Snapshottable stores

CLÉMENT ALLAIN, INRIA, France
BASILE CLÉMENT, OCamlPro, France
ALEXANDRE MOINE, INRIA, France
GABRIEL SCHERER, INRIA, France

We say that an imperative data structure is *snapshottable* or *supports snapshots* if we can efficiently capture its current state, and restore a previously captured state to become the current state again. This is useful, for example, to implement backtracking search processes that update the data structure during search.

Inspired by Baker [1978], we present a *snapshottable store*, a bag of mutable references that supports snapshots. Instead of capturing and restoring an array, we can capture an arbitrary set of references (of any type) and restore all of them at once. This snapshottable store can be used as a building block to support snapshots for arbitrary data structures, by simply replacing all mutable references in the data structure by our store references. We present use-cases of a snapshottable store when implementing type-checkers and automated theorem provers.

Our implementation is designed to provide a very low overhead over normal references, in the common case where the capture/restore operations are infrequent. Read and write in store references are essentially as fast as in plain references in most situations, thanks to a key optimization we call *record elision*. In comparison, the common approach of replacing references by integer indices into a persistent map incurs a logarithmic overhead on reads and writes, and sophisticated algorithms typically impose much larger constant factors.

The implementation, which is an extension of Baker [1978], is both fairly short and very hard to understand: it relies on shared mutable state in subtle ways. We provide a mechanized proof of correctness of its core using the Iris framework for the Coq proof assistant.

1 INTRODUCTION

1.1 Snapshots as a library

Consider an implementation of the Union-Find data structure offering the following interface:

```
type 'a node
val node : 'a -> 'a node
val find : 'a node -> 'a node
val union : ('a -> 'a -> 'a) -> 'a node -> 'a node -> unit
val equal : 'a node -> 'a node -> bool
val get : 'a node -> 'a
```

A Union-Find graph lets the user incrementally specify an equivalence relation between its nodes, and efficiently query information about the equivalence classes. In our API, each equivalence class carries a value at some type `'a`. The user can grow the equivalence relation by unifying two nodes (`union`), providing a merge function for the carried values. Unification is a destructive operation; it modifies the nodes in-place. We can ask for a representant in each equivalence class (`find`), check if two nodes belong to the same class (`equal`), and ask for the value carried by the class (`get`).

A typical implementation would use a data structure such as follows:

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

Authors' addresses: Clément Allain, INRIA, France; Basile Clément, OCamlPro, France; Alexandre Moine, INRIA, France; Gabriel Scherer, INRIA, France.

A node is just a mutable reference to some data, which indicates whether it currently is the representative of its equivalence class, or points to another node closer to the representative. The `rank` integer is used to decide who to elect as the new representative when merging two nodes.

Union-Find is a central data structure in several algorithms. For example, it is at the core of ML type inference, which proceeds by repeated unification between type variables. Union-Find can also be used to track equalities between type constructors, as introduced in the typing environment when type-checking Guarded Algebraic Data Types (GADTs) for example.

When using a Union-Find data structure to implement a type system, it is common to need backtracking, which requires the inference state to be snapshottable. For example:

(1) A single unification between two types during ML type inference translates into several unifications between type variables, traversing the structure of the two types. If we discover that the two types are in fact incompatible, we fail with a type error. However, we may want to revert the unifications that were already performed, so that the error message shown to the user does not include confusing signs of being halfway through the unification, or so that the interactive toplevel session can continue in a clean environment.
(2) Production languages unfortunately have to consider backtracking to implement certain less principled typing rules: try A, and if it fails revert to a clean state and try B instead.
(3) GADT equations are only added to the typing environment in the context of a given match clause, and must then be rolled back before checking the other clauses.

We have encountered requirements (1) and (2) in the implementation of the OCaml type-checker, and (1) and (3) in the development of Inferno [Pottier 2014], a prototype type-inference library implemented in OCaml that aims to be efficient.

Now a question for the reader: how would you change the Union-Find implementation above to support snapshots? The API needs to change a bit to let users talk about the whole Union-Find graph – otherwise, they cannot even ask to go back to a previous version of the graph. The following would be suitable, while still retaining the imperative flavor of the existing API:

```
type graph
type 'a node

val node : graph -> 'a -> 'a node
val get : graph -> 'a node -> 'a
val union : graph -> ('a -> 'a -> 'a) -> 'a node -> 'a node -> unit
val equal : graph -> 'a node -> 'a node -> bool

type snapshot
val capture : graph -> snapshot
val restore : graph -> snapshot -> unit
```

A first idea to approach our question is to browse the scientific literature for implementations of Union-Find with backtracking, for example looking at Apostolico, Italiano, Gambosi and Talamo [1994]. You would learn that there are algorithms in $O(\log n / \log \log n)$ amortized running times, and then deal with the rewarding but sizeable work of turning a dense 40 pages algorithmic paper from the 90s into runnable code. (This works because Union-Find is a well-studied problem, you would be less lucky with the same question on another, less common mutable data structure.) Unfortunately, we are too lazy to do this. We would like a *generic* approach to add snapshots to an imperative data structure, that does not require expert-level data structure knowledge.

There are two standard generic solutions that can be implemented with relatively little effort.

- **Full copy** : take a snapshot by doing a full copy of the Union-Find graph.
  This approach performs well in the case where snapshots are rare – in the extreme case where no snapshots are taken, there is zero overhead. But it can become a performance disaster when snapshots become more frequent, and the number of nodes modified between two snapshots is small – you copy all the nodes, but only touch a few of them. In one of our use-cases using Inferno, this approach makes type-inference 50× slower.
- **Full persistence** : implement the graph on top of a pure, persistent data structure. A standard approach is to change the type `'a data ref` to become just an `int` index into a persistent integer map. Implementing capture/restore is then trivial, a snapshot is just the persistent map itself. See for example the Haskell library disjoint-set. However, this adds a logarithmic overhead to each access or modification. In Inferno, we observed that this typically makes type inference about 3× slower, even in cases where no backtracking is used. (Performance is the reason why we stick to an imperative API instead of providing a functional API where modification leaves the input state unchanged and returns an updated state.)

We present a new `Store` library, which provides generic snapshottability while performing well in all situations: snapshots, easy and cheap. Unlike full persistence, it introduces no overhead when backtracking is absent or infrequent. Unlike full copy, it performs well when backtracking sections touch only a small subset of the structure.

Using our library for Union-Find requires changing the datatype definitions as follows:

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

The only change here is to replace the standard `'a ref` type of OCaml mutable references by the type `'a Store.Ref.t` of store references in our Store library, which supports snapshots. In the rest of the code, our Union-Find implementation would need to keep a store in its `graph` value, and pass this store to the `get` and `set` operations on store references. These are trivial changes.

Summary. Our Store library introduces a notion of *store*, a bag of mutable references that lets you capture and restore the state of all its references at once. Store can be used to easily make arbitrary mutable data structures snapshottable, by replacing their mutable pointers by store references.

1.2 Notions of persistence

The standard notion of *persistence* used in the algorithmics literature is one where modification operations return a different version of the data structure, without modifying the version provided in input. There are in fact many nuances to persistence, described below.

- **functional** data structures are fully immutable, as is idiomatic to implement persistent data structures in functional programming languages. (*functional* is the terminology of Demaine, Langerman and Price [2008], one may also call them *pure* data structures.) They typically rely on sharing immutable substructures between different versions, and *copying* the paths from those shared substructures to the root of the structure.
  Functional data structures have the advantage that they are thread-safe by construction: they can be accessed in parallel without any synchronization.
- **persistent** data structures may be implemented using mutable state; a typical example would be the Splay-tree data structure that performs imperative rebalancing under the hood. They may not be thread-safe. In the case of our store, our persistent snapshots are persistent in

this sense, and in particular they are not thread-safe – we cannot support restoring two snapshots in parallel.

partial persistence is a weaker notion of persistence where only the "last" version of the data structure may be updated, but read-only queries may be performed on arbitrary versions of the structure. We could expose this capability for our backtracking stores, but we do not have a clear use-case that would justify the additional implementation complexity.

confluent persistence is a stronger notion of persistence where two independent instances of a persistent data structure may be merged together – for example, merging two persistent sets or maps together. Some persistent data structures cannot offer confluence at a reasonable cost. We have not implemented confluence for our stores; the user has to plan in advance and allocate the separate data structures in the same underlying store.

semi-persistence is a weaker notion of persistence where only a linear chain of versions is maintained at any point in time, rather than a tree of versions in the general case: acting on a past version invalidates all the versions that are "after" this past version, and we cannot access them anymore.
Our store provides persistent snapshots and also exposes a semi-persistent API based on *transactions* that we describe in Section 4. This brings moderate performance benefits for use-cases that do not need full persistence; we observed no improvement on some benchmarks, 5%-10% speedups in others, and larger gains for some very specific workloads.

Use cases for persistence and semi-persistence. A semi-persistent approach suffices whenever we only ever restore ancestors of the current version. This is the case for most backtracking problems. For example, in a SAT/SMT solver, backtracking (when a conflict is found) goes back to a time when fewer decisions were made, it never jumps "forward" into a saved search state where more decisions had been made.

Some search algorithms do not perform a full depth-first search, they explore several positions in the tree in parallel, iteratively refining the more promising positions, and they may "fork" new search branches from the same promising position several times. Those require persistent snapshots. Another trite example is *saves* in video games, where players can load previous saves to move forward in game time, or go back to parallel/divergent play histories.

The original persistence use-case of Baker [1978] was the implementation of efficient dynamic binding in a Lisp interpreter. Efficient Lisp interpreters at the time would have a semi-persistent store for the dynamic environment, with a stack structure mirroring the dynamic call stack of the program – on function return they would "undo" bindings performed within the body of the function, to return to the dynamic binding environment of the caller. But this approach does not work when returning functions as first-class values, as the body of the functions (when called later) should be evaluated in the dynamic environment where it was defined, whose definitions have been undone in the meantime. Instead, Baker implemented a persistent store for its environments; first-class functions would capture a snapshot at their definition site, to be restored at call-time.

1.3 Performance model

Following Baker [1978], we implement Store as a "journaled" data structure; the current version of the store is represented in memory just like normal references, but we also keep a record of past operations to be able to go back to previous versions. If the log of operations between two snapshots A and B has size Δ, then the space cost of the log is $O(\Delta)$, and restoring the state of A when we are currently at B takes times $O(\Delta)$.

One may expect the number Δ of operations recorded to be exactly the number of operations performed between the two snapshots. For Union-Find problems the number of reference updates remains relatively small, but in general this number of operations may be large, much larger than the size of the data structure itself. We introduce a key optimization, *record elision*, where we record at most one operation per store location updated between two consecutive snapshots. As a result, our bound Δ is the number of distinct locations modified between the two snapshots, which could be much smaller than the total number of operations. Record elision does not just improve asymptotics, it is key to low-overhead implementation of `set` for store references.

We can benefit from record elision because our interface requires users to be explicit about where they take snapshots, that is, where the backtracking points are in their programs. Record elision is not available to the more elegant, more convenient and more functional interface of a persistent store, which corresponds to taking a snapshot after each update operation.

In the specific case where each snapshot is restored at most once – this is a common property of backtracking workloads, and enforced by our *semi-persistent* interface – one can amortize the cost of snapshot restoration over each operation after the snapshot is taken, so restoring a snapshot has $O(1)$ amortized complexity. This amortization does not work in the general case of persistent snapshots; for example, one could keep alternating between two snapshots without performing any operation in between. This bad interaction between persistence and amortized bounds is a well-known problem in the algorithms literature, typically solved by sophisticated *rebuilding* techniques [Chuang 1992, 1994]. We do not solve it, as our current use-cases do not need it.

When discussing our design choices, we mention constant factors a lot. Imagine that you are implementing a type checker (with type inference) for your programming language, and suddenly you realize that an oddball new feature F that you want requires backtracking inference decisions, which you did not need previously. You have to move your type-checker state to different data structures that support snapshots. You need this new capability only for programs that use feature F, but you pay the cost of the data structure all the time.[1] If you are not careful about constant factors, this implementation change could make your type-checker 2×, 5× or $\log(n)\times$ slower for *all* programs, whether they use your new feature or not. This is not acceptable.

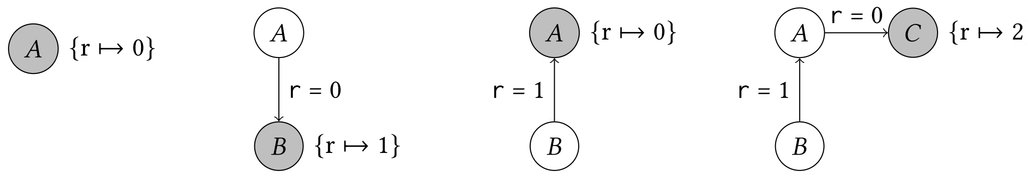
Contributions

We report on the implementation of *snapshottable stores*, a bag of mutable references that support efficiently capturing and restoring its state to implement backtracking. This abstraction can be used to easily add snapshots to complex imperative data structures. The implementation (1) is expressive, it provides persistent and not just semi-persistent snapshots, (2) is efficient, as demonstrated by benchmarks, and (3) its core mechanism is formally proved correct.

We claim the following contributions:

(1) The concept of “snapshottability” as a service worth providing in a reusable, generic way as a small software library. When we looked at existing library ecosystems (in OCaml but also Haskell, Scala, etc.) we found a few implementations of snapshottable stores in the wild, but almost always as part of a larger program that uses it exclusively, not as a shared library.
(2) An efficient OCaml implementation of a store with persistent snapshots. The implementation, extending the journaled approach of Baker [1978], is short and subtle. It is *heterogeneous*, references of different types can be tracked by the same store.
(3) A mechanized proof of correctness of persistent snapshots, using the Iris separation logic framework in the Coq proof assistant.
(4) The *record elision* optimization which is key to an almost-zero overhead on the `set` operation on `set`-heavy workloads. Forms of record elision exist in previous semi-persistent

[1] You could think of dynamically switching from one data structure to another when feature F occurs. This increases implementation complexity, and you still have the problem of not-too-slow type inference for programs that do use F.



(a) (b) (c) (d)

Fig. 1. Version trees in the example program

implementations, but combining persistent snapshots and record elision is challenging and Store is the first implementation to do so.

(5) An additional API of *semi*-persistent snapshots, which restricts ourselves to a linear history of snapshots for further efficiency benefits.
(6) Benchmarks comparing the performance of our implementation with other approaches, demonstrating that `Store` performs well on a broad variety of workloads.

2 A CORE STORE

2.1 Baker's version trees

The starting point of our implementation is Baker's *version trees* introduced in Baker [1978]. Baker's trick has been reused or rediscovered many times since, mostly in the context of implementing persistent *arrays*: homogeneous structures indexed by small integers. O'Neill and Burton [1997] give a pleasant survey of persistent arrays approaches and lists three works that reinvented Baker's trick in the late 80s.

In Baker's work, the programmer can refer to many different persistent versions of a data structure, but one is the "current version" on which access and update operations operate as usual in constant time. The "current version" uses its standard representation – for example, the current version of a Baker array is just an array. Older versions are represented by nodes in a version graph (in fact a rooted tree), whose root is the current version, and where edges log operations that were performed. Any older version can be restored by applying a "rerooting" operation on its node (it becomes the new root of the graph) which reverts all the updates that happened between that older version and the current version.

Consider the following `Store` user program:

```
let s = Store.create () in
let r = Store.Ref.make 0 in
let snap0 = Store.snapshot s in
let () = Store.Ref.set r 1 in
let () = Store.restore s snap0 in
let () = Store.Ref.set r 2
```

At the point of **`let`** `r = Store.Ref.make 0`, our version tree (shown in Figure 1a) has a single node where the reference `r` has value `0`. The mapping $\{r \mapsto 0\}$ is not stored within the node A, it describes the current state of the reference r in the current state. We place it on A to indicate that A is the current root of the version tree, which is also indicated by the darker background.

Calling `Store.Ref.set r 1` will create a second node B in the version tree, which describes the new current state (see Figure 1b). The node A now points to B, with information on how to revert to A if desired – one should restore `r` to `0`.

Calling `Store.restore s snap0` will *reroot* the version tree to have root A again – A was the current node at the time where `snap0` was captured (see Figure 1c). We do this by starting from the snapshot node A, updating the current state by using the information stored on the edges. Note that the edge between A and B has changed directions (now B points to the new current root A), and the information on the edge now describes how to restore the state of B from the state of A.

At this point, calling `Store.set s r 2` creates a new node C from A, which becomes the new current root, as shown in Figure 1d.

This representation provides constant-time access to the current state of the store, with the exact same constant factors as OCaml native references – `r` can in fact just be a native reference.

A snapshot is just a node in the version tree. Restoring the snapshot means rerooting the tree so that the snapshot node becomes the new current root – and the current state gets updated accordingly. We sketch our implementation in Section 2.3. It is obviously linear in the length of the path from the snapshot node to the current root node. The length of this path is the number of operations that happened “after” the snapshot node, in a sense that will be made precise in the next section.

2.2 A whiff of graph theory

In graph theory, an (undirected) tree is a certain kind of (undirected) graph: a graph that is acyclic (no cycle in the graph) and connected (all nodes are reachable from each other). In other words, an *undirected tree* is an undirected graph where there exists a unique path between all pairs of nodes.

The notion of "tree" that is common in programming corresponds to the notion of *rooted tree* in graph theory, a tree with a designated root node. The choice of root uniquely determines a *parent relation* that sends nodes to their parent, that is, relates A to B when A has B as parent – there is at most one parent, and the root is the only node with no parents. If we look at a given undirected tree T, and two different choices of root M and N, there is a simple relation between the parent relations of the M-rooted and N-rooted trees: all nodes have the same parent in both trees, except on the (unique) path from M to N where the parent relations are mutual inverses.

Over our version trees, there are two rooted trees (two choices of root) of interest:

(1) The *current tree*, whose root corresponds to the current state of the structure – C at the end of our example above.
(2) The *historic tree*, whose root is the initial node created when the store was first populated – A in our example. (This is a slight simplification, there is a version tree node before `r` was created that we are not showing in the version tree for simplicity.)

We call *history* of a node the path from this node to the historic root. The complexity of rerooting from the current tree A to a given snapshot tree B is exactly the length of the unique path from A to B in the version tree.

2.3 Implementing version trees

We learned of Baker’s trick from Conchon and Filliâtre [2007], which use it to define persistent arrays, on top of which they build a persistent Union-Find, with OCaml code fairly close to what we show in this section. The core of Store, described here, has the following API:

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

module Ref : sig
  type 'a t
  val make : store -> 'a -> 'a t
```

```
  val get : store -> 'a t -> 'a
  val set : store -> 'a t -> 'a -> unit
end

type snapshot
val capture : store -> snapshot
val restore : store -> snapshot -> unit
```

The `Ref` module implements mutable references inside the store. The store must be passed as argument to all operations on references, and it is an *unchecked* programming error to use a reference with a store it does not belong to. The `snapshot` type represents persistent snapshots of the state of the store at a given point in time. New snapshots for the current state are created with `capture`, and the store state can be later reset to the snapshot state using `restore`.

The version tree is a graph of mutable nodes, whose value can be `Mem` to indicate that they are the current root, or `Diff` if they log a reference write.

```
type node = data ref    and data = Mem | Diff : 'a Ref.t * 'a * node -> data
```

If A has B as parent in the current tree, its data must be `Diff(r, v,`B`)`, where `r` is a reference and `v` is the value of `r`, in A.

Finally, the store is just a mutable reference to the current root of the version tree, and a snapshot remembers which node was the current root when it was captured[2].

```
type store = { mutable root : node; }        type snapshot = { root : node; }
```

Easy parts. Creating a new store or taking a snapshot are the obvious things:

```
let create = { root = ref Mem }
let capture store : snapshot = { root = store.root }
```

References have the same representation and `get` operation as standard OCaml references:

```
module Ref = struct
  type 'a t = { mutable value : 'a; }
  let make v = { value = v }
  let get _s r = r.value
  let set s r v = ... (* to be detailed below *)
end
```

The two difficult operations are `Ref.set`, which grows the version tree with a new node, and `restore`, which reroots the version tree to a snapshot node.

Update operation: `Ref.set`. When we call `set s r v`, the current root of the version tree, which was previously a `Mem` node, becomes a `Diff` node pointing to a *new* current root. The `Diff` node carries the *previous* value of the reference, to be able to restore the reference to its previous value later on.

```
  let set s r new_val =
    let old_val = r.value in
    let new_root = ref Mem in
    let old_root = s.root in
    r.value <- new_val;
```

[2]In the actual implementation, we also remember the store, to fail at runtime if the user tries to use a snapshot with another store.

```
  old_root := Diff(r, old_val, new_root);
  s.root <- new_root
```

The code is short, but reasoning about it is difficult. It helps to define a *model* of the store and the nodes in the version tree. A node A models a functional *mapping*, denoted $\llbracket A \rrbracket$, from references to their values, as follows:

(1) The mapping of the `Mem` node maps each store reference to its current value.
(2) The mapping of a `Diff(r, v, n)` node is $\llbracket \mathtt{n} \rrbracket[\mathtt{r} \mapsto \mathtt{v}]$.

In other words, if B is the parent of A in the current tree, then the edge from A to B (stored in A's `data` in the OCaml representation) records how to transform $\llbracket B \rrbracket$ into $\llbracket A \rrbracket$.

If we look at `Ref.set` again, we can now check that, given a current mapping m, `set s r v` will move us to a new current mapping $m[\mathtt{r} \mapsto \mathtt{v}]$ (with `r.value <- new_val`). Furthermore, since `old_val` stores the value $m(\mathtt{r})$, the mapping of the old root (and hence of the existing version tree) is preserved as it becomes $m[\mathtt{r} \mapsto \mathtt{new\_val}][\mathtt{r} \mapsto \mathtt{old\_val}] = m[\mathtt{r} \mapsto m(\mathtt{r})] = m$.

Reroot, restore. The operation `reroot(`A`)` makes an arbitrary node A the new root of the current tree – without changing the model of any snapshot node in the tree. A "simple" implementation of `reroot` follows:

```
let rec reroot n =
  match !n with
  | Mem -> ()
  | Diff (r, v, n') ->
    reroot n';
    let old_v = r.value in
    r.value <- v;
    n := Mem;
    n' := Diff (r, old_v, n)
```

Before the call, `n` points to its parent node `n'`, and $\llbracket \mathtt{n} \rrbracket = \llbracket \mathtt{n'} \rrbracket[\mathtt{r} \mapsto \mathtt{v}]$.

At this point, the current model is $\llbracket \mathtt{n'} \rrbracket$.

The current model becomes $\llbracket \mathtt{n'} \rrbracket[\mathtt{r} \mapsto \mathtt{v}] = \llbracket \mathtt{n} \rrbracket$.

`n` becomes the current root, matching the current model.

`n'` gets assigned model $\llbracket \mathtt{n} \rrbracket[\mathtt{r} \mapsto \mathtt{old\_v}] = \llbracket \mathtt{n'} \rrbracket$ again.

Our actual (verified) implementation contains two improvements over this "simple" version.

(1) In this version, every recursive call in the `Diff(r, v, n')` case sets the data of both the node `n` and of its parent node `n'` – which becomes its child in the modified version tree. This means that the data of most nodes is set twice, first to `Mem` and then to their final data. Our implementation avoids these redundant modifications by setting `Mem` only once at the end, at the cost of a more complex specification for the recursive function, whose precondition is conditioned on a future update.
(2) `reroot` reverts and reverses `Diff` nodes from the root of the version tree to the snapshot node. This corresponds to undoing operations from the most recent operation to the oldest operation, as it should be. The simple version does this via a non-tail-recursive call `reroot n'` on the parent node `n'` before it handles the child `n`. Our implementation uses a tail-recursive variant where we first accumulate `Diff` nodes in a list, most recent operation at the head, and then traverse the list in order.

Finally, `restore` can be easily defined from `reroot`:

```
let restore (store : store) snapshot =
  reroot snapshot.root;
  store.root <- snapshot.root
```

Remark. This concludes the part of our exposition that is mostly a retelling of the core algorithm of Baker [1978], with an OCaml realization inspired by Conchon and Filliâtre [2007]. We consider what follows as original work.

2.4 Record elision

Record elision is a key optimization that changes the qualitative performance profile of the library. The idea is simple: if we have already performed a `set` operation on some reference `r` in "the current version" (since the last snapshot), we have created a `Diff` node with the value before that operation; so if we perform a `set` on that reference again, there is no need to log anything, as the older `Diff` node will already reset the reference to its previous value. This optimization is only valid if no snapshot was taken after the previous `Diff` node, otherwise that snapshot would get the wrong value of `r` on rerooting.

We do not wish to search the history on each `set` to check this property. In fact we cannot check it with the previous definitions, as there is no trace in our graph data structure of which nodes have been captured as snapshots. We solve both issues by introducing a notion of *generation*, an integer that counts the number of snapshots taken in the history of a node. In particular, if two nodes belong to the same history and have the same generation, there is no snapshot between them.

We keep track of generations in the store graph (the generation of the current root), in snapshots (the generation of the snapshot node), in references (the generation of the last `Diff` node on this reference), and `Diff` nodes.

```
type store = { mutable root : node; mutable generation : int; }
type 'a Ref.t = { mutable value : 'a; mutable generation : int; }
type snapshot = { store : store; root : node; generation : int; }
type node = data ref
and data = Mem | Diff : 'a Ref.t * 'a * int * node -> data
```

Creating a new snapshot increments the generation of the store:

```
let capture s =
  let snap = { store = s; root = s.root; generation = s.generation; } in
  s.generation <- s.generation + 1;
  snap
```

All the magic happens in the `Ref.set` function which updates a store reference. (We use a lighter gray color for code that is identical to the previous version.)

```
let set (s : store) (r : 'a Ref.t) (new_val : 'a) : unit =
  if s.generation = r.generation
  then r.value <- new_val
  else
    let old_val = r.value in
    let old_gen = r.generation in
    let new_root = ref Mem in
    let old_root = s.root in
    r.value <- new_val;
    r.generation <- s.generation;
    old_root := Diff(r, old_val, old_gen, new_root);
    s.root <- new_root
```

By comparing the two integers `s.generation` and `r.generation`, we check whether a snapshot was captured between the last recorded write to the reference and the current root. If no snapshot was taken, then we do *not* record the new update in the version tree – it is useless, as any `restore` call will restore an older value of the reference from the recorded write. We call this a *record elision*. If a snapshot was taken, we update the generation of the reference: we have just recorded the write, so we can elide all records for that reference until the next snapshot is taken.

In terms of model, calls to `set r v` where record elision takes place are harder to reason about, because they mutate the mapping of existing nodes in the version tree: for all the nodes from the current root (included) to the last `Diff` node on this reference excluded, their mapping gets mutated from some m to $m[\mathtt{r} \mapsto \mathtt{v}]$. In the absence of record elision, the mapping of all version tree nodes was persistent: the data on the node may change but its mapping remained unchanged. Record elision relaxes this property: the mapping of nodes that are *captured by a snapshot* is persistent, but other nodes, in fact the nodes between the last snapshot and the current root, may see their mapping changed by later operations. This weaker guarantee suffices, as we only provide persistent *snapshots* to users, they cannot observe the mapping change for other nodes.

Performance impact. Record elision has a transformative performance impact on workflows that use `Ref.set` heavily and snapshot capture rarely. (We generally assume that backtracking is rare relative to reads and writes, but many workflows are rather dominated by reads so record elision matters less.) Indeed, a record-elided `Ref.set` is just an integer comparison and a write, which is basically the same as a write: in OCaml, polymorphic writes go through a write barrier, so the cost of the write dominates the generation test. In the regime where most writes are elided, `Ref.set` is essentially as fast as OCaml primitive references, providing the almost-zero overhead we advertised. On the other hand, non-elided sets perform an extra write and two allocations. On a `get/set` microbenchmark with 16 `get` for each `set`, disabling record elision made the test 6× slower.

Record elision also has a transformative effect on the asymptotic complexity of store operations. As we detailed in the introduction (Section 1.3), the key complexity parameter of `Store` is the size Δ of the log between two consecutive snapshots. Without record elision, Δ is exactly the number of write operations that happened since the previous snapshot, which can grow arbitrarily large. Record elision reduces Δ to the number of different memory locations touched since the previous snapshot, which is much more commonly (but not always) bounded.

Notes. If one tries to implement persistent data structures on top of `Store` by capturing a snapshot after each write operation, then record elision never applies. This explains why we are not offering a persistent API for `Store`. It also probably explains why we have not found a description of this simple idea in the existing literature on more-or-less-persistent data structures.

It is tempting to think of generations as unique timestamps for snapshots, and indeed the two concepts overlap in semi-persistent implementations. Scaling record elision to the persistent setting required a more precise definition of generations that need *not* be unique. Preserving uniqueness in the persistence setting would be an instance of the *order maintenance problem*, which has amortized constant-time solutions (Bender, Cole, Demaine, Farach-Colton and Zito [2002]; but think of the constant factors!) and is a common ingredient in persistent data structure design.

2.5 Liveness

An important consideration in our choice of data structure design is *liveness*. In garbage-collected languages, the memory footprint of a data structure is determined by what other portions of memory it references, keeps *alive*. Suppose for example that a user captures a snapshot of the store, and then later drops all references to this snapshot. Can the memory corresponding to this snapshot be collected, or is it kept alive by the global Store data structure?

The version tree structure inherited from Baker [1978] has excellent liveness properties: pointers in the data representation coincide with the parent relation of the current tree, so that referencing the store only keeps the current root alive. In particular, if we do not reference any snapshot, then the whole version tree (except for the root) can be collected. Locally, only the operations that are needed to restore a snapshot that is still referenced are kept alive. This still holds if the user forgets a reference: as long as a snapshot mentioning it is kept alive, the reference will be kept alive (one could use weak pointers and ephemerons [Hayes 1997] to get better liveness properties there, at significant complexity and runtime cost). On the other hand, if the user forgets *both* the reference and all the snapshots mentioning them, then they can be collected. This is a common situation in realistic workloads such as type-checking problems where we want to forget about the inference variables created when typing a given subterm.

Another case where our implementation can "leak" values is when forgetting intermediate snapshots: if there are three consecutive snapshots A, B and C with the same reference r being written both between A and B and between B and C, forgetting B will still keep the value of r in B alive even though we can never restore B again. We could consider an implementation using weak pointers and finalizers to notice this and compress the log, but suspect that the cost in performance and code complexity would not be worth it for most applications. Our semi-persistent interface (see Section 4) provides a `commit` operation that does remove some (but not all) such unneeded records.

Most other implementation choices have worse liveness properties. Semi-persistent implementations based on a centralized journal often cannot forget any snapshot. Implementations based on functional or imperative maps (with copy) can never forget references. Another common implementation choice for persistent structures, the so-called *fat nodes* approach, keeps a list of all past values in the reference itself. This makes it impossible to forget past versions or siblings, but it allows the user to forget references.

We considered liveness properties seriously in our design, and it helped guide some implementation choices. We believe that the liveness properties of our implementation are adequate, and that it does make a positive difference in practice with respect to implementation approaches that keep all store operations alive – in the type-checking use-case, for example.

3 A COQ STORE

In this Section, we use Separation Logic to specify and verify the core of our approach: an implementation of snapshottable stores without record elision and transactions. We first introduce our formal settings (Section 3.1), then present our specifications (Section 3.2) and finally comment some details of the proof (Section 3.3).

3.1 Formal setting and Separation Logic reminder

Formally, we use the Iris Separation Logic framework [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018]. We write our programs in an untyped call-by-value λ-calculus with mutable state, similar to the HeapLang language that comes with Iris.

In the following, we write Φ for an Iris assertion, $\Phi \ast \Phi'$ for a separating conjunction, and $\Phi \mathrel{-\!\!\ast} \Phi'$ for a separating implication. If P is a proposition of the meta logic, we call P *pure* and write $\ulcorner P \urcorner$. Triples take the form $\{\Phi\}\ e\ \{\Psi\}$, where Φ is the precondition, e the verified expression and Ψ the postcondition. A postcondition Ψ is of the form $\lambda v.\, \Phi'$, where the metavariable v, which denotes the resulting value of the expression e, is bound in Φ'.

3.2 Specifications

Figure 2 presents the specification of our Coq store. To describe a store s at the logical level, we use the assertion store $s\,\sigma$ denoting that s is modeled by the (partial) mapping σ from references to

Create
$\{\text{True}\}\ \text{create}\ ()\ \{\lambda s.\ \text{store}\ s\ \emptyset\}$

Ref
$\{\text{store}\ s\ \sigma\}\ \text{ref}\ s\ v\ \{\lambda r.\ \ulcorner r \notin \text{dom}(\sigma)\urcorner * \text{store}\ s\ ([r := v]\sigma)\}$

Get
$$\frac{r \in \text{dom}(\sigma) \qquad \sigma(r) = v}{\{\text{store}\ s\ \sigma\}\ \text{get}\ s\ r\ \{\lambda v'.\ \ulcorner v' = v\urcorner * \text{store}\ s\ \sigma\}}$$

Capture
$\{\text{store}\ s\ \sigma\}\ \text{capture}\ s\ \{\lambda t.\ \text{store}\ s\ \sigma * \text{snapshot}\ s\ t\ \sigma\}$

Set
$$\frac{r \in \text{dom}(\sigma)}{\{\text{store}\ s\ \sigma\}\ \text{set}\ s\ r\ v\ \{\lambda ().\ \text{store}\ s\ ([r := v]\sigma)\}}$$

Restore
$\{\text{store}\ s\ \sigma * \text{snapshot}\ s\ t\ \sigma'\}\ \text{restore}\ s\ t\ \{\lambda ().\ \text{store}\ s\ \sigma'\}$

Fig. 2. Interface of our Coq store

values. We write $\sigma(r)$ the value associated to reference r in σ, $[r := v]\sigma$ the functional update of σ with the mapping $r \mapsto v$, and $\text{dom}(\sigma)$ the domain of σ, or the set of created references. We first present the specifications of the functions create, ref, get and set. We then devote our attention to the functions involving snapshots, namely capture and restore.

Create asserts that create () has trivial precondition and returns a store s with an empty model. Ref asserts that ref $s\ v$ creates a new reference. The precondition consumes an assertion store $s\ \sigma$ and the postcondition produces an assertion store $s\ ([r := v]\sigma)$, where r is the returned reference. The postcondition also asserts that r is fresh. Get asserts that get $s\ r$ returns the value associated to r in the model of s. The precondition consumes an assertion store $s\ \sigma$, and requires that r is in the domain of s and is mapped to the value v. The postcondition asserts that the function returns the value v, and restores the assertion store $s\ \sigma$. Set asserts that set $s\ r\ v$ correctly sets the value associated to r to v in the model of r. The precondition consumes an assertion store $s\ \sigma$ and requires that r is in the domain of σ. The postcondition produces an assertion store $s\ ([r := v]\sigma)$.

We now devote our attention to snapshots. To describe a snapshot t at the logical level, we introduce the assertion snapshot $s\ t\ \sigma$. It asserts that t is a valid snapshot of the store s, whose model was σ when the capture occurred. Crucially, the assertion snapshot $s\ t\ \sigma$ is *persistent* [Vindum and Birkedal 2021]. A persistent assertion is in particular duplicable. In our case, this means that the following entailement holds: snapshot $s\ t\ \sigma \twoheadrightarrow (\text{snapshot}\ s\ t\ \sigma * \text{snapshot}\ s\ t\ \sigma)$.

Capture asserts that capture s creates a new snapshot. The precondition requires that s is a valid store of model σ. The postcondition asserts that the store was preserved and that the function returned a snapshot t such that snapshot $s\ t\ \sigma$ holds. Restore shows that indeed, restore $s\ t$ updates the model of s to the model captured by t. The precondition consumes the assertion store $s\ \sigma$ and snapshot $s\ t\ \sigma'$, and the postcondition produces the updated assertion store $s\ \sigma'$. Notice that there is no need to repeat the assertion snapshot $s\ t\ \sigma'$ in the postcondition. Thanks to persistence, the user can duplicate the assertion before applying Restore.

3.3 Summary of the proof

To give intuition on our proofs, Figure 3 presents the definitions of the assertions occurring in our specifications. We comment them below.

The definition of the store $s\ \sigma$ assertion is central. It existentially quantifies over the root r of the current tree, a map σ_0 from location to values that we call the *global store*, representing the current values of all references ever allocated with this particular store s, the labeled graph g, a set of edges from node to node labeled with the pair of a reference and its old value, and the map M of models, associating to each node its model. M gives a formalism to the notation $[\![n]\!]$ used in Section 2.3, by posing $[\![n]\!] \triangleq M(n)$. The pure conjunction coming after describes the effective link between all these ghost variables. It asserts that the model of the current root node is effiectievly σ, and that σ

$$
\begin{array}{rcl}
\mathsf{store}\, s\, \sigma & \triangleq & \exists\, r\, \sigma_0\, g\, M.\ \ulcorner M(r) = \sigma_0 \wedge \sigma \subseteq \sigma_0 \wedge \mathsf{coherent}\, \sigma_0\, g\, M \wedge \mathsf{rooted\_dag}\, g\, r \urcorner \ast \\
& & (\ \mathop{\Large\ast}_{(n,(\ell,v),n') \in g}\ n \mapsto \mathsf{Diff}\, \ell\, v\, n'\ ) \ast (\ \mathop{\Large\ast}_{(\ell,v) \in \sigma_0}\ \ell \mapsto v\ ) \\
& & s \mapsto r \ast r \mapsto \mathsf{Root} \ast \mathsf{snapshots\_model}\, s\, M \\
\mathsf{snapshots\_model}\, s\, M & \triangleq & \exists \gamma\, C.\ \ulcorner \forall n\, \sigma.\ (n, \sigma) \in C \implies \exists \sigma'.\ M(n) = \sigma' \wedge \sigma \subseteq \sigma' \urcorner \ast \\
& & \mathsf{meta}\, s\, \gamma \ast \boxed{\bullet C}^{\gamma} \\
\mathsf{snapshot}\, s\, t\, \sigma & \triangleq & \exists \gamma\, n.\ \mathsf{meta}\, s\, \gamma \ast t \mapsto (s, n) \ast \boxed{\circ\{(n, \sigma)\}}^{\gamma}
\end{array}
$$

Fig. 3. Definition of our predicates

is included in the larger, global store σ_0. The proposition coherent $\sigma_0\, g\, M$ asserts the coherence of the information (we omit the formal definition): the nodes occurring in labels of edges of g are indeed in the domain of σ_0, and that if there is a path in g between the node n and the node n' labeled with a list of pairs or references and values, then applying this list of changes updates the model of n (as given by M) to the model of n'. The proposition rooted_dag $g\, r$ asserts that g is a directed acyclic graph (DAG), and that each node can reach the root r. Separation Logic strengthens for free this property to the fact that g is a tree. Indeed, the definition next asserts the Separation Logic ownership of the graph of nodes, an iterated conjunction over g. In particular, it asserts that nodes are unaliased: each node in g has a unique successor. In conjunction with the fact that g is a DAG, it guarantees that g is a tree. The definition then asserts the ownership of the global store as an iterated conjunction over σ_0. In the third line, the definition asserts that the store s points to the node r and that r itself represents the Root constructor. The definition finally mentions the assertion snapshots_model $s\, M$, that we describe below.

The definition of the assertion snapshots_model $s\, M$ existentially quantifies over a ghost cell γ that will be used to give meaning to snapshosts, and C, a set of pairs of nodes and models. Each pair of a node and a model describes a snapshot of the node. Notice that this is not a map: a node may have different snapshots, with different models. The pure proposition witnesses that indeed, if a node n and a model σ appear in C, then n has a larger "current" model in M. In the next line, the definition makes use a meta token, an Iris technicality [Iris Development Team 2024] that allows associating persistent information to a location. Here, the assertion meta $s\, \gamma$ permanently attaches the ghost location γ to the physical location s. It then asserts the authoritative ownership of the set C, written $\boxed{\bullet C}^{\gamma}$. When confronted with a fragmentary ownership $\boxed{\circ C'}^{\gamma}$, it allows deducing that $C' \subseteq C$. Formally, the ghost cell γ is equipped with the camera [Jung, Krebbers, Jourdan, Bizjak, Birkedal and Dreyer 2018] $Auth(Set(Location \times Map\, Location\, Value))$.

The last line of Figure 3 shows the definition of the assertion snapshot $s\, t\, \sigma$. It existentially quantifies over the ghost cell γ and a node n, asserts that γ is the unique ghost cell associated to s with the meta $s\, \gamma$ assertion. It then asserts that the snapshot points to a pair containing the store s and the node n, and the fragmentary ownership $\boxed{\circ\{(n, \sigma)\}}^{\gamma}$, witnessing that the pair (n, σ) represents a valid snapshot. Persistence of the assertion snapshot $s\, t\, \sigma$ reduces to the persistence of $\boxed{\circ\{(n, \sigma)\}}^{\gamma}$, which is guaranteed by the camera being used.

4 SEMI-PERSISTENCE THROUGH TRANSACTIONS

4.1 Introduction

The `capture` and `restore` API presented in Section 2.3 is low-level in the sense that users have to create persistent snapshots, keep track of them, and restore them manually. For some common workloads, we provide high-level wrappers that are more convenient but also less expressive.

```
val temporarily : store -> (unit -> 'a) -> 'a
```

```
val tentatively : store -> (unit -> 'a) -> 'a
```

These wrappers call the provided function, then restore the state of the Store to the state it had prior to the call either unconditionally (`temporarily`) or if an exception is raised (`tentatively`).

Both functions can be implemented by capturing a snapshot before calling `f`, and restoring it after the call if necessary. Snapshots created by these wrappers have interesting properties: not only are they restored at most once, their use follows a rigid structure dictated by scoping rules. This corresponds exactly to the notion of *semi-persistence* in the data-structure literature: there is a *stack* of versions, and versions that are removed from the stack are no longer accessible. Imposing such a linear (or affine) discipline on snapshots makes reasoning about the implementation easier, and avoids the aliasing of mutable state that makes the implementation of `restore` so subtle (Section 2.3).

One could provide an entirely different implementation of Store that only provides a semi-persistent API. It can be expected to be slightly faster, perhaps simpler to implement, but would provide less functionality than the persistent API of Store. Instead, we describe in this section an extension of the Store API with semi-persistence in the *same* implementation, providing a combination of both capabilities. We call this API *transactional*, because each semi-persistent snapshot (or transaction) is terminated by either keeping (commit) or discarding (rollback) the changes within. Users are expected to stick to the simple persistent API and the convenience wrappers `temporarily` and `tentatively`, which are implemented using the semi-persistent API for performance. In more advanced scenarios, users can directly use the transactional API, which is more difficult to use but can bring additional performance improvements.

4.2 Transactions for semi-persistence

Besides the high-level wrappers mentioned earlier, the transactional API is as follows:

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

A `transaction` represents an interval in the program execution during which an ephemeral copy of the store is preserved. The transaction is created by calling `transaction`, and terminated by calling either `rollback` or `commit`. `rollback` is similar to `restore` in the persistent API: it resets the state of the store to the one it had when the transaction started. `commit` terminates `transaction`, but the state of the store is unchanged – it merely discards the ephemeral snapshot.

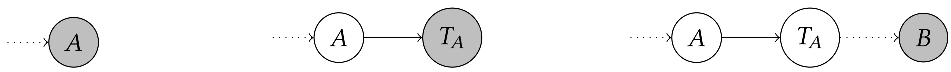
Transactions can be nested following a stack-like discipline: transactions are *valid* when created, and terminating a transaction invalidates it and all the transactions that were valid when it was created. Using an invalid transaction is a programming error and raises an `Invalid_argument` exception.

As a simple example of use of transactions, we can implement the `tentatively` convenience wrapper using the transactional API:

```
let tentatively store f =
  let trans = Store.transaction store in
  match f () with
  | v -> Store.commit store trans; v
  | exception exn -> Store.rollback store trans; raise exn
```

4.3 Combining the persistent and semi-persistent APIs

It is possible to write and reason about programs that combine both APIs.



(a) Initial state (b) After transaction (c) Before rollback

Fig. 4. Version graph during a transaction

Just like new transactions, capturing a persistent snapshot while a transaction is active creates a dependency on that transaction, and the snapshot becomes invalid if a transaction it depends on is terminated or invalidated. In other words, transactions weaken the persistency of snapshots.

Moreover, we allow leaving the scope of a transaction by restoring a snapshot captured before the transaction was created. In that case, the transaction is not invalidated: it becomes *inactive* instead, and can become active again when restoring a snapshot from inside the transaction. More precisely: transactions and snapshots can be *valid* or *invalid*, and transactions can also be *active* or *inactive*. Both *depend on* the transactions that were active and valid at the time of their creation. Terminating a transaction invalidates it and all the transactions and snapshots that depend on it. Restoring a snapshot makes all currently active transactions *inactive*, then makes all the transactions that the snapshot depends on *active* again. Terminating a transaction that is either *inactive* or *invalid* is a programming error.

These rules on the interactions between persistent snapshots and transactions are arguably complex, but provide great flexibility. For instance, they allow calling a function (maybe from a third-party library) that implements its own search sub-procedure using the full `Store` API in any context, without impacting existing snapshots and transactions. They also allow moving to a different context and then coming back, which is relevant for algebraic effects that are performed inside a transaction but whose handler needs to consult another state in the store history.

4.4 Implementing transactions

Transactions are implemented by adding a new kind of information in the graph, *transaction nodes*. Starting a transaction when the current root of the version tree is A (shown in Figure 4a) creates a new transaction node T_A that tracks the transaction (shown in Figure 4b). This does not affect the values of references: node T_A has the same mapping as node A.

When the transaction is terminated, arbitrary nodes may have been added, as shown in Figure 4c. We remove the transaction node T_A from the graph – that is, we mark the node as invalid. We also remove (invalidate) all historic descendants of T_A, so in particular the correction of the version tree is preserved. The initial state is restored: A becomes the current root again (Figure 4a). This is only valid if the current root of the version tree was "inside" the transaction, that is, if it is a node that is a current descendant of T_A. We keep track of that information in the transaction node (it is updated by `reroot`) and fail if the current root is not inside the transaction; otherwise, the transformation would end up with two root nodes in the version tree, the previous root and A.

"Removing" a node is implemented by marking it, or one of its current descendants, as `Invalid`. Which nodes to mark is an implementation detail, as long as `restore`, `commit`, and `rollback` encounter an invalid node and fail before modifying the current state. Our current implementation marks each transaction node – T_A and any child transaction – as well as the current root B.

Calling `restore` on a persistent snapshot must update the current state to apply the `Diff` nodes along the path, but also revert the edges of those `Diff` nodes and update their data to allow restoring in the other direction later. For transactions, `rollback` only updates the current state without touching the `Diff` nodes, leading to a small but measurable efficiency gain.

5 TESTING AND BENCHMARKS

5.1 Testing `Store` with `Monolith`

We used François Pottier's Monolith library to test our implementation of `Store`. Monolith [Pottier 2021] is an OCaml testing framework that implements a specific form of state-based property-based testing called *model*-based testing. It takes a description of the API to be tested, a reference implementation (*model*) of the API, generates random sequences of API calls and checks that the real implementation matches the model.

To test Store, we wrote a reference implementation, designed to be as simple and clear as possible without any efficiency requirement; one could consider it an executable specification. The property we ask Monolith to check is that the real and reference implementations agree. The reference implementation represents functional mappings as a persistent map from unique integer indices (representing references). This is a homogeneous representation (all references must have the same value type) for simplicity: we only use integer values in tests. Each snapshot carries such a functional mapping, as well as a list of transactions that it depends on (as described in Section 4.3). A transaction is a snapshot, with a mutable boolean flag indicating whether it is still valid. Finally, a store is represented by a mutable reference to a snapshot; the active transactions are the transactions that the current snapshot depends on. The data definitions of our reference implementation is available in Appendix A.

We mention our testing approach explicitly because we have found it *unreasonably effective*. The fuzzer we get from Monolith behaves, in our experience, exactly like a correctness oracle. After any code change, you run the fuzzing test, and either it finds a bug in a few seconds or the code is correct. If it finds a bug, it starts looking for a smaller test sequence that also fails, and waiting for about 10 seconds will consistently produce a small, readable sequence of operations that can be replayed to understand what is going on.

Writing complex code with a correctness oracle at hand is a liberating experience. Wondering about why a particular line of code is necessary? Remove it, run the testsuite, and you see. Thinking of reordering two state changes and wondering if there is an interaction between them? Just try it.

We believe that model-based testing is unreasonably useful for `Store` because (1) we have a relatively small and simple API, so all interesting interactions are covered by random search and (2) we gave a lot of thought to expressing clear specifications, which in turn make it easy to write a precise reference implementation.

5.2 Microbenchmarks

We studied the performance of our `Store` library on synthetic microbenchmarks that let us simulate a variety of different usage scenarios. These benchmarks perform almost only reference operations, so they magnify the performance differences between implementations compared to real-world programs – where most of the time is typically spent elsewhere. We would typically consider overheads of up to 30% as small – unlikely to be noticeable in real-world programs, 2×-5× as moderate, and above 10× as large.

Our main goal is to establish that if users need *some* form of backtracking in a (possibly small) part of their program, using `Store` is always a good choice, they will not suffer a noticeable performance degradation compared to a library that supports fewer features, in particular compared to third-party libraries specialized for semi-persistence, and compared to built-in OCaml references when no backtracking at all is used. Before our work on `Store`, when François Pottier needed a Union-Find implementation with (non-nested) backtracking, he implemented the union-find library as a functor over a store-like interface, so that users that do not need backtracking do not pay a cost – they instantiate the functor with built-in references. We want to encourage users to drop this parametrization strategy and use `Store` unconditionally, by showing that `Store` has best-in-class performance for all relevant workloads.

Implementations. We compare the following implementations:

- **Store** Our implementation.
- **Ref** Native OCaml references; they do not support backtracking of any kind, and they are the gold standard for "raw" `get`/`set` operations.
- **TransactionalRef** A "journaled" store by François Pottier, implemented in union-find for the needs of Inferno, that only supports non-nested (semi-persistent) transactions.
- **BacktrackingRef** An earlier "journaled" implementation of Store that we wrote, that only supports semi-persistence. A single dynamic array (the "log") stores all antioperations, and ephemeral snapshots are denoted by positions inside this array. `BacktrackingRef` performs a record elision optimization.
- **Facile** The backtrackable (semi-persistent) references of the Facile library, a well-established constraint-programming framework for OCaml, written with performance in mind.[3] `Facile` uses a "journaled" implementation with record elision, similar to ours. (Record elision is easier to implement for semi-persistent implementations, so it is more common there.)
- **Map** An implementation using persistent maps (the `Map` module of the OCaml standard library): $O(\log n)$ `get`/`set`, but $O(1)$ `capture`/`restore`. This corresponds to the "full persistence" approach we mentioned in the introduction. We expect it to be quite slow due to the logarithmic factor.
- **Vector** an implementation using dynamic arrays, provided by the union-find library, where backtracking operations copy the array. This corresponds to the "full copy" approach we mentioned in the introduction. It has fast get/set operations ($O(1)$), but very slow capture/restore operations ($O(n)$ in the number of references).
  We expect `Vector` to be a solid baseline for the use-cases we had in mind when implementing `Store` – infrequent backtracking operations so `get`/`set` dominate performance.

Benchmarks. We consider the following synthetic benchmarks.

- **Raw** creates 1024 references, then performs a series of 32 reads and 4 writes per reference in a loop repeated 1000 times.
- **Transactional** is the same as **Raw**, except that each iteration of the loop is performed in a failed transaction. We iterate 600 times.
  We also run the following variants, to simulate a variety of workloads:
  - **get** 128 reads per reference, no writes, 200 iterations
  - **set few** no reads, only 64 references are written to (once) in total, that is only $\frac{1}{16}$ of all references, 40000 iterations
  - **set 1** no reads, each reference is written exactly once, 6400 iterations
  - **set 16** no reads, each reference is written 16 times, 600 iterations
- **Capture-heavy** is the same as **Transactional**, but with different parameters to test the case where backtracking operations are much more frequent, with only a few reference accesses per transaction. We perform 16 writes and 64 reads per transaction in total, spread over 4 references in the "small" version (all references are touched in a single transaction) and 1024 references in the "large" version (most references are untouched in each transaction).

[3] `Facile` was written in 2005, and found to be comparable with state-of-the-art constraint solvers of the time: slower than Ilog Solver 4.3, faster than ECLiPSe 5.2.

Backtracking is the same as **Raw**, except that each iteration of the loop starts a new *nested* transaction level. All transactions are failed (rolled back) once the loop completes. The loop is repeated 1000 times, which is also the nesting depth.

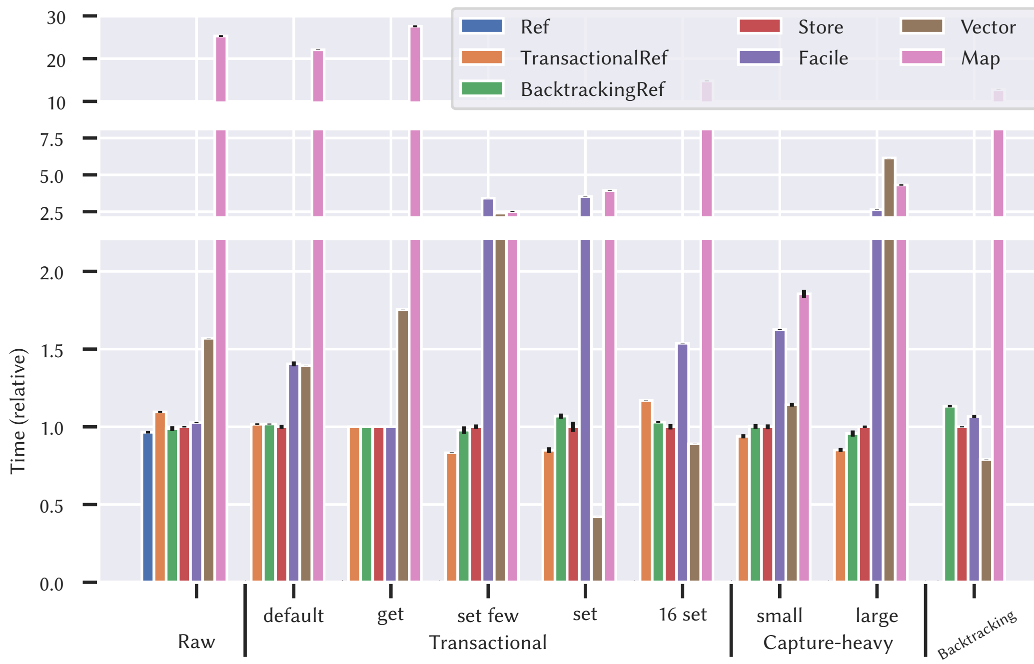


Fig. 5. Micro-benchmark results

Results summary. The results of the microbenchmarks are summarized in Figure 5. The results are normalized relative to the Store implementation to show relative performance in the different tasks. The absolute benchmarks results are available in the appendices.

For reasons of space, we only provide a high-level summary of the results here. Detailed analyses of each benchmark are included in Appendix B.

Our general conclusion is that `TransactionalRef`, `BacktrackingRef` and `Store` are the best implementations, they perform very reliably over all benchmarks, with essentially no overhead over built-in references in the `Raw` benchmark. With the exception of the "set 1" variant where `Vector` shines, they are always the *best* implementations. For the benchmarks where they are supported they have very close performance.

`BacktrackingRef` is able to perform as well as `TransactionalRef` despite supporting nested transactions, and `Store` performs as well as those two despite supporting both persistent snapshots and semi-persistent transactions. The performance of `Facile` is slower than expected: its implementation of `rollback` incurs an indirect call for each record. This suggests that our objective for `Store` of always being a good choice – despite supporting more features – is reached. It also shows the advantage of providing snapshottable stores as an independent library that can be optimized once.

Details on `Facile`. The performance of `Facile` is disappointing on `set`-heavy benchmark. This comes from the fact that `Facile` has no explicit `commit` implementation, we simply keep the snapshot around on successful transactions. In our detailed analysis in Appendix B we refined the `Transactional` benchmark into `commit`-only and `abort`-only workloads, and we see that `Facile` is in fact competitive with other semi-persistent implementations on the `abort`-only workloads.

Details on `Vector`. `Vector` performs surprisingly well, despite an extra indirection and bound checking. But it suffers from very bad behaviors on "large support" workloads, where only a few references are modified per transaction. Our "Capture-heavy (large)" test simulates them, and their `Vector` is 6× slower than `Store`. We believe that this situation is the most common in real-world workloads, and have observed even worse behaviors, for example `Vector` is 52× slower on one of our Inferno macro-benchmarks.

The best case for `Vector` is when each reference is modified exactly once per transaction. Indeed, all other implementations need to perform extra work on `set` that corresponds to a sort of per-reference copy-on-write; if we `set` all references after a snapshot, the total copy work should be at least as much as copying the dynarray on `capture`, with worse constant factors. We do observe excellent performance for `Vector` in the "set 1" variant of `Transactional`, which simulates this. But we do not know of programs in the wild with similar workloads.

If there are fewer references set per transaction, as in our "set few" variant, `Vector` is doing worse than journaled implementations. (Empirically we observed a break-even point on this benchmark when a fourth of the references are set per transaction.) On the other hand, when each reference is modified many times per transaction, as in the "set 16" variant, then journaled implementations benefit from record elision, reducing the advantage of `Vector`.

5.3 Macrobenchmarks

In order to validate the conclusions from microbenchmarks in more realistic scenarios, we adapted existing programs, that perform some sort of backtracking, to use the Store interface. This gives a more realistic view of performance differences one can expect in practice. We detail the various macro-benchmarks in Appendix C, with only a brief summary here.

	Time	Relative
`Store`	0.21s	1.0x
`Vector`	0.28s	1.3x
`Map`	0.88s	4.2x

(a) Inferno type checking (without GADTs)

	Time	Relative
`Store`	0.02s	1x
`Map`	0.08s	4x
`Vector`	1.3s	70x

(b) Inferno type checking (GADT example)

	Time	Relative
`T-Ref`	0.03s	1x
`Store`	0.03s	1x
`Map`	0.09s	3x
`Vector`	1.78s	52x

(c) Inferno type inference (short transactios)

Implementation	Time	Relative
base (hand-optimized)	1.35s	1.00
`Store`	1.63s	1.20
`Store (persistent)`	1.76s	1.30
`Vector`	4.03s	2.99

(d) Sudoku solver

Fig. 6. Macro benchmarks

Inferno re-checks the explicitly-typed programs elaborated by its type-inference engine. Our original use-case for `Store` was the introduction of GADTs, which requires backtracking changes to a Union-Find of type equations.

Figure 6a measures type-checking a large explicitly-typed term that does not actually contain GADTs (the common case). `Store` is noticeably faster than `Vector`, the previous best choice.

Figure 6b measures type-checking a small explicitly-typed GADT example. `Vector` behaves terribly (this is a "large support" situation) and `Store` is much better than other choices.

Figure 6c measures Inferno type inference on a ML program. As mentioned earlier, Inferno uses (non-nested) transactions to roll back partial unifications in case of unification failure, and the `TransactionalRef` implementation of François Pottier was written specifically for this use-case. Our results show that `Store` can replace `TransactionalRef` for this use-case.

Finally, Figure 6d represents results on a backtracking-heavy program, an optimized Sudoku solver implemented in OCaml by Alain Frisch in 2005. The original implementation uses a hand-optimized "full copy" approach, taking a copy of the Sudoku board state on backtracking points. (Our test is on a 25×25 board.) Our results show that replacing the hand-optimized backtracking logic by `Store` only results in a 20% overhead, that using the persistent API instead is slightly slower, and that `Vector` would be much worse, 3× slower than the original implementation.

6 RELATED WORK

6.1 Snapshottable references

We searched the software ecosystem for previous libraries providing "snapshots as a service" (not just the OCaml ecosystem but also Haskell, Scala, Rust), and were surprised not to find any.[4] Some larger systems implement snapshottable references internally for their own purpose, in particular SAT/SMT solvers and constraint solvers; but they did not seem to consider releasing this as its independent library. In our experience, designing `Store` as an independent library led us to consider a variety of workloads more thoroughly, and improved our design and implementation.

Union-Find. The inspiration to think of "snapshottable store" as a library of its own came from the union-find OCaml library, which provides a Union-Find implementation parametrized over a "store", a few simplistic store implementations, and the `StoreTransactionalRef` implementation supporting non-nested snapshots.

Coincidentally, the closest library we found to "snapshots as a service" is the Rust crate ena, which implements a Union-Find data structure *and* provides an `undo_log` module offering a snapshot abstraction. This crate was extracted from the codebase of `rustc`, the Rust compiler, to be shared with other Rust projects with a need for Union-Find. The implementation of `undo_log`[5] provides a semi-persistent interface with a transactional flavor (`commit` and `rollback`), implemented with a global dynamic array of changes to undo. In particular, snapshots are not persistent, with dynamic checks and explicit panics if invalid snapshots are used. It implements the simplest form of record elision, which is to skip any logging when no valid snapshots exist.

`ena` supports arbitrary edit actions with undo callbacks ("custom operations"), but provides built-in support for creating and setting references. Those references are stored in a large dynamic array, with indices passed to the user. In consequence, a given undo log is parametrized over a

[4]The undo-redo Rust crate is the closest we found. It keeps a history of "edit events" on some structure, and can call an "undo" callback associated to each event. It seems designed to record events at the scale of human interactions – human modifications to a document, etc. – rather than fine-grained changes, and would be fairly inefficient for our use-cases. It provides "record", with a linear history (like most semi-persistent implementations) and "histories", which allows a branching history with a git-like model of explicit branches.

[5]https://github.com/rust-lang/ena/blob/12584218/src/undo_log.rs

fixed type of values, and references of different types cannot be combined in a single undo log – this makes using them more cumbersome for some applications, see our discussion of the Rust type-checker in Appendix D. In contrast, our heterogeneous store can contain references of any type.

Search monads. If we cannot find "snapshots as a service", we looked for such code bundled into a larger abstraction, namely a backtracking/search library. We have not found interesting code to snapshot state in search monads or logic programming monads.

Software Transactional Memory. Software Transactional Memory libraries are designed for concurrency rather than sequential use. In particular, their main concern is to detect races with another transaction running concurrently. STM libraries typically do implement a form of journaling, but with different requirements that makes a comparison difficult. In particular, the implementations that we studied cannot implement record elision, as they need to track the previous *and* final value of each transaction variable – they cannot elide all tracking even if the variable was already modified by the continuation.

Bespoke implementations in types, solvers. We surveyed implementations of snapshottable stores hidden inside type checkers (we surveyed GHC, Scala 2 and 3, Rust, OCaml), SAT/SMT solvers (CVC5 explicitly mentions, but all solvers implement something like this) and a few constraint solvers. For reasons of space, this content is moved in Appendix D.

We found that most implementations are specialized for semi-persistent snapshots, solvers implement record elisions while type checkers are typically more naive. The OCaml type-checker implementation stands out (their implementation is independent from ours) in having a Baker-inspired structure that would allow persistent snapshots. `ocamlc` also implements a weaker form of record elision based on the birth-date of references rather than the time of the last write, that seems to work very well for type-checking workflows thanks to a generational phenomenon: most type variables are modified shortly after they are created.

6.2 Mutable and persistent interfaces

Our API provides a *mutable* interface: mutation operations modify the input store directly: `update : store * params -> unit`. Another choice would be to provide a *persistent* interface, where mutation operations leave the input store unchanged, and return another store containing the modification. We write `pstore` to emphasize that the store is persistent:

```
val update : pstore * params -> pstore
```

Functional programming typically encourages persistent data structures, whose transparential referency helps for program reasoning. Using linear types (when provided by the source language) can provide similar benefits for mutable interfaces, reformulated using a linear function that consumes its input:

```
val update : store * params ⊸ store
```

Conversely, the mutable (or linear) interface is often preferred for performance reasons. Some structures have efficient persistent implementations, but other structures have mutable versions with better complexity or noticeably lower constant factors.

Some implementations expose a persistent interface only, but they rely on reference-counting schemes to know when the input store is uniquely owned, and perform a mutable update in that case – they dynamically switch to the linear API. See for example Puente [2017], [Stokke 2018], or the Functional but In-Place style popularized by Koka [Reinking, Xie, de Moura and Leijen 2021]. This has the potential to be a "best of both worlds" solution, but only in systems where the cost of reference counting is already paid by the runtime or accepted as standard practice – it is a diffuse cost that must be paid by all users to enable this capability.

6.3 Transient views of persistent data structures

Some persistent data structures provide a *transient* view into the data structure, on which mutable updates can be applied imperatively, which can then be turned back into a persistent state:

```
val transient : pstate -> state      val mutably : (* higher-order combinator *)
val persistent : state -> pstate       pstate -> (state -> unit) -> pstate
```

The `transient` combinator can be used, for example, to efficiently add a lot of elements at once into a persistent collection. This is a pattern popularized by the Clojure community [Hickey and contributors 2024], based on seminal ideas by Bagwell [2001]. Transient data structures can be found in many languages. For example, transient vectors and hash-maps can be found in Scala's standard library, but also in the JavaScript library `immutable.js` [Byron 2024], and in the Python library `pyrsistent` [Gustafsson 2023]. The C++ library `immer` [Puente 2017] provides transients Relaxed Radix Balanced (RRB) vectors.

Our interface is the other way around: we expose the mutable API by default, but our snapshots are persistent, letting users capture persistent versions at point of interest in their code, typically around an operation they may want to backtrack over.

The two styles are equally expressive: we can implement a persistent store API with transient views, and conversely a mutable-with-snapshot API can be built on top of persistent-with-transient-views APIs. Our work focuses on enabling forms of persistence for data structures that are typically provided with a mutable API only, with an easy migration path for existing users.

Moine, Charguéraud and Pottier [2022] proposes the only formal verification of a transient data structure that we are aware of. They verify both functional correctness and time complexity of a transient stack in Separation Logic, using CFML [Charguéraud 2022]. They represent the shared mutable state between snapshots using a dedicated assertion, which complexifies specifications. Thanks to Iris support for monotone ghost state, we remove the need for this assertion.

6.4 State-of-the-art algorithms

The value proposition of our work is to provide an *easy* way to equip an imperative data structure with backtracking – more generally, persistent snapshots. We of course do not expect the result to be competitive with specialized algorithms.

The standard complexity of a Union-Find implementation is $O(n\alpha(n))$ for a sequence of n `union` and `find` operations, with a $O(\log n / \log\log n)$ worst-case complexity for each operation in the sequence. If we require backtracking support (an operation to undo the last `union` operation), Westbrook and Tarjan [1989] prove a lower-bound of $\Omega(n \log n / \log\log n)$ for n operations, and Apostolico, Italiano, Gambosi and Talamo [1994] provide an optimal implementation providing an $O(\log n / \log\log n)$ worst-case bound per `union` and `find` operation, with a total space cost of $O(n)$ for the whole sequence of operations. Their `backtrack : graph -> int -> unit` operation runs in time $O(1)$, and it is in fact able to undo the n most recent `union` operations.

We have not implemented this algorithm, nor are we aware of existing implementations, but our intuition is that this algorithm would have noticeably higher constant factors than the traditional Union-Find implementation. In contrast, our approach requires no new algorithmic expertise (except to implement our `Store` library once and for all), it provides a much worse complexity of $O(n)$ for the backtracking operation (that is infrequent in the workloads we are considering, relatively to `find` and `get` queries), and very low constant factor overheads for existing operations – which are performance-critical for our workloads. Our space overhead is $O(n)$, as with state-of-the-art algorithms.

Demaine, Langerman and Price [2008] presents a persistent trie data structure, which is unrelated to our current interest, but it is of interest to us for two reasons. First, to our non-expert knowledge it presents a state-of-the-art implementation of persistent dynamic arrays (which can be resized dynamically), using a sophisticated "rebuilding" approach to interleave resizing work with updates – if you know of Okasaki's technique to amortize the reversal of a list to implement a persistent queue, think of a much harder version of this idea. Second, it contains a very useful, detailed discussion of notions of persistence used in algorithmic research, which we tried to summarize in our introduction. Coming back to persistent (resizable) arrays: the standard approach for persistent arrays comes from Dietz [1989], where each access operation has cost $O(\log \log n)$ in expectation (it is randomized), where n is the total number of operations performed so far. This dependence on the number of operations is problematic for many use-cases, including ours – we only have such a dependence on `backtrack` operations, and want to avoid them on access operations. Demaine, Langerman and Price [2008] lowers it to $O(\log \log \Delta)$, where Δ is the total size of the array.

Driscoll, Sarnak, Sleator and Tarjan [1989] exposes generic techniques to add partial persistence and full persistence to existing data structures; they are not exposed as support libraries, applying them requires changing the data structure and its operations in a systematic way. These techniques apply to all data structures that can be seen as a graph of nodes with *bounded in-degree* – there is a global bound on the number of parents of each node. The techniques are designed to provide $O(1)$ access to any version in the tree, and typically have higher constant factors than we would like. As it happens, the usual Union-Find data structure does not have bounded in-degrees, as an arbitrary number of nodes can point to the same representant.

6.5 Static checking and formal verification

Conchon and Filliâtre [2008] presents a static checking discipline for semi-persistent data structures, based on ghost updates in Why3, a programming language designed for deductive verification. One could also use linear types or unique ownership to capture semi-persistence. Our OCaml implementation performs no static checking, but we invalidate our data structures at runtime in such a way that incorrect use results in a clear dynamic failure rather than unspecified behavior.

Conchon and Filliâtre [2007] propose persistent arrays and a persistent Union-Find library written in OCaml, and verify them in Coq. (The Union-Find implementation is built on the persistent arrays, so in particular it has bad liveness properties, it retains the memory of all nodes forever.) They use a shallow embedding of OCaml in Coq with an explicit heap, and express specifications using dependent types. This approach leads to verbose specifications. On the contrary, we benefit from Separation Logic and provide simpler specifications. Conchon and Filliâtre [2007] verify the termination of functions of the library, which we do not. We posit that we can enhance our specifications and proofs with time credits [Charguéraud and Pottier 2019] to verify both the termination and the time complexity of our implementation. Our proof does establish that the version graph remains acyclic, which is the key argument needed for termination.

7 FUTURE WORK

Verification. We verified the persistent core of `Store`, forcing us to build a very good mental model of the subtle implementation, without record elision. The next step is the verification of record elision. We have already sketched the proof and do not expect any conceptual difficulty. In particular, the specifications of Section 3.2 remain the same: record elision is only an internal optimization. After, it would be nice to include complexity bounds in the specifications, and to extend the mechanized proofs to the semi-persistent API, which requires invalidating snapshots (and transactions).

Custom operations. `Store` currently supports a single mutable datatype, namely references. This is enough, as all mutable datatypes can be built on top of mutable references. For example, one can define a snapshottable dynamic array as a store reference over an array of store references, and build snapshottable hashtables on top of it.

We believe however that some datatypes would benefit performance-wise from being integrated more directly into our stores, by extending our version nodes with higher-level operations – adding a value to a dynamic array, writing a table at a given key, etc.

One could of course hardcode such higher-level operations in the `Store` implementation (the backtrackable trail of Z3 is hardcoded in this way), but we would prefer to let users define "custom operations" following a certain abstract interface (the context-depedent objects of CVC5 provide this). We have started working on this abstract interface and played with several iterations of this idea; in particular, we believe that it is possible to combine custom operations with record elision. A difficulty is to find the right balance between generality and performance: some interfaces are more expressive than others, but they suffer from higher constant factors.

Confluence. Consider a user manipulating two snapshottable union-find graphs, each with its own store. They may decide to "merge" the graphs together – and start unifying nodes from both sides. We do not provide support for this. It is possible to just keep a product of stores, and restore/capture them together (`rustc` does this), but better support for this use-case could be useful in some scenarios – that we have not encountered yet.

Rebuilding. Journaled implementations, including `Store`, are optimized for "single-threaded" computations where switching from one snapshot to another is rare. Their performance breaks down if trying, for example, to evolve two different versions in lockstep. This is a limit to the generality of our implementation. Improving on this probably requires being able to track several copies of the "global state" simultaneously. For example, one could ask to *rebuild* a given snapshot, a costly operation that would turn it into an independent copy of the state – in particular, its validity would not depend on active transactions anymore.

The algorithmics literature studies how to perform this rebuilding implicitly, whenever edit chains become long enough that it is worth it – the most elaborate works in this direction are Chuang [1992, 1994]. This introduces other costs, in particular in space, and makes it harder for users to reason about performance. We would rather keep this an explicit operation.

Our current implementation choice, where each reference really has a unique field storing its current state – instead of being an index into a copiable structure – is in tension with rebuilding, we do not see how to do it. It seems challenging to offer this capability without hurting constant factors and/or our memory-liveness properties (Section 2.5).

Acknowledgments hidden for anonymity.

ACKNOWLEDGMENTS

Acknowledgments.

REFERENCES

Alberto Apostolico, Giuseppe F. Italiano, Giorgio Gambosi, and Maurizio Talamo. 1994. The Set Union Problem with Unlimited Backtracking. *SIAM J. Comput.* 23, 1 (1994), 50–70.

Phil Bagwell. 2001. *Ideal Hash Trees.* Technical Report. EPFL. http://infoscience.epfl.ch/record/64398

Henry G. Baker. 1978. Shallow binding in Lisp 1.5. *Commun. ACM* 21, 7 (jul 1978), 565–569. https://doi.org/10.1145/359545.359566

Haniel Barbosa, Clark W. 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. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13243)*, Dana Fisman and Grigore Rosu (Eds.). Springer, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA 2002) (Lecture Notes in Computer Science, Vol. 2461)*. 152–164.

François Bobot, Bruno Marre, Guillaume Bury, Stéphane Graham-Lengrand, and Hichem Rami Ait El Hara. 2022. *Colibri2: a constraint-programming solver for smtlib.* https://colibri.frama-c.com

Lee Byron. 2024. *Immutable.js library for JavaScript.* https://github.com/immutable-js/immutable-js/

Arthur Charguéraud. 2022. The CFML tool and library. http://www.chargueraud.org/softs/cfml/.

Arthur Charguéraud and François Pottier. 2019. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning* 62, 3 (March 2019), 331–365. http://cambium.inria.fr/~fpottier/publis/chargueraud-pottier-uf-sltc.pdf

Tyng-Ruey Chuang. 1992. Fully persistent arrays for efficient incremental updates and voluminous reads. In *ESOP '92: 4th European symposium on programming*. 110–129.

Tyng-Ruey Chuang. 1994. A randomized implementation of multiple functional arrays. In *ACM conference on LISP and functional programming*. 173–184.

Sylvain Conchon and Jean-Christophe Filliâtre. 2007. A Persistent Union-Find Data Structure. In *ACM SIGPLAN Workshop on ML*. ACM Press, Freiburg, Germany, 37–45. http://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf

Sylvain Conchon and Jean-Christophe Filliâtre. 2008. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*. http://www.lri.fr/~filliatr/ftp/publis/spds-rr.pdf

Erik Demaine, Stefan Langerman, and Eric Price. 2008. Confluently Persistent Tries for Efficient Version Control. *Algorithmica* 57 (07 2008), 462–483. https://doi.org/10.1007/s00453-008-9274-z

Paul F. Dietz. 1989. Fully persistent arrays. In *Algorithms and Data Structures*. 67–74.

J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. 1989. Making data structures persistent. *J. Comput. System Sci.* 38, 1 (1989), 86–124. https://www.cs.cmu.edu/~sleator/papers/another-persistence.pdf

Tobias Gustafsson. 2023. *Pyrsistent library for Python.* https://github.com/tobgu/pyrsistent

Barry Hayes. 1997. Ephemerons: a new finalization mechanism. *SIGPLAN Not.* 32, 10 (oct 1997), 176–183. https://doi.org/10.1145/263700.263733

Hickey and contributors. 2024. Clojure Reference Manual on Transient Data Structuresransient. https://clojure.org/reference/transients.

Iris Development Team. 2024. `iris.base_logic.lib.gen_heap`. https://plv.mpi-sws.org/coqdoc/iris/iris.base_logic.lib.gen_heap.html.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf

Alexandre Moine, Arthur Charguéraud, and François Pottier. 2022. Specification and verification of a transient stack. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Philadelphia, PA, USA) *(CPP 2022)*. Association for Computing Machinery, New York, NY, USA, 82–99. https://doi.org/10.1145/3497775.3503677

Melissa E. O'Neill and F. Warren Burton. 1997. A new method for functional arrays. *J. Funct. Program.* 7, 5 (sep 1997), 487–513. https://doi.org/10.1017/S0956796897002852

François Pottier. 2014. Hindley-Milner elaboration in applicative style. In *International Conference on Functional Programming (ICFP)*. http://cambium.inria.fr/~fpottier/publis/fpottier-elaboration.pdf

François Pottier. 2021. Strong Automated Testing of OCaml Libraries. In *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs*. Saint Médard d'Excideuil, France. https://inria.hal.science/hal-03049511

Juan Pedro Bolívar Puente. 2017. Persistence for the masses: RRB-vectors in a systems language. *Proc. ACM Program. Lang.* 1, ICFP, Article 16 (aug 2017), 28 pages. https://doi.org/10.1145/3110260

Alex Reinking, Ningning Xie, Leonardo de Moura, and Daan Leijen. 2021. Perceus: garbage free reference counting with reuse. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. 96–111. https://doi.org/10.1145/3453483.3454032

Bodil Stokke. 2018. `im` *crate in Rust: in-place mutation*. https://docs.rs/im/latest/im/index.html#in-place-mutation

Simon Friis Vindum and Lars Birkedal. 2021. Contextual refinement of the Michael-Scott queue. In *Certified Programs and Proofs (CPP)*. 76–90. https://cs.au.dk/~birke/papers/2021-ms-queue-final.pdf

Jeffery Westbrook and Robert E. Tarjan. 1989. Amortized Analysis of Algorithms for Set Union with Backtracking. *SIAM J. Comput.* 18, 1 (1989), 1–11. https://doi.org/10.1137/0218001

A MONOLITH INTERFACE

```
type 'a sref = { key : int; default : 'a }
type 'a mapping = 'a Map.Make(Int).t

type 'a snapshot = { state : 'a mapping;
                     transactions : 'a transaction list; }
and 'a transaction = { snapshot : 'a snapshot;
                       mutable terminated : bool; }
and 'a store = 'a snapshot ref
```

B DETAILED MICROBENCHMARKS RESULTS AND ANALYSES

We introduce our microbenchmarks in Section 5.2, but for reasons of space we only gave a high-level summary of the results. The current appendix contains more details on our benchmarking setup, the results of each benchmark, and a summary analysis of the results.

B.1 Methodology

Performing accurate microbenchmarks is very difficult.

We account for runtime noise by running benchmarks many times, and can provide intervals / error estimates (we use the `hyperfine` tool). All the micro benchmarks are run on a machine with an AMD Ryzen Threadripper 3990X processor and 264Go of RAM. Hyper-threading and frequency scaling are disabled, the frequency is set to its maximum of 2.9GHz, and the benchmarks are run sequentially on a single isolated core, so that the noise level of running the same binary repeatedly is very low.

Other sources of measurement biases are harder to detect and control. Our general approach is to ensure that we know how to explain the benchmark results, and carefully study each result that we do not understand – more often than not, this comes from a measurement bias that must be fixed to give accurate results. For example, we found performance swings of up to 10% due to code alignment effects. (We now run our benchmarks with 16 different alignments to control this.)

In our opinion, the main threat to validity of the results below is that we have had access one noise-controlled benchmarking machine with a specific AMD ThreadRipper processor, and that some of the fine-grained qualitative comparisons may be different on other processors or architectures. This is an issue with microbenchmarks, which give a very detailed view of performance but are more sensitive to system differences. The macrobenchmark discussed in Section 5.3 are more robust in that regard.

B.2 Benchmark parameters

All benchmarks are purely synthetic, and they are parametrized by the following environment variables.

ROUNDS the benchmark does *something* in a loop, ROUNDS time; the total time should scale linearly with this variable (but this may not be just a **`for`** loop, there may be an environment growing from one round to the next).

NCREATE, NREAD, NWRITE : the logarithm of the number of references to create,read,write each round.

We use three sets of parameters to simulate different workloads:

default represents our default workload where backtracking operations are rare, and reads dominate writes. We use NCREATE=10, NREAD=16, NWRITE=12, with 4*1024 writes and 32*1024 writes per transaction. All references are touched in each transactio in each transactionn, so this is an ideal case for `Vector`.

capture-heavy tests a limit case where backtracking operations are much more frequent, with only a handful of `get`/`set` calls per transaction. We use NCREATE=2, NWRITE=4, NREAD=6, with 16 writes and 64 reads per transaction (spread over 4 references).

capture-heavy-large-support is a variant of `capture-heavy` where there are many references around, but only a few of them are touched by each transaction. We use NCREATE=10, NWRITE=4, NREAD=6, with 16 writes and 64 reads per transaction (spread over 1024 references; most references are untouched at each round).

B.3 Colibri2

ompared to the microbenchmarks summary in Section 5.2, we include an additional third-party implementation, `Colibri2`. Colibri2 [Bobot, Marre, Bury, Graham-Lengrand and Rami Ait El Hara 2022] is a constraint-programming and SMT solver written in OCaml, with an implementation of backtrackable references.

We wanted to measure the performance of `Colibri2` because it is uses a different design from our implementation or `Facile`. It uses a "fat node" representation where the previous values of each reference are stored within the reference itself. The work of restoring an earlier version is done lazily, on demand. `rollback` is constant-time and does not update references; when we `get` a reference we check that it has the current version or rewind the reference state. Note that storing the history locally in each reference improves the memory-liveness properties compared to our implementation, where each snapshot retains old versions of all references it recorded.

As you will see in this section, we found that this on-demand approach has a noticeable overhead due to the extra check in the performance-critical operation `get`. We discussed this with the authors of Colibri2, and in January 2024 they changed their implementation to be very close to ours. (The numbers below correspond to the previous, distinctive implementation.)

B.4 Commit-only and abort-only results

In the results that we have shown in the summary in Section 5.2, the `Transactional` benchmark and its `Capture-heavy` variant test a mix of successful and failed transactions, `commit` and `restore`. On the other hand, our third-party implementations `Facile` and `Colibri2` were written for solver-backtracking use cases and do not implement a dedicated `commit` operation. It is possible to just *do nothing* on commit – we leave the snapshot in history, they support nested snapshots, but this provides noticeably worse performance than our implementations that compress the history on `commit` – new records can be elided.

In this more detailed section, we provide a more fine-grained comparison by separating the `Transactional` and `Capture-heavy` benchmarks in two variants, an `abort`-only variant and a `commit`-only variant. This changes the qualitative comparison, as `Facile` becomes competitive with our semi-persistent implementations in the `abort`-only scenarios.

Figure 7 provides an overview of the `abort`-only results, and Figure 8 an overview of the `commit`-only results.

B.5 Per-benchmark results and analysis

B.5.1 Raw.

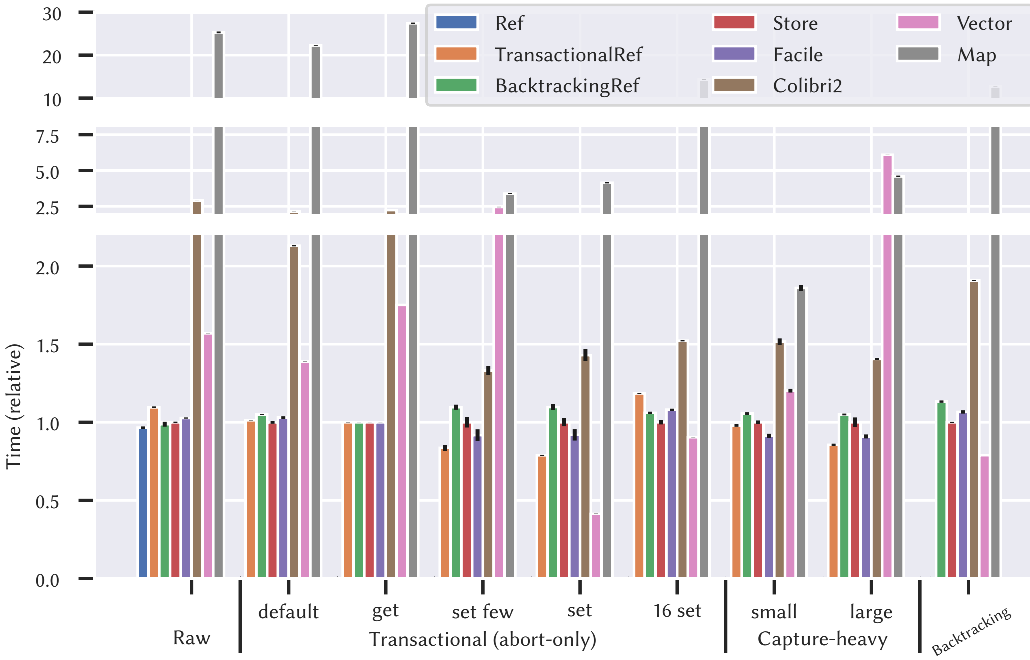


Fig. 7. Micro-benchmark results (abort-only)

Implementation	Time (ms)	Relative
Ref	77.8 ± 0.65	1.00 ± 0.01
BacktrackingRef	79.6 ± 1.27	1.02 ± 0.02
Store	80.6 ± 0.33	1.04 ± 0.01
Facile	82.8 ± 0.31	1.06 ± 0.01
TransactionalRef	88.3 ± 0.45	1.14 ± 0.01
Vector	126.5 ± 0.06	1.63 ± 0.01
Colibri2	233.8 ± 0.08	3.01 ± 0.03
Map	2038.1 ± 17.47	26.20 ± 0.31

`Ref` is the gold standard for this benchmark. `Store`, `BacktrackingRef` and `Facile` have a small overhead (2%-3%). `TransactionalRef` is a bit slower (12% overhead): it performs two writes per `set` instead of one. `Vector` is even slower (67% overhead), probably due to additional indirections and bound checks, and `Map` is an order of magnitude slower than the rest (25-27 times).

`TransactionalRef` has a slower `set` operation in the absence of backtracking (two polymorphic writes instead of one), but it keeps the same code in the presence of backtracking (thanks to its restriction to non-nested transactions). It will perform better (relatively to `BacktrackingRef`, `Facile`, `Store`) in the Transactional benchmarks that follow.

B.5.2 Transactional, `abort`*-only.* In a transactional scenario, `Ref` cannot be used. `Store`, `TransactionalRef`, `BacktrackingRef` and `Facile` are the fastest and all within 5% of each other; `Vector` is about 40% slower, `Colibri2` is about 2× slower and `Map` is 22× slower.

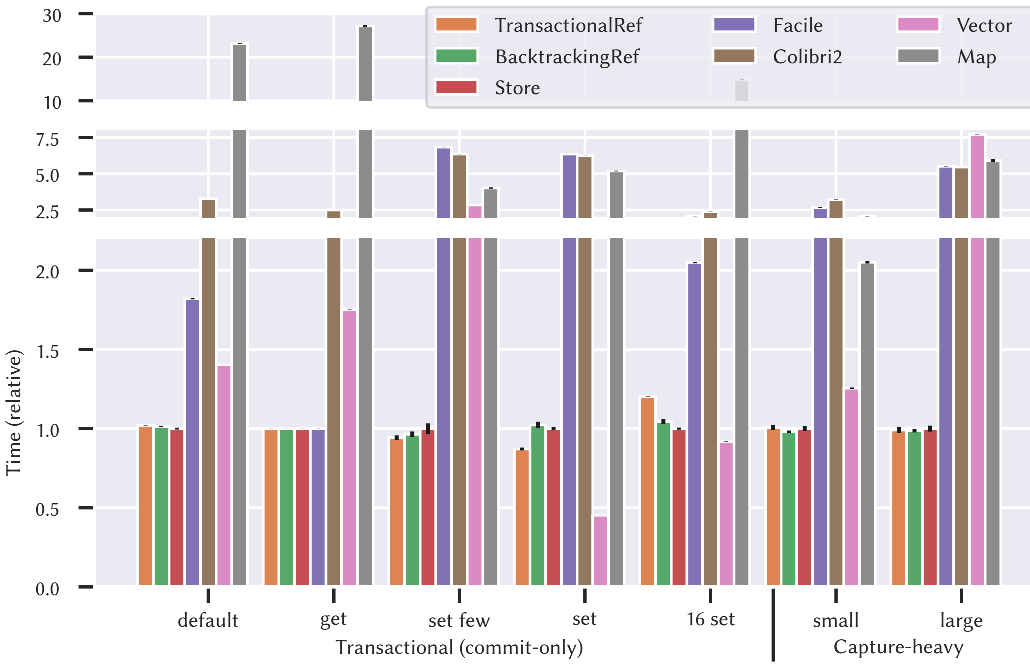


Fig. 8. Micro-benchmark results (commit-only)

Mixed get/set workload	Time (ms)	Relative
Store	70.1 ± 0.64	1.00 ± 0.01
TransactionalRef	71.0 ± 0.12	1.01 ± 0.01
Facile	72.2 ± 0.56	1.03 ± 0.01
BacktrackingRef	73.6 ± 0.23	1.05 ± 0.01
Vector	97.3 ± 0.05	1.39 ± 0.01
Colibri2	149.3 ± 0.29	2.13 ± 0.02
Map	1560.4 ± 5.07	22.26 ± 0.22

`Get`*-only workload.* `Vector` is 70% slower than the other implementations on the “get” variant, partially due to performing many unnecessary copies.

Get	Time (ms)	Relative
BacktrackingRef	73.7 ± 0.02	1.00 ± 0.00
Facile	73.7 ± 0.03	1.00 ± 0.00
Store	73.7 ± 0.02	1.00 ± 0.00
TransactionalRef	73.7 ± 0.10	1.00 ± 0.00
Vector	129.1 ± 0.02	1.75 ± 0.00

Continued on next page

Get	Time (ms)	Relative
Colibri2	164.8 ± 0.02	2.24 ± 0.00
Map	2021.1 ± 11.93	27.42 ± 0.16

Varying ratios of `set`. For `set`-only benchmarks we measure three different ratios of write: in "set few", only one out of 16 references is modified (once) at each round. In "set 1", each reference is modified exactly once. In "set 16", each reference is modified 16 times.

Set few	Time (ms)	Relative
TransactionalRef	43.4 ± 1.01	1.00 ± 0.03
Facile	47.6 ± 1.92	1.10 ± 0.05
Store	51.9 ± 1.75	1.20 ± 0.05
BacktrackingRef	56.9 ± 0.84	1.31 ± 0.04
Colibri2	69.1 ± 1.47	1.59 ± 0.05
Vector	127.1 ± 1.13	2.93 ± 0.07
Map	175.5 ± 1.62	4.04 ± 0.10

Set1	Time (ms)	Relative
Vector	52.9 ± 0.29	1.00 ± 0.01
TransactionalRef	101.0 ± 0.42	1.91 ± 0.01
Facile	117.7 ± 4.62	2.22 ± 0.09
Store	128.1 ± 3.30	2.42 ± 0.06
BacktrackingRef	140.6 ± 2.33	2.66 ± 0.05
Colibri2	183.2 ± 4.85	3.46 ± 0.09
Map	530.3 ± 5.45	10.02 ± 0.12

Set16	Time (ms)	Relative
Vector	53.3 ± 0.08	1.00 ± 0.00
Store	59.0 ± 0.82	1.11 ± 0.02
BacktrackingRef	62.5 ± 0.46	1.17 ± 0.01
Facile	63.7 ± 0.40	1.19 ± 0.01
TransactionalRef	69.9 ± 0.19	1.31 ± 0.00
Colibri2	89.7 ± 0.25	1.68 ± 0.01
Map	845.5 ± 5.54	15.85 ± 0.11

`Vector` shines on the "set" variant where it is 2× faster than other implementations. The "set" variant is the best-case scenario for the "full copy" approach, since all other implementations degrade to also doing a full copy with worse constant factors. This advantage goes away if many `set` operations are performed in a transaction and record elision kicks in: in the "16 set" variant, Vector is only about 10-15% faster than the other implementations. It also goes away if only a subset of the references are modified: in the "set few" variant, it is 3× slower than the best implementation.[6].

The relative performance of TransactionalRef can be explained by its set implementation: while its *elided* write is slower than the other journaled implementations, its *non-elided* write is simpler due to not supporting nesting. This gives it a performance boost in scenarios that do not allow record elision (the "set" variant and the "large" capture-heavy variant); that goes away as the number of writes per reference increases (in the "16 set" variant and "small" capture-heavy varieant).

More generally, the difference in performance between the journaled implementations boils down to relative efficiency of elided and non-elided writes. The default and set 16 configurations compare write performance, and the set and set few configurations compare non-elided write performance. TransactionalRef has a single write implementation that is faster than non-elided writes of other implementations but slower than their elided writes. Facile has fast non-elided writes, but slow elided writes. Store has fast elided writes, but slow non-elided writes (with an extra caml_modify compared to Facile). BacktrackingRef has slow elided and non-elided writes.

Colibri2 is generally slow, partially due to get operations being slower but also set operations are slower in general.

Capture-heavy variants. The Map implementation has a much smaller overhead in the "small" capture-heavy variant; however, even in this ideal scenario (few references and few read/write operations per transaction), it is still twice as slow as the journaled implementations. When the number of references increases, the logarithmic overhead shows up, as in the "large" capture-heavy variant – where Vector also performs much worse.

Capture-heavy, small support	Time (ms)	Relative
Facile	41.4 ± 0.62	1.00 ± 0.02
TransactionalRef	44.4 ± 0.38	1.07 ± 0.02
Store	45.3 ± 0.52	1.10 ± 0.02
BacktrackingRef	47.8 ± 0.37	1.16 ± 0.02
Vector	54.5 ± 0.55	1.32 ± 0.02
Colibri2	68.7 ± 0.94	1.66 ± 0.03
Map	84.3 ± 0.86	2.04 ± 0.04

Capture-heavy, large support	Time (ms)	Relative
TransactionalRef	60.3 ± 0.56	1.00 ± 0.01
Facile	64.1 ± 0.91	1.06 ± 0.02
Store	70.5 ± 2.17	1.17 ± 0.04
BacktrackingRef	74.0 ± 0.42	1.23 ± 0.01
Colibri2	99.0 ± 0.46	1.64 ± 0.02

Continued on next page

[6]In this particular benchmark, we find that the break-even point is when around one-fourth of the references are modified per transaction.

Capture-heavy, large support	Time (ms)	Relative
Map	323.5 ± 3.85	5.37 ± 0.08
Vector	429.6 ± 0.51	7.13 ± 0.07

B.6 Transactional, `commit`-only

The results for `commit`-only transactional benchmarks are similar for most implementations, except for `Facile` and `Colibri2` which do not support efficient `commit` operations. `TransactionalRef` and `BacktrackingRef` have very fast `commit` operations. `Store` is marginally slower but still competitive.

Mixed get/set workload	Time (ms)	Relative
Store	69.2 ± 0.41	1.00 ± 0.01
BacktrackingRef	70.1 ± 0.35	1.01 ± 0.01
TransactionalRef	70.6 ± 0.10	1.02 ± 0.01
Vector	97.0 ± 0.03	1.40 ± 0.01
Facile	125.9 ± 0.25	1.82 ± 0.01
Colibri2	226.2 ± 0.32	3.27 ± 0.02
Map	1602.4 ± 5.21	23.16 ± 0.16

Get	Time (ms)	Relative
BacktrackingRef	73.7 ± 0.01	1.00 ± 0.00
TransactionalRef	73.7 ± 0.02	1.00 ± 0.00
Store	73.7 ± 0.02	1.00 ± 0.00
Facile	73.7 ± 0.02	1.00 ± 0.00
Vector	129.1 ± 0.07	1.75 ± 0.00
Colibri2	184.1 ± 0.04	2.50 ± 0.00
Map	2002.2 ± 18.20	27.17 ± 0.25

Set few	Time (ms)	Relative
TransactionalRef	41.4 ± 0.63	1.00 ± 0.02
BacktrackingRef	42.4 ± 0.78	1.02 ± 0.02
Store	44.0 ± 1.45	1.06 ± 0.04
Vector	125.0 ± 0.46	3.02 ± 0.05
Map	176.6 ± 1.97	4.26 ± 0.08
Colibri2	279.4 ± 0.81	6.74 ± 0.10
Facile	300.4 ± 0.63	7.25 ± 0.11

Set 1	Time (ms)	Relative
Vector	52.5 ± 0.06	1.00 ± 0.00
TransactionalRef	100.9 ± 1.06	1.92 ± 0.02
Store	115.9 ± 1.25	2.21 ± 0.02
BacktrackingRef	118.5 ± 2.41	2.26 ± 0.05
Map	601.2 ± 3.32	11.46 ± 0.06
Colibri2	722.7 ± 1.05	13.77 ± 0.02
Facile	736.5 ± 1.61	14.04 ± 0.03

Set 16	Time (ms)	Relative
Vector	53.3 ± 0.12	1.00 ± 0.00
Store	58.1 ± 0.38	1.09 ± 0.01
BacktrackingRef	60.7 ± 0.93	1.14 ± 0.02
TransactionalRef	69.8 ± 0.13	1.31 ± 0.00
Facile	119.0 ± 0.27	2.23 ± 0.01
Colibri2	139.1 ± 0.34	2.61 ± 0.01
Map	867.2 ± 8.36	16.28 ± 0.16

Capture-heavy, small support	Time (ms)	Relative
BacktrackingRef	36.8 ± 0.26	1.00 ± 0.01
Store	37.5 ± 0.57	1.02 ± 0.02
TransactionalRef	37.8 ± 0.54	1.03 ± 0.02
Vector	47.1 ± 0.16	1.28 ± 0.01
Map	76.9 ± 0.30	2.09 ± 0.02
Facile	100.4 ± 0.90	2.73 ± 0.03
Colibri2	120.4 ± 0.75	3.27 ± 0.03

Capture-heavy, large support	Time (ms)	Relative
BacktrackingRef	53.7 ± 0.62	1.00 ± 0.02
TransactionalRef	53.9 ± 1.02	1.00 ± 0.02
Store	54.4 ± 1.03	1.01 ± 0.02
Colibri2	297.0 ± 0.53	5.53 ± 0.06
Facile	300.1 ± 0.96	5.59 ± 0.07
Map	322.0 ± 5.64	5.99 ± 0.13
Vector	419.6 ± 0.48	7.81 ± 0.09

B.6.1 `Backtracking`.

Implementation	Time (ms)	Relative
Vector	164.3 ± 0.12	1.00 ± 0.00
Store	208.1 ± 0.72	1.27 ± 0.00
Facile	221.8 ± 2.20	1.35 ± 0.01
BacktrackingRef	235.6 ± 1.50	1.43 ± 0.01
Colibri2	396.9 ± 0.71	2.42 ± 0.00
Map	2647.8 ± 7.74	16.12 ± 0.05

This benchmark tests deeply nested backtracking chains, with our standard set parameters where all references are set 4 times and read 16 time in each round. This scenario is again favorable to our full-copy baseline `Vector`, with "journaled" implementations being somewhat slower at 29%-43% overhead. `Map` remains very slow, 16× slower than `Vector`. (`TransactionalRef` does not support nested transactions, so it cannot be used here.)

B.6.2 Persistent API. Finally, we use the `Backtracking` benchmark, which performs deeply nested backtracking, to measure the performance difference between the persistent and semi-persistent operations of `Store` – we run the same workload with the `tentatively` function reimplemented on top of `capture`/`restore`. On this test, we observe a 50% overhead for the persistent API.

Implementation	Time (ms)	Relative
Backtracking-abort	210.2 ± 0.30	1.00 ± 0.00
Backtracking-persistent	314.7 ± 1.46	1.50 ± 0.01

Remark. We conclude that there are *some* workloads where the semi-persistent API provides a noticeable performance difference. The difference, however, remains fairly small for a microbenchmark, and would typically not be noticeable for many end-user applications.

C MACROBENCHMARKS DETAILS

This appendix contains the full details on the macrobenchmarks mentioned in Section 5.3.

C.1 System F type-checking in Inferno

The Inferno project implements type-inference for a small ML language, and for well-typed terms it produces a "witness" or an "elaboration", which is an explicitly-typed version of the input program in a variant of System F. Inferno includes a type-checker for this explicitly language, which is much simpler than type inference and can be used to catch bugs in the type inference machinery.

This explicit type checker uses a Union-Find data structure to check equality between types. We worked on a prototype extension of Inferno with GADTs, which required to add backtracking to the Union-Find graph of System F types to support local type-equality assumptions that are undone when leaving the scope of a GADT equation.

This was our initial motivation for implementing Store, and an ideal scenario for journaled implementations. `Vector` is a bad choice because we are in the "large support" worst-case: most backtracking points (that is, pattern-matching clauses containing GADTs) are short-lived and modify only a few Union-Find nodes. On the other hand, `Map` introduces an important overhead, even when the code does not use GADTs.

Now that we have `Store` implemented we can replace `Vector` with it and compare performance. We use Inferno's own performance test, which is to generate a large *random* term (with a generator design to produce well-typed terms), infer its type and check its explicitly-typed version.

The results in Figure 6a show that in this real program performing many other operations than `Store` operations, using `Vector` is 1.3× slower than using our `Store` implementation, and using `Map` is 4.2× slower. Adopting `Store` is easy and comes with a direct, noticeable performance improvement.

The large random term type-checked in the test above does *not* contain any GADTs[7] (the random generator does not know about them), so no snapshots are actually taken when running this test. This is a best case for `Vector` – it does not suffer from the "large support" situation.

We do not have good, representative test programs that contain a reasonable frequency of GADT constructs, but as a limit case we checked the performance of the type-checker on a small GADT example – a very short program that *only* checks GADT features, checked 1000 times in a loop. The results (below) should be taken with a grain of salt, as this is closer to microbenchmark territory again. For this limit test shown in Figure 6b, the System F type-checker remains 4× slower with `Map` than with `Store`, but using `Vector` now performs terribly, almost 70× slower, due to the "large support" situation.

C.2 System F type inference with GADTs (Inferno)

The previous test measures the performance of type-checking of explicitly-typed terms in Inferno. Inferno also uses a Union-Find data structure during inference of ML terms, performing inference via unification as usual. As we explained previously, Inferno implements a transactional behavior for unification of types: a single unification constraint is decomposed in many variable-variable unifications, but if any of those fail, we revert all changes to the inference state caused by this unification constraint in order to generate clear error messages. We measure the type-inference work for (again) a large randomly-generated ML term, with our Union-Find graph instantiated by different store implementations.

This workload has a relatively high number of backtracking points, most of which perform little work (most type-type unification are on small types that perform few variable-variable unifications). This workload is a worst-case scenario for full-copy implementations such as `Vector`, but it is a best case for full-persistence implementations such as `Map`. There are no nested transactions, so François Pottier's `TransactionalRef` implementation can be used – in fact, it was designed precisely for this use-case, so it is the gold standard for this test.

We see in Figure 6c that `Store` has the same performance as `TransactionalRef` despite being much more general; Map is much slower, and `Vector` is unacceptably slow.

C.3 Sudoku solver

We wanted to test backtracking programs that are not doing type-checking of any form. We are interested in using `Store` in SAT or SMT context, but SAT/SMT engines have deeply ingrained forms of backtracking and it is not so easy to port existing solvers to Store. Instead we looked for Sudoku solvers written as constraint-solving programs, which are typically simpler. We found an OCaml implementation of a Sudoku solver[8] written by Alain Frisch in 2005 with performance in mind, and we adapted it to use `Store`.

[7]The implementation of the type-checker must support GADTs, and thus use a snapshottable store. For this specific benchmark without GADTs, we tried using built-in references out of curiosity, and the performance is the same as `Store`.

[8]http://alain.frisch.fr/sudoku.html

A constraint-based Sudoku solver operates on a "board state", which tracks the possible values (the "domain") of each board position. Whenever the domain of a board position is refined, we propagate constraints to other positions whose domain could be refined in turn (in the same row, column or block). Once all constraints have been propagated fully, we have to perform backtracking: choose a yet-undetermined position, and try each of the possible value of its domain – backtracking any state change after each attempt fails. Sudoku solvers must represent the board state efficiently (this solver uses an array of integers, where integers are used as bitsets to represent the domains), propagate constraints efficiently, and use good heuristics to decide which position to backtrack on.

Alain Frisch's Sudoku solver uses a hand-crafted "full copy" implementation, that copies the full board state at each backtracking point. The implementation is careful about reusing buffers to avoid allocations when possible. The state is fixed and relatively small, so copy is cheap – we used a test benchmark on a 25×25 sudoku board, so the state is an array of 625 integers.

`base` is Alain Frisch's hand-crafted implementation, and it remains the fastest. `Store` adds 20% overhead. `Store (persistent)` uses our persistent API rather than our semi-persistent API; it performs slightly worse at 30% overhead. Finally, `Vector` is 3× slower. `Vector` is noticeably slower because it induces a memory representation that is less compact than the hand-written implementation[9] and cannot reuse buffers.

Our conclusion is that even though `Store` does not beat a hand-crafted full-copy implementation of backtracking in this case, its low overhead remains acceptable on backtracking-intensive programs. Using `Store` instead of carefully copying temporary buffers may be a good deal for some programmers.

D RELATED WORK: BESPOKE IMPLEMENTATIONS IN TYPERS AND SOLVERS

Type checkers. The GHC type-checker does not implement backtracking of any form.

The Scala 2 type-checker implements journaled backtracking for its type inference variables, a simple semi-persistent implementation with a global list of undo actions.[10]. No record elision. Interestingly, another custom undo log is maintained in the function inliner – the project could benefit from generic snapshottability support.

The Scala 3 type-checker implements a `snapshot/restore` interface for the entire type-checking state[11], but the `snapshot` logic is intentionally trivial as all this state is maintained in fully persistent data structures. (Looking for use-cases of the `snapshot` function shows all the places where the type-checker resorts to backtracking.)

The Rust type-checker implements "undo logs" for its mutable state, using the `undo_log` module of the `ena` crate we mentioned earlier. Because undo logs are homogeneous, different components of the type-checking state are stored in different undo logs. A module in the type-checker gathers all these logs[12], with a single function to snapshot and restore them all at once.

The OCaml type-checker implements a snapshottability mechanism for its type variables, whose implementation is also inspired by (or a rediscovery of) Baker.[13]. The implementation seems to support full persistence, but it seems that it is only used in a semi-persistent way in the compiler codebase. This implementation performs a simplified form of record elision, based on the birth

[9] To measure the importance of the compact memory implementation, we replaced the `int array` implementation of Alain Frisch by an exactly equivalent `int ref array` implementation, introducing one indirection in the memory represent. This introduces a 48% overhead, larger than `Store`.

[10] https://github.com/scala/scala/blob/577ab8e0/src/reflect/scala/reflect/internal/tpe/TypeConstraints.scala#L26-L76

[11] https://github.com/lampepfl/dotty/blob/7f410a/compiler/src/dotty/tools/dotc/core/TyperState.scala#L29-L43

[12] https://github.com/rust-lang/rust/blob/9afdb8d1/compiler/rustc_infer/src/infer/undo_log.rs#L19-L32

[13] https://github.com/ocaml/ocaml/blob/572aeb5f/typing/types.ml#L490-L514, https://github.com/ocaml/ocaml/blob/572aeb5f/typing/types.ml#L755-L759, https://github.com/ocaml/ocaml/blob/572aeb5f/typing/types.ml#L851-L874

date of the reference rather than the timestamp or generation of its last write. Indeed, each type variable has a unique identifier implemented as consecutive integers starting at 0, which can also serve as a "birth date" for the type variable. The snapshot implementation tracks the value of the type identifier counter when the last snapshot was taken. When performing a write on a type, it performs record elision if the type has a higher identifier than the last snapshot – it was created after the snapshot was taken. This heuristic is less precise than our record elision, but it comes for free once type identifiers are there. It seems fairly effective for a type-checker due to a sort of generational phenomenon: most type variables are modified a lot shortly after they are created, and more rarely afterward. (Disabling this form of elision makes type-checking about 5% slower on some files of the compiler codebase.)

Constraint solvers and SAT/SMT solvers. Based on discussions with implementors of automated theorem projects, we conjecture that all SMT solvers include some version of a general snapshottable store – but of course they did not tell anyone until we explicitly asked them. The only explicit mention we found is in the recent overview paper on CVC5, Barbosa, Barrett, Brain, Kremer, Lachnitt, Mann, Mohamed, Mohamed, Niemetz, Nötzli, Ozdemir, Preiner, Reynolds, Sheng, Tinelli and Zohar [2022], which describes "Context-Dependent Data Structures" (Section 2.4)[14], and currently supports context-dependent maybe/option values, append-only lists, dequeues, insert-only hashsets, and hashmaps. Z3 simply adds support for adding arbitrary edit events on the "trail", and does not seem to support record elision.[15] The implementations in SMT solvers are semi-persistent, and their API is influenced by the internal vocabulary of SAT search algorithms; typically, one does not backtrack to a given snapshot, but to a "decision level".

Constraint-based solvers seem to also implement semi-persistent snapshottable structures, and we have found implementations of record elision, which is relatively natural in the semi-persistent case. We mentioned Facile, an OCaml implementation, but for example the Java constraint solver `choco-solver` also has support for generic "trails", and performs record elision[16].

[14]https://github.com/cvc5/cvc5/blob/92caabc7/src/context/context.h
[15]https://github.com/Z3Prover/z3/blob/2880ea39/src/util/trail.h
[16]https://github.com/chocoteam/choco-solver/blob/efb697ea/solver/src/main/java/org/chocosolver/memory/trailing/EnvironmentTrailing.java, https://github.com/chocoteam/choco-solver/blob/efb697ea/solver/src/main/java/org/chocosolver/memory/trailing/StoredInt.java#L33-L48