Storable types: free, absorbing, custom

Basile Clément, Camille Noûs, Gabriel Scherer

Storable types: free, absorbing, custom

Basile Clément[1], Camille Noûs[2], and Gabriel Scherer[3]

[1]OCamlPro, Paris, France
[2]Laboratoire Cogitamus
[3]INRIA & IRIF, Université Paris Cité, France

The Store library (Noûs and Scherer, 2023; Allain, Clément, Moine, and Scherer, 2024) provides a *store* datastructure that tracks a bag of mutable references, and provides an efficient way to *capture* its current state and *restore* it later on. This is useful to implement backtracking algorithms over mutable state, as found in type-checkers and automated theorem provers.

We extend the Store library to store not only mutable references, but also richer data structures such as dynamic arrays and hash tables. Instead of hard-coding specific data structures within Store, we capture broad classes of *storable* data structures through generic interfaces. These interfaces are determined by the properties of the (anti)operations of each data structure. We describe three interesting classes of operations – *absorbing*, *free*, and *custom* (the general case) – and the corresponding APIs and implementations.

1 Introduction

1.1 Use-cases for storable types

The use-case of the Store library is to allow to take snapshots of a mutable data structure. For example, we use it in type-checking libraries to backtrack over a union-find graph of type unifications. To do this, we change the definition of the union-find graph to use "stored references" instead of normal reference:

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

In general, users interested in adopting store must change their mutable data definitions, replacing any reference or mutable field by a stored reference of type `_ Store.Ref.t`. Any mutable data structure can be systematically modified in this way. For example, a hash table can be implemented as a reference to an array of singly-linked lists; we would use a stored reference for each element of the array, for the reference to the array itself (which is mutated on resizes), and for the element and tail pointer of each cell in the linked list.

We would like to extend Store to support a richer set of built-in data structures for convenience, and also to support generic *storable types*, letting users version pre-existing data structures in the store by implementing a generic interface to track their modifications. There are at least two reasons to do this:

1. Pre-existing data structures may be faster and more sophisticated than what we, as Store authors or users, are willing to duplicate and maintain. For example, maybe

some library authors maintain an extremely sophisticated hash table implementation, and are not interested in integrating Store directly – nor us in duplicating their code.

2. If we implement all mutable pointers in the structure with stored references, we get the same backtracking behavior for all parts of the structure. But sometimes more clever policies are desirable. For example, in a dynamic/resizable array, mutations to array elements need to be backtracked, but we may want to *not* undo the the transition to a larger backing array. Similarly, some data structures in SAT or SMT solvers are designed to *not* require any work when backtracking.

For example, if the user has already chosen a third-party hash table implementation, and they want to integrate it in a Store-powered backtracking program, they want to implement a "storable hash table" using their hash table library. They would have to write some glue code to record operations in the store and let them be rolled back when an older version is restored. What interface should Store demand for this glue code? This is the question that we set out to answer in the present article – and in an upcoming release of the Store library.

We are using Store for type-inference implementations within the Inferno project (Pottier, Martinot, and Scherer, 2023). We would like to also adopt Store in the Alt-Ergo SMT solver (Bury, Clément, Coquereau, Conchon, Contejean, de Olivera, El Hara, Iguernlala, Lescuyer, Mebsout, Roux, and Villemot, 2015), and we believe that storable types are a necessary milestone to adopt it there.

1.2 Storable types of interest

Here are a few examples of data structures which we would like to support as storable types. Having a diverse set of structures in mind avoids over-specializing the design.

references: Store already has built-in support for references, and they are going to be faster than an implementation through a generic mechanism. Implementing them as storable types should still be possible, with only a constant-factor performance overhead.

dynarray: we want to support dynamic/extensible arrays. If a dynamic array transitions to a larger backing array to store more elements, we would like backtracking to keep this larger array, instead of reverting to the previous, smaller backing array. (Going back to the smaller backing array is another implementation that could make sense for some memory-conscious applications, and it is easier to express. We want both choices to be expressible.)

hash table or kvset: hash tables with backtracking support are common in SMT solvers, they are a natural data structure to expect. Key-value sets (kvsets) are a simpler data structure where key-value pairs can be added and removed, without support for modifying the value bound to a key.

n-watching sets of optional references: an *n-watching set* is a bag of at least n optional references – they may or may not carry a value. Users can create several n-watching sets, and each optional reference can be part of several sets, with a *global* operation to efficiently find all the n-watching sets where no more than n references do not carry a value. This is a generalization of "watched literals" in SAT solvers, designed to offer efficient backtracking.

1.3 Anti-operations and their composition

Our idea for storable types is that when a user performs a modifying operation on a data structure in the store, the store should record what we call an *anti-operation*, a description of how to undo this operation to restore the current value of the data structure.

Different data structures offer different sets of operations, and they also have different sets of anti-operations.

For example, references can be described as supporting operations of the form `Set(v)`, where `v` is a new value for the reference. If the current value of a reference is `v`, then an anti-operation that can undo the operation `Set(v')` would be `Unset(v)`, which simply remembers the current value of the reference.

In a kvset, operations can be described as `Insert(k, v)` and `Delete(k)`, and the corresponding anti-operations would be `Undelete(k,v)`, which undoes the delection of `k` by reinstating the key `v`, and `Uninsert k`.

In these two examples, operations and anti-operations are very symmetric, so it is tempting to suggest to use the same type for both. Yet, this is not always the case. For example, an at-most-one-assignment reference (references start uninitialized, and can no longer change value once initialized) may have an operation `Init(v)` and anti-operation `Uninit`.

Composition When the user performs several operations on a storable type after taking a snapshot, we can record a list of anti-operations, one for each operation, but often we can combine several anti-operations into a single anti-operation. This saves space, and it can be faster to rollback.

In general we ask that anti-operations for a given storable type form a monoid, with a multiplication or composition operation: if a is the anti-operation for the operation o in the current state, and b is the anti-operation for p in the state resulting from o, we can write $a.b$ for the composed anti-operation, that is able to un-do the sequential composition $(o; p)$ from the current state. In general, supporting composition requires enriching the set of anti-operations with new kinds of anti-operations, or even changing the type of anti-operations. Our earlier idea of storing a list of anti-operations in fact corresponds to the most general way to do this: if we designed a type A with an anti-operation for each operation of the structure (but no support for composition), then we can use the type $\mathsf{SnocList}(A)$ of lists that grow on the right, with list concatenation for composition, as the free monoid of (composable) anti-operations.

1.4 Properties of composition: absorbing, free, custom

We found that the composition of anti-operations has different properties depending on the data structure – and on the signature of the anti-operations. We distinguish three classes of anti-operations of interest.

Absorbing composition We say that a monoid of anti-operations is *absorbing* if, for all composable anti-operations a, b, we have $a.b = a$. This property implies that the anti-operation for any operation o contains enough information to un-do not only o, but any future extension of o into a larger sequence of operations $(o; p; \dots)$. This is in the particular the case when a contains the full value of the current state of the structure.

For example, the reference anti-operation `Unset(v)` is absorbing: when the current value of the reference is `v`, this information contains enough information to undo any operation of the form `Set(v1)`, but also any sequence of updates `(Set(v1); Set(v2); ...)`. Indeed, `Unset(v)` contains the current value `v` in full, it captures the current state of the reference.

Consider a storable type of counters, with operations `Incr` and `Decr` to increment and decrement the counter. We could take integers as our monoid of anti-operations, composed by addition: the anti-operation for the sequence `Incr; Decr; Incr; Incr` would be (-2) – it does not depend on the current value of the counter. This monoid of anti-operations is not absorbing: subtracting 2 only allows to go back to the initial state from the current state, not from a future state where we increment the counter again. On the other hand, if we take as anti-operation the current value of the counter, we have an absorbing monoid for the same data structure.

Free composition We say that composition is *free* when the monoid of anti-operations is exactly (isomorphic to) a list of simple anti-operations with concatenation for composition. A sequence of free anti-operations cannot be compressed/summarized in any particular way and is recorded as-is.

Consider a storable type of monotone sets, whose only operations are of the form `Add(v)` – we can only add, never delete elements. A natural choice of composable anti-operations is to consider lists of simple anti-operations of the form `Delete(v)`, with composition defined as concatenation of lists. This is a free monoid of anti-operations. In fact the ordering of deletions may not matter, so we could store *sets* of values instead of lists, which is non-free again. But there would be no particular performance benefit in doing so.

Custom composition We say that composition of anti-operations is *custom* when it is neither absorbing nor free. For example, if instead of monotone sets (only additions) we consider sets with both addition `Add(v)` and deletion `Delete(v)`. We could choose as (composable) anti-operations just lists of operations, so the anti-operation for the operation sequence `(Add(v1); Add(v2); Delete(v3))` would be the list `[Add(v3); Delete(v2); Delete(v1)]`. But sometimes two operations cancel out: for example, the anti-operation for the sequence `(Add(v); Delete(v))` can be just the empty list. To implement this efficiently, we would define our anti-operations as partial maps from values to either `Added` or `Deleted`. This monoid of anti-operations is neither absorbing nor free, it is *custom*.

1.5 Different interfaces for different properties

The key idea of our implementation of storable types in Store is that these three properties correspond to three important classes of storable types, that each deserve specific support. We offer:

- A simple API for *free* anti-operations, which logs a separate anti-operation for each operation performed. It is easy to use, but using it for absorbing or custom anti-operations could lead to worse asymptotics in memory usage and backtracking time.
- A simple API for *absorbing* operations, which is efficient but not general: it can only be used for storable types whose anti-operations are absorbing.
- A complex API for *custom* anti-operations, that lets user implement arbitrary non-free composition of anti-operations. This is the most general interface and it could be used for any storable type.

Contributions We consider the contribution of our work to be the separation of storable types in these three classes (absorbing, free, custom), the design of an API for each class that offers an adequate balance between complexity and efficiency, and the implementation of the corresponding logic in the Store library.

We implemented representative data structures in each of these classes, and the final design results from several iterations on those implementations. We report on these implementations, which demonstrate that the APIs are usable and expressive. We do not make any claim regarding efficiency in this article – on the overhead of implementing a data structure with one of those generic interfaces instead of built-in support – and will not provide benchmarks as part of our evaluation, thank you very much.[1]

[1] The joke is that running good benchmarks is *brutally hard*. Last time we decided to benchmark our Store implementation, it took us *weeks* of full-time work to converge to a result that we believe was a correct experimental validation of our qualitative performance claims. The amount of effort required for our benchmarks was comparable to the effort of writing a mechanized Coq/Rocq proof of correctness for the persistent part of Store.

Our implementation vehicle is an experimental version of Store that is available as open-source software, and will hopefully be integrated shortly as a new Store release. It is available at `https://gitlab.com/basile.clement/store/-/tree/jfla2025`.

2 A Store reminder

Allain, Clément, Moine, and Scherer (2024) provide a complete description of the design and implementation of the `Store` library (without storable types), with a partial formal verification and detailed benchmarks. Reading this previous work is not strictly necessary to follow the present article, as we include a self-contained reminder – but if anything is unclear below, please have a look at that paper.

The current article is focused on storable types, and in particular our Related Work discussion focuses on works that are relevant for storable types, not other works that are related to Store in more general ways. For a discussion of those, again, please have a look at Allain, Clément, Moine, and Scherer (2024).

2.1 Interface

The Store library defines a type `store` of stores.

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

In the Store library as described in Allain, Clément, Moine, and Scherer (2024), the only data structure that can be put in a store are references:

```
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
```

The backtracking support of Store is offered through two different APIs. The *persistent* API offers immutable *snapshots* that can be restored an arbitrary number of times: `capture store` takes a snapshot of the current value of all structures in the store, and `restore store snapshot` reverts the store to the state it had when the snapshot was taken.

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

The *semi-persistent* API offers ephemeral *transactions*, that can be terminated at most once and follow a last-in-first-out discipline: terminating a transaction invalidates all transactions and snapshots created within that transaction. A transaction can be terminated by `rollback`, which reverts the store to the state it had when the transaction started, or by `commit`, which leaves the store state unchanged.

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

Terminating a transaction using `rollback` or `commit` invalidates all transactions or snapshots which were created within the transaction. On the other hand, it remains possible to restore a previous snapshot, or to terminate an enclosing transaction.

Finally, we provide two high-level wrappers:

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

`temporarily st f` takes a store `st` and a function `f` that performs an arbitrary computation, calls `f ()`, and then reverts any change to the store that would have been performed by `f`. `tentatively` reverts the changes only if `f` raises an exception, and persists the changes if it returns a value.

Note that this gives a different programming style than the semi-persistent APIs as originally presented by Conchon and Filliâtre (2008). In our API, users explicitly capture and restore transactions or snapshots – unless their uses are simple enough that the high-level wrappers suffice. The API of Conchon and Filliâtre (2008) gives a programming style that is closer to programming with pure, persistent data structures, with no explicit manipulation of snapshot/transactions; but it is also less expressive, for example one cannot easily expose `commit` in this model.

2.2 Examples

2.2.1 Unification with rollback on error

Suppose we have a type-checker that uses our union-find from Section 1.1, with the type `ty node` used to represent a type, which is also a node in a Union-Find graph. We have an implementation of the function

```
exception Unification
val unify_head :
  Store.t -> (ty node * ty node) Queue.t ->
  ty node -> ty node -> unit
```

which unifies two type nodes, raises an exception if they are incompatible, and pushes the children of the nodes into a queue of pairs to be unified later. For example, unifying `X -> Y` and `X' -> int` pushes the pairs `(X, X')` and `(Y, int)` into the unification queue.

We now want to wrap this in a higher-level function that unifies two types in depth, recursively unifying its children. If some unification fails along the way, we want all unifications performed so far to be rolled back: we will print an error message to the user that shows the types involved, and we do not want the types shown to include some partial unification information, depending on the traversal order of the unifier. This can be done by capturing the state of the store before the unifications start, and restoring this initial state on error.

```
let unify st ty1 ty2 =
  let snapshot = Store.capture st in
  try
    let q = Queue.create () in
    Queue.push (ty1, ty2) q;
    while not (Queue.is_empty q) do
      let (v1, v2) = Queue.take q in
      unify_head st q v1 v2
    done
  with Unification as err ->
    Store.restore st snapshot;
    raise err
```

The snapshot `snapshot` is only restored at most once in this example, so it fits the constraints of the semi-persistent API, which would make the code slightly more efficient:

```
let unify st ty1 ty2 =
  let transaction = Store.transaction st in
  try
    let q = Queue.create () in
    Queue.push (ty1, ty2) q;
    while not (Queue.is_empty q) do
      let (v1, v2) = Queue.take q in
      unify_head st q v1 v2
    done;
    Store.commit st transaction;
  with Unification_error as err ->
    Store.rollback st transaction;
    raise err
```

(Slightly more efficient: in theory there can be arbitrary savings in time and memory usage, but in practice in most cases we would not observe a performance difference. In some use-cases we observed that the semi-persistent version is about 20% faster. It also has better memory-usage properties: in technical terms that will only be clear later, `commit` compresses anti-operations by increasing the opportunities for record elision.)

This use-case of rolling back a transaction on error is common enough that Store proposes a high-level wrapper for this, `Store.tentatively`:

```
let unify st ty1 ty2 =
  Store.tentatively st (fun () ->
    let q = Queue.create () in
    Queue.push (ty1, ty2) q;
    while not (Queue.is_empty q) do
      let (v1, v2) = Queue.take q in
      unify_head st q v1 v2
    done
  )
```

2.2.2 A persistent union-find

One of the type-checkers that we implemented in this style really wants to see the type-checking environment as a persistent data structure: it is used inside a non-determinism monad that runs many different variants of the same type-checking computation, reusing the same solver state many times.

Suppose we want to provide the following interface, which exposes a persistent "equation environment" which records equations between types and can then check whether two types are equal:

```
module Env : sig
  type t (* a persistent typing environment *)

  val equal : t -> ty node -> ty node -> bool
  (* are two types equal in the given environment? *)

  val unify : t -> ty node -> ty node -> t
  (* add an equation between two types,
     resulting in a new equation environment *)
end
```

This can be implemented by storing the store *and* a snapshot of the store in the type `t` representing the persistent environment:

```
type t = {
  store: Store.t;
  snapshot: Store.snapshot;
}
```

Manipulating types in a given environment is done by first restoring the snapshot of the environment:

```
let equal env ty1 ty2 =
  Store.restore env.store env.snapshot;
  UnionFind.equal env.store ty1 ty2
```

Unifying two types is followed by taking a new snapshot, to return a semantically distinct type environment:

```
let unify env ty1 ty2 =
  Store.restore env.store env.snapshot;
  unify env.store ty1 ty2;
  { env with store = Store.capture env.store }
```

Here we really need to use the persistent API, as the user may alternate between different equation environments. It may seem inefficient to call `Store.restore` so often, but in fact this is efficient in practice as `Store.restore` is a no-op when the store is already in the desired state. This also illustrates that `Store` allows switching from its more complex API to a simpler persistent (or semi-persistent) API at an arbitrary abstraction level. The `env` type above provides a fully persistent interface while still benefitting from record elision in the implementation of `unify`.

2.3 Implementation

The implementation of Store maintains a tree-shaped directed graph, where each node denotes a version of the store, and edges between nodes carry information on how to mutate the store state to move from one version to another. We call this graph the *store graph*. It has a designated root node, which corresponds to the current state. Operations that need to record a change create a new node, create an edge between the current root and this new node that carries an anti-operation, and make that new node the new root of the store.

The store graph has the shape of a *tree*, there is a unique path to the root from any node. Nodes of the graph are represented directly as OCaml references, whose content store their parent node in the graph and the information on the edge between the two nodes. Each store reference is a mutable record that stores its current value (just like normal OCaml references); only older versions are stored in edges of the store graph.

Taking a snapshot is $O(1)$, we just return a reference to the current root node. Operations on the mutable structures are also $O(1)$, and in particular reading a stored reference has exactly the same implementation and performance as reading a standard, non-stored reference. Restoring a previous state of the store traverses all edges between the captured node and the current root, updating the stored references and the store state in the process, by undoing each recorded operation. The complexity is linear in the number of operations recorded between the two versions. Moving between states using the persistent API requires some extra book-keeeping compared to the semi-persistent version (still $O(1)$ per operation), as we revert and update the edges to allow going back later.

Parsimony Store has a *parsimonious* implementation: the OCaml value that denotes a snapshot only refers to anti-operations that happened *after* the snapshot was taken, not before. In particular, if a snapshot becomes unreachable as an OCaml value, then all anti-operations that are only reachable from this snapshot can be collected by the garbage collector. This would not be the case if we used, for example, a single global dynamic array to store all anti-operations (which was the implementation technique in Noûs and Scherer

(2023)); then holding any snapshot or just the store would force all anti-operations to remain alive, retaining more memory than the user expects.

2.4 Record elision

Store implements a key optimization we call *record elision*. When a reference has already been updated since the last snapshot was taken, we have the anti-operation `Unset(v)` somewhere on the path from the current node to the last snapshot. There is no need to record further updates to the same reference until the next snapshot, as the existing anti-operation is sufficient for the reference to be correctly restored to `v`. This implicitly relies on the fact that references have *absorbing* anti-operations.

To take advantage of this fact, we track *generations*, which are integers such that:

- Each snapshot in history has a distinct generation.
- The current root carries a generation, larger than the generation of all snapshots in history, which is stored in the store metadata.
- Each stored reference carries a generation, which is the generation that the store had on the last time the reference was updated.

When a referenced is updated, the implementation compares its generation with the store generation. If the reference has an older generation, it means that the last update to the reference took place before the last snapshot was taken. In this case, Store records an anti-operation for this update in the store graph. But if the reference has the same generation as the store, it means that the store was modified after the last snapshot was taken, and Store does not need to record another anti-operation — it *elides* this extra work.

Record elision has a transformative impact on performance. On the workflows we are interested in, backtracking is a relatively infrequent operation compared to operations on the stored data structures. In this regime, only the first update of each reference after a snapshot incurs some versioning overhead, and the many following updates have the same performance as writing to a non-stored reference. In practice, using Store in this regime has no noticeable overhead over using global mutable state that does not support backtracking.

Record elision is a key design question when generalizing Store from (absorbing) references to other storable types. To what extent can we perform record elision for those operations?

Elision on `commit` There is another situation where we perform a less-direct form of runtime elision. When we `commit` a transaction, we must keep the anti-operations logged during the transaction somewhere in the store graph to allow going back to an earlier snapshot. We implement this by, morally, re-recording each operation in turn, in the context of the *previous* snapshot — as the snapshot corresponding to the transaction is invalidated by the commit. In particular, some anti-operations that had to be recorded when the transaction was performed (there was no previous record in the current transaction) can now be elided (there is a previous record in the outer snapshot or transaction), and this performs a form of compression of the history.

For example, consider the following sequence of operations:

```
let snap1 = Store.capture s in
Store.Ref.set s r v1;
let snap2 = Store.transaction s in
Store.Ref.set s r v2;
Store.commit s snap2;
```

The second call to `Store.Ref.set` is the first `set` on `r` since the last snapshot (the transaction `snap2`) was taken, so it creates a node in the graph — it is not elided. But when we decide to `commit` this transaction `snap2`, the snapshot `snap2` is removed from the

graph, so the changes since `snap2` now have `snap1` as their latest snapshot. In particular, the capture of `r` corresponding to the second `set` is not necessary anymore, as the value of `r` at `snap1` was already recorded on the first set. This second write is removed from the graph during `commit`, using a comparison of generations.

3 Our interfaces: free, absorbing, custom.

3.1 Storing free operations

We offer a module `Store.Free` to implement storable types with free operations.

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

When they want to perform a free operation on data structure `data` at a storable type, the user should call `Free.modify store data ~op ~anti`, where `op` is a function that performs the operation and `anti` is a function that reverts the operation. (We guarantee that `anti` is only called in the state produced by the `op` function.). The triple `(data, op, anti)` is be stored as a new edge in the graph, ready to be reverted or re-applied in the future.

Example: `RareRef` As an example, we can implement `RareRef`, a type of references that are updated rarely, as follows:

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

This module is observationally equivalent to the built-in module `Store.Ref` of stored references, but the performance tradeoffs are fairly different. `RareRef` does not perform record elision, so it is noticeably slower than `Store.Ref` if a reference is modified a lot between two snapshots. But on the other hand `RareRef.t` does not need to carry a generation for elision, so it consumes less space than `Store.Ref`, and can be a good choice for certain read-dominated workflows.

3.2 Storing absorbing operations

We offer a module `Store.Absorbing` to implement storable types with absorbing operations.

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

To implement an absorbing operation, the user must provide the following functions as part of the "description" record:

- A `capture` function that saves the current state of the storable structure as an anti-operation.

- A `rollback` function that consumes the anti-operation to revert to the previous state; this is used by the semi-persistent API.
- an `undo` function that consumes the anti-operation, reverts to the previous state, but produce an operation in `'op` along the way, that can be used to come back to the captured state by calling `redo`; this is used by the persistent API.

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

`Absorbing.make descr v` creates an instance of an absorbing structure whose current value is `v`. `Absorbing.get` can be used to read the current value of the structure, and `Absorbing.modify` must be called before any modification to the structure — unlike `Free.modify`, it does not perform the modification itself, the user is in charge of the modification. The `modify` function may capture the current state using the `capture` callback, but it can perform record elision if that storable value was already modified since the last snapshot.

Internally the abstract type `Absorbing.t` looks like a reference, with a generation and an extra field storing the description of the absorbing operations. (We use GADT syntax to hide the internal type parameters `'op`, `'anti`.)

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

Note that the `value` field is not marked mutable: the `'a` itself is usually a mutable type, for example `'a Dynarray.t` in the example that follows, and the user updates it directly. Providing API support for updating the `'a` value wholesale would mostly be useful for stored references, and those already have a dedicated implementation.

Example: push-only queue The `PushQueue` module implements a queue in which only pushing is allowed. Unlike queues that can be both pushed and popped, push-only queue are absorbing, as their current state can be captured in a future-proof way by remembering the current length of the queue.

```
module PushQueue = struct
  type 'a t = ('a Dynarray.t, 'a array, int) Absorbing.t
```

The state of a `PushQueue` is a dynamic array. Our type of operations is `'a array`, representing a sequence of values to push to the queue. Our type of anti-operations is just `int`, the current length of the array. A previous state can be restored by just truncating the dynarray to the previous length (the backing array is not shrunk); in `undo` we first save the suffix of the dynarray, all elements after this previous length, for later re-application.

```
  let capture = Dynarray.length
  let rollback = Dynarray.truncate

  let undo arr n =
    let len = Dynarray.length arr - n in
    let suffix = Array.init len (fun i -> Dynarray.get arr (n + i)) in
    Dynarray.truncate arr n;
    suffix
```

```
  let redo arr suffix =
    let len = Dynarray.length arr in
    Dynarray.append_array arr suffix;
    len

  let descr = { capture; rollback; undo; redo; }
```

Once we have defined how to apply operations and anti-operations, we define useful functions on our queues, as a thin layer over `Dynarray`:

```
  let create _s =
    Absorbing.make descr (Dynarray.create ())

  let push s d v =
    Absorbing.modify s d;
    Dynarray.add_last (Absorbing.get d) v

  let length _s d =
    Dynarray.length (Absorbing.get d)

  let get _s d i =
    Dynarray.get (Absorbing.get d) i
end
```

Notice how `push` calls `Absorbing.modify`, which saves the current state of the queue in the store graph, except when record elision is possible. After calling `Absorbing.modify`, the user modifies the dynamic array themselves.

3.3 Storing custom operations

The `Custom` interface is similar to the `Absorbing` interface, but the user needs to implement composition of anti-operations on their own.

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

A `'trail` is a user-selected mutable type whose state represents a composition of anti-operations. The `record` operation starts a new trail from the current state. The user has to mutate the trail on each operation — `Store` has no callback to extend the trail with a single anti-operation. But `Store` needs to be able to concatenate two trails, composing their anti-operations together, this is the `append` function: `append trail1 trail2` consumes `trail2` and transfers its anti-operations into `trail1`. `undo` and `redo` are as in the `Absorbing` module, with a subtle difference: `redo` takes the trail as an argument instead of returning a new trail. `Store` only replays operations with `redo` that were created through `undo`, and for a given operation, always passes in the same (physical) trail to `redo` — it is up to the data structure implementer whether to leave the trail untouched in `undo`, or to clear it and record the appropriate anti-operations in `redo`.

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

The rest of the module is similar to `absorbing`, but the `modify` operation returns a trail. For absorbing types, the Store logic would take care of recording every anti-operation, but for custom types this is the responsibility of the user, who does it by mutating the trail returned by `modify`.

Internally the implementation of `Custom.t` is similar to `Absorbing.t`, with an extra field to store the trail. In this case only `'op` is hidden as an internal detail, as the `'trail` is exposed in the user-facing API.

```
type (_, _) t = Custom : {
  value : 'a;
  mutable generation : Generation.t;
  descr : ('a, 'op, 'trail) custom_descr;
  mutable trail : 'trail
} -> ('a, 'trail) t
```

Example: push-pop queue Previously we noticed that removing the `pop` operation from a queue makes the structure absorbing. If we add `pop`, we get a custom storable type.

```
module Queue = struct
  type 'a op = { truncate: int; suffix : 'a Dynarray.t }
  type 'a trail = { mutable truncate: int; mutable suffix : 'a list }
```

Both operations and anti-operations truncate the queue to a certain position and push a new suffix of elements. Anti-operations accumulate elements to push one by one, so they use a list to represent the suffix: when the user pops the queue, the trail is updated to push an extra element at the *beginning* of the suffix, so the elements in the suffix list are in-order. Operations are built in one go by `undo`, so they can use an (immutable) array instead of a list.

The implementation of those callbacks is rather similar to `PushQueue`:

```
  let record arr = { truncate = Dynarray.length arr; suffix = [] }
  let append trail1 trail2 = [...]

  let rollback arr trail =
    Dynarray.truncate arr trail.truncate;
    Dynarray.append_list arr trail.suffix
```

Note that in `undo` we leave the trail untouched, so that we do not have to recreate it in `redo`.

```
  let undo arr trail =
    let suffix =
      Dynarray.init (Dynarray.length arr - trail.truncate)
        (fun i -> Dynarray.get arr (trail.truncate + i)) in
    rollback arr trail;
    { truncate = trail.truncate; suffix; }

  let redo arr op =
    Dynarray.truncate arr op.truncate;
    Dynarray.append arr op.suffix

  let descr = { record; append; rollback; undo; redo; }
```

We will not repeat all `Queue` operations here, but focus on `push` and `pop` that update the trail to log an anti-operation.

```
let push s d v =
  let _trail = Custom.modify s d in
  Dynarray.add_last (Custom.get d) v
```

The `push` function is identical to the previous, absorbing version. No trail update is necessary, because the rollback logic starts by truncating the dynarray, ignoring all later pushes.
On the other hand, `pop` may need to update the trail. The `truncate` length represents the end of the dynarray prefix that was unchanged by the sequence of `push` and `pop` operations accumulated since the last `capture`; in other words, it is the low-water mark of changes at the end of the queue. If a `pop` occurs strictly above that limit, it does not touch the unchanged prefix, and no logging is necessary. If it happens at the limit, we need to decrement the limit, and to record the popped element in the `suffix` part to be able to restore it later.

```
let pop s d =
  let trail = Custom.modify s d in
  let arr = Custom.get d in
  let len = Dynarray.length arr in
  let last = Dynarray.pop_last arr in
  if len > trail.truncate then ()
  else begin
    trail.truncate <- len - 1;
    trail.suffix <- last :: trail.suffix;
  end;
  last
end
```

Commit compression As we recalled in Section 2.4, record elision happens when the user performs an operation, but also on `commit`, where it is used to "compress" the changes that have been recorded during the transaction.

We want to preserve this phenomenon in our generic custom interface, and it is the purpose of the `append : 'trail -> 'trail -> unit` callback. On `commit`, if there exists a trail for the operation before the committed transaction (and after the last snapshot) and a trail after the committed transaction, then they are merged together by calling `append`.

To implement this, when we create a new trail for a custom data structure (and record it in the store graph), we keep a pointer to its previous trail, that records the operations performed before the last transaction or snapshot. We did not need to do this for built-in references, or in general for absorbing structures, where `commit` compression can be implemented simply by discarding certain anti-operations in the transaction with a generation check; it does not require looking at what was recorded before the transaction. For custom operations, compression depends on the anti-operations recorded before, so we need to track them.

In our example of add-remove set, anti-operations are a map from set elements to `Added | Removed`, and commit compression is a merge of the two maps where elements present in both maps are omitted in the merged map. Indeed, if an element occurs in both maps, it has been added (just before the transaction) and then removed (during the transaction), or removed and then re-added. In both cases this is a no-op, and we can remove this key from the single map resulting from `commit`, reducing memory usage and speeding up future operations.

Keeping a pointer to a trail from before the last snapshot weakens our parsimony property that only later anti-operations are reachable from the OCaml representation of a snapshot or transaction. Specifically to provide `commit` compression on custom data structures, we

moved to a less parsimonious implementation. We believe that this is acceptable in practice. Two remarks:

- The user is in control of the `'trail` type, so they can choose the tradeoff between performance, implementation complexity, and parsimony. For example, it is possible to store anti-operations in a mutable linked list, with the `'trail` type being a reference to a last, empty cell at the end of the list — this is a complicated way to reimplement the `Free` interface in terms of the `Custom` API. This implementation choice preserves the strong parsimony of Store, while removing the ability to compress anti-operations. In general the user wants to keep more data reachable from the trail, enough to enable effective compression, but they have control over it.
- We could use a weak pointer to track this older trail, recovering strong parsimony properties. But so far we were happy to not need weak pointers at all, which are an advanced OCaml runtime feature that is harder to reason about and less portable to alternative implementations.

3.4 Three implementations

In our new version of Store with storable types, we have a built-in implementation for each of the three interface modules described above: `Free`, `Absorbing`, `Custom`. That is, Store adds four kinds of built-in edges, one for `Free`, one for `Absorbing` (that can store either operations or anti-operations), and two for `Custom`, one for anti-operations and one for operations (they have a different shape).

This is a counter-intuitive choice. Our first inclination would be to only implement built-in support for the most general, expressive interface (here `Custom`), and implement the other interfaces on top of it. However, our current judgment is that having three separate implementations is better in this case, for two reasons:

1. This provides better constant factors. For example, a `Custom.t` value stored in the graph is a record of four fields: the value, the description record, the generation, and the trail. An `Absorbing.t` has only three fields, as it does not keep a trail. `Absorbing` also has better liveness properties than `Custom`, as it does not keep backward pointers for compression of `commit` operations.

   Performance-wise, the use of a more specific API can provide benefits: re-implementing the original `Store` reference using the `Absorbing` API is roughly 20% slower (mostly due to the overhead of indirect calls), while re-implementing it using the `Custom` API is roughly 40% slower. A `Free` implementation would have entirely different performance characteristics due to the absence of record elision.

2. Because `Free` and `Absorbing` are much simpler to implement than `Custom`, the extra code is noticeably shorter than the generic implementation, keeping the maintenance cost acceptable. In the case of `Free`, implementing it on top of `Custom` would require more code duplication than adding built-in support.

4 Related Work

Several automated solvers (SMT solvers or constraint solvers) implement generic interfaces to record changes to imperative data structures and backtrack them. We did not find in the literature a clear discussion of the performance-vs-expressivity tradeoffs involved in designing those interfaces. The only clear mention we found of this design choice is in the tool description paper for the CVC5 solver: Barbosa, Barrett, Brain, Kremer, Lachnitt, Mann, Mohamed, Mohamed, Niemetz, Nötzli, Ozdemir, Preiner, Reynolds, Sheng, Tinelli, and Zohar (2022), “Context-Dependent Data Structures” (Section 2.4). One important

aspect to note is that those implementations only support a semi-persistent interface, not a persistent interface where snapshots are persistent and can be resumed several times. Supporting persistence (while preserving record elision) is a distinguishing feature of Store. This affects the storable type interface, as semi-persistent-only structures do not attempt to store operations, only anti-operations.

We looked at the implementations themselves to understand what is used in practice. Here is what we found:

choco the Choco constraint solver implements a trail that can store references to various builtin types (bool, int, double), and also a generic `IOperation` class that only provides an `undo()` method for undoing some changes — this corresponds to the free interface. For example, its backtracking sets log an anti-operation[2] on each update.

z3 does not seem to provide a generic interface for backtrackable objects, its trail hard-codes a set of builtin types[3], that are generally implemented by storing an anti-operation for each update separately on the trail — the free interface again.

cvc5 supports context-dependent maybe/option values, append-only lists, dequeues, insert-only hash sets, and hash maps. Its generic interface for context-dependent objects[4] resembles our custom interface, if we think of the `ContextObj` class as representing a trail. The cvc5 authors appear to have found complex storable types difficult to implement[5], and their main strategy is to focus on append-only structures that are easier to backtrack.

Acknowledgments The design discussions for storable types in Store consumed most of the free time of Basile Clément and Gabriel Scherer during the JFLA 2024; they almost missed their return trains to continue iterating on ever-more-complex APIs. The bulk of the implementation work after that was done by Basile Clément in September 2024, with many iterations, only some of which had acceptable complexity — if Basile does not know how to write the code, then for sure the design needs to be rethought. The bulk of the writing work for the paper was done by Gabriel Scherer. François Pottier provided illuminating review comments as always, and we also thank our anonymous JFLA reviewers for their excellent feedback.

References

Clément Allain, Basile Clément, Alexandre Moine, and Gabriel Scherer. Snapshottable stores. In *ICFP*, 2024.

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. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 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*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[2] `https://github.com/chocoteam/choco-solver/blob/758489f2dcf6e9da0fbc3667552f2de4c59277b8/solver/src/main/java/org/chocosolver/util/objects/setDataStructures/StdSet.java#L75`

[3] `https://github.com/Z3Prover/z3/blob/2880ea39/src/util/trail.h`

[4] `https://github.com/cvc5/cvc5/blob/92caabc77cf8b347cadf1517c75b2af2357d419c/src/context/context.h#L340-L380`

[5] `https://github.com/cvc5/cvc5/blob/0570e78fb972e53e0b39cb29718f2a329bff613a/src/context/cdhashmap.h#L27-L30`

Guillaume Bury, Basile Clément, Albin Coquereau, Sylvain Conchon, Evelyne Contejean, Steven de Olivera, Hichem Rami Ait El Hara, Mohamed Iguernlala, Stephane Lescuyer, Alain Mebsout, Mattias Roux, and Pierre Villemot. the alt-ergo smt solver, 2015.

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

Camille Noûs and Gabriel Scherer. Backtracking reference store. In *JFLA*, January 2023.

François Pottier, Olivier Martinot, and Gabriel Scherer. the inferno type-inference library, 2023.