Efficient Verified Implementation of Introsort and Pdqsort

Peter Lammich

The University of Manchester

July 2020
Motivation + Overview

• Verification of efficient software
  • stepwise refinement: separation of concerns
    • algorithmic idea, data structures, optimizations, ...
  • interactive theorem prover: flexible, mature
    • easily proves required background theory
Motivation + Overview

• Verification of efficient software
  • stepwise refinement: separation of concerns
    • algorithmic idea, data structures, optimizations, ...
  • interactive theorem prover: flexible, mature
    • easily proves required background theory

• Isabelle Refinement Framework
  • tools for stepwise refinement in Isabelle/HOL
  • already used for complex software: Model-Checkers, UNSAT-Certifiers, Graph Algorithms, ...
Motivation + Overview

• Verification of efficient software
  • stepwise refinement: separation of concerns
    • algorithmic idea, data structures, optimizations, ...
  • interactive theorem prover: flexible, mature
    • easily proves required background theory

• Isabelle Refinement Framework
  • tools for stepwise refinement in Isabelle/HOL
  • already used for complex software: Model-Checkers, UNSAT-Certifiers, Graph Algorithms, ...

• Backends (refinement target)
  • limited by Isabelle’s code generator
  • purely functional code: slow
  • functional + imperative (e.g. Standard ML): faster
Motivation + Overview

• Verification of efficient software
  • stepwise refinement: separation of concerns
    • algorithmic idea, data structures, optimizations, ...
  • interactive theorem prover: flexible, mature
    • easily proves required background theory

• Isabelle Refinement Framework
  • tools for stepwise refinement in Isabelle/HOL
  • already used for complex software: Model-Checkers, UNSAT-Certifiers, Graph Algorithms, ...

• Backends (refinement target)
  • limited by Isabelle’s code generator
  • purely functional code: slow
  • functional + imperative (e.g. Standard ML): faster
  • cannot compete with good C/C++ compiler!
Isabelle-LLVM

- Fragment of LLVM semantics formalized in Isabelle/HOL
  - code generator for LLVM code and C/C++ headers
  - integration with Isabelle Refinement Framework
  - slim trusted code base (vs. functional lang. compiler)

- Can now compete with C/C++ implementations
- Less features (datatype, poly, ...) require more complex refinement
- Higher-level refinements can typically be reused
Isabelle-LLVM

- Fragment of LLVM semantics formalized in Isabelle/HOL
  - code generator for LLVM code and C/C++ headers
  - integration with Isabelle Refinement Framework
  - slim trusted code base (vs. functional lang. compiler)
- Can now compete with C/C++ implementations
  - less features (datatype, poly, ...) require more complex refinement
  - higher-level refinements can typically be reused
Case study how fast we can get
This Paper: Overview

Case study how fast we can get

- Verify state-of-the-art generic sorting algorithms
  - Introsort (std::sort in libstdc++)
  - Pdqsort (from Boost C++ Libraries)
This Paper: Overview

Case study how fast we can get

- Verify state-of-the-art generic sorting algorithms
  - Introsort (std::sort in libstdc++)
  - Pdqsort (from Boost C++ Libraries)
- Using Isabelle Refinement Framework
  - separate optimizations from algorithmic ideas
  - usable as building-blocks for other verifications
This Paper: Overview

Case study how fast we can get

- Verify state-of-the-art generic sorting algorithms
  - Introsort (std::sort in libstdc++)
  - Pdqsort (from Boost C++ Libraries)
- Using Isabelle Refinement Framework
  - separate optimizations from algorithmic ideas
  - usable as building-blocks for other verifications
- As fast as their unverified counterparts
  - on an extensive set of benchmarks
The Introsort Algorithm

- Combine quicksort, heapsort, and insort to fast $O(n \log n)$ algorithm.

```
1: procedure INTROSORT(xs, l, h)
2:     if h − l > 1 then
3:         INTROSORT_AUX(xs, l, h, 2[log_2(h − l)])
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 ← PARTITION_PIVOT(xs, l, h)
10:               INTROSORT_AUX(xs, l, m, d − 1)
11:               INTROSORT_AUX(xs, m, h, d − 1)
```
The Introsort Algorithm

- Combine quicksort, heapsort, and insort to fast $O(n \log n)$ algorithm.
  - if quicksort recursion too deep, switch to heapsort

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 $h - l > \text{threshold}$ then
7:      if $d = 0$ then HEAPSORT($xs, l, h$)
8:      else
9:          $m \leftarrow \text{PARTITION\_PIVOT}(xs, l, h)$
10:         INTROSORT_AUX($xs, l, m, d - 1$)
11:         INTROSORT_AUX($xs, m, h, d - 1$)
The Introsort Algorithm

• Combine quicksort, heapsort, and inset sort to fast $O(n \log n)$ algorithm.
  • if quicksort recursion too deep, switch to heapsort
  • use insertion sort for small partitions
    • final inset sort on array sorted up to threshold

1: procedure INTROSORT(xs, l, h)
2:    if $h - l > 1$ then
3:        INTROSORT_AUX(xs, l, h, 2\lceil \log_2(h - l) \rceil)
4:    final_INSORT(xs, l, h)
5: procedure INTROSORT_AUX(xs, l, h, d)
6:    if $h - l > \text{threshold}$ then
7:        if $d = 0$ then HEAPSORT(xs, l, h)
8:        else
9:            $m \leftarrow \text{PARTITION\_PIVOT}(xs, l, h)$
10:           INTROSORT_AUX(xs, l, m, d - 1)
11:           INTROSORT_AUX(xs, m, h, d - 1)
Verification Methodology: Modularity

- Specifications for subroutines, e.g. `heapsort ≤ sort_spec`
  - proof only uses specification
  - independent of impl details of subroutines
Verification Methodology: Modularity

- Specifications for subroutines, e.g. `heapsort \leq sort\_spec`
  - proof only uses specification
  - independant of impl details of subroutines

\[
\text{partition\_spec } xs \equiv \text{ any non-trivial partitioning}
\]
\[
\text{assert } (\text{length } xs \geq 4);
\text{spec } (xs_1, xs_2). \text{ mset } xs = \text{ mset } xs_1 + \text{ mset } xs_2 \land xs_1 \neq [] \land xs_2 \neq []
\land (\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y)
\]
Verification Methodology: Modularity

- Specifications for subroutines, e.g. $\text{heapsort} \leq \text{sort_spec}$
- proof only uses specification
- independant of impl details of subroutines

\[
\text{partition_spec } xs \equiv \text{— any non-trivial partitioning as a set of partitions}
\]
\[
\text{assert } (\text{length } xs \geq 4);
\]
\[
\text{spec } (xs_1, xs_2). \text{ mset } xs = \text{ mset } xs_1 + \text{ mset } xs_2 \land xs_1 \neq [] \land xs_2 \neq []
\]
\[
\land (\forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y)
\]

\[
\text{part_sorted_spec } xs \equiv \text{— sort up to threshold as a set of partitions}
\]
\[
\text{spec } xs'. \text{ mset } xs' = \text{ mset } xs \land \text{part_sorted_wrt } (\leq) \text{ threshold } xs'
\]

where

\[
\text{part_sorted_wrt } n xs \equiv \exists ss. \text{ is_slicing } n xs ss \land \text{sorted_wrt slice_lt ss}
\]
\[
\text{is_slicing } n xs ss \equiv xs = \text{concat } ss \land (\forall s \in \text{set } ss. s \neq [] \land \text{length } s \leq n)
\]
\[
\text{slice_lt } xs ys \equiv \forall x \in \text{set } xs. \forall y \in \text{set } ys. x \leq y
\]
Verification Methodology: Refinement

• E.g. lists → slices of lists → arrays; \( \mathbb{N} \rightarrow \text{uint64}_t \)
Verification Methodology: Refinement

- E.g. lists → slices of lists → arrays; \( \mathbb{N} \rightarrow \text{uint64}_t \)

\[
\text{introsort\_aux1 } d \ x s \leq \text{part\_sorted\_spec } x s \quad \text{— sort whole list}
\]

\[
(x s i, x s) \in \text{slice\_rel } l \ h \quad \Longrightarrow \quad \text{— sort slice}
\]

\[
\text{introsort\_aux2 } d \ x s i \ l \ h \leq \downarrow (\text{slice\_rel } x s i \ l \ h) \ (\text{introsort\_aux1 } d \ x s)
\]

\[
(\text{introsort\_aux\_impl}, \text{introsort\_aux2}) \quad \text{— sort arrays, indices as uint64}
\]

\[
: \text{nat}_64 \rightarrow \text{array}^d \rightarrow \text{nat}_64 \rightarrow \text{nat}_64 \rightarrow \text{array}
\]
Verification Methodology: Refinement

• E.g. lists → slices of lists → arrays; \( \mathbb{N} \rightarrow \text{uint64}_t \)
  
  \[
  \text{introsort\_aux1} \ d \ \text{xs} \leq \ \text{part\_sorted\_spec} \ \text{xs} \quad \text{— sort whole list}
  \]

  \[
  (\text{xsi, xs}) \in \text{slice\_rel} \ l \ h \quad \Longrightarrow \quad \text{— sort slice}
  \]

  \[
  \text{introsort\_aux2} \ d \ \text{xsi} \ l \ h \leq \downarrow (\text{slice\_rel} \ \text{xsi} \ l \ h) \ (\text{introsort\_aux1} \ d \ \text{xs})
  \]

  \[
  (\text{introsort\_aux\_impl}, \ \text{introsort\_aux2}) \quad \text{— sort arrays, indices as uint64}
  \]

  \[
  : \ \text{nat}_{64} \rightarrow \text{array}^d \rightarrow \text{nat}_{64} \rightarrow \text{nat}_{64} \rightarrow \text{array}
  \]

• Transitivity yields

  \[
  (\text{introsort\_aux\_impl}, \ \lambda d. \ \text{slice\_part\_sorted\_spec})
  \]

  \[
  : \ \text{nat}_{64} \rightarrow \text{array}^d \rightarrow \text{nat}_{64} \rightarrow \text{nat}_{64} \rightarrow \text{array}
  \]

where

\[
\text{slice\_part\_sorted\_spec} \ \text{xs} \ l \ h \equiv \ldots \quad \text{sort xs}[l..h] \ \text{up to threshold}
\]
Verification Methodology: Refinement

- E.g. lists → slices of lists → arrays; \( \mathbb{N} \rightarrow \text{uint64}_t \)
  
  \[
  \text{introsort\_aux1} \ d \ \text{xs} \leq \ \text{part\_sorted\_spec} \ \text{xs} \quad \text{— sort whole list}
  \]

  \[
  (\text{xsi,xs}) \in \text{slice\_rel} \ l \ h \quad \implies \quad \text{— sort slice}
  \]

  \[
  \text{introsort\_aux2} \ d \ \text{xsi} \ l \ h \leq \ (\text{slice\_rel} \ \text{xsi} \ l \ h) \ (\text{introsort\_aux1} \ d \ \text{xs})
  \]

  \[
  (\text{introsort\_aux\_impl}, \ \text{introsort\_aux2}) \quad \text{— sort arrays, indices as \text{uint64}}
  \]

  \[
  : \ \text{nat}_64 \rightarrow \text{array}^d \rightarrow \text{nat}_64 \rightarrow \text{nat}_64 \rightarrow \text{array}
  \]

- Transitivity yields
  
  \[
  (\text{introsort\_aux\_impl}, \ \lambda d. \ \text{slice\_part\_sorted\_spec})
  \]

  \[
  : \ \text{nat}_64 \rightarrow \text{array}^d \rightarrow \text{nat}_64 \rightarrow \text{nat}_64 \rightarrow \text{array}
  \]

  where

  \[
  \text{slice\_part\_sorted\_spec} \ \text{xs} \ l \ h \equiv \ldots \quad \text{sort \text{xs}[l..h] up to threshold}
  \]

- From here on, impl-details and internal refinement steps are irrelevant
Some of the Optimizations

1: procedure INSERT(\( G, xs, l, i \))
2: \( \text{tmp} \leftarrow xs[i] \)
3: while \((\neg G \lor l < i) \land \text{tmp} < xs[i - 1]\) do
4: \(xs[i] \leftarrow xs[i - 1]\)
5: \(i \leftarrow i - 1\)
6: \(xs[i] \leftarrow \text{tmp}\)
Some of the Optimizations

- unguarded insertion sort
  - omit index check in insert, if $\exists$ smaller element
  - guard controlled by flag. ($\text{insort } G \; \text{xs} \; l \; h$)
  - specialized for $G=\{true, false\}$

1: **procedure** INSERT($G, xs, l, i$)
2: $\quad tmp \leftarrow xs[i]$
3: $\quad$ while $(\neg G \lor l < i) \land tmp < xs[i - 1]$ do
4: $\quad xs[i] \leftarrow xs[i - 1]$
5: $\quad i \leftarrow i - 1$
6: $\quad xs[i] \leftarrow tmp$
Some of the Optimizations

- unguarded insertion sort
  - omit index check in insert, if \( \exists \) smaller element
  - guard controlled by flag. (\textit{insert} \( G \ \text{xs} \ l \ h \))
  - specialized for \( G = \{ \text{true}, \text{false} \} \)
- move instead of swap (insert, sift-down)
  - element gets overwritten in next loop iteration anyway
  - insert: directly implemented
  - sift-down: by refinement from version with swap

1: \textbf{procedure} \text{INSERT}(G, xs, l, i)
2: \quad \texttt{tmp} \leftarrow xs[i]
3: \quad \textbf{while} \ ((\neg G \lor l < i) \land \texttt{tmp} < \texttt{xs}[i - 1]) \ \textbf{do}
4: \quad \quad \texttt{xs}[i] \leftarrow \texttt{xs}[i - 1]
5: \quad \quad \texttt{i} \leftarrow \texttt{i} - 1
6: \quad \texttt{xs}[i] \leftarrow \texttt{tmp}
Some of the Optimizations

- unguarded insertion sort
  - omit index check in insert, if $\exists$ smaller element
  - guard controlled by flag. \( \textit{insort} \ G \ x s \ l \ h \)
  - specialized for $G=\{true, false\}$
- move instead of swap (insert, sift-down)
  - element gets overwritten in next loop iteration anyway
  - insert: directly implemented
  - sift-down: by refinement from version with swap
- manual tail-recursion optimization
  - replace second \texttt{INTROSORT\_AUX} call by loop
  - omitted in formalization
  - but done by LLVM optimizer!
Pdqsort: Algorithm

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 \( h - l < \text{threshold} \) then INSORT(lm, xs, l, h)
5:     else
6:         PIVOT\_TO\_FRONT(xs, l, h)
7:         if \( \neg lm \land xs[l - 1] \not< xs[l] \) then
8:             m ← 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) ← PARTITION\_RIGHT(xs, l, h)
13:         if \( m - l < \lceil(h - l)/8\rceil \lor h - m - 1 < \lfloor(h - l)/8\rfloor \) then
14:             if \( \neg\neg d = 0 \) then HEAPSORT(xs,l,h); return
15:             SHUFFLE(xs,l,h,m)
16:         else if ap \land MAYBE\_SORT(xs, l, m) \land 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)
Pdqsort: Verification

- Similar to introsort, but
  - more complex
  - different depth-limit implementation ($\max \#\text{unbalanced partitions}$)
  - insert inside algorithm (rather than final insert)

Verification went mostly smoothly
- heapsort, and parts of insert could be re-used
- had learned our lessons from introsort verification
- slightly more coarse-grained refinement steps
- in-bound proofs overwhelmed Isabelle's simplifier
  - solved by 'hiding' arithmetic operations behind custom constants
Pdqsort: Verification

• Similar to introsort, but
  • more complex
  • different depth-limit implementation (max # unbalanced partitions)
  • insort inside algorithm (rather than final insort)
• Verification went mostly smoothly
  • heapsort, and parts of insort could be re-used
  • had learned our lessons from introsort verification
    • slightly more coarse-grained refinement steps
  • in-bound proofs overwhelmed Isabelle’s simplifier
    • solved by ’hiding’ arithmetic operations behind custom constants
Benchmarks: Introsort (64 bit integers) (Intel laptop)

Sorting $100 \cdot 10^6$ uint64s on Intel Core i7-8665U CPU, 32GiB RAM
Benchmarks: Pdqsort (64 bit integers) (Intel laptop)

Sorting $100 \cdot 10^6$ uint64s on Intel Core i7-8665U CPU, 32GiB RAM
Benchmarks: Introsort (strings) (Intel laptop)

Sorting $10 \cdot 10^6$ strings on Intel Core i7-8665U CPU, 32GiB RAM
Benchmarks: Pdqsort (strings) (Intel laptop)

Sorting $10 \cdot 10^6$ strings on Intel Core i7-8665U CPU, 32GiB RAM
Benchmarks: Pdqsort (64 bit integers) (AMD server)

Sorting $100 \cdot 10^6$ uint64s on AMD Opteron 6176, 128GiB RAM
In the paper/formalization

- Sorting of strings
  - requires borrowing to access complex elements of array
In the paper/formalization

• Sorting of strings
  • requires borrowing to access complex elements of array
• Sorting with parameterized comparison functions
  • E.g. $i < j$ iff $a[i] < a[j]$, for array $a$
  • Engineering challenge
  • Refinement: late introduction of parameter, abstract proofs unchanged
In the paper/formalization

- Sorting of strings
  - requires borrowing to access complex elements of array
- Sorting with parameterized comparison functions
  - E.g. $i < j$ iff $a[i] < a[j]$, for array $a$
  - Engineering challenge
  - Refinement: late introduction of parameter, abstract proofs unchanged
- More benchmarks
Conclusions

- Verified state-of-the-art sorting algorithms
  - using Isabelle Refinement Framework with LLVM backend
  - as fast as libstdc++/Boost implementations
  - \( \sim 9000 \) lines of proof text, \( \sim 130 \) person hours
- Future work
  - branch aware optimization of pdqsort
  - stable sorting (mergesort, timsort, ...)
  - non-comparative/hybrid sorting (radix-sort, boost::spreadsort, ...)
- Verification Engineering (analogous to software engineering)
  - correctness + efficiency, scalability, adaptability, reusability, dev-cost, ...

Formalization, benchmarks & more
https://www21.in.tum.de/~lammich/isabelle_llvm/

Considering a PhD in formal verification?
https://tinyurl.com/PhdIsabelleLLVM