Deductive Verification of Sparse Sets in Why3

Catherine Dubois

▶ **To cite this version:**

Catherine Dubois. Deductive Verification of Sparse Sets in Why3. VSTTE 2024 - 16th International Conference on Verified Software: Theories, Tools, and Experiments, Azalea Raad; Jonathan Protzenko, Oct 2024, Prague, Czech Republic. ⟨hal-04863059⟩

HAL Id: hal-04863059

https://hal.science/hal-04863059v1

Submitted on 3 Jan 2025

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

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

HAL Authorization

Deductive Verification of Sparse Sets in Why3

Catherine Dubois[0000−0002−9477−8109]

ENSIIE, INRIA, Université Paris-Saclay, LMF, France
catherine.dubois@ensiie.fr

Abstract. To represent finite sets of integers on an interval 0..n, Briggs and Torczon studied a very simple data structure in 1993, called *sparse sets*. With this representation, initialization, membership test, insertion and deletion of an element are O(1) operations. This data structure is often used in compilers to allocate registers or to represent the objects in a video game. A variant of this data structure is also used in finite domain constraint solvers to represent the domains of integer variables. This variant makes it a backtrackable data structure. We have formalized and verified the original data structure and its variant in Why3. Set operations such as intersection and union are formally verified, even though they are less commonly used with this representation of sets. To our knowledge this is the first formal verification of the backtrackable variant of sparse sets used as domains.

1 Introduction

Sets are seldom primitive objects in programming languages or specification formalisms. There are such objects in the old programming language Setl [18] or the logic programming language {log} [8] and also in the formal languages B [1] or Event-B [2] and TLA+ [12]. More usually, they are available as implementations in libraries, based on underlying data structures such as sorted lists, red-black trees, AVL trees, B trees, skiplists, etc. In this paper, we focus on sparse sets, studied by Briggs and Torczon [4] in 1993, also appearing as an exercise in the famous book "The Design and Analysis of Computer Algorithms" written by Aho and Hopcroft [3]. This data structure dates back to computer folklore, it is used in different applications like register allocation, video game, constraint solving. With this mutable representation based on arrays and simple manipulations, initialization, membership test, insertion and deletion of an element are O(1) operations. Many implementations exist on the web, in several languages, e.g. Java, C++, C, Rust.

Sparse sets appear as a benchmark (Constant-time sparse array) of VACID-0, a suite of benchmark verification problems proposed in 2010 [15]. The sparse sets data structure as described by Briggs and Torczon is a particular case of the latter in which there is one less indirection. A solution[1] where 3 operations

[1] available at https://toccata.gitlabpages.inria.fr/toccata/gallery/vacid_0_sparse_array.en.html

(membership, add and remove) are implemented, has been given using Why3 by Filliâtre and Paskevich.

Sparse sets are also used in constraint solvers as an alternative to range sequences or bit vectors for implementing domains of integer variables [13] which are nothing else than mathematical finite sets of integers. Sparse sets as domains are slightly different from sparse sets introduced by Briggs and Torczon making them very easy to store and restore when backtracking for finding solutions.

Our main contribution is a formally verified implementation of sparse sets as domains and its various operations, developed with the deductive verification tool Why3 [10], extracted in OCaml. In addition to classical set operations (test membership, remove, etc.), we specify and verify an operation that allows the user to undo some operations very easily (in one simple assignment). This contribution brings some more confidence in the data structures used in constraint solvers, as it has been done by Ledein and Dubois [14] for the traditional implementation of domains as range sequences.

In [7], Cristiá and the author have formalized this sparse set as domain variant and verified three simple operations (remove, bind and membership) in three formalisms, Why3, EventB and {log}. However they do not address the verification of the backtrackable dimension and in particular the undo operation.

The article is structured as follows. In Section 2 we give an informal overview of sparse sets. In Section 3, we briefly introduce Why3 and WhyML. In Section 4, we detail our WhyML implementation of sparse sets following Briggs and Torczon and discuss its deductive verification. In Section 5, we introduce the modifications to the previous data structure when it is used to represent the domain of integer variables in constraint solvers. Then, in Section 6, we present the WhyML formalization of this backtrakable variant by focusing mainly on the additional artefacts we used to verify the undo operation. Section 7 presents some experimentations performed on the OCaml code extracted from our models. Finally we conclude and present some future work.

All the code described in this paper is available on `https://gitlab.com/cdubois/why3_sparsesets`.

2 Sparse sets

Sparse sets are used to represent subsets of natural numbers up to $N-1$, where N is any non-zero natural number. The range $0..N-1$ is called the universe of the sparse set in the following. A sparse set D is represented by two arrays of length N called $Dense$ and $Sparse$, and a natural number $sizeD$[2]. The current elements of the finite set are those in $Dense[0, sizeD-1]$ — let us call this subarray the effective part —, the rest of the array being *garbage*. The array $Sparse$ maps any value $v \in [0, N-1]$ to an index ind_v in $Dense$ or is not initialized. Thus, for the current elements v of the set, $Sparse[v]$ has a value i

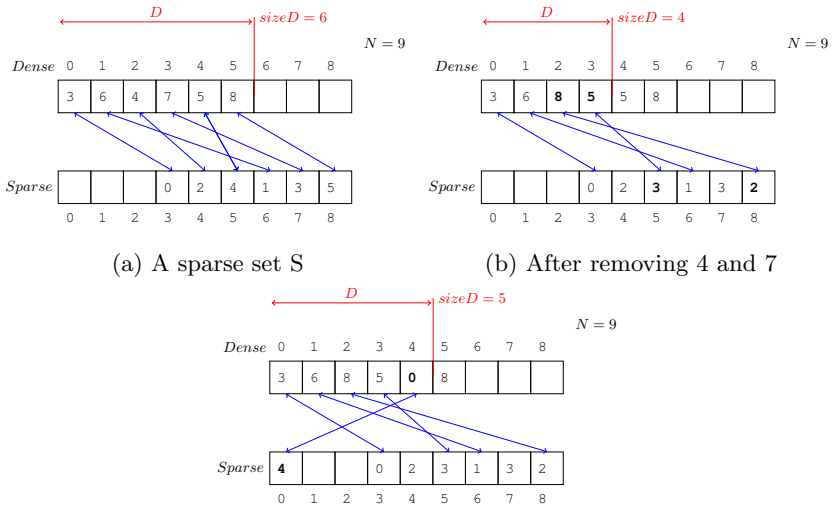
[2] The name of this data structure may be explained by the fact that the $Sparse$ array may have holes whereas the $Dense$ array is more compact.

in the range $[0, sizeD - 1]$ and $Dense[i]$ is equal to v. If D is empty (resp. the full set), $sizeD$ is equal to 0 (resp. N).

The two invariants of the data structure representing the set D are as follows:

$$D \subseteq 0..N-1 \wedge D = \{Dense[i] \mid 0 \leq i < sizeD\} \quad (P_1)$$
$$v \in D \iff 0 \leq Sparse[v] < sizeD \wedge Dense[Sparse[v]] = v \quad (P_2)$$

Fig. 1a illustrates this representation. This state has been reached after inserting the elements 3, 6, 4, 7, 5 and 8 in the empty set. The blue arrows emphasize the invariant P_2.



(a) A sparse set S

(b) After removing 4 and 7

(c) After adding 0

Fig. 1: Three States of a Sparse Set

Checking if an element v belongs to the sparse set D simply consists in the evaluation of the expression $0 \leq Sparse[v] < sizeD$ && $Dense[Sparse[v]] = v$. Removing an element consists in replacing v in $Dense$ with the last element e of the $Dense$ effective part ($e = Dense[sizeD - 1]$), decrementing $sizeD$ and updating $Sparse[e]$. This operation is illustrated in Fig. 1b: 4 and 7 are removed in this order from the sparse set represented in Fig. 1a. We can see two occurrences of both 8 and 5 but their presence in $Dense[sizeD..N]$ does not matter.

Inserting an element v is implemented as follows: put v in $Dense$ at the position $sizeD$, update $Sparse[v]$ with $sizeD$ and increment $sizeD$. In Fig. 1c, 0 has been inserted in the sparse set represented in Fig. 1b.

Clearing the sparse set, that is making it represent the empty set, is very efficient, just set $sizeD$ to 0. The cardinality of a sparse set is exactly the value

of $sizeD$. All the previous operations are constant-time. Operations like forall, exists, union, intersection, equality only require to explore the elements in the effective part of $Dense$, and are thus in $O(sizeD)$.

3 Why3 and WhyML

Why3 [10] is a platform for deductive program verification that provides a specification and programming language called WhyML. It relies on external automated and interactive theorem provers to discharge automatically generated verification conditions (VC). The SMT provers Alt-ergo, CVC4 and Z3 are used here. Transformations, aka tactics, are also provided, making Why3 an interactive proof environment. Why3 supports modular verification and includes some mechanisms for managing modularity, abstraction and genericity [11].

WhyML allows the user to write functional or imperative programs featuring polymorphism, algebraic data types, pattern-matching, exceptions, mutable variables, arrays, etc. These programs can be specified by using contracts (pre- and post- conditions) and assertions (e.g. variants, loop invariants). User-defined types with invariants can be introduced, invariants are verified at the function call boundaries. Furthermore to prevent logical inconsistencies, Why3 generates a verification condition to ensure that such a type is inhabited. To help the verification, a witness can be explicitly given by the user (`by` clause in Fig. 3). The `old` operator can be used inside post-conditions to refer to the value of a term at the call program point.

In addition, as with other verification tools, ghost code can be used to annotate the source code in order to to make it easier to verify. Ghost code is regular WhyML code, except that ghost variables or record fields are introduced using the keyword `ghost`.

Correct-by-construction OCaml (and, more recently, C) programs can be automatically extracted from verified WhyML programs. More detail is provided throughout the paper as necessary.

4 Formal Verification of Sparse Sets

This section deals with the data structure as it is described in Briggs and Torczon's paper [4]. We provide a WhyML specification and an implementation of the data structure and its operations.

4.1 Abstract Specification

We start with a high-level module that contains the abstract specification of type `t` and operations on that type, where `t` is the type of subsets of an interval of natural numbers (beginning at 0 as in [4]). Fig. 2 contains an excerpt of that module. The type `t` here is specified as a record with two fields only used for specification: the size `n` of the support universe and `setD` which has to be

understood as the high-level model of the data structure. The `fset` logical type constructor is defined in the module `set.FsetInt` of the standard library. Set mathematical symbols that appear in the contracts are used here and in the rest of the paper to denote the mathematical set operations acting on mathematical sets also defined in the module `set.FsetInt`. The == infix operator is the mathematical set equality. The `writes` clause in a contract indicates that the corresponding function updates its argument. The operations are implemented in the refining module that also provides a full definition for the type. We describe this refining module in the next subsection.

```
module FiniteNatSet
use int.Int
use set.FsetInt

type t  = abstract {n : int ;
                    mutable setD : fset int ; }
invariant {setD ⊆ (interval 0 n)}

val empty_set (nn : int) : t
requires {0<=nn}
ensures {result.setD = ∅}
ensures {result.n = nn}

val member (v : int) (a : t) : bool
requires {0<=v}
ensures {result = v ∈ a.setD}

val cardinal_sparse (a : t) : int
ensures {result = |a.setD|}

val add (v : int) (a : t) : unit
requires {0<=v<a.n}
ensures {a.setD ==  (old a.setD) ∪ {v} }
writes {a.setD}

val remove (v : int) (a : t) : unit
requires {0<=v<a.n}
ensures {a.setD == (old a.setD) − {v}}
writes {a.setD}

...

end
```

Fig. 2: Abstract Specification

4.2 Concrete Implementation

The abstract type `t` is implemented as the record type `tsparse` (see Fig.3) whose fields are the size of the universe `n`, the two mutable arrays `dense` and `sparse`, the mutable bound `sizeD` and the ghost mathematical and abstract model `setD`. Why3 will generate verification conditions to ensure that the concrete implementation respects the abstract specification.

This record type definition is constrained by invariant properties: the length of both arrays is `n` which is a positive number, contents are belonging to the integer range $0..n-1$ (Inv1), `sizeD` is between 0 and `n` (Inv2), the two arrays must be consistent for those elements in the set (Inv3) (P_2 in Sect. 2). Furthermore the last property, Inv5, relates the abstract model with the concrete representation as in the property P_1 of Sect. 2.

In [4], Briggs and Torczon emphasize the fact that the two arrays do not require to be initialized when allocated. In the solution given by Filliâtre and Paskevich to the formal verification of sparse arrays, the arrays are not initialized too [9]. They specify a non initialized memory with the help of a `malloc` function. In our implementation we initialize the arrays `dense` and `sparse` with a negative value (-1) when they are created. We could have reused Filliâtre and Paskevich's approach in our formalization, but we did not in order to stay in line with the variant developed in the next section.

```
let constant initval : int = -1

predicate dom_ran (a : array int) (n: int) =
0 <= n && a.length = n && forall i. 0<=i<n -> initval<=a[i]< n

type tsparse =      { n : int;
                      mutable dense: array int;
                      mutable sparse: array int;
                      mutable sizeD: int;
                      mutable ghost setD : fset int; }
invariant {
  (*Inv1*)   dom_ran dense n && dom_ran sparse n &&
  (*Inv2*)   0 <= sizeD <= n &&
  (*Inv3*)   (forall i:int. 0 <= i < sizeD ->
               (0 <= dense[i] && sparse[dense[i]] = i)) &&
  (*Inv4*)   setD ⊆ (interval 0 n) &&.
  (*Inv5*)   forall x: int. 0<= x < n -> (x ∈  setD <->
               (0 <= sparse[x] < sizeD && dense[sparse[x]] = x))
}
by {n = 0; dense = make 0 initval; sparse = make 0 initval;
    sizeD = 0; setD = ∅}
```

Fig. 3: WhyML Type of a Sparse Set

The code of the operations on sparse sets are the straightforward translation of the algorithms in [4], except for the supplementary ghost code (e.g. the last statement in `remove_sparse`) which updates the abstract model `a.setD`. The deletion operation, named here `remove_sparse`, is shown in Fig. 4.

In addition to the previous constant-time operations, the following functions have been implemented and verified:

- forall: check if all the elements of the sparse set satisfy a predicate (linear with respect to the number of elements in `a.setD`);
- exists: check if one element of the sparse set satisfies a predicate (linear with respect to the number of elements in `a.setD`);
- tolist: compute the list of elements (linear with respect to the number of elements in `a.setD`);
- filter: remove the elements that do not satisfy a predicate (linear with respect to the number of elements in `a.setD`);
- copy: create a copy of a sparse set (linear wrt the number of elements in `a.setD`);
- union of 2 sparse sets : create a new sparse set containing the elements of the 2 arguments (linear wrt the number of elements in each set);
- in place union: update the first argument required to have the largest universe with the union of the 2 arguments (linear wrt the number of elements in the second argument);
- intersection of 2 sparse sets: create a new sparse set (linear wrt the number of the smallest set).
- in place intersection: update the first argument required to have the smallest universe with the intersection of the 2 arguments (linear wrt the number of elements in the first argument);

Let us notice that for the filter and in place intersection operations, iteration and removal are performed at the same time.

The deductive verification of all these operations required to invent and add some formal annotations such as loop invariants, ghost code and lemma functions. A typical example is the implementation of `cardinal_sparse` shown in Fig. 4. Its code is very simple since the number of elements in the sparse set `a` is exactly `a.sizeD` but a lemma-function, `cardinal_sizeD`, is used to prove the function's contract as a lemma that will be provided to the provers. The latter states that `|a.setD| = a.sizeD` by going through the `dense` array up to `sizeD`, collecting and counting its elements. In all the operations or logical functions that require an iteration on the effective part of the `Dense` array, a ghost variable collects the visited elements (and also the elements removed, e.g. in the in place intersection operation) and allows the computation to be observed. This makes the WhyML code very verbose[3] but it is the prize to pay to have automatic proofs.

VCs for the functions concern the conformance of the code to the post-condition and also to the invariant attached to the `tsparse` type.

[3] 18 lines of logical code are added to the 8 lines of computational code in the in-place intersection operation.

```
let remove_sparse (v : int) (a : tsparse)
requires {0<=v<a.n}
ensures {a.setD ==  (old a.setD) − {v}}
=
let i = a.sparse[v] in
if 0 <= i < a.sizeD && a.dense[i]=v then
  let e = a.dense[a.sizeD - 1] in
  a.dense[i] <- e ; a.sparse[e] <- i ;
  a.sizeD <- a.sizeD - 1;
  a.setD <- a.setD − {v}

(*  a lemma function to help the verification  *)
let lemma cardinal_sizeD  (a : tsparse)
ensures {|a.setD| = a.sizeD}
=
let ghost ref s = FsetInt.empty in
let ghost ref nb = 0 in
for i = 0 to a.sizeD - 1  do
  invariant {forall x:int. (exists j. 0<=j<i && x = a.dense[j]) <->  x ∈ s}
  invariant {nb = |s| && nb = i}
  s <- s  ∪ {a.dense[i]};
  nb <- nb + 1
done ;
assert {a.setD == s && nb = a.sizeD }

let cardinal_sparse (a : tsparse) : int
ensures {result =  |a.setD|}
=
return a.sizeD
```

Fig. 4: Implementation of Some Sparse Set Operations in WhyML

4.3 Proofs

The proof of all the VCs are done automatically using three automatic provers, CVC4, Alt-Ergo and Z3, using the strategy Auto Level 2[4]. Statistics per prover, number of proofs, time (minimum/maximum/average) in seconds, are recorded in Fig. 5.

Prover	*nb.proofs*	*min.time(s)*	*max.time(s)*	*av.time(s)*
Z3 4.8.9	1	0.06	0.06	0.06
Alt-Ergo 2.5.1	29	0.03	2.12	0.55
CVC4 1.6	276	0.03	1.80	0.15

Fig. 5: Sparse sets - Statistics per Prover: Number of Proofs, Time (minimum/-maximum/average) in Seconds

4.4 Extraction of OCaml Executable Code

To extract OCaml executable code from this development, we modified the previous WhyML code to use machine integers instead of mathematical integers. However mathematical integers are still manipulated in most logical assertions or ghost code. In our case it requires only syntactical modifications regarding the type of integer variables and arrays and some insertions of coercions between machine integers and mathematical integers in the logical assertions. The proofs remain all automatic.

This data structure is also often proposed as a bounded data structure, in which the set is constrained to have at most a given cardinality m. We can find several implementations of this variant on the Web. In that case the length of the `dense` array is m. The abstract type `t` and the concrete type `tsparse` are modified to take into account this maximal capacity. Some functions (e.g. add and union) are also concerned with this limit. This new requirement does not bring any difficulty for the verification. We have implemented this variant in WhyML and verified it with Why3. When machine integers are used, the `union` function requires an additional pre-condition for not going to an overflow.

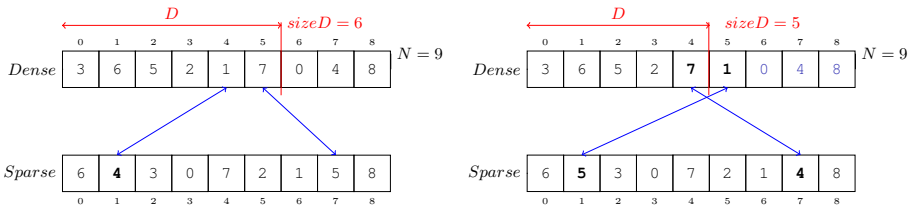
5 Backtrackable Sparse Sets as Domains

In this section we focus on a variant of sparse sets used in some constraint solvers (e.g. MiniCP [16], OsCaR [17]) to represent the domain of an integer variable, i.e. the finite set of possible values for that variable [13]. In such a context, to find a solution to a collection of constraints on some variables, or to show that the problem is unsatisfiable, the use case is as follows: for a variable X,

[4] and only one assertion in the lemma function about the cardinality.

initialize $Domain(X) = 0..N-1$, for some N, then propagate constraints to prune $Domain(X)$, then set $Domain(X)$ to a singleton containing a value of the pruned domain, propagate again, etc., backtrack if necessary. Thus, once the domain is initialized, there is no need to add any value, only deletions are performed. The advantage of sparse sets, as we have seen, is that membership and deletion operations can be performed in constant time. Furthermore, with a simple variation, these data structures are easy to restore when exploring solutions in an imperative setting, making backtracking cheap. Even if they are not used in constraint solving, we keep in our verified implementation the `add` and `union` operations but we will have to take care of the fact that they break reversibility. In the rest of the paper, to refer to this variant, we sometimes use the expression *sparse sets as domains* or shortly *domains*.

In this variant, the property P_2 is enforced for every value in $Dense$ (not only in $Dense[0..sizeD-1]$): $Sparse[Dense[i]] = i$ for all $i \in 0..N-1$, called now P_2'. Checking the membership of value v becomes trivial: just check $Sparse[v] < sizeD$. Removing an element v now consists of swapping v with the last element in $Dense$, decrementing $sizeD$ and also updating $Sparse$. An example is shown in Fig. 6. As pointed out in [13], the values in $dense[sizeD..N-1]$ are not changed by any operation, in particular by a deletion. Let us call this property P_3. This property can easily be added as an additional post-condition of the remove operation. The other operations remain the same (even if `add` and `union` are not used in constraint solving). We introduce a new function `bind`, which takes an argument v and reduces the set to the singleton $\{v\}$. It is useful in the context of constraint solving, to bind the value of a variable when exploring the search space. Its behaviour is very similar to `remove`: v is swapped with the last element in *sparse*, *dense* is updated accordingly, and $sizeD$ is set to 1. Illustrations are given in Fig. 6.

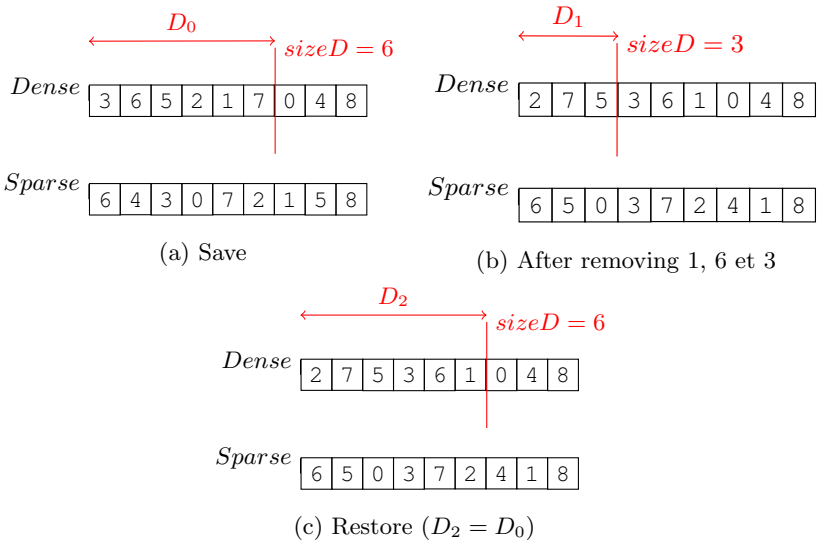


(a) a domain (b) remove 1

Fig. 6: A sparse set as Domain and a Deletion

Sparse sets in this variant are now easily backtrackable (or reversible), the only element to be stored and restored being the value of $sizeD$. Fig. 7 illustrates this with a simple example. Let a be the sparse set in Fig. 7a denoting the set D_0. We store the current value of $sizeD$, which is 6. Then we remove 1, 6 and 3 from a, whose resulting value is described on Fig. 7b. To restore the initial

situation, it is sufficient to set $sizeD$ to the value previously stored, i.e. 6, see Fig.7c. We recover exactly the same mathematical set of elements, even if the values of the two arrays are different from the value in Fig. 7a. The behaviour is the same when backtracking after a `bind` operation, a `clear` operation or a destructive intersection operation, or a combination of all of these. However insertion and destructive union operations break this possibility, we keep them in our formalization but after their use, all checkpoint information is lost. We introduce the operation `undo` to come back to a previously reached situation, characterized by the value p of $sizeD$. Its algorithmic content is very simple, i.e. set $sizeD$ to p but its specification requires more work and it will affect all the operations since new invariants will be needed. The main idea is to keep, for a domain, the collection of all previous states to which we can go back.



(a) Save

(b) After removing 1, 6 et 3

(c) Restore ($D_2 = D_0$)

Fig. 7: Backtracking on a Sparse Set as Domain

6 Formal Verification of Sparse Sets as Domains

We follow the same approach with an abstract specification and a concrete implementation. To take into account P_2', we have to change the type invariant of the `tsparse` type. To specify `undo`, the modification is deeper and impacts both the abstract type `t` and the concrete type `tsparse`.

6.1 Abstract Specification

The type definition of the abstract type `t` is shown on Fig. 8. We introduce an additional abstract variable, `states`, of type `fmap (fset int)`, which

stores the different successive states of the set, i.e. the successive mathematical models. It is defined as a partial function mapping cardinalities of `setD` to their corresponding model. For example, if `n` is 5, starting with the full set, removing 0 and then binding to the singleton containing 2, would result in the partial function `states` which maps 5 to the set $\{0, 1, 2, 3, 4\}$, 4 to $\{1, 2, 3, 4\}$ and 1 to $\{2\}$ and which is defined nowhere else.

A theory of partial functions is available in the Why3 standard library. It specifies a partial function as a WhyML total function and a mathematical set corresponding to its definition domain. Several logical functions and predicates are provided. For example, the formula `mapsto i s f` expresses that `i` is in the definition domain of the partial function `f` and relates `i` with its image `s`.

Each time an operation that modifies the sparse set, is performed, its current mathematical model is stored in `states`: `states (|setD|)` is then defined and equal to `setD`. This is part of the invariant that needs to be preserved. As said before, backtracking is only possible if elements are only removed from the sparse set, this implies that `states` is defined from the maximum cardinality, i.e. `n`, to the current cardinality, and that the corresponding images are subsets of each other. Furthermore, because of the `bind` operation, we may step from a cardinality `i` to 1, which means that the domain of states may not be an interval. The predicate `valid_states` specifies the invariant properties expected of `states`. So the abstract type `t` is modified to take account of this additional invariant (in Fig. 8, modifications are marked with 2 stars).

Fig. 8 also contains the abstract specification of the `undo` operation. It takes an argument that is the cardinality of the set to which to come back. This one must be greater than the cardinality of the current set and the corresponding state must have been encountered in the past, so its mathematical model must have been registered in `states`. The post-condition specifies that after calling the operation the current model is the one stored in `states(p)`, and the previous states have not changed for cardinalities greater than `p`. The new value of `states` has to comply with the `valid_states` invariant property, so `states` is not defined anymore for cardinalities smaller than `p` after the `undo` operation. The abstract specification of other operations has to be adapted consequently (see `undo` and `add` in Fig. 8). The additional post-condition of `add` enforces that backtracking is no longer possible by restricting the definition domain of `states` to the singleton `{|a.setD|}`.

6.2 Concrete Implementation

The `tsparse` type is adapted in the same way as the abstract type `t` (see Fig. 9). An additional ghost variable `states` is introduced and constrained according to the `valid_states` property. Furthermore a new property, $Inv7$, makes the connection between `states` and `sparse`: if s is the mathematical set registered in `states` for i, then its elements are exactly those that are in `dense`$[0..i]$. Again as a property in the type invariant, it must be preserved by any operation modifying an argument of type `tsparse`. Furthermore $Inv3$ is

```
predicate valid_states (states : fmap int (fset int) (setD : fset int) (n : int) =
-- domain of states is included in 0..n
   (dom states) ⊆ (interval |setD| (n+1)) &&
-- current state is registered
   mapsto |setD| setD states &&
-- each registered state contains the current one
   forall i:int, s : fset int. mapsto i s states ->
      s ⊆ (interval 0 n) &&  |setD| = i &&  setD ⊆ s

type t  = abstract {n : int ; mutable setD : fset int ;
                    mutable states:  fmap int (fset int);    (* *)
                   }
invariant {setD ⊆ (interval 0 n) &&
           valid_states states setD n}   (* *))

val remove (v : int) (a : t) : unit
requires {0<=v<a.n && v ∈ a.setD}
ensures {a.setD == remove v (old a.setD)}
ensures {forall i. 0 <= i <= a.n ->  i ≠ |a.setD| ->
            (mapsto i x a.states <-> mapsto i x (old a).states)}     (* *)

val add (v : int) (a : t) : unit
requires {0<=v<a.n && v ∉ a.setD}
ensures {a.setD == add v (old a.setD)}
ensures {dom a.states = {|a.setD|}}   (* *))

val undo (a : t) (p : int) : unit
requires {|a.setD| < p <= a.n}
requires {p ∈ (dom states)}
writes {a.setD, a.states}
ensures {exists s. mapsto p s (old a).states && a.setD == s}
ensures {forall i. p < i <= a.n ->
            (mapsto i v a.states <-> mapsto i v (old a).states)}
```

Fig. 8: Abstract Type t and undo Abstract Specification

modified to take into account that `dense` and `sparse` are now inverse and $Inv5$ is simplified.

The code of the `undo` operation is shown in Fig. 9. Its contract is similar to that of the abstract specification. Its computational part is only the last statement, the rest is some ghost code to update the model (`a.setD` and `a.states`). In particular, to maintain the invariant, all states between `a.sizeD` and `p` are deleted in `a.states`, thanks to the `remove_set_from_domain` operation which removes from the definition domain of a partial function all the elements of its first argument. The operations for removing and inserting an element are also illustrated in that figure. The computational part is composed of the two first statements. Besides the modification of the `a.setD` ghost model, the ghost code updates the `a.states` partial function: the former operation just stores the current state while the latter also erases all the previous stored models.

6.3 Proofs

The proof of all the VCs are done automatically using three automatic provers, CVC4 and Alt-Ergo using the strategy Auto Level 2. Statistics per prover, number of proofs, time (minimum/maximum/average) in seconds, are recorded in Fig. 10.

6.4 Extraction of OCaml Executable Code

OCaml executable code has been extracted. Again the previous implementation of sparse sets as domains has been modified to deal with machine integers without any difficulty. Proofs are still automatic.

6.5 A Formally Verified Defensive Implementation of Sparse Sets as Domains

Let us look at the test function written in WhyML in Fig. 11. The last statement is incorrect since we want to go back to a non-existing previous state. In Why3, when a call to a function is performed, we have to prove that each pre-condition is satisfied. So we need to prove that 2 is in the domain of `states`, which is not the case here. However, when using the extracted code on the same program, the invariant is broken. A solution to this problem consists in making the `undo` operation more defensive by testing if its argument refers to a correct state. It implies to keep track of the domain of `states` in the executable code. For this purpose, we propose to implement the bound `sizeD` by a *reversible* integer, i.e. a structure containing the current value of the bound and a list of its previous values. This proposition is inspired by the Java implementation of reversible integers in MiniCP[16].

The `rint` type of reversible integers is defined in Fig. 12. The list of previous values, `back`, is sorted in increasing order and has no duplicates. So each time an element is removed from the sparse set as domain, the current value in `sizeD` is

```
type tsparse =      { n : int;
                      mutable dense: array int;
                      mutable sparse: array int;
                      mutable sizeD: int;
                      mutable ghost setD: fset int;
                      mutable ghost states: fmap (fset int);   (* *)
                      }
invariant {
(*Inv1 *)   dom_ran dense n && dom_ran sparse n &&
(*Inv2 *)   0 <= sizeD <= n &&
(*Inv3' *)  (forall i:int. 0 <= i < sizeD &&  0<=v<n->
               (dense[i]=v <-> sparse[v]=i))
(*Inv4 *)   setD ⊆ (interval 0 n) &&.
(*Inv5' *)  (forall x: int. 0<= x < n ->
               (x ∈  setD <-> sparse[x] < sizeD)) &&
(*Inv6 *)  valid_states states setD n &&.    (* *)
(*Inv7 *)  (forall i, s. 0<=i<=n -> states i =  s ->
               (forall x. 0<=x<n  -> (sparse[x]<i <-> mem x s)))   (* *)
}
by ...

let undo_sparse (a : tsparse) (p : int) : unit
...
=
let ghost v = fmap_apply a.states p in
 a.setD <-  v ;
 a.states <- remove_set_from_domain (interval 0 p) a.states;
 a.sizeD <- p

let remove_sparse (v : int) (a : tsparse)
...
=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] (a.sizeD - 1);
a.sizeD <- a.sizeD - 1;
a.setD <- remove v a.setD;
a.states <- fmap_add a.sizeD a.setD a.states    (* *)

let add_sparse (v : int) (a : tsparse)
...
=
swap_two_arrays a.dense a.sparse a.n a.sparse[v] a.sizeD;
a.sizeD <- a.sizeD + 1;
a.setD <- add v a.setD;
a.states <- fmap_add a.sizeD a.setD fmap_empty    (* *)
```

Fig. 9: Concrete Implementation of Domains

Prover	*nb.proofs*	*min.time(s)*	*max.time(s)*	*av.time(s)*
Z3 4.8.9	39	0.02	0.96	0.08
Alt-Ergo 2.5.1	115	0.01	3.74	0.22
CVC4 1.6	216	0.04	1.07	0.15

Fig. 10: Statistics per prover: number of proofs, time (minimum/maximum/average) in seconds

```
let test () : unit =
let a = full_sparse(8) in
remove_sparse 0 a; remove_sparse 6 a;
remove_sparse 1 a; remove_sparse 4 a;
remove_sparse 2 a;
bind_sparse a 5;
assert {2 ∉ (dom states)};   assertion: proved
undo_sparse a 2              pre-condition: proof failed
```

Fig. 11: An Incorrect Program in WhyML

pushed in the `back` list and *value* is decreased. Each time an element is added, `back` becomes empty and the value is increased. When `undo p` is performed, the operation first searches `p` in `back` and removes all the values until `p` in this list before assigning `p` as the new value of `sizeD`. In fact the two first actions are done simultaneously, raising an exception if `p` is not a correct argument.

```
type rint = { mutable value : int;
              mutable back : list int;
            }
invariant { sorted back && distinct back &&
            forall n. mem n back -> value < n }
```

Fig. 12: WhyML Implementation of Reversible Integers

The WhyML code of the abstract and concrete modules for sparse sets as domains are modified to use reversible integers. To ensure consistency between `sizeD` and `states`, we add the following property in the invariant of the types `t` and `tsparse` :

```
forall x. (x = sizeD.value || mem x sizeD.back) <->  x ∈ dom states
```

Proofs of VCs remain automatic.

7 Some Experimentations

OCaml code was extracted from the WhyML models (using machine integers) of all the variants we have developed. We implemented a naive implementation of the Erathosthenes Sieve algorithm following a Web article[5] presenting sparse set implementations in C++, using the OCaml extracted code of four variants of sparse sets: sparse sets *à la* Briggs and Torczon with and without limited capacity, sparse sets as domains with bounds implemented as simple integer numbers and sparse sets as domains with bounds implemented as reversible integers. We also implemented this algorithm using the OCaml standard library module Set and an implementation of sets as hash tables[6].

This algorithm performs many insertions, deletions, membership tests and a final call to the operation that computes the cardinality of the sparse set that, at the end, contains the prime numbers up to P, the parameter of the algorithm. In our experimentation whose results are shown on Fig. 13, P varies from one hundred to one million. On the x-axis are the execution times in seconds and on the y-axis the values of P.

On this example sparse sets in their four versions outperform the two set other representations but it is a bit unfair since we are comparing a mutable representation with functional ones. Regarding the four variants, they are equivalent. Maintaining the links for removed elements do not impact significantly the execution time, the same remark applies for using reversible integers.

Our extracted code of sparse sets as domains (with bounds implemented as simple integer numbers and with bounds implemented as reversible integers) has been used with a simple sudoku solver originally written in OCaml by Filliâtre[7]. That example intensively uses backtracking and thus the `undo` operation. It has been evaluated on a large number of sudoku puzzles.

8 Conclusion

In this paper we presented the formal Why3 development for sparse sets and for sparse sets as domains used in constraint solvers. The former refines and extends a partial solution for a more general data structure, sparse arrays, done by Filliâtre and Paskevich some years ago. The latter is a variant of the former, but as far as we know it is the first formalization of this backtrackable data structure that allows the representation of domains of integer variables. We have extracted efficient OCaml code from these formally verified models, which we have experimented on simple test cases, the Erathosthenes Sieve algorithm and a naive sudoku solver. One perspective of this work is the extraction of C code.

[5] `https://www.codeproject.com/Articles/859324/Fast-Implementations-of-Sparse-Sets-in-Cplusplus`

[6] `https://github.com/backtracking/hashset`

[7] `https://github.com/backtracking/ocaml-bazaar/blob/main/sudoku.ml`

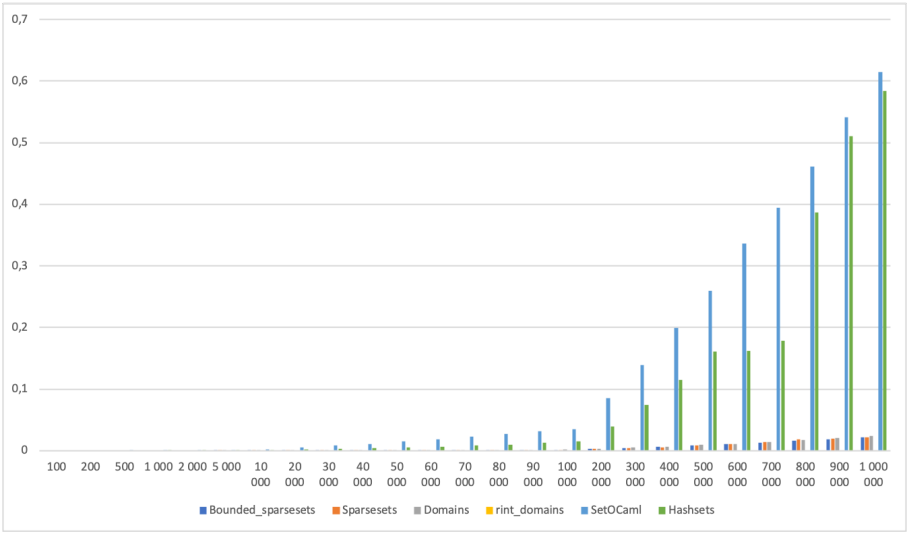


Fig. 13: Comparison of Execution Times on a Naive Implementation of Erathosthenes Sieve Algorithm

The technique used to be able to specify and prove the `undo` operation has implications for the whole formal development. It allows the use of WhyML and the deductive verification engine of Why3 to prove a property that involves more than a pre- and a post- state, and is close to a dynamic or temporal property.

In the case of very sparse sets or domains, using an array to implement the `sparse` structure is not optimal in terms of memory space. The data structure could be made more interesting by using another fast access structure, e.g. a hashmap (idea also suggested in [13]). So we could also suggest extending our current work to use such an alternative. It would be more interesting to make this `sparse` structure a generic parameter of the formalization in order to choose the right implementation *à la carte*.

As future work, we would also like to integrate sparse sets as domains in a finite domain constraint solver, e.g. in CoqBinFD, a formally verified constraint solver formally verified in Coq [6] or in FaCile, an OCaml constraint library [5].

Acknowledgements The author would like to thank the Why3 development team at LMF, Valentin Blot and Jean-Paul Bodeveix, for their interest and helpful comments during an initial presentation of this work. She also thanks the anonymous reviewers for their suggestions and careful reading.

References

1. J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

2. J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
3. A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1974.
4. P. Briggs and L. Torczon. An efficient representation for sparse sets. *LOPLAS*, 2(1-4):59–69, 1993.
5. P. Brisset and N. Barnier. FaCiLe : a Functional Constraint Library. In *CICLOPS 2001, Colloquium on Implementation of Constraint and LOgic Programming Systems*, Paphos, Cyprus, 2001.
6. M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 116–131, Paris, France, 2012.
7. M. Cristiá and C. Dubois. Comparing EventB, {log} and Why3 Models of Sparse Sets. In *35es Journées Francophones des Langages Applicatifs (JFLA 2024)*, Saint-Jacut-de-la-Mer, France, Jan. 2024.
8. M. Cristiá and G. Rossi. {*log*}: set formulas as programs. *Rend. Ist. Mat. Univ. Trieste*, 53:24, 2021. Id/No 23.
9. J.-C. Filliâtre and A. Paskevich. Why3 version of the sparse arrays example, the first example of the vacid-0 benchmarks, `https://toccata.gitlabpages.inria.fr/toccata/gallery/vacid_0_sparse_array.en.html`.
10. J.-C. Filliâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *22nd European Symposium on Programming, ESOP 2013, Held as Part of ETAPS 2013, Rome, Italy, Proceedings*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
11. J.-C. Filliâtre and A. Paskevich. Abstraction and genericity in why3. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I*, volume 12476 of *Lecture Notes in Computer Science*, pages 122–142. Springer, 2020.
12. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
13. V. Le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.
14. A. Ledein and C. Dubois. Facile en coq : vérification formelle des listes d'intervalles. In *31ème Journées Francophones des Langages Applicatifs*, 2019.
15. K. R. M. Leino and M. Moskal. Vacid-0: Verification of ample correctness of invariants of data-structures, edition 0. In *Proceedings of Tools and Experiments Workshop at VSTTE*, 2010.
16. L. Michel, P. Schaus, and P. Van Hentenryck. Minicp: a lightweight solver for constraint programming. *Mathematical Programming Computation*, 13(1):133–184, 2021.
17. P. Schaus and R. D. Landtsheer. Oscar user-guide, 2016. Available from `http://oscarlib.org`.
18. J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.