Efficient Verified Implementation of Introsort and Pdqsort

Peter Lammich

The University of Manchester, UK

Abstract. Sorting algorithms 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. The Boost C++ Libraries implement pdqsort, an extension of introsort that achieves linear runtime on inputs with certain patterns.
We verify introsort and pdqsort in the Isabelle LLVM verification framework, closely following the state-of-the-art implementations from GNU and Boost. On an extensive benchmark set, our verified implementations perform on par with the originals.

1 Introduction

Sorting algorithms are an important part of any standard library. The GNU C++ Library (libstdc++) [15] implements Musser's introspective sorting algorithm (introsort) [28]. It is a combination of quicksort, heapsort, and insertion sort, which has the fast average case runtime of quicksort and the optimal $O(n\log(n))$ worst-case runtime of heapsort. The Boost C++ Libraries [6] provide a state-of-the-art implementation of *pattern-defeating quicksort* (pdqsort) [29], an extension of introsort to achieve better performance on inputs that contain certain patterns like already sorted sequences. Verification of these algorithms and their state-of-the-art implementations is far from trivial, but turns out to be manageable when handled with adequate tools.

Sorting algorithms in standard libraries have not always been correct. The timsort [30] 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 [10]. Also, many real-world mergesort implementations suffered from an overflow bug [5]. Finally, LLVM's libc++ [26] 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].

In this paper, we present efficient implementations of introsort and pdqsort that are verified down to their LLVM intermediate representation [27]. The

[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 April 2020.

verification uses the Isabelle Refinement Framework [24], and its recent Isabelle-LLVM backend [23]. We also report on two extensions of Isabelle-LLVM, to handle nested container data structures and to automatically generate C-header files to interface the generated code. Thanks to the modularity of the Isabelle Refinement Framework, our verified algorithms can easily be reused in larger verification projects.

While sorting algorithms are a standard benchmark for theorem provers and program verification tools, verified real-world implementations seem to be rare: apart from our work, we are only aware of two verified sorting algorithms [10, 3] from the Java standard library.

The complete Isabelle/HOL formalization and the benchmarks are available at `http://www21.in.tum.de/~lammich/isabelle_llvm/`.

2 The Introsort and Pdqsort Algorithms

The introsort algorithm by Musser [28] is a generic unstable sorting algorithm that combines the good average-case runtime of quicksort [18] with the optimal $O(n \log(n))$ worst-case complexity of heapsort [1]. The basic idea is to use quicksort as main sorting algorithm, insertion sort for small partitions, and heapsort when the recursion depth exceeds a given limit, usually $2\lfloor \log_2 n \rfloor$ for n elements.

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

Algorithm 1: Introsort

Algorithm 1 shows our implementation of introsort, which closely follows the implementation in libstdc++ [15]. The function INTROSORT sorts the slice from index l (inclusive) up to index h (exclusive) of the list[3] xs. If there is more than one element (line 2), it initializes a depth counter and calls the function INTROSORT_AUX (line 3), which partially sorts the list such that every element is no more than threshold positions away from its final position in the sorted

[3] Our formalization initially uses lists to represent the sequence of elements to be sorted, and refines them to arrays later (cf. Sec. 4).

list. The remaining sorting is then done by insertion sort (line 4). The function INTROSORT_AUX implements a recursive quicksort scheme: recursion stops if the slice becomes smaller than the threshold (line 6). If the maximum recursion depth is exhausted, heapsort is used to sort the slice (line 7). Otherwise, the slice is partitioned (line 9), and the procedure is recursively invoked for the two partitions (line 10–11). Here, PARTITION_PIVOT moves the pivot element to the first element of the left partition, and returns the start index of the right partition.

Note that we do not try to invent our own implementation, but closely follow the existing (and hopefully well-thought) libstdc++ implementation. This includes the slightly idiosyncratic partitioning scheme, which leaves the pivot-element as first element of the left partition. Moreover, the libstdc++ implementation contains a manual tail-call optimization, replacing the recursive call in line 11 by a loop. While we could easily add this optimization in an additional refinement step, it turned out to be unnecessary, as LLVM recognizes and eliminates this tail call automatically.

```
 1: procedure PDQSORT($xs, l, h$)
 2:     if $h - l > 1$ then PDQSORT_AUX(true, $xs, l, h, \log(h - l)$)
 3: procedure PDQSORT_AUX($lm, xs, l, h, d$)
 4:     if $h - l <$ threshold then INSORT($lm, xs, l, h$)
 5:     else
 6:         PIVOT_TO_FRONT($xs, l, h$)
 7:         if $\neg lm \wedge xs[l-1] \not< xs[l]$ then
 8:             $m \leftarrow$ PARTITION_LEFT($xs, l, h$)
 9:             assert $m + 1 \leq h$
10:             PDQSORT_AUX(false, $xs, m + 1, h, d$)
11:         else
12:             $(m, ap) \leftarrow$ PARTITION_RIGHT($xs, l, h$)
13:             if $m - l < \lfloor (h - l)/8 \rfloor \vee h - m - 1 < \lfloor (h - l)/8 \rfloor$ then
14:                 if $--d = 0$ then HEAPSORT(xs,l,h); return
15:                 SHUFFLE(xs,l,h,m)
16:             else if $ap \wedge$ MAYBE_SORT($xs, l, m$) $\wedge$ MAYBE_SORT($xs, m + 1, h$) then
17:                 return
18:             PDQSORT_AUX($lm, xs, l, m, d$)
19:             PDQSORT_AUX(false, $xs, m + 1, h, d$)
```

Algorithm 2: Pdqsort

Algorithm 2 shows our implementation of pdqsort. As for introsort, the wrapper PDQSORT just initializes a depth counter, and then calls the function PDQSORT_AUX (line 2), which, in contrast to introsort, completely sorts the list, such that no final insertion sort is necessary. Again, the PDQSORT_AUX function implements a recursive quicksort scheme, however, with a few additional optimizations. Slices smaller than the threshold are sorted with insertion sort (line 4). If the current slice is not the leftmost one of the list, as indicated by the parameter *lm*, the element before the start of the slice is guaranteed to be smaller than any element of the slice itself. This can be exploited to omit a comparison in the inner loop of insertion sort (cf. Sec. 3.3). If the slice is not smaller than the threshold, a pivot element is selected and moved to the front of the slice (line 6). If the pivot is equal to the element before the current slice (line 7), this indicates a lot of equal elements. The PARTITION_LEFT function (line 8) will put them in the left partition, and then only the right partition needs to be sorted recursively (line 10). Otherwise, PARTITION_RIGHT (line 12) places elements equal to the pivot in the right partition. Additionally, it returns a flag *ap* that indicates that the slice was already partitioned. Next, we check for a highly unbalanced partitioning (line 13), i.e., if one partition is less than 1/8th of the overall size. After encountering a certain number of highly unbalanced partitionings, pdqsort switches to heapsort (line 14). Otherwise, it will shuffle some elements in both partitions, trying to break up patterns in the input (line 15). If the input was already partitioned wrt. the selected pivot (indicated by the flag *ap*), pdqsort will optimistically try to sort both partitions with insertion sort (line 17). However, these insertion sorts abort if they cannot sort the list with a small number of swaps, limiting the penalty for being too optimistic. Finally, the two partitions are recursively sorted (lines 18–19).

Our implementation of pdqsort closely follows the implementation we found in Boost [6]. Again, we omitted a manual tail call optimization that LLVM does automatically. Moreover, for certain comparison functions, Boost's pdqsort uses a special branch-aware partitioning algorithm [11]. We leave its verification to future work, but note that it will easily integrate in our existing formalization.

While introsort and pdqsort are based on the same idea, this presentation focuses on the more complex pdqsort: apart from the more involved PDQSORT_AUX function, PIVOT_TO_FRONT uses Tukey's 'ninther' pivot selection [4], while introsort uses the simpler median-of-three scheme. It has two partitioning algorithms used in different situations, and the PARTITION_RIGHT algorithm also checks for already partitioned slices. Finally, with INSORT and MAYBE_SORT, it uses two different versions of insertion sort.

3 Verification

We use the Isabelle Refinement Framework [24, 23] to formally verify our algorithms. It provides tools to develop algorithms by stepwise refinement, and generates code in the LLVM intermediate representation [27].

A program returns an element of the following datatype:

$$\alpha\ \mathit{nres} \equiv \texttt{fail} \mid \texttt{spec}\ (\alpha \Rightarrow \mathit{bool})$$

Here `fail` represents possible non-termination or assertion violation, and `spec` P a result nondeterministically chosen to satisfy predicate P. Note that we use $\equiv$ to indicate defining equations. We define a *refinement ordering* on *nres* by

$$\texttt{spec}\ P \leq \texttt{spec}\ Q \equiv \forall x.\ P\ x \implies Q\ x \qquad \texttt{fail} \not\leq \texttt{spec}\ Q \qquad m \leq \texttt{fail}$$

Intuitively, $m_1 \leq m_2$ means that m_1 returns fewer possible results than m_2, and may only fail if m_2 may fail. Note that $\leq$ is a complete lattice, with top element `fail`. The *monad combinators* are then defined as

$\texttt{return}\ x \equiv \texttt{spec}\ y.\ y{=}x$
$\texttt{bind}\ (\texttt{spec}\ P)\ f \equiv \bigsqcup\{f\ x \mid P\ x\}$ $\qquad$ $\texttt{bind fail}\ f \equiv \texttt{fail}$

Here, `return` x deterministically returns x, and `bind` $m\ f$ chooses a result of m and then applies f to it. If m may fail, then the bind may also fail. We write $x{\leftarrow}m;\ f\ x$ for `bind` $m\ (\lambda x.\ f\ x)$, and $m_1;\ m_2$ for `bind` $m_1\ (\lambda\_.\ m_2)$.

Arbitrary recursive programs can be defined via a fixed-point construction [20]. An *assertion* fails if its condition is not met, otherwise it returns the unit value:

$\texttt{assert}\ P \equiv \texttt{if}\ P\ \texttt{then return}\ ()\ \texttt{else fail};$

Assertions are used to express that a program m satisfies the Hoare triple with precondition P and postcondition Q:

$m \leq \texttt{assert}\ P;\ \texttt{spec}\ x.\ Q\ x$

If the precondition is false, the right hand side is `fail`, and the statement trivially holds. Otherwise, m cannot fail, and every possible result x of m must satisfy Q.

While the Isabelle Refinement Framework provides some syntax to express programs, for better readability, we use the slightly more informal syntax that we have already used in Algorithms 1 and 2. In particular, we treat lists as if they were updated in place, while our actual formalization is purely functional, i.e., generates a new version of the list on each update, which is explicitly threaded through the program. Destructively updated arrays will only be introduced in a later refinement step (cf. Sec. 4).

3.1 Specification of Sorting Algorithms

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

$sort\_spec\ xs\ l\ h$
$\quad \equiv \texttt{assert}\ l{\leq}h \wedge h{\leq}|xs|;\ \texttt{spec}\ xs'.\ xs =_{l,h} xs' \wedge sorted\ (xs'[l..{<}h])$

here $|xs|$ is the length of the list xs and $xs[I]$ is the slice of the list xs for indexes in the interval I[4]. The equivalence relation $xs =_{l,h} xs'$ relates lists xs and xs' iff they are equal outside the slice $l..{<}h$ and xs is a permutation of xs'. To simplify the presentation, we assume a linear ordering on the elements. Note that both C++ and our actual formalization support arbitrary weak orderings [19].

3.2 Quicksort Scheme

We split a call of PDQSORT_AUX into phases, described by the following predicates:

[4] An interval from index l (inclusive) to h (exclusive) is denoted as $l..{<}h$. If both indexes are exclusive, we write $l{<}..{<}h$.

$$
\begin{aligned}
&\mathit{pvt}\ \mathit{xs} \equiv a_0 =_{l,h} \mathit{xs} \wedge (\exists i \in l{<}..{<}h.\ \mathit{xs}[i] \le \mathit{xs}[l]) \wedge (\exists i \in l{<}..{<}h.\ \mathit{xs}[i] \ge \mathit{xs}[l])\\
&\mathit{part}\ m\ \mathit{xs} \equiv a_0 =_{l,h} \mathit{xs} \wedge l{\le}m \wedge m{<}h\\
&\qquad\qquad \wedge (\forall i \in l..{<}m.\ \mathit{xs}[i]{\le}\mathit{xs}[m]) \wedge (\forall i \in m{<}..{<}h.\ \mathit{xs}[m]{\le}\mathit{xs}[i])\\
&\mathit{sortl}\ m\ \mathit{xs} \equiv \mathit{part}\ m\ \mathit{xs} \wedge \mathit{sorted}\ (\mathit{xs}[l..{<}m])\\
&\mathit{sortr}\ m\ \mathit{xs} \equiv \mathit{sortl}\ m\ \mathit{xs} \wedge \mathit{sorted}\ (\mathit{xs}[m{<}..{<}h])
\end{aligned}
$$

Let a_0 denote the original list. First, a pivot element is selected and moved to the beginning of the slice (phase *pvt*). The pivot is selected in a way such there is at least one smaller ($\le$) and one greater ($\ge$) element, e.g., by a median-of-three selection. This knowledge can later be exploited to optimize the inner loops of the partitioning algorithm. After the partitioning (phase *part* m), m points to the pivot element, and all elements before m are smaller, and all elements after m are greater. Then, first the left (phase *sortl* m), and then the right (phase *sortr* m) partition gets sorted, while the list remains partitioned around m.

This approach allows us to prove correct the algorithm, without assuming too many details of the underlying subroutines. The following is all we need to know about the subroutines:

$$
\begin{aligned}
&(a)\ \mathit{lm} \vee \mathit{notleft}\ \mathit{xs}\ l\ h \implies \mathit{insort}\ \mathit{lm}\ \mathit{xs}\ l\ h \le \mathit{sort\_spec}\ \mathit{xs}\ l\ h\\
&(b)\ l{+}4{<}h \implies \mathit{pivot\_to\_front}\ \mathit{xs}\ l\ h \le \texttt{spec}\ \mathit{xs}'.\ \mathit{pvt}\ \mathit{xs}'\\
&(c)\ \mathit{pvt}\ \mathit{xs} \implies \mathit{partition\_right}\ \mathit{xs}\ l\ h \le \texttt{spec}\ (\mathit{xs}',m,\_).\ \mathit{part}\ m\ \mathit{xs}'\\
&\qquad\qquad\qquad \wedge\ \mathit{partition\_left}\ \mathit{xs}\ l\ h \ \ \le \texttt{spec}\ (\mathit{xs}',m,\_).\ \mathit{part}\ m\ \mathit{xs}'\\
&(d)\ \mathit{heapsort}\ \mathit{xs}\ l\ h \le \mathit{sort\_spec}\ \mathit{xs}\ l\ h\\
&(e)\ \mathit{part}\ m\ \mathit{xs} \implies \mathit{shuffle}\ \mathit{xs}\ l\ h\ m \le \texttt{spec}\ \mathit{xs}'.\ \mathit{part}\ m\ \mathit{xs}'\\
&(f)\ i{\le}j \wedge j{\le}|\mathit{xs}|\\
&\quad \implies \mathit{maybe\_sort}\ \mathit{xs}\ i\ j \le \texttt{spec}\ (b,\mathit{xs}').\ \mathit{xs}{=}_{i,j}\mathit{xs}' \wedge (\neg b \vee \mathit{sorted}\ \mathit{xs}'[i..{<}j])
\end{aligned}
$$

where $\mathit{notleft}\ \mathit{xs}\ l\ h \equiv 0{<}l \wedge \forall i \in l..{<}h.\ \mathit{xs}[l{-}1] \le \mathit{xs}[i]$ states that the element $\mathit{xs}[l{-}1]$ before the slice is smaller than any element of the slice. Note that we explicitly mention the changed list xs' in these specifications, while we left the list changes implicit in the algorithm description.

Intuitively, (a,d,f) state correctness of the sorting subroutines, (b) states that pivot selection goes to phase *pvt*, (c) states that partitioning transitions from phase *pvt* to phase *part*, and (e) states that shuffling preserves phase *part*. From the above, we easily prove the following lemmas:

$$
\begin{aligned}
&(g)\ \mathit{part}\ m\ \mathit{xs} \wedge \mathit{notleft}\ \mathit{xs}\ l\ h \wedge \mathit{xs}[m] \le \mathit{xs}[l{-}1] \implies \mathit{sorted}\ \mathit{xs}[l..{<}m]\\
&(h)\ \mathit{part}\ m\ \mathit{xs} \wedge (\ \mathit{xs}{=}_{l,m}\mathit{xs}' \vee \mathit{xs}{=}_{m+1,h}\mathit{xs}') \implies \mathit{part}\ m\ \mathit{xs}'\\
&(i)\ \mathit{sortl}\ m\ \mathit{xs} \wedge \mathit{xs}{=}_{m+1,h}\mathit{xs}' \implies \mathit{sortl}\ m\ \mathit{xs}'\\
&(j)\ \mathit{part}\ m\ \mathit{xs} \implies \mathit{sort\_spec}\ \mathit{xs}\ l\ m \le \texttt{spec}\ \mathit{xs}'.\ \mathit{sortl}\ m\ \mathit{xs}'\\
&(k)\ \mathit{sortl}\ m\ \mathit{xs} \implies \mathit{sort\_spec}\ \mathit{xs}\ (m{+}1)\ h \le \texttt{spec}\ \mathit{xs}'.\ \mathit{sortr}\ m\ \mathit{xs}'\\
&(l)\ \mathit{sortr}\ m\ \mathit{xs} \implies \mathit{sorted}\ \mathit{xs}[l..{<}h]
\end{aligned}
$$

The correctness statement for PDQSORT_AUX is:

$$\mathit{lm} \vee \mathit{notleft}\ \mathit{xs}\ l\ h \implies \mathit{pdqsort\_aux}\ \mathit{lm}\ \mathit{xs}\ l\ h\ d \le \mathit{sort\_spec}\ \mathit{xs}\ l\ h$$

The proof is done by using the Refinement Framework's verification condition generator, and then discharging the generated VCs using the above lemmas. The line numbers in the following brief sketch refer to Algorithm 2. As termination measure for the recursion, we use the size $h - l$ of the slice to be sorted. If we switch to insertion sort in line 4, (a) implies that the slice gets sorted, and we are done. Otherwise, we select a pivot in line 6, going to phase *pvt* (b). When the equals optimization is triggered in line 7, we transition to phase *part* (c), and the left partition is already sorted[5] (g), such that we can transition to phase *sortl* (j), and, via a recursive call in line 10 to phase *sortr* (k). This implies that the slice is sorted (l), and we are done. When the equals optimization is not used, (c) shows that we transition to phase *part* in line 12. If the partition is unbalanced, we either use heapsort (line 14) to directly sort the slice (d), or shuffle the elements (line 15) and stay in phase *part* (e). In line 17, the algorithm may attempt to sort the slice. If this succeeds, we are done (f). Otherwise, we stay in phase *part* (h), and the recursive calls in lines 18 and 19 will take us to phase *sortr* (j,k), which implies sortedness of the slice (l).

Using the above statement, and an analogous statement for introsort, we can prove the main correctness theorem:

Theorem 1. *pdqsort xs l h* $\leq$ *sort_spec xs l h*
and *introsort xs l h* $\leq$ *sort_spec xs l h*

Note that we could prove the correctness of our algorithm with only minimal assumptions about the used subroutines. This decoupling of the algorithm from its subroutines simplifies the proof, as it is not obfuscated with unnecessary details. For example, correctness of the algorithm does not depend on the exact partitioning scheme being used, as long as it implements a transition from the *pvt* to the *part* phase. It also simplifies changing the subroutines later, e.g., adding further optimizations such as branch-aware partitioning [11].

Breaking down an algorithm into small and decoupled modules is often the key to its successful verification. Note that the original implementation in Boost is more coarse grained, inlining much of the functionality into the main algorithm. After having proved correct an algorithm, we can always do the inlining in a later refinement step, or rely on the LLVM optimizer to do the inlining for us. In our formalization, we use the inlining feature of Isabelle-LLVM's preprocessor.

```
procedure INSORT($G, xs, l, h$)
    if $l = h$ then return
    $i \leftarrow l + 1$
    while $i < h$ do
        INSERT($G, xs, l, i$)
        $++i$

procedure INSERT($G, xs, l, i$)
    $t \leftarrow xs[i]$
    while $(\neg G \vee l < i) \wedge t < xs[i-1]$ do
        $xs[i] \leftarrow xs[i-1]$
        $--i$
    $xs[i] \leftarrow t$
```

Algorithm 3: Insertion Sort

[5] actually all elements in the left partition are equal to the pivot.

3.3 Insertion Sort

Algorithm 3 shows our implementation of insertion sort. The INSORT procedure repeatedly calls INSERT to add elements to a sorted prefix of the list. The flag G controls the *unguarded* optimization: if it is false, we assume that INSERT will hit a smaller element before underflowing the list index i, and thus omit the comparison $l < i$ (line 9) in the inner loop. We later specialize the INSORT algorithm for the two cases of G, and simplify the loop conditions accordingly.

Again, we split the insertion sort algorithm into two smaller parts, which are proved separately via the following specification for INSERT:

$\texttt{assert}\ \textit{sorted}\ \textit{xs}[l..{<}i] \wedge l{\le}i \wedge i{<}|\textit{xs}| \wedge (G \vee \textit{xs}[l{-}1] \le \textit{xs}[i]);$
$\texttt{spec}\ \textit{xs}'.\ \textit{xs}{=}_{l,i+1}\textit{xs}' \wedge \textit{sorted}\ \textit{xs}[l..{<}i{+}1]$

This captures the intuition that INSERT goes from a slice that is sorted up to index i to one that is sorted up to index $i + 1$.

3.4 The Remaining Subroutines

The proofs of the remaining subroutines follow a similar plot, and are not displayed here in full. Most of them were straightforward, and we could use existing Isabelle proofs as guideline [25, 16, 22]. For the SHUFFLE and PIVOT_TO_FRONT procedures, which contain a large number of indexing and update operations, we ran into a scalability problem: the many partially redundant in-bound statements for the indexes overwhelmed the linear arithmetic solver that is hard-wired into the simplifier. We worked around this problem by introducing auxiliary definitions, which hide the in-bound statements from the simplifier, and allow us to precisely control when it sees them.

Finally, we point out another interesting application of refinement: the SIFT_DOWN function of heapsort restores the heap property by *floating down* an element[6]. A straightforward implementation swaps the element with one of its children, until the heap property is restored (Alg. 4 (left)). However, the element that is written to $xs[\text{right}(i)]$ or $xs[\text{left}(i)]$ by the swap will get overwritten in the next iteration. 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. 4 (right)). Note that the INSERT procedure of insertion sort (cf. Alg. 3) does a similar optimization. However, for INSERT, 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 abstract list xs and the concrete list xs' is $xs = xs'[i{:=}t]$. Using the tool support of the Isabelle Refinement Framework, the proof that the optimized version refines the version with swaps requires only about 20 lines of straightforward Isabelle script.

[6] see, e.g., [9, Ch. 6] or [33, Ch. 2.4] for a description of heapsort.

```
procedure SIFT_DOWN($xs, i$)
    while has_right($i$) do
        if $xs[\text{left}(i)] < xs[\text{right}(i)]$ then
            if $xs[i] < xs[\text{right}(i)]$ then
                SWAP($xs[i], xs[\text{right}(i)]$)
                $i \leftarrow \text{right}(i)$
            else return
        else if $xs[i] < xs[\text{left}(i)]$ then
            SWAP($xs[i], xs[\text{left}(i)]$)
            $i \leftarrow \text{left}(i)$
        else return
    if has_left($i$) $\wedge$ $xs[i] < xs[\text{left}(i)]$ then
        SWAP($xs[i], xs[\text{left}(i)]$)
```

```
procedure SIFT_DOWN_OPT($xs', i$)
    $t \leftarrow xs'[i]$
    while has_right($i$) do
        if $xs'[\text{left}(i)] < xs'[\text{right}(i)]$ then
            if $t < xs'[\text{right}(i)]$ then
                $xs'[i] \leftarrow xs'[\text{right}(i)]$
                $i \leftarrow \text{right}(i)$
            else return
        else if $t < xs[\text{left}(i)]$ then
            $xs'[i] \leftarrow xs'[\text{left}(i)]$
            $i \leftarrow \text{left}(i)$
        else return
    if has_left($i$) $\wedge$ $t < xs'[\text{left}(i)]$ then
        $xs'[i] \leftarrow xs'[\text{left}(i)]$
        $i \leftarrow \text{left}(i)$
    $xs'[i] \leftarrow t$
```

Algorithm 4: The standard (left) and optimized (right) sift-down function.

4 Imperative Implementation

We have presented a refinement based approach to verify the introsort and pdqsort algorithms, including most optimizations we found in their libstdc++ and Boost implementations. However, the algorithms are still expressed as nondetermistic programs on functional lists and unbounded natural numbers. In this section, we use the Isabelle-LLVM framework [23] to (semi-)automatically refine them to LLVM programs on arrays and 64 bit integers.

4.1 The Sepref Tool

The Sepref tool [21, 23] symbolically executes an abstract program in the *nres*-monad, keeping track of refinements for every abstract variable to a concrete representation, which may use pointers to dynamically allocated memory. During the symbolic execution, the tool synthesizes an imperative Isabelle-LLVM program, together with a refinement proof. The synthesis is automatic, but usually requires some program-specific setup and boilerplate. For a detailed discussion of Sepref and Isabelle-LLVM, we refer the reader to [21, 23].

Sepref comes with standard setup to refine lists to arrays. List updates are refined to destructive array updates, as long as the old version of the list is not used after the update. It also provides setup to refine unbounded natural numbers to bounded integers. It tries to discharge the resulting in-bounds proof obligations automatically. If this is not possible, it relies on hints 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 PDQSORT_AUX algorithm (Alg. 2, line 9), the assertion $m{+}1 \leq h$ ensures that the addition $m{+}1$ in the next line cannot overflow. This assertion adds a proof obligation to the correctness proof of PDQSORT_AUX, which is easily discharged (we are in phase *part*, which guarantees $m<h$). When refining PDQSORT_AUX to an implementation with bounded integers, one can assume $m+1\leq h$ to discharge the non-overflow proof obligation. Note that re-proving $m+1\leq h$ when doing the refinement would require duplicating large parts of the correctness proof. Thus, assertions provide a convenient tool to pass properties down the refinement chain. Our actual formalization contains multiple such assertions, which we have omitted in this presentation for the sake of readability.

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

Fig. 1: Implementation of the INSERT procedure for 64bit unsigned integer elements and $G{=}false$, which is generated by the Sepref tool. This definition lies within the executable fragment of Isabelle-LLVM, i.e., the Isabelle LLVM code generator can translate it to LLVM intermediate representation. Note that the function does not depend on the lower bound parameter l any more, as this was only required in the guarded version. Inlining will remove this bogus parameter.

Using the Sepref tool, it is straightforward to refine the sorting algorithms and their subroutines to an Isabelle-LLVM program. For example, Figure 1 shows the Isabelle-LLVM code that is generated for the INSERT procedure for unsigned 64 bit integer elements and $G{=}false$ (cf. 3.3). Moreover, the Sepref tool proves that the generated program actually implements the abstract one:

$$(\mathit{ug\_insert\_impl},\ \mathit{insert\ False}) : \mathit{arr}^d \times \mathit{nat}_{64}^k \times \mathit{nat}_{64}^k \rightarrow \mathit{arr}$$

This specifies the refinement relations for the parameters and the result, where *arr* relates arrays with lists, and nat_{64} relates 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 original array is destroyed.

The final correctness statement for our implementations is:

Theorem 2. $(\mathit{introsort\_impl},\ \mathit{sort\_spec}) : \mathit{arr}^d \times \mathit{nat}_{64}^k \times \mathit{nat}_{64}^k \rightarrow \mathit{arr}$
and $(\mathit{pdqsort\_impl},\ \mathit{sort\_spec}) : \mathit{arr}^d \times \mathit{nat}_{64}^k \times \mathit{nat}_{64}^k \rightarrow \mathit{arr}$

Here, *introsort_impl* and *pdqsort_impl* are the Isabelle-LLVM programs generated by Sepref from INTROSORT and PDQSORT (Algs. 1 and 2). The theorem is easily proved by combining Theorem 1 with the theorems generated by Sepref.

4.2 Separation Logic and Ownership

Internally, the Sepref tool represents the symbolic state that contains all abstract variables and their refinements to concrete variables as an assertion in separation logic [31, 8]. Thus, two variables can never reference the same memory. This is a problem for nested container data structures like arrays of strings: when indexing the array, both the array element and the result of the indexing operation would point to the same string. In the original Sepref tool [21], which targeted Standard ML, we worked around this problem by always using functional data types (e.g. lists) to represent the inner type of a nested container. This workaround is no longer applicable for the purely imperative LLVM, such that we could not use Sepref for nested container data structures[7].

We now describe an approach towards solving this problem for Sepref. Abstractly, we model an array by the type α *option list*[8], where *None* means that the array does currently not own the respective element. The abstract indexing operation then moves the element from the list to the result:

$$\mathit{move}\ \mathit{xs}\ i \equiv \texttt{assert}\ i{<}|\mathit{xs}| \wedge \mathit{xs}[i]{\neq}\mathit{None};\ \texttt{return}\ (\mathit{the}\ (\mathit{xs}[i]),\ \mathit{xs}[i{:=}\mathit{None}])$$

As no memory is shared between the result and the array, we can show the following refinement:

$$(\lambda a\ i.\ (a[i],a),\ \lambda \mathit{xs}\ i.\ \mathit{move}\ \mathit{xs}\ i) : \mathit{oarr}^d \times \mathit{nat}^k_{64} \rightarrow A \times \mathit{oarr}$$

where *oarr* is the relation between an array and an α *option list*, and A is the relation for the array elements. Note that this operation does not change the concrete array a. The movement of ownership is a purely abstract concept, which results in no implementation overhead.

The transition from α *list* to α *option list* can typically be done in an additional refinement step, and thus does not obfuscate the actual correctness proofs, which are still done on plain α *list*. Moreover, the α *option list* representation is only required for subroutines where extracted array elements are actually visible. For example, we define an operation to compare two array elements:

$$\mathit{cmp\_idxs}\ \mathit{xs}\ i\ j \equiv \texttt{assert}\ i{<}|\mathit{xs}| \wedge j{<}|\mathit{xs}|;\ \texttt{return}\ \mathit{xs}[i] < \mathit{xs}[j]$$

Inside this operation, we have to temporarily extract the elements i and j from the array, requiring an intermediate refinement step to α *option list*. However, at the start and end of this operation, the array owns all its elements. For the whole operation, we thus get a refinement on plain arrays:

$$(\mathit{cmp\_impl},\ \mathit{cmp\_idxs}) : \mathit{arr}^k \times \mathit{nat}^k_{64} \times \mathit{nat}^k_{64} \rightarrow \mathit{bool}$$

In our case, we only have to explicitly refine INSERT and SIFT_DOWN. The other subroutines use *cmp_idxs* and *swap* operations on plain lists.

[7] We could still reason about such structures on a lower level.

[8] Here, α *option* $=$ *None* $|$ *Some* α is Isabelle's option datatype, and $\mathit{the}\ (\mathit{Some}\ x) \equiv x$ is the corresponding selector function.

4.3 The Isabelle-LLVM Code Generator

The programs that are generated by Sepref (cf. Fig. 1) lie in the fragment for which Isabelle-LLVM [23] can generate LLVM text. For example, the pdqsort algorithm for strings yields an LLVM function with the signature:

```
{ i64, { i64, i8* } }* @str_pdqsort({ i64, { i64, i8* } }*, i64, i64)
```

Here, the type `{ i64, { i64, i8* } }` represents dynamic arrays of characters[9], represented by length, capacity, and a data pointer.

The generated LLVM text is then compiled to machine code using the LLVM toolchain. To make the generated program usable, one has to link it to a C wrapper, which handles parsing of command line options and printing of results. However, the original Isabelle-LLVM framework provides no support for interfacing the generated code from C: one has to manually write a C header file, which hopefully matches the object file generated by the LLVM compiler. If it doesn't, the program has undefined behaviour[10].

To this end, we extended Isabelle-LLVM to also generate a header file for the exported functions. For example, the 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; ›
```

will check that the specified signature actually matches the Isabelle definition, and generate the following C declarations:

```
typedef struct { int64_t size; struct { int64_t capacity; char *data; }; } llstring;
llstring* str_introsort(llstring*, int64_t, int64_t);
```

5 Benchmarks

The Boost library comes with a sorting algorithm benchmark suite, which we extended with further benchmarks indicated in [4]: apart from sorting random lists 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 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).

[9] For strings, we use the verified dynamic array implementation provided by Isabelle-LLVM. Note that C++ uses a similar representation, with an additional optimization for small strings.

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

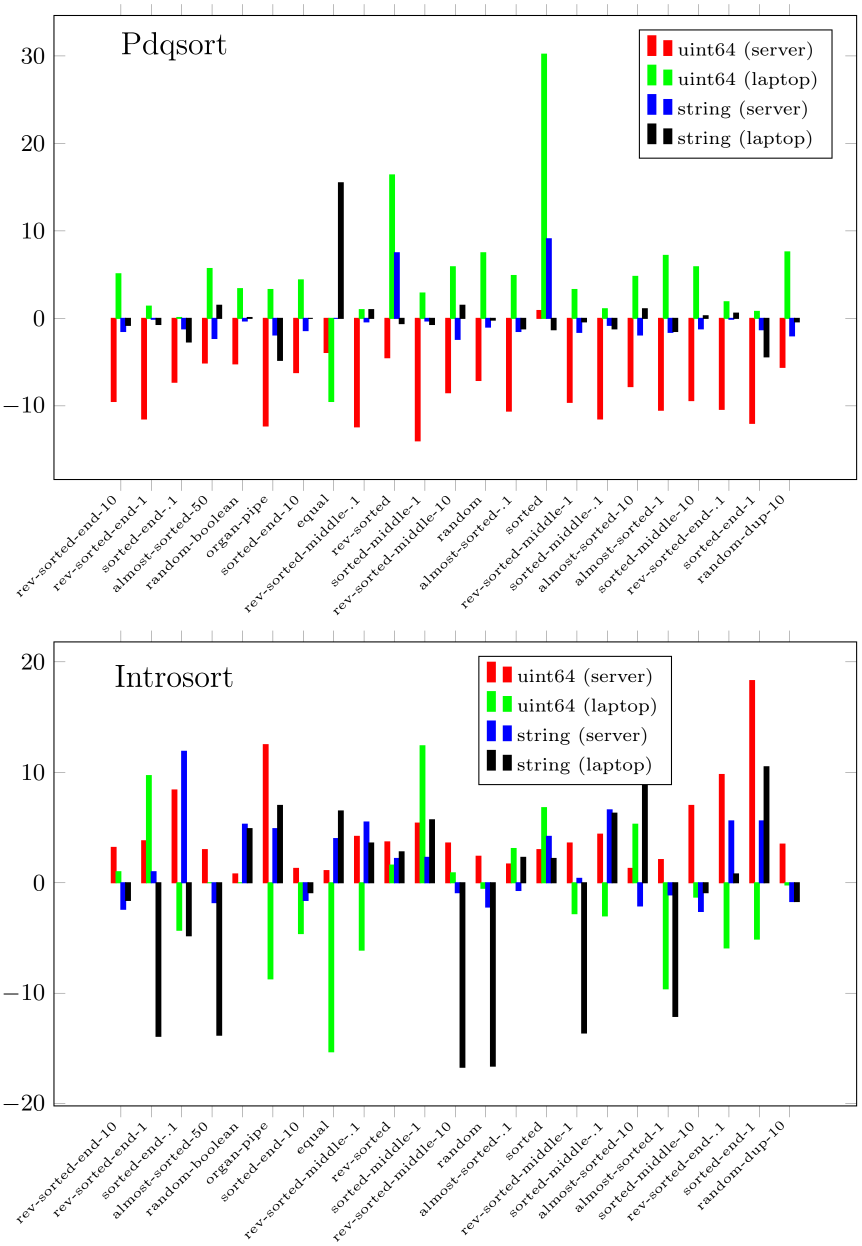

Fig. 2: Benchmarking our verified implementations against the unverified originals. For each element type, machine, and distribution, the value $(t_1/t_2 - 1) * s$ is shown, where t_1 is the slower time, t_2 is the faster time, and $s = 100$ if our implementation is slower, and $s = -100$ if the original implementation is slower. That is, a positive value p indicates that our implementation is slower, requiring $100 + p$ percent of the run time of the original. Analogously, a negative value $-p$ means that the original implementation is slower, requiring $100 + p$ percent of our implementation's run time.

We sorted integer arrays with $n = 10^8$ elements, and string arrays with $n = 10^7$ elements. For strings, all implementations use the same data structure and compare function. For integers, we disable pdqsorts branch-aware partitioning, which we have not yet verified. For strings, it does not apply anyway.

We compile both, the verified and unverified algorithms with clang-6.0.0, and run them 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 same algorithm should take exactly the same time when repeatedly run on the same data and machine. However, in practice, we encountered some noise up to 17%. Thus, we have repeated each experiment at least ten times, and more often to confirm outliers where the verified and unverified algorithms' run times differ significantly. Assuming that the noise only slows down an algorithm, we take the fastest time measured over all repetitions. The results are displayed in Figure 2.

They indicate that both our pdqsort and introsort implementations are competitive. There is one outlier for pdqsort for already sorted integer arrays on the laptop. We have not yet understood its exact reason. The remaining cases differ by less than 20%, and in many cases our verified algorithm is actually faster.

6 Conclusions

We have presented the first verification of the introsort and pdqsort algorithms. We verified state-of-the-art implementations, down to LLVM intermediate representation. On an extensive set of benchmarks, our verified implementations perform on par with their unverified counterparts from the GNU C++ and Boost C++ libraries. Apart from our work, the only other verified real-world implementations of sorting algorithms are Java implementations of timsort and dual-pivot quicksort that have been verified with KeY [10, 3].

Compared to other program verification methods, the trusted code base of our approach is small: apart from the well-tested and widely used LLVM compiler, it only includes Isabelle's logical inference kernel, and the relatively straightforward Isabelle-LLVM semantics and code generation [23]. In contrast, 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.

Our verified algorithms can readily be used in larger verification projects, and we have already replaced a naive quicksort implementation that caused a stack overflow in an ongoing SAT-solver verification project [13]. For fixed element types and containers based on arrays (e.g. std::vector), we can use our verified algorithms as a drop-in replacement for C++'s std::sort. A direct verification of the C++ code of std::sort, however, would require a formal semantics of C++, including templates and the relevant concepts from the standard template library (orderings, iterators, etc.). To the best of our knowledge, such a semantics has not yet been formalized, let alone been used to verify non-trivial algorithms.

The verification of introsort took us about 100 person hours. After we had set up most of the infrastructure for introsort, we could verify the more complex pdqsort in about 30h. The development consists of roughly 8700 lines of Isabelle text, of which 2400 lines are for introsort and 2200 lines for pdqsort. The rest is boilerplate and libraries shared between both algorithms, among them 1500 lines for the verification of heapsort.

6.1 Related Work

Sorting algorithms are a standard benchmark for program verification tools, such that we cannot give an exhaustive overview here. Nevertheless, we discuss a few notable examples: the arguably first formal proof of quicksort was given by Foley and Hoare himself [14], 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 Magaud [12], who prove correct imperative versions of quicksort, heapsort, and insertion sort in Coq. However, they use a simplistic partitioning scheme, do not report on code generation or benchmarking, nor do they combine their separate algorithms to get introsort.

The timsort algorithm, which was used in the Java standard library, has been verified with the KeY tool [10]. 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.

6.2 Future Work

An obvious next step is to verify a branch-aware partitioning algorithm [11]. Thanks to our modular approach, this will easily integrate with our existing formalization. We also plan to extend our work to stable sorting algorithms. Recently, we have extended the Refinement Framework to support reasoning about algorithmic complexity [17]. Once this work has been integrated with Isabelle-LLVM, we can also prove that our implementations have a worst-case complexity of $O(n \log(n))$, as required by the C++ standard. Finally, we proposed an explicit ownership model for nested lists. We plan to extend this to more advanced concepts like read-only shared ownership, inspired by Rust's [32] ownership system. Formally, this could be realized with fractional permission separation logic [7].

Acknowledgements We received funding from DFG grant LA 3292/1 "Verifizierte Model Checker" and VeTSS grant "Formal Verification of Information Flow Security for Relational Databases".

Bibliography

[1] Algorithm 232: Heapsort. *Commun. ACM*, 7(6):347–349, June 1964.

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

[3] B. Beckert, J. Schiffl, P. H. Schmitt, and M. Ulbrich. Proving jdk's dual pivot quicksort correct. In *VSTTE*, 2017.

[4] J. L. Bentley and M. D. McIlroy. Engineering a sort function. *Softw. Pract. Exper.*, 23(11):1249–1265, Nov. 1993.

[5] J. Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken.

[6] Boost C++ libraries.

[7] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.

[8] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS 2007*, pages 366–378, July 2007.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[10] S. de Gouw, J. Rot, F. S. de Boer, R. Bubel, and R. Hähnle. Openjdk's java.utils.collection.sort() is broken: The good, the bad and the worst case. In *CAV*, 2015.

[11] S. Edelkamp and A. Weiß. Blockquicksort: How branch mispredictions don't affect quicksort. *CoRR*, abs/1604.06697, 2016.

[12] J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the coq system. 1999.

[13] M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using Imperative HOL. In *Proc. of CPP*, pages 158–171, 2018.

[14] M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *The Computer Journal*, 14(4):391–395, 01 1971.

[15] The GNU C++ library. Version 7.4.0.

[16] S. Griebel. Binary heaps for imp2. *Archive of Formal Proofs*, June 2019. `http://isa-afp.org/entries/IMP2_Binary_Heap.html`, Formal proof development.

[17] M. Haslbeck and P. Lammich. Refinement with time – refining the run-time of algorithms in isabelle/hol. In *ITP2019: Interactive Theorem Proving*, 6 2019.

[18] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[19] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley Professional, 2nd edition, 2012.

[20] A. Krauss. Recursive definitions of monadic functions. In *Proc. of PAR*, volume 43, pages 1–13, 2010.

[21] P. Lammich. Refinement to Imperative/HOL. In *ITP*, volume 9236 of *LNCS*, pages 253–269. Springer, 2015.

[22] P. Lammich. Refinement based verification of imperative data structures. In J. Avigad and A. Chlipala, editors, *CPP 2016*, pages 27–36. ACM, 2016.

[23] P. Lammich. Generating Verified LLVM from Isabelle/HOL. In J. Harrison, J. O'Leary, and A. Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] P. Lammich and T. Tuerk. Applying data refinement for monadic programs to Hopcroft's algorithm. In L. Beringer and A. P. Felty, editors, *ITP 2012*, volume 7406 of *LNCS*, pages 166–182. Springer, 2012.

[25] P. Lammich and S. Wimmer. Imp2 – simple program verification in isabelle/hol. *Archive of Formal Proofs*, Jan. 2019. `http://isa-afp.org/entries/IMP2.html`, Formal proof development.

[26] "libc++" c++ standard library.

[27] LLVM language reference manual.

[28] D. R. MUSSER. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[29] Pattern-defeating quicksort.

[30] T. Peters. Original description of timsort. Accessed 2019-10-21.

[31] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc of. Logic in Computer Science (LICS)*, pages 55–74. IEEE, 2002.

[32] The rust programmin language.

[33] R. Sedgewick and K. Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.