HAL
archives-ouvertes.fr

Simpler Proofs with Decentralized Invariants

Jean-Christophe Filliâtre

▶ **To cite this version:**

Jean-Christophe Filliâtre. Simpler Proofs with Decentralized Invariants. 2020. hal-02518570

HAL Id: hal-02518570
https://hal.inria.fr/hal-02518570

Preprint submitted on 25 Mar 2020

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.

Simpler Proofs with Decentralized Invariants

Jean-Christophe Filliâtre

Université Paris-Saclay, CNRS, Laboratoire de Recherche en Informatique
Inria Saclay – Île-de-France
91405 Orsay, France

Abstract

When verifying programs where the data have some recursive structure, it is natural to make use of global invariants that are themselves recursively defined. Though this is mathematically elegant, this makes the proofs more complex, as the preservation of these invariants now requires induction. In particular, this makes the proofs less amenable to automation. An alternative is to use local invariants attached to individual components of the structure and which only involve a bounded number of elements. We call these *decentralized invariants*. When the structure is updated, the footprint of the modification only impacts a limited number of invariants and reestablishing them does not require induction. In this paper, we illustrate this idea on three non-trivial programs, for which we achieve fully automated proofs.

Keywords: Deductive Verification, Invariants, Induction, Automated Theorem Provers

1. Introduction

In the realm of deductive program verification, an efficient approach consists in annotating programs with a behavioral specification, as well as proof hints such as assertions and invariants, then to compute verification conditions, and finally to pass them to automated theorem provers. This is called *auto-active verification*, a term coined by Leino and Moskal [14]. It contrasts

Email address: Jean-Christophe.Filliatre@lri.fr (Jean-Christophe Filliâtre)
URL: https://www.lri.fr/~filliatr/ (Jean-Christophe Filliâtre)

[1]This research was partly supported by the French National Research Organization (project VOCAL ANR-15-CE25-008).

with other approaches where user input is rather provided during the proof itself, *e.g.*, using some interactive proof assistant. Auto-active verification benefits from tremendous progress in automated theorem proving, in particular through SMT solvers, and flourishes through many tools these days, such as Dafny [13], Why3 [4], or VeriFast [11], to name a few.

Auto-active verification has some limits, though. One is that most automated theorem provers do not perform any kind of proof by induction. This is typically the case for SMT solvers used behind many program verifiers. As a consequence, recursively-defined functions and predicates are quickly show-stoppers. Of course, there are many ways to perform a proof by induction nonetheless. One can resort to a lemma function, for instance, that is a recursively-defined ghost function whose sole purpose is to establish a universal property by induction. Another solution is to switch to some interactive theorem prover to discharge the verification conditions that require induction. Whatever the solution, this increases user input a lot and slows down the verification dramatically.

When the data manipulated by the program have some recursive structure, it is natural to make use of global invariants that are themselves recursively defined. Then, reestablishing invariants after some update typically requires a proof by induction, which means some painful interaction. For this reason, we defend here the use of local invariants attached to individual components of the structure and which only involve a bounded number of elements. We call these *decentralized invariants*. When the structure is updated, the footprint of the modification only impacts a limited number of invariants. Automated theorem provers are thus able to reestablish them without resorting to induction, using only case analysis.

We illustrate this idea using three non-trivial examples: a union-find data structure; a heap-based priority queue data structure; and an algorithm to inverse a permutation in-place. The verification uses the Why3 tool, version 1.3.1 [4]. The source files are available at `http://why3.lri.fr/spdi/`. The verification is fully automated, using a combination of several SMT solvers, namely Alt-Ergo [3], CVC4 [2], and Z3 [6]. The web page details the provers used and the time spent to discharge the various verification conditions (186 in total). The time is typically under one second. A handful of VC require several seconds, but never more than 10 seconds.

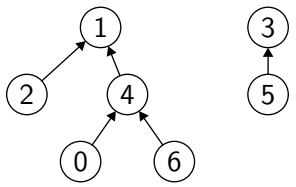
Purposely, this paper does not describe the code of these three examples, nor the function contracts of the various functions. Instead, we focus on the invariants, being either data structure invariants from the first two examples

and loop invariants for the third example. Through the URL given above, the reader has access to the full Why3 sources, where the code and specification are small and self-explanatory.

2. Union-Find

The union-find data structure maintains a set of disjoint classes. It provides two main operations, `find` to retrieve the representative element of a class and `union` to merge two classes, hence its name *union-find*. Its implementation is straightforward and achieves near-constant, optimal complexity. The union-find data structure is credited to McIlroy and Morris [1] and its complexity was studied by Tarjan, Fredman, and Saks [15, 9].

The principle behind the union-find data structure is simple: in each class, the elements are chained in such way that all paths lead to a single element, that is the representative of that class. For instance, a partition of the set $\{0, 1, 2, 3, 4, 5, 6\}$ into the two classes $\{0, 1, 2, 4, 6\}$ and $\{3, 5\}$ could be as follows:



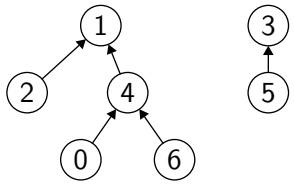
Here, the representative elements are respectively 1 and 3. The `find` operation follows the links, until it reaches the representative element. The `union` operation first finds out the representative elements of the two classes, using `find`, and then makes one point to the other. To achieve an optimal complexity, the union-find data structure uses two additional ideas:

- Once `find` has found the representative element, it makes all intermediate elements along the path pointing to it, to speed up further `find` operations on these elements. This is called *path compression*. On the example above, a call to `find`(0) would link 0 to 1 before returning 1.

- When performing a `union` operation, we link the representative element of the "smallest" class to the one of the "largest" class. This is called *weighted union*. To decide which one is the largest, we maintain for each class an upper bound of the length of each path, called the *rank*. On the example above, the class $\{0, 1, 2, 4, 6\}$ has rank 2 and the class

$\{3, 5\}$ has rank 1, so that a union of these two classes would make 3 point to 1. In case of a rank tie, we choose arbitrarily and we increase the rank by one.

To make things simpler, but not too simple, we can assume that we are only interested in partitions of the set $\{0, 1, \ldots, N-1\}$. In that case, the data structure can be readily implemented with two arrays `link` and `rank`.



	0	1	2	3	4	5	6
link	4	1	1	3	1	3	4

	0	1	2	3	4	5	6
rank	0	2	0	1	1	0	0

A representative element, such as 1 and 3 here, is linked to itself. In the array `rank`, values are only meaningful for representative elements.

Our Why3 implementation declares the union-find data structure as a record, with fields containing the value of N, the two arrays, and a ghost function mapping each element to the representative element of its class. The latter is notably used to specify the operations (not shown here, but available online).

```
type uf = {
  size: int;
  link: array int;
  rank: array int;
  mutable ghost rep: int → int;
}
```

We now come to the point of this paper, that is to state the invariants of this data structure. In one way or another, we need to state that the contents of `link` indeed forms paths connecting each element x to the element `rep` x. As a consequence, it *would be* natural to introduce a notion of path, for instance as an inductively-defined predicate, as follows,

$$\frac{}{path\ x\ x} \qquad \frac{path\ \texttt{link}[x]\ y}{path\ x\ y}$$

and then to state the existence of adequate paths, that is, something like

$$\forall x.\, path\ x\ (\texttt{rep}\ x). \tag{1}$$

(We omit bounds on x and y for simplicity.) One nice feature of such an inductively-defined predicate is that it captures the finiteness of paths. Alternatively, predicate *path* could also include a length as a third argument, or the full list of all elements along the path, which is then amenable to a recursive definition instead, as follows:

$$path\ x\ y\ n \stackrel{\text{def}}{=} \begin{cases} x = y & \text{if } n = 0, \\ path\ \texttt{link}[x]\ y\ (n-1) & \text{otherwise.} \end{cases}$$

The problem with such a predicate *path*, being it inductively or recursively defined, lurks in the preservation of property (1) whenever the array `link` is modified to account either for a path compression or for the union of two classes. Indeed, some paths are preserved, possibly being shortened, some paths are no more valid, and new paths are created. In the example above, the path compression resulting from the assignment $\texttt{link}[1] \leftarrow 0$ means that the paths $2 \rightarrow 1$, $6 \rightarrow 4 \rightarrow 1$, and $5 \rightarrow 3$ are left unchanged, the path $0 \rightarrow 1$ is shortened, and the path $0 \rightarrow 4$ disappear.

More precisely, we now have two versions of the array `link`, before and after the assignment, and we seek to establish the new path properties, for the new version of the array, from the former path properties given by the old version of the array. And this *requires proofs by induction*. This is true even for paths that are left unchanged. In particular, the need for induction places such proofs out of the scope of automated theorem provers. Of course, we could do such proofs with some interactive theorem prover, or resort to the use of recursive lemma functions, but that is a lot of work.

Our trouble comes from the *global* nature of the *path* predicate, which involves an arbitrarily large number of elements[2]. So we should rather look for invariants that only involve a bounded number of elements. And this is indeed possible for the union-find data structure. Here are such invariants, in Why3's syntax:

```
invariant { ∀ x. 0 ≤ x < size → 0 ≤ link[x] < size      }
invariant { ∀ x. 0 ≤ x < size → 0 ≤ rep x   < size      }
invariant { ∀ x. 0 ≤ x < size → rep (rep x) = rep x     }
invariant { ∀ x. 0 ≤ x < size → x = link[x] ↔ rep x = x }
invariant { ∀ x. 0 ≤ x < size → rep link[x] = rep x     }
```

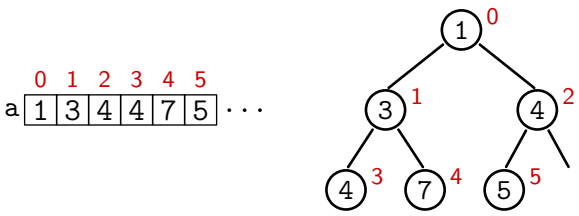
[2]This is called the *footprint*, a terminology introduced by Peter O'Hearn.

As we can see, each invariant only involves a limited number of elements, namely `x`, `link[x]`, and `rep x`. This means that restoring such invariants after an assignment will only involve a *case analysis*, not a *proof by induction*. Sometimes there are many cases to consider (`x`/`link[x]`/`rep x` is the assignment element or not), but still the number of cases is bounded. In particular, this is amenable to automated theorem proving. Since such invariants do not state any global property about the data structure, but rather state local, individual properties about the element, we call them *decentralized invariants*.

Using these invariants, we could verify an implementation of the `find` and `union` operations, including their functional specification in terms of `rep`. The verified program also contains additional fields and invariants to prove the termination of `find` and `union`. They are also stated as decentralized invariants. It is worth pointing out that, though our code is making use of the array `rank` to achieve the expected complexity, our invariants say nothing about it since we are not concerned in verifying the complexity. This could be done, for instance following the proof by Charguéraud and Pottier [5]. We anticipate that the properties related to `rank` could also be stated in terms of decentralized invariants.

3. Heap-Based Priority Queue

To implement a priority queue in a space and time efficient way, one of the best option is to use a heap, that is a binary tree where each node contains a value no greater that its two successors, and to encode that tree within an array. The latter is readily achieved by storing the successors of the node at index i at indices $2i + 1$ and $2i + 2$. Here is a heap containing the multiset $\{1, 3, 4, 4, 5, 7\}$, stored within the first six elements of an array `a`, the array indices being displayed in red:



The array can be larger, to accommodate later additions within the limits of the array length, or, better, can be implemented as a resizable array. The priority queue provides three main operations, as follows:

Minimal element The minimal element appears to be `a[0]`, by the heap property. So it is obtained in constant time.

Insertion To insert a new element, we place it in the next available slot of array `a` and then we "move it up" by repeatedly swapping it with its parent node as long as necessary.

Deletion To remove the minimal element, we replace it by the rightmost element of the array `a` (that would be 5 in the example above) and then we "move it down" by repeatedly swapping it with the smallest of its two successors as long as necessary.

In particular, it is clear that insertion and deletion have a time complexity that is logarithmic in the size of the queue, since the tree is always balanced by construction.

When it comes to implement the priority queue, we declare a record with two fields, an array `a` storing the elements and the size `len` of the queue:

```
type t = {
  a: array int;
  mutable len: int;
}
```

This means that the first `len` elements in `a` are used to store the contents of the queue, the remaining elements being unused. This also means that the queue has a fixed capacity, which is the size of the array. We could easily accommodate an arbitrary number of elements using a resizable array instead, as mentioned earlier, but that is not the point here. Similarly, we assume the queue elements to be integers, for simplicity, but that could be generalized easily.

The point here is to find an efficient way to state the invariant of the type `t` above, to express that the first `len` elements of array `a` indeed encode a binary tree that enjoys the heap property. One way to do so *would be* to introduce an inductively-defined predicate *heap*, where *heap* k means "the tree rooted at index k is a heap", as follows:

$$\frac{2k+1 < \texttt{len} \Rightarrow \texttt{a}[k] \leq \texttt{a}[2k+1] \wedge \mathit{heap}\ (2k+1) \qquad 2k+2 < \texttt{len} \Rightarrow \texttt{a}[k] \leq \texttt{a}[2k+2] \wedge \mathit{heap}\ (2k+2)}{\mathit{heap}\ k}$$

Then, we simply need to maintain *heap* 0 as the sole invariant of the data structure. The author has done exactly this in the past [8], in a former

proof of this data structure using the Coq proof assistant. Such a way to do, however, involves a lot of proofs by induction. Indeed, any assignment in array `a` requires to reestablish the global invariant *heap* 0, which means proofs by induction. In particular, one has to reestablish the *heap* property even for sub-trees not impacted by the assignment.

Instead, we can use a decentralized invariant that states the heap property *locally*, at each node in the tree, as follows:

```
∀ i. 0 ≤ i < len →
  (0 < 2*i+1 < len → a[i] ≤ a[2*i+1]) ∧
  (0 < 2*i+2 < len → a[i] ≤ a[2*i+2]) ∧
  (0 < i     < len → a[(i-1)/2] ≤ a[i])
```

This invariant purposely contains a bit of redundancy, to relieve the SMT solvers from swapping between multiplication and division by two. With such an invariant, assignments in array `a`, when moving elements up and down, only require to reestablish the heap property at a bounded number of indices. This is not completely trivial, as we need to consider whether the index of the assignment element is of the form $2i + 1$, $2i + 2$, or $\lfloor (i - 1)/2 \rfloor$. Yet, this is within the reach of SMT solvers. Indeed, we are able to verify the insertion and deletion operations in a fully automated way.

We are left with a single proof by induction, namely to show that `a[0]` is indeed the smallest element in the heap. We can perform this induction using a lemma function, as follows:

```
let lemma first_is_min (h: t)
  ensures { ∀ i. 0 ≤ i < h.len → h.a[0] ≤ h.a[i] }
```

The body of this lemma function is straightforward and its verification condition is easily discharged automatically.

4. Inverse of a Permutation

The last example is that of a cute algorithm to inverse a permutation. Consider an integer array A of size N that contains a permutation of the set $\{0, 1, \ldots, N - 1\}$. It is straightforward to compute the inverse of that permutation using a second array, say B, by assigning i to $B[A[i]]$ for each i. But say that we want to inverse the permutation *in-place*, for the sake of the game or because we cannot afford allocating another array. It means that we are limited to moving elements around within array A. Additionally, we seek for a solution with a linear time complexity.

0	1	2	3	4	5	
4	3	0	1	5	2	start a new cycle at index 5
4	3	0	1	5	-1	place sentinel -1 and jump to index 2
4	3	-6	1	5	-1	mark we came from 5 and jump to index 0
-3	3	-6	1	5	-1	mark we came from 2 and jump to index 4
-3	3	-6	1	-1	-1	mark we came from 0 and jump to index 5
-3	3	-6	1	-1	4	back to the sentinel, from index 4
-3	3	-6	1	0	4	find negative value at index 4
-3	3	-6	1	0	4	start a new cycle at index 3
-3	3	-6	-1	0	4	place sentinel -1 and jump to index 1
-3	-4	-6	-1	0	4	mark we came from 3 and jump to index 3
-3	-4	-6	1	0	4	back to the sentinel, from index 1
-3	-4	5	1	0	4	find negative value at index 2
-3	3	5	1	0	4	find negative value at index 1
2	3	5	1	0	4	find negative value at index 0

Figure 1: Running Algorithm I on the permutation (1 3) (0 4 5 2).

In volume 1 of *The Art of Computer Programming*, Knuth presents a version nice solution to that problem, as Algorithm I [12, Sec. 1.3.3, page 176]. This is not, strictly speaking, an in-place algorithm, as it requires one extra bit per array element for marking purposes. Yet we can consider it in-place in practice, as a permutation stored within an array of *signed* integers has unused sign bits that we can freely use as marks.

Algorithm I, due to Huang [10], proceeds as follows. It inverts the various cycles of the permutation, one at a time. The inversion of a cycle consists in traversing the cycle, assigning to each array cell the index of the previous cell in the cycle. While doing so, we mark these elements as being inverted using the sign bit. A sentinel value -1 is used to mark the first element of the cycle, so that we can detect when we are done with the cycle. A main loop scans the array from right to left (it could be the other way round), examining each value. If the value is nonnegative, it belongs to a new cycle, and we start inverting it using an inner loop. If on the contrary the value is negative, it means that it is a marked value that belongs to some cycle we already inverted, and we simply erase the mark. Figure 1 illustrates algorithm I on a 6-element permutation made of two cycles.

When it comes to implement algorithm I, it is convenient to mark the

elements using the function $x \mapsto -x - 1$. First, this is an involution, which means that it is used for both setting and erasing marks. Second, it coincides with the bitwise NOT operation, which means it is already implemented efficiently by the machine. For this reason, we introduce it as a prefix `~` operation:

```
let function (~_) (x: int) : int = -x-1
```

The implementation of algorithm I is then a matter of two nested loops and four variables, with a little bit of care to the details. As we did in the previous two sections, the code is omitted here (but is available online) and we rather focus on the way to state the invariants so that the proof is as automated as possible.

As with the union-find data structure, it *would be* natural to introduce some inductive or recursive notion of path induced by array A, for instance as follows:

$$\frac{}{path\ x\ x} \qquad \frac{path\ A[x]\ y}{path\ x\ y}$$

This way, we could define the notion of cycles, maintain a loop invariant for the inner loop stating that there is a path from the starting point to the current point, etc. And there is no reason we could not succeed in proving this algorithm this way[3]. But for the exact same reasons we did not do that for the union-find data structure, we are not going to do that.

Instead, we seek for a decentralized invariant for our algorithm. It means that we wish to state a local property, holding universally at each array index. Assuming that the initial contents of array A is in variable `olda`, its current contents in variable `a`, and that the start of the current cycle is in variable `m`, we come up with the following invariant:

```
1. ∀ e. 0 ≤ e < n →
2.   (-n ≤ a[e] < n) ∧
3.   (m < e → 0 ≤ a[e]) ∧
4.   (m < e → olda[a[e]] = e) ∧
5.   (e < m → a[e] ≥ 0 → a[e] = olda[e]) ∧
6.   (e ≤ m → a[e] ≤ m) ∧
7.   (e < m → a[e] < 0 →
```

[3]For instance, Dufourd has a Coq library with similar definitions [7] and he used it to verify similar algorithms.

8. `olda[~a[e]] = e ∧ (~a[e] ≤ m → a[~a[e]] < 0))`

Line 2 sets bounds for the elements, the marked elements being negative and the unmarked elements being nonnegative. Line 3 states that any element at a position above `m` is unmarked. Indeed, `m` is the start of the cycle and we are scanning the array from the right to the left. Line 4 states that the segment of the array above `m` is already inverted. Line 5 states that unmarked elements at positions below `m` are still unmodified. Line 6 states that elements at positions below `m` cannot be greater than `m`. Indeed, a greater value would be part of a cycle that is already inverted but then it would be marked, and thus negative. Last, lines 7 and 8 state that negative values at positions below `m` are already inverted and, when they lead to values at positions below `m`, these are marked values as well. It is worth pointing out that the invariant does not make any particular case for the sentinel value.

Here is an illustration of this invariant taken from Fig. 1, just after with start inverting the second cycle, with `m` = 3.

0	1	2	m=3	4	5
-3	3	-6	-1	0	4

Values at positions above `m`, namely 0 and 4, are already the final values. Values at positions below `m` are either negative, such as -3 and -6, which means they are already inverted but yet to be unmarked, or positive, such as 3, which means they are still to be inverted, either as part of the current cycle being inverted (that is the case, here) or as part of yet another cycle.

The code online contains a few other loop invariants, namely bureaucratic invariants related to the values of the few variables of the program. It also contains a proof of termination of the inner, using a variant that counts the number of nonnegative values in the array. All the verification conditions are discharged automatically by SMT solvers.

5. Conclusion

In this paper, we have illustrated the idea of *decentralized invariants*, that are local invariants attached to a bounded number of data components, as opposed to recursively-defined invariants. The benefits of decentralized invariants lie in greater proof automation. We have illustrated this on three non-trivial programs, for which we have achieved fully automated proofs.

This concept is likely to be folklore, known and used already by people doing auto-active verification on a daily basis. Yet, as we say in French,

"l'éducation, c'est la répétition" (teaching means repetition). And we believe that the three examples given in this paper might be useful. The author himself tediously verified the first two examples using the Coq proof assistant a long time ago and embarked himself in some unnecessarily complex proof of the third example. Achieving fully automated proofs of these three programs was a nice surprise.

Acknowledgments. I'm grateful to my colleague Andrei Paskevich, who coined the term *decentralized invariants*. Additionally, I would like to thank all the people who contributed to the three Why3 proofs described in this paper, in one way or another: Martin Clochard, Simão Melo de Sousa, Andrei Paskevich, Mário Pereira, Aymeric Walch.

References

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11. Springer-Verlag, 2011. http://cvc4.cs.stanford.edu/web/.

[3] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover, 2008. http://alt-ergo.lri.fr/.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform. http://why3.lri.fr/.

[5] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, 62(3):331–365, March 2019.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3, an efficient SMT solver. http://research.microsoft.com/projects/z3/.

[7] J-F. Dufourd. Formal study of functional orbits in finite domains. *Theoretical Computer Science*, 564:63–88, Jan 2015.

[8] Jean-Christophe Filliâtre and Nicolas Magaud. Certification of Sorting Algorithms in the System Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999.

[9] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 345–354, New York, NY, USA, 1989. ACM.

[10] Bing-Chao Huang. An algorithm for inverting a permutation. *Information Processing Letters*, 12(5):237–238, October 1981.

[11] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[12] Donald E. Knuth. *The Art of Computer Programming, volume 1 (3rd ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., 1997.

[13] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[14] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, Redmond, WA, USA, November 2010.

[15] Robert Endre Tarjan. *Efficiency of a Good But Not Linear Set Union Algorithm. J. ACM*, 22(2):215–225, 1975.