Efficient Verified Implementation of Introsort

Anonymous Author(s)

Abstract

Sorting algorithm are an important part of most standard libraries, and both, their correctness and efficiency is crucial for many applications.

As generic sorting algorithm, the GNU C++ Standard Library implements the Introsort algorithm, a combination of quicksort, heapsort, and insertion sort.

We verify this algorithm in the Isabelle LLVM verification framework, including most of the optimizations from the GNU implementation. On an extensive benchmark set, our verified algorithm performs on par with the original.

1 Introduction

The goal of this paper is to provide a case study to show that the verification of state-of-the art implementations of medium-complex algorithms is feasible. We chose Musser's introspective sorting algorithm (Introsort) [33], and its implementation in the GNU C++ Library (libstdc++) [19]. Using the Isabelle-LLVM verification framework [29], we verify the same algorithm, including most of the optimizations from libstdc++. The result is a verified sorting algorithm, which performs on-par with the original implementation on an extensive set of benchmarks.

Sorting algorithms are an important part of any standard library. The Introsort algorithm, being a combination of quicksort, heapsort, and insertion sort, is far from being trivial, but still has a manageable complexity. Finally, libstdc++ provides a state-of-the art implementation with several non-trivial but crucial optimizations. This makes Introsort a good candidate for our case study.

Moreover, sorting algorithms in standard libraries have not always been correct. The Timsort [36] algorithm in the Java standard library has a history of bugs[1], the (hopefully) last of which was only found by a formal verification effort [14]. Also, many real-world mergesort implementations suffered from an overflow bug [7]. Finally, LLVM's libc++ [32] implements a different quicksort based sorting algorithm. While it may be functionally correct, it definitely violates the C++ standard by having a quadratic worst-case run time[2].

To the best of our knowledge, this paper provides the first verification of the Introsort algorithm. This algorithm and two Java sorting algorithms [3, 14] are the only verified "real-world" implementations of sorting algorithms we are aware of. On the technical side, this paper provides a case study of using stepwise refinement for the design of verified algorithms. Moreover, we report on two extensions to the Isabelle-LLVM framework, to handle nested container data structures and to automatically generate C-header files to interface the generated code. These were required for our benchmark programs, which sort arrays of arrays of characters, and interface the generated LLVM code from C++.

The complete Isabelle/HOL formalization and the benchmarks are submitted as artefact accompanying this paper.

2 The Introsort Algorithm

The Introsort algorithm by Musser [33] is a sorting algorithm that combines the good average-case runtime of quicksort [23] with the worst-case complexity of heapsort [40]. The basic idea is to use quicksort as main sorting algorithm, but switch to heapsort when the recursion depth exceeds a given limit, usually $2\lfloor \log_2 n \rfloor$ for n elements.

Introsort is the default generic sorting algorithm in the GNU C++ Library [19]. Alg. 1 displays its pseudo-code. The

```
 1: procedure PARTIAL_INTROSORT($a, l, h, d$)
 2:     if $h - l >$ threshold then
 3:         if $d = 0$ then
 4:             HEAPSORT($a, l, h$)
 5:         else
 6:             $m \leftarrow$ PARTITION_PIVOT($a, l, h$)
 7:             PARTIAL_INTROSORT($a, l, m, d - 1$)
 8:             PARTIAL_INTROSORT($a, m, h, d - 1$)
 9: procedure INTROSORT($a, l, h$)
10:     if $h - l > 1$ then
11:         PARTIAL_INTROSORT($a, l, h, 2\lfloor \log_2(h - l) \rfloor$)
12:         FINAL_INSORT($a, l, h$)
```

Algorithm 1. Introsort

INTROSORT function sorts the slice $[l..<h]$ of an array a. For this, it first calls PARTIAL_INTROSORT to partially sort the array, i.e., all elements are no more than threshold from their

[1] see https://bugs.java.com/bugdatabase/view_bug.do?bug_id=8011944

[2] See https://bugs.llvm.org/show_bug.cgi?id=20837. This has not been fixed by October 2019.

Conference'17, July 2017, Washington, DC, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

final position, and then finishes the sorting by invoking insertion sort. The PARTIAL_INTROSORT procedure implements a standard quicksort, that switches to heapsort when the recursion depth gets greater than d, and stops when the partition size becomes less than or equal to threshold. Note that the actual implementation in the GNU C++ Library contains a manual tail-call optimization, replacing the second recursive call to PARTIAL_INTROSORT by looping. For our formalization, we chose to not implement this optimization, in particular due to the fact that the LLVM optimizer recognizes and eliminates this tail call automatically.

2.1 Partitioning

The PARTITION_PIVOT(a, l, h) algorithm must re-arrange the elements in a and return a *pivot* index $m \in [l{<}..{<}h]$, such that all elements $a[l..{<}m]$ are less than or equal to $a[m]$, and all elements $a[m..{<}h]$ are greater than or equal to $a[m]$. Note that the performance of quicksort crucially depends on pivot selection: Ideally, the pivot index should be close to $(h-l)/2$, i.e., the pivot element should be close to the median. The algorithm implemented by the GNU C++ Library is sketched in Alg. 2. It selects the median of the three elements $a[l+1]$, $a[\lfloor(h-l)/2\rfloor]$, and $a[h-1]$ as pivot (l. 10). It then swaps the selected median element to the front $a[l]$, and invokes the standard Hoare partitioning algorithm, which searches for pairs of mismatched elements simultaneously from the start and end of the array, and swaps them. The chosen pivot element remains at position $a[l]$.

```
1:  function PARTITION($a, l, h, p$)
2:      while True do
3:          while $a[l] < a[p]$ do $++l$
4:          $--h$
5:          while $a[p] < a[h]$ do $--h$
6:          if $\neg l < h$ then return $l$
7:          SWAP($a[l], a[h]$)
8:          $++l$
9:  function PARTITION_PIVOT($a, l, h$)
10:     $p \leftarrow$ MEDIAN_OF_THREE($a, l+1, \lfloor h-l \rfloor/2, h-1$)
11:     SWAP($a[l], a[p]$)
12:     return PARTITION(a,l+1,h,l)
```

Algorithm 2. Partitioning

Note that this function contains a subtle optimization: The loops in lines 3 and 5 will iterate until they find a mismatched element. This iteration will not exceed the array bounds, only because the median selection placed *stopper elements* at the beginning and end of the array. In particular when the element comparison function is cheap (e.g. for integer arrays), omitting an extra bounds check in the very inner loop of the sorting procedure can make a difference in practice.

2.2 Unguarded Insertion Sort

Insertion sort is a well-known simple sorting algorithm, which works by inserting the elements of the array, one after the other, into the sorted prefix of the array. Its worst case complexity is quadratic in general, but linear for partially sorted arrays with a constant threshold. For small thresholds, it performs better than quicksort, such that it is effective to stop the quicksort procedure early, and do the final sorting with insertion sort. Algorithm 3 shows the algorithm we implemented[3].

```
1:  procedure INSERT($G, a, l, i$)
2:      $t \leftarrow a[i]$
3:      while $(\neg G \lor l < i) \land t < a[i-1]$ do
4:          $a[i] \leftarrow a[i-1]$
5:          $--i$
6:      $a[i] \leftarrow t$
7:  procedure GEN_INSORT($G, a, l, i, h$)
8:      while $i < h$ do
9:          INSERT($G, a, l, i$)
10:         $++i$
11: procedure FINAL_INSORT($a, l, h$)
12:     if $h - l \leq$ threshold then
13:         GEN_INSORT(true, $a, l, l+1, h$)
14:     else
15:         GEN_INSORT(true, $a, l, l+1, l+$ threshold)
16:         GEN_INSORT(false, $a, l, l+$ threshold, $h$)
```

Algorithm 3. Final Insertion Sort

This algorithm contains a similar optimization as the partitioning algorithm: As the array is already partially sorted, we know that elements beyond the threshold index won't be inserted at the very beginning of the array. This is exploited to get rid of the bounds check in line 3. Only the first threshold elements of the array are sorted with activated bounds check. In the displayed code, we control the bounds check by the Boolean parameter G, and assume that the algorithm will be specialized for both values of G.

3 Notation

Throughout this paper, we use some specific notation. We write function application in curried form, i.e., $f\, x_1 \, \ldots \, x_n$. Definitions of types and functions are denoted by $\equiv$. Indexes start at zero. The ith element of a list is $xs[i]$, the slice of a list from index i inclusive to j exclusive is $xs[i..{<}j]$. We also use the notations $xs[i{<}..{<}j]$, $xs[i{<}..j]$, and $xs[i..j]$ to specify exclusiveness of bounds. Moreover, an omitted lower bound denotes index 0, and an omitted upper bound denotes the length of the list, e.g., $xs[..{<}n]$ is the list of the first n elements

[3]The guarded version (G = true) slightly differs from the implementation in the GNU C++ Library. However, as it is only called for a small number of elements, we don't expect this to have any significant effect.

of xs, and $xs[n..]$ is the list of the remaining elements. List concatenation is written as $xs_1 @ xs_2$, and $xs[i{:=}x]$ is the list xs, with index i updated to element x. The length of list xs is denoted by $|xs|$.

4 The Isabelle Refinement Framework

The Isabelle Refinement Framework [30] provides tools for verified program development by stepwise refinement, using the Isabelle HOL proof assistant [35]. It is based on a nondeterminism-error monad, defined as follows:

$\alpha\ nres \equiv FAIL \mid RES\ (\alpha\ set)$
$\texttt{return}\ x \equiv RES\ \{x\}$
$\texttt{bind}\ FAIL\ f \equiv FAIL \mid \texttt{bind}\ (RES\ X)\ f = \bigsqcup x{\in}X.\ f\ x$

Intuitively, $FAIL$ represents a program that may fail (e.g. not terminate or fail explicitly by violating an assertion), and $RES\ X$ represents a program that, nondeterministically, returns a value $x{\in}X$. By extending the subset ordering, with $FAIL$ as greatest element, $\alpha\ nres$ becomes a complete lattice. The program $\texttt{return}\ x$ returns exactly the value x. The program $\texttt{bind}\ m\ f$ sequentially composes m and f: it nondeterministically chooses a return value x of m, and then behaves like $f\ x$. If either m, or one of the $f\ x$ may fail, then also $\texttt{bind}\ m\ f$ may fail.

We use sequential notation for bind, i.e., $x{\leftarrow}m;\ f\ x$ stands for $\texttt{bind}\ m\ (\lambda x.\ f\ x)$, and $m_1;\ m_2$ stands for $\texttt{bind}\ m_1\ (\lambda\_.\ m_2)$, i.e., the result of m_1 is ignored. We also use the standard HOL functions `if−then−else` and `let`, and define the following notations:

$\texttt{assert}\ \Phi \equiv \texttt{if}\ \Phi\ \texttt{then return}\ ()\ \texttt{else}\ FAIL$
$\texttt{spec}\ x.\ \Phi\ x \equiv RES\ \{x \mid \Phi\ x\}$

An assertion will fail if its argument is false, and $\texttt{spec}\ x.\ \Phi\ x$ describes all results that satisfy Φ.

Recursion is defined by the greatest fixed point over a monotonic functor F[4] , and while loops are defined via recursion:

$\texttt{rec}\ F\ x \equiv \texttt{assert}\ (mono\ F);\ fixp\ F\ x$
$\texttt{while}\ b\ c\ s \equiv \texttt{rec}\ (\lambda W\ s.$
$\quad \texttt{if}\ b\ s\ \texttt{then}\ s' \leftarrow c\ s;\ W\ s'\ \texttt{else return}\ s)\ s$

From now on, we will omit the `rec` syntax, and define recursive monadic functions by their recursion equation, e.g.:

$\texttt{while}\ b\ c\ s \equiv$
$\quad \texttt{if}\ b\ s\ \texttt{then}\ s' \leftarrow c\ s;\ \texttt{while}\ b\ c\ s'\ \texttt{else return}\ s$

The ordering on $nres$ can be interpreted as refinement: $m_1 \leq m_2$ means that m_1 returns fewer possible results than m_2, and may only fail if m_2 may fail. The statement

$f\ x \leq \{\ \texttt{assert}\ (P\ x);\ \texttt{spec}\ r.\ Q\ r\}$

means that function f satisfies the specification with precondition P and postcondition Q.

[4]Monadic programs are monotonic by construction [26]. Moreover, on monadic programs, the greatest fixed point over the $nres$-ordering coincides with the fixed point over a flat CCPO, which is the standard construction for recursion. In the actual formalization, monotonicity wrt. both orderings is asserted.

5 Verifying Introsort

The first step to verify a sorting algorithm is to specify the desired result. We specify a sorting algorithm as follows:

$sort\_spec\ xs \equiv \texttt{spec}\ xs'.\ sorted\ xs' \wedge mset\ xs' = mset\ xs$
where
$sorted\ xs \equiv \forall i{<}j{<}|xs|.\ \neg(xs[i] > xs[j])$

That is, the result will be sorted and a permutation of the input ($mset\ xs$ denotes the multiset of elements of list xs).

Note that $<$ is a *strict weak ordering*, i.e., a relation with:

$$a{<}b \implies \neg b{<}a \quad \textbf{and} \quad a{<}c \implies a{<}b \vee b{<}c$$

Weak orderings are used by C++ for sorting [24]. They are a natural model for objects that are sorted by linearly ordered keys: An equivalent characterization can be given by an equivalence relation on the elements (e.g. objects with same keys), and a linear ordering on the equivalence classes. In particular, any linear ordering is a weak ordering, with the equivalence relation being equality, and $\neg(a < b)$ meaning $b \leq a$. While our formalization is over arbitrary weak orderings, we assume a linear ordering for the rest of this paper, to simplify the presentation.

For a partial sorting algorithm wrt. threshold n, we replace *sorted* by:

$part\_sorted\ xs \equiv \exists ss.\ is\_slicing\ xs\ ss \wedge sorted_{\leq}\ ss$
where
$is\_slicing\ xs\ ss \equiv xs = concat\ ss \wedge (\forall s{\in}ss.\ |s| \leq n)$
$xs \leq ys \equiv \forall x{\in}xs.\ \forall y{\in}ys.\ x \leq y$

That is, a list is partially sorted if we can split it into slices that are not longer than the threshold n, and each element in a slice is less than or equal to any element in a subsequent slice.

The next step to verify quicksort like algorithms is to specify a partitioning function:

$partition\_spec\ xs \equiv \texttt{assert}\ (|xs| \geq 4);$
$\quad \texttt{spec}\ (\lambda(xs_1,xs_2).\ xs_1{\neq}[] \wedge xs_2{\neq}[]$
$\qquad \wedge\ mset\ xs = mset\ xs_1 + mset\ xs_2 \wedge xs_1 \leq xs_2)$

Given a list xs of sufficient length, two non-empty lists xs_1,xs_2 are returned, which together contain the same elements as xs, and each element of the first list is less than or equal to any element of the second list. We assume lists of at least four elements, because this is the size where the pivot selection makes sense (cf. Alg. 2). In practice, the lists will always be longer than the threshold, which is typically 16.

Finally, we can specify the partial Introsort algorithm in the Refinement Framework:

```
$partial\_introsort\ xs\ d \equiv$
  if $|xs| > threshold$ then {
    if $d{=}0$ then $sort\_spec\ xs$
    else {
      $(xs_1,xs_2) \leftarrow partition\_spec\ xs;$
      $xs_1 \leftarrow partial\_introsort\ xs_1\ (d{-}1);$
      $xs_2 \leftarrow partial\_introsort\ xs_2\ (d{-}1);$
      return $(xs_1 @ xs_2)$ }}
  else return $xs$
```

A straightforward Isabelle proof yields:

Theorem 5.1. $partial\_introsort\ xs\ d \leq part\_sort\_spec\ xs$

5.1 Refinement

The partial introsort algorithm that we have specified clearly captures the idea of the algorithm. However, it splits the input list and later concatenates it, which makes it unsuitable for an in-place implementation on arrays. In this section we will *refine* this algorithm into one that is better suited for an implementation on arrays. Note that refinement preserves correctness, such that the new algorithm will automatically be correct.

Given a relation R that relates *concrete* with *abstract* values, and a program m that returns abstract values, the program $\Downarrow R\ m$ is the biggest program that returns only concrete values with corresponding abstract values in m. Moreover, $\Downarrow R\ m$ may fail if and only if m may fail. Formally:

$$\Downarrow R\ FAIL \equiv FAIL$$
$$\Downarrow R\ (RES\ A) \equiv RES\ \{\ c.\ \exists a{\in}A.\ (c,a){\in}R\ \}$$

Note that for the identity relation I, we have $\Downarrow I\ m = m$.

Example 5.1. As a simple but constructed example, consider the specification of a program that returns some set of numbers divisible by n:

$$dvd\_set\_spec\ n \equiv \texttt{assert}\ (n{>}0);\ SPEC\ S.\ \forall m{\in}S.\ n\ dvd\ m$$

A possible implementation of this program is

$$dvd\_set_1\ n \equiv \texttt{return}\ \{n,2{*}n\}$$

and we have $dvd\_set_1\ n \leq dvd\_set\_spec\ n$. This restricts the number of possible results.

Now consider the relation $list\_set\_rel$ that relates distinct lists to the set of their elements:

$$list\_set\_rel \equiv \{\ (l,\ set\ l)\ |\ distinct\ l\ \}$$

We have $\texttt{return}\ [n,2{*}n] \leq \Downarrow list\_set\_rel\ (dvd\_set_1\ n)$. This changes the representation from sets to lists. Note that, by transitivity and monotonicity of $\Downarrow$ we also have:

$$\texttt{return}\ [n,2{*}n] \leq \Downarrow list\_set\_rel\ (dvd\_set\_spec\ n)$$

Intuitively, this limits the number of possible results and changes their representation at the same time.

We now refine the specifications and the introsort algorithm to work on a *slice* of a list rather than on the whole list. The relation $slice\_rel\ xsr\ l\ h$ relates a list with its slice from l to h. Outside the slice, the list must be equal to xsr:

$$slice\_rel\ xsr\ l\ h \equiv \{(xs,\ xs[l..{<}h])\ |\ l \leq h \wedge h \leq |xs|$$
$$\wedge\ |xs|{=}|xsr| \wedge xs[..{<}l] = xsr[..{<}l] \wedge xs[h..] = xsr[h..]$$
$$\}$$

We then define:

```
$slice\_sort\_spec\ xs\ l\ h \equiv \Downarrow (slice\_rel\ xs\ l\ h)$ (
  assert $(l \leq h \wedge h \leq |xs|);$
  $sort\_spec\ (xs[l..{<}h])$ )
```

That is, for valid ranges l,h, we return a list where the range is sorted, and the rest is unchanged. Partial sorting of a slice is specified analogously.

Similarly, we refine the specification for partitioning:

```
$slice\_partition\_spec\ xs\ l\ h \equiv \Downarrow(\{((xs',m),(xs_1,xs_2)).$
  $(xs',xs_1 @ xs_2){\in}slice\_rel\ xs\ l\ h \wedge m = l + |xs_1|\ \})$
  (assert $(l{\leq}h \wedge h{\leq}|xs|);\ partition\_spec\ (xs[l..{<}h])$))
```

This returns a pair of a list xs' and an index m. This list xs' contains the two partitions $xs_1 @ xs_2$ at range l, h, and is equal to the input list xs outside this range. Moreover, the index m is the index where the second partition starts in xs'.

Having defined the concrete specifications, we define a more concrete version of the Introsort algorithm:

```
$slice\_partial\_introsort\ xs\ l\ h\ d \equiv$
  assert $(l{\leq}h);$
  if $h{-}l > threshold$ then {
    if $d{=}0$ then $slice\_sort\_spec\ xs\ l\ h$
    else {
      $(xs,m) \leftarrow slice\_partition\_spec\ xs\ l\ h;$
      $xs \leftarrow slice\_partial\_introsort\ xs\ l\ m\ (d{-}1);$
      $xs \leftarrow slice\_partial\_introsort\ xs\ m\ h\ (d{-}1);$
      return $xs$ }}
  else return $xs$
```

To show that this algorithm refines $slice\_sort\_spec$, we first show that it refines $partial\_introsort$:

Theorem 5.2. $l{\leq}h \wedge h{\leq}|xs| \implies$
$slice\_partial\_introsort\ xs\ l\ h\ d$
$\leq \Downarrow (slice\_rel\ xs\ l\ h)\ (partial\_introsort\ (xs[l..{<}h])\ d)$

The proof is straightforward: First, we use the verification condition generator of the Refinement Framework[5]. This reduces the proof to a set of proof obligations on lists. With some help from the sledgehammer tool [5], these could be solved easily. For example, a crucial proof obligation relates the first abstract and recursively sorted partition with the respective slice of the concrete result:

[5]we had to do a bit of manual hinting to get the recursion refinement right.

```
lemma $(xs', xs_1 @ xs_2) \in slice\_rel\ xs\ l\ h$
    $\Longrightarrow xs_1 = xs'[l..<l+|xs_1|]$
  apply $(clarsimp\ simp: slice\_rel\_def)$
  by $(metis\ slice\_def\ add\_diff\_cancel\_left'\ append.assoc \ldots)$
```

The first line of the proof unfolds the definition of *slice_rel* and simplifies the goal. These simplification steps are standard, and usually don't require too much thinking. The second line of the proof was then automatically found by sledgehammer.

Finally, we combine the refinement statement (Thm 5.2) with the correctness result for *partial_introsort* (Thm 5.1). By transitivity and monotonicity of $\Downarrow$ we get correctness of *slice_partial_introsort*:

Theorem 5.3.
$slice\_partial\_introsort\ xs\ l\ h\ d \leq slice\_part\_sort\_spec\ xs\ l\ h$

Note how refinement allowed us to split this correctness proof into two independent parts: We first established the correctness of Introsort on a whole list, using a partition function that returns two separate lists. Although this *abstract* representation is not suitable for an in-place implementation with arrays, the correctness proof is much simpler than a direct correctness proof on slices. The second part of the proof is independent from the abstract correctness proof. It just shows how to implement the abstract algorithm on the *concrete* data representation.

The main Introsort algorithm is simple enough to define it on slices directly: The additional overhead introduced by refinement would outweigh the simplification of the proof.

We first specify the final sorting step to completely sort an already partially sorted slice[6]:

```
$slice\_final\_sort\_spec\ xs\ l\ h \equiv$
 assert $(h-l > 1 \land part\_sorted\ (xs[l..<h]))$;
 $slice\_sort\_spec\ xs\ l\ h$
```

The Introsort algorithm then simply combines partial sorting and final sorting:

```
$introsort\ xs\ l\ h \equiv$
  assert $(l \leq h)$;
  if $h-l>1$ then {
    $xs \leftarrow slice\_part\_sort\_spec\ xs\ l\ h$;
    $xs \leftarrow final\_sort\_spec\ xs\ l\ h$;
    return $xs$
  } else return $xs$
```

Its correctness proof is straightforward, and we get:

Theorem 5.4. $introsort\ xs\ l\ h \leq slice\_sort\_spec\ xs\ l\ h$

[6] We will check for an empty or singleton slice, which is trivially sorted, in the main algorithm. Thus, the additional precondition $h-l > 1$ allows us to omit this check in the final sorting algorithm.

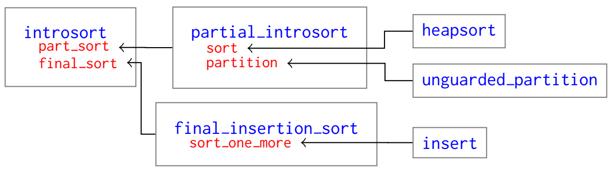

Figure 1. Modular verification of Introsort. The boxes show the different algorithms and the specifications they depend on. Arrows point from an algorithm to the specification that it implements.

6 Assembling the Implementation

Note that our *introsort* algorithm itself is independent from both, the partial and the final sorting algorithms: It only uses abstract specifications of them. However, in order to arrive at a concrete implementation, one has to *plug in* concrete implementations for these specifications. Similarly, our *slice_partial_introsort* algorithm depends on abstract specifications of partitioning and sorting (when the recursion depth limit is hit). Figure 1 shows the complete architecture of our Introsort implementation.

6.1 Unguarded Insertion Sort

We also use refinement for the implementation of insertion sort. As suggested in the pseudocode (Alg. 3), we use a Boolean variable to discriminate between the guarded and unguarded versions. Only in the last refinement step we will specialize the algorithm for both cases.

For the correctness proof, we first specify insertion as sorting one more element of an already sorted list:

```
$sort\_one\_more\_spec\ G\ xs\ i \equiv$
 assert $(i<|xs| \land sorted\ (xs[0..<i]))$;
 assert $(\neg G \longrightarrow 0<i \land \neg(xs[i] < xs[0]))$;
 spec $xs'.\ inres\ (slice\_sort\_spec\ xs\ 0\ (i+1))\ xs'$
      $\land (\neg G \longrightarrow xs'[0] = xs[0]))$
```

Assuming the initial segment of the list up to i is already sorted, afterwards, the segment up to $i+1$ will be sorted, and the rest of the list will be unchanged. We reuse *slice_sort_spec* for this (*inres m x* states that x is a possible result of m). In the unguarded case ($\neg G$), we assume that the first element of the list to be sorted is not greater than the element to be inserted, which guarantees that insertion will stop latest at this element. Moreover, we explicitly guarantee that the first element of the list will not change[7].

Using this specification, and its canonical refinement to slices, it is straightforward to prove correct the final insertion sorting algorithm (cf. Alg. 3). We now focus on the actual insertion procedure. In a first step, we refine the specification

[7] This is stronger than required, as in the general case of weak orderings, it would be fine to exchange the first element by an equivalent one. However, it holds for our implementation and slightly simplifies the proofs.

to a version that describes what the insertion procedure actually does, rather than focusing on the sorting:

```
insert_spec G xs i ≡
  assert (i<|xs| ∧ (¬G ⟶ 0<i ∧ ¬(xs[i] < xs[0])));
  spec xs'. ∃i'≤i.
    xs'=xs[0..<i'] @ xs[i] @ xs[i'..<i] @ xs[i<..]
  ∧ ∀j∈{i'<..i}. xs[i] < xs'[j]
  ∧ i'>0 ⟶ ¬(xs[i] < xs'[i'−1])
```

This precisely describes how the element at index i is moved backwards over greater elements. Its new index is i', and if this is not the beginning of the list, the element at $i'-1$ was not greater, such that the moving could be stopped.

This specification refines *sort_one_more_spec*:

Theorem 6.1. *insert_spec xs i ≤ sort_one_more_spec xs i*

The proof requires some basic arguments about relating equalities on list indices to equalities on lists and sets. Although straightforward in principle, it required a few auxiliary lemmas. Again, refinement allows us to do this reasoning in isolation, and then focus on how to implement the moving of the element.

We implement the insert procedure on whole lists first, and then refine it to slices beginning at index l. We only display the version on slices here:

```
slice_insert G xs l i ≡
  x ← xs[i];
  (xs,i) ← while_I (λ(xs,i). (¬G ∨ i>l) ∧ x<xs[i−1]) (λ(xs,i).
    xs ← xs[i:=xs[i−1]]
    return (xs,i−1) ) (xs,i)
  xs ← xs[i:=x]
  return xs
```

As this is a functional program, the variables xs and i have to be threaded through the loop explicitly. Moreover, instead of updating a single index, we update the whole list and rebind the updated list to the same name as the old list ($xs \leftarrow xs[i{:=}x]$). Finally, the loop is annotated with an invariant I. It is completely standard and not displayed here.

We easily prove that *slice_insert* refines *slice_insert_spec*, and, by transitivity, it also refines *slice_sort_one_more_spec*. Thus, we can plug it into the insertion sort algorithm, which, in turn, is plugged into the main Introsort algorithm (cf. Fig. 1).

6.2 Heapsort and Partition

The proofs of heapsort and partitioning follow a similar plot, and are not displayed here in full. We used some existing Isabelle proofs as guideline [21, 28, 31]. However, we point out one more interesting application of refinement: recall the sift-down function on heaps, that restores the heap property by *floating down* an element[8]. Usually, this function is presented by swapping the element with one of its children, until the heap property is restored (Alg. 4). However, the element that is written to a[right i] or a[left i] by the swap will get overwritten in the next loop iteration. Thus, a common optimization to save half of the writes is to store the element to be moved down in a temporary variable, and only assign it to its final position after the loop (Alg. 5). Note that the insertion procedure of insertion sort does a similar optimization. However, for the insertion procedure, it was feasible to prove the optimization together with the actual algorithm. For the slightly more complicated sift-down procedure, we first prove correct the simpler algorithm with swaps, and then refine it to the optimized version. Inside the loop, the refinement relation between the concrete array a' and the

```
procedure SIFT_DOWN($a$, $i$)
    while has right child $i$ do
        if $a$[left $i$] < $a$[right $i$] then
            if $a$[$i$] < $a$[right $i$] then
                SWAP($a$[$i$], $a$[right $i$])
                $i$ ← right $i$
            else return
        else if $a$[$i$] < $a$[left $i$] then
            SWAP($a$[$i$], $a$[left $i$])
            $i$ ← left $i$
        else return
    if has left child $i$ and $a$[$i$] < $a$[left $i$] then
        SWAP($a$[$i$], $a$[left $i$])
```

Algorithm 4. Pseudocode of sift-down function with swap

```
procedure SIFT_DOWN_OPT($a'$, $i$)
    $t$ ← $a'$[$i$]
    while has right child $i$ do
        if $a'$[left $i$] < $a'$[right $i$] then
            if $t$ < $a'$[right $i$] then
                $a'$[$i$] ← $a'$[right $i$]
                $i$ ← right $i$
            else return
        else if $t$ < $a$[left $i$] then
            $a'$[$i$] ← $a'$[left $i$]
            $i$ ← left $i$
        else return
    if has left child $i$ and $t$ < $a'$[left $i$] then
        $a'$[$i$] ← $a'$[left $i$]
        $i$ ← left $i$
    $a'$[$i$] ← $t$
```

Algorithm 5. Pseudocode of optimized sift-down function

[8]see, e.g., [13, Ch. 6] or [39, Ch. 2.4] for a description of heapsort.

abstract array a states that we obtain a as $a'[i{:=}t]$. The proof that the optimized version refines the version with swaps requires only about 20 lines of straightforward Isabelle script.

7 Imperative Implementation

In the previous sections, we have presented a refinement based approach to verify the Introsort algorithm, including most optimizations we found in its GNU C++ Library implementation. However, the algorithm is still expressed as a nondeterministic monadic program that modifies functional lists. In this section, we discuss how the Isabelle LLVM framework [29] is used to (semi-)automatically transfer the algorithm into an LLVM program on arrays.

7.1 The Sepref Tool

The idea of the Sepref tool [27, 29] is to symbolically execute an abstract program in the *nres*-monad, keeping track of refinements of every abstract variable to a concrete representation that may use pointers to dynamically allocated memory. During this symbolic execution, the tool can synthesize an imperative program in the Isabelle-LLVM semantics, together with a refinement proof. The tool works automatically, but usually requires some program-specific setup and boilerplate. For a detailed discussion of the Sepref tool and the Isabelle-LLVM framework, we refer the reader to [27, 29].

Isabelle-LLVM comes with standard setup to refine lists to arrays. List update operations are refined to destructive array element updates, as long as the old version of the abstract list is not used any more after the update. Note that, by rebinding the updated list to the same variable name as the original list[9], this property is syntactically ensured. Moreover, Isabelle-LLVM provides setup to refine the unbounded integers of Isabelle/HOL to bounded integers. The resulting proof obligations to show that the integers are in bounds are discharged automatically, but typically require some hinting from the user. A common technique to provide such hints is to insert additional assertions into the abstract program. Usually, these can be proved easily. For example, in the definition of *slice_partial_introsort* (Sec. 5.1), we have inserted the assertion $l \leq h$ at the very beginning. This is required by the Sepref tool to prove that the operation $h-l$ in the next line cannot underflow. This assertion adds a proof obligation to the proof that *slice_partial_introsort* refines *slice_part_sort_spec* (Thm. 5.3), which is, however, trivial, as the specification explicitly states $l \leq h$ as precondition. On the other hand, when refining *slice_partial_introsort* to an implementation with bounded integers, one can assume $l \leq h$, thus discharging the non-underflow proof obligation for the $h-l$ operation. Thus, assertions provide a convenient tool to pass properties down the refinement chain.

Using the Sepref tool, it is straightforward to refine the Introsort algorithm to an Isabelle-LLVM program that sorts arrays of integers. For example, Figure 2 shows the Isabelle-LLVM code that is generated for the *insert* procedure with *G=false* (cf. 6.1). Moreover, the Sepref tool will generate a theorem that the generated program actually implements the abstract one:

```
ug_insert_impl ≡ λxs l i. doM {
   x ← array_nth xs i;
   (xs, i) ← llc_while (λ(xs, i). doM {
      bi ← ll_sub i 1;
      t ← array_nth xs bi;
      ll_icmp_ult x t
   }) (λ(xs, i). doM {
      i' ← ll_sub i 1;
      t ← array_nth xs i';
      xs ← array_upd xs i t;
      i ← ll_sub i 1;
      return (xs, i)
   }) (xs, i);
   array_upd xs i x
 }
```

Figure 2. Implementation for *insert* procedure with *G=false* generated by the Sepref tool, for 64bit signed integer elements. This definition is within the executable fragment of Isabelle-LLVM, i.e., the Isabelle LLVM code generator can, after preprocessing, translate it to LLVM-IR. Moreover, note that the function does not depend on the lower bound parameter l any more, as this was only required in the guarded version.

Theorem 7.1. (*ug_insert_impl, slice_insert False*)
$: arr\_u64^d \times snat64^k \times snat64^k \to arr\_u64$

This specifies the relations for the parameters and the result, where *arr_u64* relates arrays of unsigned 64-bit integers with lists of integers, and *snat64* relates non-negative 64-bit integers with natural numbers. The $\cdot^d$ annotation means that the parameter will be *destroyed* by the function call, while $\cdot^k$ means that the parameter is *kept*. Here, the insertion is done in place, such that the input list has no corresponding refinement after the function call.

The final correctness statement for our Introsort implementation (for integers) is

Theorem 7.2. (*introsort_impl, slice_sort_spec*)
$: arr\_u64^d \times snat64^k \times snat64^k \to arr\_u64$

Here, *introsort_impl* is the Isabelle-LLVM program generated by Sepref from *introsort* (cf. Sec. 5.1). This theorem combines the refinement theorem generated by Sepref, and Theorem 5.4.

[9]as e.g. in $xs \leftarrow xs[i{:=}x]$, cf. Sec. 6.1

```
define i64* @ug_insert_impl(i64* %xs, i64 %l, i64 %i) {
start:
  %x3 = getelementptr i64, i64* %xs, i64 %i
  %r = load i64, i64* %x3, align 8
  br label %while_start

while_start:
  %bib.sink = phi i64 [ %bib, %while_body ], [ %i, %start ]
  %bib = add i64 %bib.sink, −1
  %xda = getelementptr i64, i64* %xs, i64 %bib
  %ra = load i64, i64* %xda, align 8
  %xg = icmp ult i64 %r, %ra
  %p1 = getelementptr i64, i64* %xs, i64 %bib.sink
  br i1 %xg, label %while_body, label %while_end

while_body:
  store i64 %ra, i64* %p1, align 8
  br label %while_start

while_end:
  store i64 %r, i64* %p1, align 8
  ret i64* %xs
}
```

Figure 3. LLVM-IR generated by Isabelle-LLVM for the definition from Figure 2. We display the code after an LLVM optimization pass that eliminates some redundant operations generated by Isabelle-LLVM, and thus makes the code more readable. Note that we kept the unused parameter %l. LLVM will eliminate it by inlining the function.

7.2 The Isabelle-LLVM Code Generator

Isabelle-LLVM comes with a code generator that exports this program to actual LLVM text, which is subsequently compiled using the LLVM toolchain. Fig. 3 shows the code generated for the *slice_insert* procedure.

To make the generated programs usable, one has to link them to C wrappers, that handle parsing of command line options and printing of results. However, the original Isabelle-LLVM framework provides no support for this: one has to manually write a C header file, that hopefully matches the object file generated by the LLVM compiler. If it doesn't, the program has undefined behaviour[10]. What makes the situation even worse is, that the C compiler sometimes changes function signatures: For example, a function with a non-primitive return value (e.g. a structure) is changed to a void function with an extra reference parameter for the return value. LLVM, on the other hand, does not perform this change, which

[10]In practice, this means it will probably SEGFAULT. However, it also might return wrong results, or be prone to various kinds of exploits.

Isabelle command

```
export_llvm
  str_sort_introsort_impl
    is llstring* str_introsort(llstring*, int64_t, int64_t)
  defines ‹
    typedef struct {
      int64_t size; struct {int64_t capacity; char *data;};
    } llstring; ›
```

Generated C header

```
typedef struct {
  int64_t size;
  struct {
    int64_t capacity;
    char*data;
  };
} llstring;

llstring* str_introsort(llstring*, int64_t, int64_t);
```

Figure 4. An example Isabelle command to export LLVM code for a string sorting function, and the generated header file. Strings are represented as dynamic arrays of characters. Isabelle will refuse to export functions whose signatures would be changed by C. Moreover, structure fields in LLVM are indexed by consecutive numbers, rather than named by meaningful names. Our extension allows to define properly named structures for the header file, and checks that these definitions are actually compatible with the LLVM signatures.

makes the function generated by LLVM incompatible with the function call generated by the C compiler.

To this end, we extended Isabelle-LLVM to also generate a header file for the exported functions, refusing to export functions whose signatures would be altered by the C compiler. Figure 4 shows the extended Isabelle LLVM export command and the generated header file for our Introsort algorithm on strings.

7.3 Separation Logic and Ownership

Internally, the Sepref tool represents the symbolic state, which contains all abstract variables and their refinements to concrete variables, as an assertion in separation logic [11, 37]. For example, the assertion

$array\ xs_1\ p_1 * array\ xs_2\ p_2$

describes a symbolic state that refines the two abstract list variables xs_1 and xs_2 by two arrays at addresses p_1 and p_2. Separation logic guarantees that the arrays at p_1 and p_2 share no common memory, i.e., updates to the first array cannot change the second array. As long as the array elements are primitive values like integers, this works as expected, and a list of integers can be refined to an array of integers. An array lookup operation will simply return a copy of the integer value at the specified index. It can be specified by the following Hoare-triple, which can be used by Sepref to synthesize LLVM code for the operation $r \longleftarrow xs[i]$.

$$\{\ \mathit{array}\ xs\ p * i{<}|xs|\ \}\ r = p[i]\ \{\ \mathit{array}\ xs\ p * r{=}xs[i]\ \}$$

Note that we use C like notation for the program, and abstracted away the fact that the i in the program is a bounded integer register from LLVM, while the i in the assertions refers to its content as an unbounded Isabelle integer.

Now, let's assume we have a list of strings, where a string is itself a list of characters. The following assertion refines both, the outer and the inner lists to arrays:

$$\begin{aligned}&\mathit{sarray}\ xs\ p \equiv \exists ys.\\ &\quad |ys|{=}|xs| * \mathit{array}\ ys\ p * \mathit{array}\ xs_1\ ys_1 * \ldots\ \mathit{array}\ xs_n\ ys_n\end{aligned}$$

Here, ys represents the list of string pointers, which point to their own separate array each. The canonical Hoare triple for array lookup would be:

$$\{\ \mathit{sarray}\ xs\ p * i{<}|xs|\ \}\ r = p[i]\ \{\ \mathit{sarray}\ xs\ p * \mathit{array}\ (xs[i])\ r\ \}$$

However, this does not hold, as the inner array at position i is now pointed to twice: by r and by the $ys[i]$ from within the *sarray* assertion. The only way to make a Hoare triple of this form valid is to make a copy of the inner array, which would come with an unacceptable performance penalty.

The Rust Programming Language [38] has to solve a similar problem. There, they use the concept of borrowing: the inner array is *borrowed* from the outer one, i.e., ownership is transferred to the variable r. Only at the end of the lifetime of r, ownership will return to the outer array[11]. We realize a similar concept in Sepref, by refining abstract lists of optional values to arrays, that only own those elements which are present (not None) in the abstract list[12]:

$$\begin{aligned}&\mathit{oarray}\ xs\ p \equiv \exists ys.\ \mathit{array}\ ys\ p * A'\ xs_1\ ys_1 * \ldots\ * A'\ xs_n\ ys_n\\ &\mathit{where}\\ &A'\ \mathit{None}\ \_ = \square\ \ |\ \ A'\ (\mathit{Some}\ x)\ y = \mathit{array}\ x\ y\end{aligned}$$

We define abstract operations to borrow and return (insert) elements, as well as to convert between a standard list and a list with explicit ownership:

```
borrow xs i ≡ assert ($i{<}|xs| \land xs[i]{\neq}None$);
          return ($the\ (xs[i]),\ xs[i{:=}None]$)
ins xs i x ≡ assert ($i{<}|xs| \land xs[i]{=}None$);
          return $xs[i{:=}Some\ x]$
to_olist xs = return ($map\ Some\ xs$)
to_slist xs = assert ($None \notin xs$); return ($map\ the\ xs$)
```

Here, $\mathit{the}\ (\mathit{Some}\ x) = x$ is the selector function of the option data type. The *borrow* function assumes that the element is still present in the list, and then returns the element and a new list where the element is gone. Symmetrically, the *ins* function assumes that there is no element present at the specified index. The *to_olist* and *to_slist* functions convert between lists of plain values and lists of optional values, where *to_slist* assumes that all values in the list are actually present.

This approach allows us to explicitly model ownership transfer of array elements in the *nres* monad, and let Sepref generate efficient code, sparing explicit copies of elements.

The necessary conversion operations can usually be inserted by a straightforward refinement step. For example, we refine the *insert* function (Sec. 6.1) as follows:

```
is_insert_o G xs' i ≡
  xs' ⟵ to_olist xs';
  (x,xs') ⟵ borrow xs' i;
  (xs',i) ⟵ while
    ($\lambda(xs',i).\ (\neg G \lor i{>}0) \land xs' < xs'[i{-}1]$) ($\lambda(xs',i)$.
      (t,xs') ⟵ borrow xs' (i–1);
      xs' ⟵ ins xs' i t;
      return (xs', i–1)
    ) (xs',i);

  xs' ⟵ ins xs' i x;
  return (to_slist xs')
}
```

First, this function converts the input list xs' to one with explicit ownership. It then borrows the element i from this list. In each loop iteration, it moves an element from $i{-}1$ to i, by borrowing and immediately re-inserting it. Finally, the element borrowed at the beginning is re-inserted, and the list is converted back to implicit ownership. The refinement proof is, again, straightforward. The refinement relation for the loop states that the abstract and concrete array contain the same values, except at index i, where the concrete value is not present:

$$\begin{aligned}&|xs'| = |xs|\\ &\land (\forall j{<}|xs|.\ j{\neq}i \longrightarrow xs'[j] = \mathit{Some}\ (xs[j])) \land xs'[i]{=}\mathit{None}\}\end{aligned}$$

Using this explicit ownership extension to Sepref, we can also synthesize the Introsort algorithm for arrays of arrays of characters, where the inner arrays are compared lexicographically.

[11] Actually, borrowing in Rust does not work for array elements, as the type-checking would become undecidable. Fortunately, in an interactive theorem prover, we can burden the user with parts of this type checking.

[12] For simplicity, the presentation sticks to the array of strings example. Actually, our implementation is parameterized over the refinement for the inner type. We believe that a further generalization to include outer types other than lists (e.g. maps) is possible, but leave it for future work.

8 Benchmarks

The goal of this paper was to verify an existing real-world implementation of a medium-complex algorithm. We chose the Introsort implementation from the GNU C++ library. As we did not directly verify the existing C++ code, but re-implemented the algorithm using Isabelle-LLVM, we have to test how close we came to the performance of the original.

Our benchmark set is based on the Boost C++ Library's [8] sorting algorithm benchmarks, extended with further benchmarks indicated in [4]. Apart from sorting a random list of elements that are mostly different (random), we also sort lists of length n that contain only $n/10$ different elements (random-dup-10), and random lists of only two different elements (random-boolean), as well as lists where all elements are equal (equal). We also consider already sorted sequences (sorted, rev-sorted), as well as a sequence of $n/2$ elements in ascending order, followed by the same elements in descending order (organ-pipe). We also consider sorted sequences where we applied $pn/100$ random swap operations (almost-sorted-p). Finally, we consider sorted sequences with $pn/100$ random elements inserted at the end or in the middle ([rev-]sorted-end-p, [rev-]sorted-middle-p).

We sorted integer arrays with $n = 10^8$ elements, and string arrays with $n = 10^7$ elements. To represent strings, we used dynamic arrays of characters, and used the same compare function for our algorithm and std::sort, in order to eliminate differences in the compare function implementation[13].

We compile both, our verified algorithm and the std::sort benchmark with clang-6.0.0. The benchmarks were run on a laptop with an Intel(R) Core(TM) i7-8665U CPU and 32GiB of RAM, as well as on a server machine with 24 AMD Opteron 6176 cores and 128GiB of RAM. Ideally, the sorting algorithm should take exactly the same time when run on the same data and machine. However, in practice, we encountered some noise. We encountered both, *high frequency* noise, that slowed down single sorting runs, and *low frequency* noise, that slowed down the machine for several minutes. To counter the high-frequency noise, we ran each experiment 10 times in a row, dropped the 6 extremal results, and report the average of the remaining 4 results. If the smallest remaining result was more than 2% smaller than the largest remaining result, we repeated the experiment. Countering the low-frequency noise is more time consuming. For all benchmarks, we repeated the slower experiment when the difference between our algorithm and std::sort was more than 15%[14]. Moreover, for the random and organ-pipe integer benchmarks, we ran all the experiments in a loop for several hours, and report the intervals of the averages of the median 4 values. The results are displayed in Tables 1 and 2.

[13]Initially, we used std::string elements for std::sort, for which the compare function seems to be significantly slower than our compare function on dynamic arrays.

[14]The outlier for sorted-end-1 on the server machine seems to be genuine. We have no explanation for that.

Data Set	uint64		llstring	
	Lap	Srv	Lap	Srv
random	−0.5	2.4	−14.5	−2.1
random-dup-10	−0.2	3.2	−1.2	−1.7
random-boolean	−0.1	−0.7	4.7	5.3
equal	−13.3	0.9	6.9	4.0
sorted	6.6	3.0	2.2	4.2
rev-sorted	1.9	3.8	2.1	2.0
organ-pipe	−8.1	12.8	7.0	4.9
almost-sorted-10	3.8	1.3	−0.8	−2.0
almost-sorted-.1	3.2	1.7	2.3	−0.7
almost-sorted-1	−8.7	2.2	−11.1	−1.1
almost-sorted-50	0.0	2.5	−12.2	−1.8
sorted-end-10	−4.3	1.3	−1.0	−1.4
sorted-end-.1	−4.0	8.3	−4.7	11.9
sorted-end-1	−5.2	19.9	10.4	5.6
sorted-middle-10	−1.1	3.2	−0.7	−2.7
sorted-middle-1	12.5	4.5	5.4	2.1
sorted-middle-.1	−2.8	4.4	6.1	6.6
rev-sorted-end-10	1.1	3.1	−1.5	−2.4
rev-sorted-end-.1	5.6	8.7	0.9	5.4
rev-sorted-end-1	9.7	3.9	−12.6	0.6
rev-sorted-middle-10	1.2	6.2	−14.3	−0.6
rev-sorted-middle-1	−2.3	3.6	−12.0	0.9
rev-sorted-middle-.1	−5.7	4.2	3.4	5.5

Table 1. Benchmark results, as the ratio $(i/c-1)*100$, where i and c are the average wall-times of verified Introsort and std::sort. That is, negative values indicate that Introsort was faster, and positive values indicate that std::sort was faster.

Algorithm	random			organ-pipe		
	min	max	r	min	max	r
isabelle::sort	7.35	8.58	16.7	8.43	9.86	17.0
std::sort	7.38	8.63	17.0	8.97	10.49	16.9

Table 2. Intervals of the results (wall time in seconds) for repeating each experiment 74 times, in an interleaved fashion. The r fields display $(\max/\min-1)*100$, i.e., the value that would appear in Table 1 when comparing the slowest and fastest run of the same experiment.

The results clearly show that our verified sorting algorithm performs on-par with the GNU C++ Library's implementation, within the expected noise of up to 17%. Table 2 even indicates that our algorithm performs slightly better on our laptop machine. We leave it to future work to do more precise measurements on different machines.

9 Conclusions

In this paper we have verified the state-of-the art Introsort sorting algorithm using Isabelle-LLVM. We included most optimizations found in the GNU C++ Library's (libstdc++) implementation. On an extensive set of benchmarks, our verified implementation performs on par with the libstdc++ implementation.

The verification took us about 100 person hours distributed over 4 weeks. After two weeks and roughly 60 person hours, we had arrived at an algorithm that was only about 10% slower than std::sort. However, it only worked for integers, and always used guarded insertion sort. The remaining time was spent on implementing explicit ownership for Sepref, such that we could sort strings and other objects, and on verifying the unguarded insertion sort optimization.

This shows that Isabelle-LLVM can be used to generate competitive verified implementations of medium-complex algorithms. The Refinement Framework has been successfully used for much larger projects, e.g., model-checkers [10, 16, 41]. However, these developments produce implementations that use arbitrary precision integers and target Standard ML, which means a principal performance deficit compared to more low-level implementations. Using Isabelle-LLVM as backend to the Refinement Framework, these limitations can now be overcome. Thanks to the modular structure of the Refinement Framework, we can even hope to gradually transform existing verification projects to Isabelle-LLVM, e.g., replacing invocations of functional mergesort on lists by our Introsort algorithm on arrays in a first step.

While we can easily instantiate our sorting algorithm to arbitrary weak orderings on arbitrary element types *within* the Refinement Framework, we cannot so easily replace C++'s std::sort. The reason is that std::sort is a template which is instantiated by the C++ compiler, i.e., before LLVM intermediate code is even generated. Verifying the std::sort template itself would require a formal semantics of C++, including its template mechanism and the relevant traits from the standard template library (orderings, iterators, etc.). To the best of our knowledge, such a semantics has not yet been formalized, let alone being used to verify non-trivial algorithms. However, we can *specialize* the std::sort template to use our Introsort algorithm for any fixed combination of element type and ordering, on containers backed by an array (e.g. std::vector).

Finally, the trusted code base of Isabelle-LLVM is relatively small, even compared to other common program verification methods. Following the LCF tradition [20], the Isabelle prover is designed around a small inference kernel, which is the only code whose bugs can affect the correctness of proved theorems. The Isabelle-LLVM framework contains a relatively straightforward formalization of the LLVM semantics, with a straightforward code generator [29] to LLVM intermediate language. Finally, the LLVM compiler is an industrial strength, well-tested, and widely used compiler. In contrast, the default code generators of most theorem provers target high level languages, which includes the compilers and the high-level language semantics into the trusted code base. Deductive verification tools like KeY [2] depend on the correct axiomatization of the highly complex Java semantics, as well as on several automatic theorem provers, which, themselves, are highly complex and optimized C programs.

To reduce the trusted code base even further, one can include a compiler correctness proof in the theorem prover: The Verifiable C project [1] provides a verification infrastructure on top of the verified CompCert compiler [6]. However, to the best of our knowledge, this has only been applied to verify rather simple programs [12]. The CakeML project [25, 34] has developed a code generator for HOL4 and a verified ML compiler. However, the performance of functional high-level languages seems to be inherently limited wrt. low-level imperative languages, and we are not aware of any high-performance algorithm that has been verified using CakeML.

9.1 More Related Work

Sorting algorithms are a standard benchmark for program verification tools, such that we cannot give an exhaustive overview. Nevertheless, we discuss a few notable examples: The arguably first formal proof of quicksort was given by Foley and Hoare himself [18], though, due to the lack of powerful enough theorem provers at these times, it was only done on paper.

One of the first mechanical verifications of imperative sorting algorithms is by Filliâtre and Magaut [17], who prove correct imperative versions of quicksort, heapsort, and insertion sort in Coq. However, they do not report on code generation or benchmarking, nor do they combine their separate algorithms to Introsort.

The Timsort algorithm, which was used in the Java standard library, has been verified with the KeY tool [14]. A bug was found and fixed during the verification. Subsequently, KeY has been used to also verify the dual-pivot quicksort algorithm from the Java standard library [3]. This time, no bugs were found.

9.2 Future Work

Actually, the C++ standard requires the std::sort algorithm to not only be functional correct, but to also have a worst-case complexity of $O(n \log n)$. Recently, the Refinement Framework has been extended to reason about time complexity [22], and it has been proposed that Isabelle-LLVM could be used as a backend. This would enable us to also formally prove the worst-case complexity of Introsort.

Another obvious direction is to formalize other state-of-the-art implementations of sorting algorithms, for example std::stable_sort, or Boost's pattern defeating quicksort [8] with branch prediction optimization [15].

Finally, we proposed an explicit ownership model for nested lists. However, we only support write-ownership. A next step would be to include read-ownership, which allows aliasing on read-only objects. Formally, this could be realized with fractional permissions [9].

Acknowledgements Hidden for blind review.

References

[1] Andrew W. Appel, Robert Dockins, Aquinas Hobor, Lennart Beringer, Josiah Dodds, Gordon Stewart, Sandrine Blazy, and Xavier Leroy. 2014. *Program Logics for Certified Compilers*. Cambridge University Press, New York, NY, USA.

[2] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. 2007. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg.

[3] Bernhard Beckert, Jonas Schiffl, Peter H. Schmitt, and Mattias Ulbrich. 2017. Proving JDK's Dual Pivot Quicksort Correct. In *VSTTE*.

[4] Jon L. Bentley and M. Douglas McIlroy. 1993. Engineering a Sort Function. *Softw. Pract. Exper.* 23, 11 (Nov. 1993), 1249–1265. https://doi.org/10.1002/spe.4380231105

[5] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C. Paulson. 2013. Extending Sledgehammer with SMT Solvers. *J. Autom. Reasoning* 51, 1 (2013), 109–128. https://doi.org/10.1007/s10817-013-9278-5

[6] Sandrine Blazy and Xavier Leroy. 2009. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning* 43, 3 (2009), 263–288. http://xavierleroy.org/publi/Clight.pdf

[7] Joshua Bloch. [n. d.]. Extra, Extra - Read All About It: Nearly All Binary Searches and Mergesorts are Broken. ([n. d.]). http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html

[8] boost [n. d.]. Boost C++ Libraries. ([n. d.]). https://www.boost.org/

[9] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 2005. Permission Accounting in Separation Logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, NY, USA, 259–270. https://doi.org/10.1145/1040305.1040327

[10] Julian Brunner and Peter Lammich. 2018. Formal Verification of an Executable LTL Model Checker with Partial Order Reduction. *J. Autom. Reasoning* 60, 1 (2018), 3–21. https://doi.org/10.1007/s10817-017-9418-4

[11] C. Calcagno, P.W. O'Hearn, and Hongseok Yang. 2007. Local Action and Abstract Separation Logic. In *LICS 2007*. 366–378.

[12] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 2018. VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs. *Journal of Automated Reasoning* 61 (02 2018). https://doi.org/10.1007/s10817-018-9457-5

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[14] Stijn de Gouw, Jurriaan Rot, Frank S. de Boer, Richard Bubel, and Reiner Hähnle. 2015. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In *CAV*.

[15] Stefan Edelkamp and Armin Weiß. 2016. BlockQuicksort: How Branch Mispredictions don't affect Quicksort. *CoRR* abs/1604.06697 (2016). arXiv:1604.06697 http://arxiv.org/abs/1604.06697

[16] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. 2013. A Fully Verified Executable LTL Model Checker. In *CAV*. LNCS, Vol. 8044. Springer, 463–478.

[17] Jean-Christophe Filliâtre and Nicolas Magaud. 1999. Certification of Sorting Algorithms in the Coq System. (1999). https://www.lri.fr/~filliatr/ftp/publis/Filliatre-Magaud.ps.gz

[18] M. Foley and C. A. R. Hoare. 1971. Proof of a recursive program: Quicksort. *Comput. J.* 14, 4 (01 1971), 391–395. https://doi.org/10.1093/comjnl/14.4.391 arXiv:http://oup.prod.sis.lan/comjnl/article-pdf/14/4/391/927573/140391.pdf

[19] GNU C++ Library [n. d.]. The GNU C++ Library. ([n. d.]). https://gcc.gnu.org/onlinedocs/libstdc++/ Version 7.4.0.

[20] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. 1979. *Edinburgh LCF: A Mechanised Logic of Computation*. LNCS, Vol. 78. Springer.

[21] Simon Griebel. 2019. Binary Heaps for IMP2. *Archive of Formal Proofs* (June 2019). http://isa-afp.org/entries/IMP2_Binary_Heap.html, Formal proof development.

[22] Maximilian P.L. Haslbeck and Peter Lammich. 2019. Refinement with Time – Refining the Run-time of Algorithms in Isabelle/HOL. In *ITP2019: Interactive Theorem Proving*.

[23] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321–. https://doi.org/10.1145/366622.366644

[24] Nicolai M. Josuttis. 2012. *The C++ Standard Library: A Tutorial and Reference* (2nd ed.). Addison-Wesley Professional.

[25] YONG KIAM TAN, MAGNUS O. MYREEN, RAMANA KUMAR, ANTHONY FOX, SCOTT OWENS, and MICHAEL NORRISH. 2019. The verified CakeML compiler backend. *Journal of Functional Programming* 29 (2019), e2. https://doi.org/10.1017/S0956796818000229

[26] Alexander Krauss. 2010. Recursive definitions of monadic functions. In *Proc. of PAR*, Vol. 43. 1–13.

[27] Peter Lammich. 2015. Refinement to Imperative/HOL. In *ITP*. LNCS, Vol. 9236. Springer, 253–269.

[28] Peter Lammich. 2016. Refinement based verification of imperative data structures. In *CPP 2016*, Jeremy Avigad and Adam Chlipala (Eds.). ACM, 27–36.

[29] Peter Lammich. 2019. Generating Verified LLVM from Isabelle/HOL. In *10th International Conference on Interactive Theorem Proving (ITP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, John Harrison, John O'Leary, and Andrew Tolmach (Eds.), Vol. 141. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 22:1–22:19. https://doi.org/10.4230/LIPIcs.ITP.2019.22

[30] Peter Lammich and Thomas Tuerk. 2012. Applying Data Refinement for Monadic Programs to Hopcroft's Algorithm. In *ITP 2012 (LNCS)*, Lennart Beringer and Amy P. Felty (Eds.), Vol. 7406. Springer, 166–182.

[31] Peter Lammich and Simon Wimmer. 2019. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs* (Jan. 2019). http://isa-afp.org/entries/IMP2.html, Formal proof development.

[32] libc++ [n. d.]. "libc++" C++ Standard Library. ([n. d.]). https://libcxx.llvm.org/

[33] DAVID R. MUSSER. 1997. Introspective Sorting and Selection Algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993. https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#

[34] Magnus O. Myreen and Scott Owens. 2014. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.* 24, 2-3 (2014), 284–315. https://doi.org/10.1017/S0956796813000282

[35] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer.

[36] Tim Peters. [n. d.]. Original Description of Timsort. ([n. d.]). https://bugs.python.org/file4451/timsort.txt Accessed 2019-10-21.

[37] John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc of. Logic in Computer Science (LICS)*. IEEE, 55–74.

[38] Rust Programmin Language [n. d.]. The Rust Programmin Language. ([n. d.]). https://www.rust-lang.org/

[39] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms* (4th ed.). Addison-Wesley Professional.

[40] J. W. J. WILLIAMS (Ed.). 1964. Algorithm 232: Heapsort. *Commun. ACM* 7, 6 (June 1964), 347–349. https://doi.org/10.1145/512274.512284

[41] Simon Wimmer and Peter Lammich. 2018. Verified Model Checking of Timed Automata. In *TACAS 2018*. 61–78.