxiliary function on stacks, `move_stack s1 ~into:s2`. It moves all elements of `s1` into `s2`, in the same order as they were in `s1`, which is now empty. In other words, if before the call the list of elements of `s1` and s2 were `l1` and `l2` respectively, then after the call it is `[]` and `l1 @ l2`.

```
let commit s =
  match !s with
  | [] ->
    invalid_arg "rollback: the root version cannot be committed"
  | diff :: [] ->
    (* Current state:
```

$$\left\{ \begin{array}{ccc} \textit{current version} & \xrightarrow{\textit{diff}} & \textit{root version} \\ \textit{in state } A & & \textit{in state } B \end{array} \right\}$$

```
    *)
    Stack.clear diff;
    (* Final state:
```

$$\left\{ \begin{array}{ccc} \textit{current version} & \overset{\emptyset}{=\!=} & \textit{root version} \\ \textit{in state } A & & \textit{in state } A \end{array} \right\}$$

```
    *)
  | diff :: parent_diff :: _ ->
    (* Current state:
```

$$\left\{ \begin{array}{ccccc} \textit{current version} & \xrightarrow{\textit{diff}} & \textit{parent version} & \xrightarrow{\textit{parent diff}} & \textit{parent parent version} \\ \textit{in state } A & & \textit{in state } B & & \textit{in state } C \end{array} \right\}$$

```
     *)
    move_stack diff ~into:parent_diff;
    (* Final state:
```

$$\left\{ \begin{array}{ccccc} \textit{current version} & \overset{\emptyset}{=\!=} & \textit{parent version} & \xrightarrow{\textit{diff} + \textit{parent diff}} & \textit{parent parent version} \\ \textit{in state } A & & \textit{in state } A & & \textit{in state } C \end{array} \right\}$$

```
    *)
```

**Complexity** Ideally we want each operation to take constant time, possibly amortized. Amortized reasoning works well with `rollback`. It traverses the undo log of the current version to perform each undo action, removing it from the diff. Each undo action is rolled back at most once, and we can consider that its rollback cost was paid in advance by the corresponding `set`.

This reasoning fails for `commit` unfortunately. `move_stack` traverses the current undo log, and this traversal cost can be amortized, but the undo actions are not dropped after traversal, they remain in the diff of another version. One would need `set` to pre-pay for each version, which is not a constant cost. Amortized reasoning fails, and in fact we can observe quadratic complexity in the worst case: consider a sequence of $N$ `branch` calls, then $N$ writes, then $N$ (`commit`; `terminate_nodiff`) sequences, this runs in $O(N^2)$. (This is not a concern if you only use backtracking and never the transactional interface, as in this case you always `rollback` and never `commit`.)

It would be possible to change the `diff` data-structure from stacks to something that provides constant-time concatenation, such as doubly-linked lists. Instead we move to our second implementation, which uses simpler data structures and can thus be expected to have lower constant factors – at the cost of being less abstract.

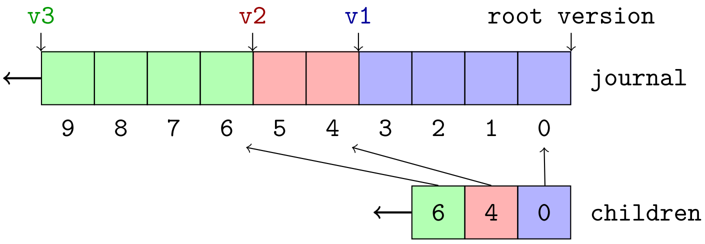
## 3.2 Stack and indices

In our second iteration, we use a single stack to store the undo actions of all versions. We remember the boundaries of each version in a separate list.

```
type store = { journal: undo_action Stack.t;
               mutable children: int list;   }
```

For each child (non-root) version, the list `children` stores the index of this version in the `journal`, that is, the starting position of the diff of this version. The children versions are stored in the list from the most recent (the current version) to the oldest (the immediate child of the root version). Note that the root version keeps no diff / undo log, so its immediate child always starts at position 0.

For example, a store with three children versions in addition to the root version may have the following representation – we draw the stacks and lists as growing from the right to the left.



**Easy changes** Most functions are trivially adapted to this new representation. In the case of `set`, we have to check the list `s.children` to tell whether we are at the root version (which stores no diff) or not, and only push to `journal` in the non-root case:

```
let set (s : store) (x : 'a rref) (v : 'a) : unit =
  begin match s.children with
    | [] -> ()
    | _ :: _ ->
      Stack.push (Set (x, x.current)) s.journal;
  end;
  x.current <- v
```

`branch` uses the current length of the journal as the starting position of the new version:

```
let branch s =
  s.children <- Stack.length s.journal :: s.children
```

Finally, `rollback` performs some length computation to determine how many undo actions to roll back from the journal:

```
let rollback s =
  match s.children with
  | [] ->
    invalid_arg "rollback: the root version cannot be rolled back"
  | curr_start :: _ ->
    let current_version_length = Stack.length s.journal - curr_start in
    for _i = 1 to current_version_length do
      match Stack.pop s.journal with
      | Set (x, old) ->
        x.current <- old
    done;
    assert (Stack.length s.journal = curr_start);
```

**Commit** This function motivated the change of representation. In the case where the parent version is not the root version, `commit` does not touch the journal at all – the undo actions remain in the same position in the stack, we only needs to adjust the version boundaries. We move the start of the current version to the end of the stack, so that the parent version adopts the undo actions that were previously in the current version.

```
let commit s =
  match s.children with
  | [] ->
    invalid_arg "rollback: the root version cannot be committed"
  | _curr_start :: parent_start :: rest ->
    (* Current state:
          ⎧   end of stack      current diff   pos. curr_start    parent diff      pos. parent_start     ⎫
          ⎨  current version  <-------------   parent version   <-------------  parent parent version  ⎬
          ⎩     state: A                          state: B                            state: C           ⎭
       We move the boundary of our current version to the current
       stack length. This moves the content of our diff to our
       parent version. *)
    s.children <- Stack.length s.journal :: parent_start :: rest;
    (* Final state:
          ⎧   end of stack        ∅    pos. Stack.length      current diff       pos. parent_start     ⎫
          ⎨  current version  ======    parent version      + parent diff    parent parent version  ⎬
          ⎩     state: A                   state: A        <----------------       state: C           ⎭
    *)

  | curr_start :: [] ->
    (* Current state:
          ⎧   end of stack      current diff   pos. curr_start = 0   ⎫
          ⎨  current version  <-------------      root version       ⎬
          ⎩     state: A                            state: B          ⎭
       The current version is the immediate child of the root version,
       which has no undo log. We know that the current version starts in
       position [0], and can discard the changes in our undo log. *)
    assert (curr_start = 0);
    Stack.clear s.journal;
    (* Final state:
          ⎧   end of stack        ∅    pos. curr_start = 0   ⎫
          ⎨  current version  ======      root version       ⎬
          ⎩     state: A                    state: A          ⎭
    *)
```

In the case where the root version is the parent version, a linear cost may remain — depending on the complexity of `Stack.clear`. But there is no complexity issue there, this is amortized constant time. Indeed, we only pay this cost once per undo action, and then the actions are removed from the journal. We can consider that `set` paid for this cost in advance.

## 3.3 Record elision

Our last iteration is a small modification of our previous version that avoids recording several writes to the same cell in a given version. Recording only the first write suffices to rollback the changes if necessary.

During `set` we could walk back the undo log of the current version, and exit early if we find another write to the same reference. This would be very slow. Instead we store, in each reference, information on when the last write to this reference was recorded – using positions inside the journal as a natural notion of *timestamp*.

```
type 'a rref = {
  mutable current: 'a;
    (** The value of this reference in the current version. *)
  mutable last_record: int;
    (** [last_record] is the position of the most recent
        record of this reference recorded in the journal,
        or -1 this reference has no record. *)
}

let make (_s : store) (v : 'a) : 'a rref =
  { current = v; last_record = -1; }
```

(Our actual code uses an abstract type `Pos_option.t` instead of `int` to hide the `-1` encoding, but auxiliary abstractions would make the presentation heavier here.)

We need to save this `last_record` field in our undo actions to be able to restore it correctly during `rollback`:

```
type undo_action =
  | Set : {
    ref: 'a rref;
      (** the reference that was written *)
    previous: 'a;
      (** the value of the reference before the write *)
    previous_record: int;
      (** the [last_record] value at write time *)
  } -> undo_action
```

**Set** We can skip recording the write if the last record belongs to the current version. We update the `last_record` field when we extend the journal.

```
let set (s : store) (x : 'a rref) (v : 'a) : unit =
  begin match s.children with
    | [] -> ()
    | current_version_start :: _ ->
      if x.last_record >= current_version_start then ()
      else begin
        let new_record = Stack.length s.journal in
        Stack.push (Set {
          ref = x;
          previous = x.current;
          previous_record = x.last_record;
        }) s.journal;
        x.last_record <- new_record;
      end
  end;
  x.current <- v
```

Note that while the space-overhead complexity reduction may feel nebulous for many applications, this optimization also makes a qualitative difference in performance. Indeed, the (polymorphic) writes dominate the cost of `set`; with the previous implementations, each `set` would perform two such writes, one to the reference field and the other by pushing into the journal. With the new implementation, in the common case we perform a single write. Outside the root-version fast path, our previous `set` implementations was at least twice slower than with standard references, while the overhead of the new version becomes neglectible for write-heavy workflows. Said otherwise, this implementation is low-cost even in presence of infrequent branching.

**Commit** There is a subtlety in the case where our parent version is the root version. The previous implementation simply clears the journal in this case, but doing so invalidates the `last_record` fields of the references mentioned in this journal, pointing to journal entries that no longer exist. This turns into a bug later where `set` does record a reference, falsely believing that it is already recorded. To avoid creating invalid `last_record` positions, we must clear this field for the references mentioned in the journal. We highlight the only lines of code that change since the last version:

```
let commit s =
  match s.children with
  | [] -> invalid_arg "rollback: the root version cannot be committed"
  | _curr_start :: parent_start :: rest ->
    s.children <- Stack.length s.journal :: parent_start :: rest;
  | curr_start :: [] ->
    assert (curr_start = 0);
    s.journal |> Stack.iter (function
      | Set {ref; previous = _; previous_record = _} ->
        ref.last_record <- -1;
    );
    Stack.clear s.journal;
```

Note that, when the parent version is not the root version, it may be the case that a single reference was recorded in the current diff and also in the parent diff. In this case it ends up mentioned twice in the final diff of the parent version. We do not preserve the invariant that a reference is recorded at most once in each version. Enforcing this invariant saves no work, and the obvious way to do it is to perform a linear traversal during `commit`, which would create again a quadratic worst case.

# 4 Interface(s)

```
module BtRef = StoreBacktrackingRef
let rec typeof env = function (*...*) | Use (t, u) ->
  match typeof env t with
  | TyEq (ty1, ty2) ->
      BtRef.branch env.store;
      let finally () =
        BtRef.rollback env.store;
        BtRef.terminate_nodiff env.store; in
      Fun.protect ~finally @@ fun () ->
      introduce_equalities env ty1 ty2;
      typeof env u
```

The primitives defined previously are enough to use our Union-Find with backtracking. In this section we build two higher-level interfaces, to show that our primitives are expressive enough. One expresses the transactional API of François Pottier, the other expresses the original semi-persistent API of Conchon and Filliâtre (2008).

## 4.1 Transactional API

François Pottier exposes the following function for his transactional reference implementation, StoreTransactionalRef:

```
(**[tentatively s f] runs the function [f] within a new transaction on the
   store [s]. If [f] raises an exception, then the transaction is aborted, and
   all updates performed by [f] on references in the store [s] are rolled
   back. If [f] terminates normally, then the updates performed by [f] are
   committed.

   Two transactions on a single store cannot be nested.

   A cell that is created during a transaction still exists after the
   transaction, even if the transaction is rolled back. In that case, its
   content should be considered undefined. *)
val tentatively: 'a store -> (unit -> 'b) -> 'b
```

We can implement this API for our backtracking references, with two improvements:

1. Our transactions can be nested at will. (This single change to the StoreTransactionalRef specification suffices to express arbitrary backtracking.)
2. Our specification naturally gives a meaning to the content of a reference that was created in a version since rolled back. In our specification, a reference is defined in all versions, and its initial value (in all versions) is the parameter provided to `make`.

As we remarked earlier, StoreTransactionalRef also provides the important property of being low-cost in the sense that read and writes outside a `tentatively` call are essentially as fast as raw references. We also preserve this property, thanks to the absence (in our implementations) of an undo log for the root version.

```
let tentatively (s : store) (f : unit -> 'b) : 'b =
  branch s;
  match f () with
  | v ->
    commit s;
    terminate_nodiff s;
    v
  | exception e ->
    let b = Printexc.get_raw_backtrace() in
    rollback s;
    terminate_nodiff s;
    Printexc.raise_with_backtrace e b
```

## 4.2 Semi-persistent API

The API proposed in Conchon and Filliâtre (2008) has a more declarative, less imperative flavor than ours.

```
module SemiPersistent : sig
  type versioned_store

  val new_store : unit -> versioned_store
  val branch : versioned_store -> versioned_store

  val make : versioned_store -> 'a -> 'a rref
  val get : versioned_store -> 'a rref -> 'a
  val set : versioned_store -> 'a rref -> 'a -> unit
end
```

This API manipulates *versioned stores*, whereas our imperative API acts on a global store. With this versioned API, `branch` returns a new store version. The functions `branch`, `get` or `set` do not need to be called on the current version, but they backtrack to the version they were given, invalidating/aborting any child version. There is no explicit way to terminate a version.

This gives nice, declarative code for our store-backtracking function.

```
module SPRef = StoreBacktrackingRef.SemiPersistent
let rec typeof env = function (*...*) | Use (t, u) ->
  match typeof env t with
  | TyEq (ty1, ty2) ->
      let env = { env with store = SPRef.branch env.store } in
      introduce_equalities env ty1 ty2;
      typeof env u
```

On the other hand, we are not sure how to gracefully integrate the transactional features, the difference between `commit` and `rollback`. This is related to the fact that termination of versions is implicit. We could add a `commit` function, but it might break user expectations by having the user of a child version modify its parent version.

(It is also somewhat awkward that `make` takes a versioned store as input but adds the reference to all versions.)

In fact, we can define a generic functor that takes a data-structure with our imperative API, and implements the declarative API on top.

```
module type Backtracking = sig
  type t
  val branch : t -> unit
  val terminate : t -> unit
end
```

(With our primitives, `terminate` is defined by calling `rollback` then `terminate_nodiff`.)

```
module type SemiPersistent = sig
  type t
  type version

  (* A root version with the given initial state;
     it is initially the current version for this state. *)
  val new_root : t -> version
```

```
    (* Branches a new version from a given version
       (any previous transitive child of the given version is invalidated);
       the new version becomes the new current version. *)
    val branch : version -> version

    (* The given version becomes the current version
       (any transitive child is invalidated);
       the corresponding global state is returned. *)
    val access : version -> t
  end

  module Make (D : Backtracking)
    : SemiPersistent with type t = D.t
  = struct ... end
```

The type version provided by this functor keeps the global state of the datastructure, along with a mutable status field that tracks whether it is currently the current version, is the parent of another version, or has been invalidated by backtracking.

```
  type t = D.t
  type version = {
    global : D.t;
    mutable status : status;
  }
  and status =
  | Current
  | Parent_of of version
  | Invalid

  let new_root global =
    { global; status = Current; }
```

Branching modifies the status of the branched version to track its child. We first implement branch_current, which assumes that it gets the current version.

```
  let branch_current version =
    assert (version.status = Current);
    D.branch version.global;
    let child = { global = version.global; status = Current } in
    version.status <- Parent_of child;
    child
```

We then implement a backtrack primitive that backtracks the global state to make a given version the current version.

```
  let backtrack version =
    (* [collect] accumulates the transitive children of a version. *)
    let rec collect children version =
      match version.status with
      | Current -> children
      | Invalid -> invalid_arg "backtrack: this version is invalid"
      | Parent_of child -> collect (child :: children) child
    in
    (* terminate and invalidate all children versions. *)
    collect [] version |> List.iter (fun child ->
      child.status <- Invalid;
      D.terminate version.global;
    );
    version.status <- Current
```

Finally, `branch` and `access` first backtrack to the given version, then branch a new version or access the underlying global structure.

```
let branch version =
  backtrack version;
  branch_current version

let access version =
  backtrack version;
  version.global
```

## 5 Related Work

**StoreTransactionalRef** Within the library ecosystem we are familiar with, our work is fairly similar to François Pottier's StoreTransactionalRef impleementation. We provide more features (nested transactions) with no additional costs. We haven't discussed this with François Pottier yet, but we would expect our proposal to replace his StoreTransactionalRef module completely.

**In automated solvers** Automated solvers, for example SMT solvers, rely heavily on efficient backtracking. When we looked in the opam package repository for an implementation of backtrack-able Union-Find or backtrack-able references, we could not find anything; but since we worked on this paper every author of a solver we met tells us with "l'air penaud" that they have a version of this, deep in the middle of their own code, that they have never shown to anyone. Boo!

A kind anonymous reviewer also pointed us to the CVC5 overview paper: Barbosa, Barrett, Brain, Kremer, Lachnitt, Mann, Mohamed, Mohamed, Niemetz, Nötzli, Ozdemir, Preiner, Reynolds, Sheng, Tinelli, and Zohar (2022). In Section 2.4, "context-dependent data structures", they mention a general idea of backtrackable data structures indexed by a *context* with an imperative push/pop interface – as common in SAT/SMT solvers. CVC5 supports various data-structures as first-class members of those contexts, which also correspond to our stores. It is not clear to non-experts as we are whether one should try to continue with just backtrackable stores of references as a simple building block, and define elaborate data structures on top of it (Union-Find, hashtables, etc.), or whether we really get noticeable efficiency benefits by adding more data structures as first-class concepts in the backtrackable store, creating opportunities for specialized implementations or more compact representations.

**Hidden OCaml implementations** After submitting this article we were pointed to, in particular, (unpublished) implementations of backtracking stores of references in the FaCiLe constraint-solving library and in the Colibri2 constraint solver, which more generally implements the context interface of CVC5 for various data structures. Note that SMT implementations typically support only backtracking, that is an `abort` operation, but not the `commit` operation of François Pottier's transactional references. FaCiLe does support a `commit` operation which is described as a `cut` from logic programming.

**Specialized implementations** Finally, in the late 80s there was apparently a lot of work on specialized implementations of Union-Find with various forms of built-in backtracking support, see for example Apostolico, Italiano, Gambosi, and Talamo (1994). We don't know of released implementation of these algorithms. They may be marginally faster than building on top of stores of references, but cannot be reused for other data structures.

# 6 Benchmarks

We wrote some benchmarks to confirm experimentally our overall claim: the implementation we present in this work (the latest iteration, with the record-elision optimization) provides references with backtracking support at low cost. They are suitable to build backtracking data structures.

You can find our detailed performance results at `https://gitlab.inria.fr/gscherer/unionfind/-/blob/jfla-benchmarks/bench/README.Store.JFLA.md` A subset of these results are mentioned in Appendix A.

In our performance tests, the Facile implementation is around 15% slower than ours, while Colibri2 is around 60% slower than our code. (These overheads are small and become neglegible for most workflows that are not dominated by reference performance; for comparison, using a persistent Map is 13x slower than our implementation, and our attempt to use CCHashTrie with transient updates fared even worse.)

We believe that the Colibri2 slowdown comes from an "on-demand" design where the backtracking work is done the next time a reference is accessed, instead of backtracking all references eagerly at abort/rollback time. This on-demand logic may save effort in some workloads, but it adds an extra `rewind` function call in the hot paths of `get` and `set`. In the common case the `rewind` function returns immediately because no backtracking is going on, or because it has already been performed, but a function call that returns immediately is enough in a very fast path for a noticeable performance difference.

# References

Alberto Apostolico, Giuseppe F. Italiano, Giorgio Gambosi, and Maurizio Talamo. The set union problem with unlimited backtracking. *SIAM Journal on Computing*, 23(1):50–70, 1994.

Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.

Sylvain Conchon and Jean-Christophe Filliâtre. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*, pages 37–45, Freiburg, Germany, October 2007. ACM Press.

Sylvain Conchon and Jean-Christophe Filliâtre. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*, April 2008.

# A Boring Benchmarks

We benchmarked several implementations of stores of references, some backtracking, some supporting only non-nested transactions, some supporting no transaction at all:

**Ref** is just using raw references. No transactions or backtracking. transactional or backtracking worfklows.

**Map** is François Pottier's StoreMap implementation. It supports constant-time backtracking, but the logarithmic overhead on each access is prohibitive.

**Vector** is François Pottier's implementation storing all references in a dynamic array. The array can be copied, which gives us a way to implement backtracking.

**BacktrackingRefv1** was our first implementation of backtracking reference store in Section 3.1. It contains a complexity bug for the `commit` operation.

**BacktrackingRefv2** was our second implementation of backtracking reference store in Section 3.2, fixing the complexity issue for `commit`.

**BacktrackingRef** is our final, third implementation of backtracking reference store in Section 3.3. It contains a record-elision optimization that elides redundant writes to the journal at the same version, improving space usage and also time constant factors (only one polymorphic write in the common case).

**TransactionalRef** is François Pottier's implementation of "transactional" references. It contains its own sort of record-eliding optimization, and should behave similarly well in write-heavy workflows.

**Facile** is an implementation of our interface on top of FaCiLe's backtracking references: `https://github.com/Emmanuel-PLF/facile/blob/master/lib/fcl_stak.mli` FaCiLE also implements a record-elision optimization.

**Colibri2** is an implementation of our interface on top of Colibri2's references in backtracking contexts. `https://git.frama-c.com/pub/colibrics/-/blob/bobot/abs/colibri2/stdlib/context.mli` Colibri2 also implements a record-elision optimization.

(For full scripts and even more results, see `https://gitlab.inria.fr/gscherer/unionfind/-/blob/jfla-benchmarks/bench/README.Store.JFLA.md`)

For most applications of references, reads dominate writes. The benchmarks that follow perform $2^{15}$ writes and $2^{20}$ reads (so 32x more reads) to an array of $2^{10}$ references. (This workload is then iterated a certain number of times.)

## A.1 Raw benchmarks

These benchmarks only test read and write, no transactional or backtracking workflows. (No transaction is ever started or no backtracking snapshot taken.)

| impl. | time | memory |
|---|---|---|
| Ref | 1.37s | 2.4Mio |
| TransactionalRef | 1.64s | 2.4Mio |
| BacktrackingRefv1 | 1.55s | 2.4Mio |
| BacktrackingRefv2 | 1.52s | 2.5Mio |
| BacktrackingRef | 1.53s | 2.4Mio |
| Facile | 1.74s | 2.4Mio |
| Colibri2 | 2.45s | 2.4Mio |
| Vector | 1.62s | 2.4Mio |
| Map | 20.00s | 4.5Mio |

We see that there is a very small performance difference between simple references and the transactional or backtracking version – mostly due to code size / inlining factors, simple references have shorter code but they do the same thing in the hot path.

Facile and Colibri2 are slower than our implementations. Facile is 15% slower, Colibri2 is 60% slower. (See our Related Work section 5 for our understanding of the performance disadvantage of Colibri2.)

Persistent maps have much worse performance than all other due to their logarithmic access overhead.

## A.2 Transactional benchmarks

This benchmark repeatedly performs the raw alloc-read-write workflow under one transaction, which is either committed or aborted.

| impl. | time | memory |
|---|---|---|
| TransactionalRef | 1.14s | 2.4Mio |
| BacktrackingRefv1 | 1.42s | 12.2Mio |
| BacktrackingRefv2 | 1.52s | 12.3Mio |
| BacktrackingRef | 1.13s | 2.5Mio |
| Facile | 1.20s | 2.5Mio |
| Colibri2 | 1.63s | 2.5Mio |
| Vector | 1.66s | 4.1Mio |
| Map | 12.48s | 4.5Mio |

TransactionalRef, BacktrackingRef, Facile and Colibri2 perform well under this benchmark, with similar relative differences between them.

On the other hand, previous versions v1 and v2 of BacktrackingRef pay a 2x time overhead due to running under a transaction. Their memory usage is also noticeably higher, due to the absence of record-elision logic: they add each write to the log. (All further benchmarks show the same memory advantage to our final version of BacktrackingRef, corresponding to the better space-complexity claims.)

# B Backtracking benchmarks

| impl. | time | memory |
|---|---|---|
| BacktrackingRefv1 | 18.96s | 458.0Mio |
| BacktrackingRefv2 | 1.63s | 231.1Mio |
| BacktrackingRef | 0.89s | 2.5Mio |
| Facile | 1.25s | 2.5Mio |
| Colibri2 | 1.77s | 2.6Mio |
| Vector | 1.87s | 4.1Mio |
| Map | 17.12s | 4.6Mio |

We can observe the complexity bug in `commit` for BacktrackingRefv1.

In our previous backtracking benchmark, the nesting depth is 100, with $2^{20}$ reads and $2^{15}$ writes at each level. We could observe better results with Map by having a longer nesting depth, and fewer reads and writes.

If we use a nesting depth of 30000, and only $2^5$ reads and $2^5$ writes at each level, we observe the following results:

| impl. | time | memory |
|---|---|---|
| BacktrackingRefv1 | 0.51s | 160.0Mio |
| BacktrackingRefv2 | 0.50s | 161.5Mio |
| BacktrackingRef | 0.09s | 32.2Mio |
| Facile | 0.10s | 40.9Mio |
| Colibri2 | 0.13s | 33.4Mio |
| Vector | 0.78s | 542.6Mio |
| Map | 0.80s | 42.0Mio |

Map looks much better, but the record-eliding versions (BacktrackingRef, Facile, Colibri2) perform even better. Vector consumes a lot of memory due to frequent copies; copies corresponding to previous versions of the structure cannot be reclaimd by garbage collection, as we may still restore them on backtracking.

We are not sure whether this high-nesting-low-work scenario is relevant to any real-world use case. Automated theorem provers have very deep nestings of backtracking; but even SMT solvers tend to do a lot of propagation work in-between two backtracking points.