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)

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

This Paper: Overview

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.

procedure INTROSORT(xs, l, h)
    if $h - l > 1$ **then**
        INTROSORT_AUX($xs, l, h, 2\lfloor \log_2(h - l) \rfloor$)
        FINAL_INSORT(xs, l, h)
procedure INTROSORT_AUX(xs, l, h, d)
    if $h - l > \text{threshold}$ **then**
        if $d = 0$ **then** HEAPSORT(xs, l, h)
        else
            $m \leftarrow$ PARTITION_PIVOT(xs, l, h)
            INTROSORT_AUX($xs, l, m, d - 1$)
            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

```
procedure INTROSORT(xs, l, h)
    if h − l > 1 then
        INTROSORT_AUX(xs, l, h, 2⌊log2(h − l)⌋)
        FINAL_INSORT(xs, l, h)
procedure INTROSORT_AUX(xs, l, h, d)
    if h − l > threshold then
        if d = 0 then HEAPSORT(xs, l, h) ←
        else
            m ← PARTITION_PIVOT(xs, l, h)
            INTROSORT_AUX(xs, l, m, d − 1)
            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
  - use insertion sort for small partitions
    - final insort on array sorted up to threshold

```
procedure INTROSORT(xs, l, h)
    if h − l > 1 then
        INTROSORT_AUX(xs, l, h, 2⌊log₂(h − l)⌋)
        FINAL_INSORT(xs, l, h)  ←
procedure INTROSORT_AUX(xs, l, h, d)
    if h − l > threshold then  ←
        if d = 0 then HEAPSORT(xs, l, h)
        else
            m ← PARTITION_PIVOT(xs, l, h)
            INTROSORT_AUX(xs, l, m, d − 1)
            INTROSORT_AUX(xs, m, h, d − 1)
```

Verification Methodology: Modularity

- Specifications for subroutines, e.g. *heapsort* $\leq$ *sort_spec*
  - proof only uses specification
  - independant 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

```
partition_spec xs ≡ — any non-trivial partitioning
  assert (length xs ≥ 4);
  spec (xs₁,xs₂). mset xs = mset xs₁ + mset xs₂ ∧ xs₁≠[] ∧ xs₂≠[]
      ∧ (∀x∈set xs. ∀y∈set ys. x ≤ y)
```

Verification Methodology: Modularity

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

partition_spec xs ≡ — any non-trivial partitioning
  assert (length xs ≥ 4);
  spec (xs_1,xs_2). mset xs = mset xs_1 + mset xs_2 ∧ xs_1≠[] ∧ xs_2≠[]
      ∧ (∀x∈set xs. ∀y∈set ys. x ≤ y)

part_sorted_spec xs ≡ — sort up to threshold
  spec xs′. mset xs′ = mset xs ∧ part_sorted_wrt (≤) threshold xs′

where
  part_sorted_wrt n xs ≡ ∃ss. is_slicing n xs ss ∧ sorted_wrt slice_lt ss
  is_slicing n xs ss ≡ xs = concat ss ∧ (∀s∈set ss. s≠[] ∧ length s ≤ n)
  slice_lt xs ys ≡ ∀x∈set xs. ∀y∈set ys. x ≤ y

Verification Methodology: Refinement

- E.g. lists $\rightarrow$ slices of lists $\rightarrow$ arrays; $\mathbb{N} \rightarrow$ uint64_t

Verification Methodology: Refinement

- E.g. lists $\rightarrow$ slices of lists $\rightarrow$ arrays; $\mathbb{N} \rightarrow$ uint64_t

  introsort_aux1 d xs $\leq$ part_sorted_spec xs — sort whole list

  (xsi,xs)$\in$slicep_rel l h $\Longrightarrow$ — sort slice
  introsort_aux2 d xsi l h $\leq \Downarrow$(slice_rel xsi l h) (introsort_aux1 d xs)

  (introsort_aux_impl, introsort_aux2) — sort arrays, indices as uint64
  : $nat_{64} \rightarrow array^{d} \rightarrow nat_{64} \rightarrow nat_{64} \rightarrow array$

Verification Methodology: Refinement

- E.g. lists → slices of lists → arrays; $\mathbb{N}$ → uint64_t

  introsort_aux1 d xs $\leq$ part_sorted_spec xs — sort whole list

  (xsi,xs)$\in$slicep_rel l h $\Longrightarrow$ — sort slice
  introsort_aux2 d xsi l h $\leq$ $\Downarrow$(slice_rel xsi l h) (introsort_aux1 d xs)

  (introsort_aux_impl, introsort_aux2) — sort arrays, indices as uint64
  : $nat_{64} \rightarrow array^{d} \rightarrow nat_{64} \rightarrow nat_{64} \rightarrow array$

- Transitivity yields

  (introsort_aux_impl, λd. slice_part_sorted_spec)
  : $nat_{64} \rightarrow array^{d} \rightarrow nat_{64} \rightarrow nat_{64} \rightarrow array$

  where
  slice_part_sorted_spec xs l h $\equiv$ ... sort xs[l..h] up to threshold

Verification Methodology: Refinement

- E.g. lists → slices of lists → arrays; $\mathbb{N}$ → uint64_t

  introsort_aux1 d xs ≤ part_sorted_spec xs — sort whole list

  (xsi,xs)∈slicep_rel l h ⟹ — sort slice
  introsort_aux2 d xsi l h ≤ ⇓(slice_rel xsi l h) (introsort_aux1 d xs)

  (introsort_aux_impl, introsort_aux2) — sort arrays, indices as uint64
  : $nat_{64} \to array^d \to nat_{64} \to nat_{64} \to array$
- Transitivity yields

  (introsort_aux_impl, λd. slice_part_sorted_spec)
  : $nat_{64} \to array^d \to nat_{64} \to nat_{64} \to array$

  where
  slice_part_sorted_spec xs l h ≡ ... sort xs[l..h] up to threshold
- From here on, impl-details and internal refinement steps are irrelevant

Some of the Optimizations

procedure $\textsc{Insert}(G, xs, l, i)$

  $tmp \leftarrow xs[i]$

  while $(\neg G \vee l < i) \wedge tmp < xs[i-1]$ **do**

    $xs[i] \leftarrow xs[i-1]$

    $i \leftarrow i - 1$

  $xs[i] \leftarrow tmp$

Some of the Optimizations

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

procedure INSERT(G, xs, l, i)

$\quad tmp \leftarrow xs[i]$

$\quad$ **while** $(\neg G \vee l < i) \wedge tmp < xs[i-1]$ **do**

$\quad\quad xs[i] \leftarrow xs[i-1]$

$\quad\quad i \leftarrow i - 1$

$\quad xs[i] \leftarrow tmp$

Some of the Optimizations

- unguarded insertion sort
  - omit index check in insert, if $\exists$ smaller element
  - guard controlled by flag. (*insort G xs 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

procedure INSERT(G, xs, l, i)

$\quad tmp \leftarrow xs[i]$

$\quad$ **while** $(\neg G \lor l < i) \land tmp < xs[i-1]$ **do**

$\quad\quad xs[i] \leftarrow xs[i-1]$

$\quad\quad i \leftarrow i - 1$

$\quad xs[i] \leftarrow tmp$

Some of the Optimizations

- unguarded insertion sort
  - omit index check in insert, if $\exists$ smaller element
  - guard controlled by flag. (*insort G xs 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 INTROSORT_AUX call by loop
  - omitted in formalization
  - but done by LLVM optimizer!

Pdqsort: Algorithm

```
procedure PDQSORT(xs, l, h)
    if h − l > 1 then PDQSORT_AUX(true, xs, l, h, log(h − l))
procedure PDQSORT_AUX(lm, xs, l, h, d)
    if h − l < threshold then INSORT(lm, xs, l, h)
    else
        PIVOT_TO_FRONT(xs, l, h)
        if ¬lm ∧ xs[l − 1] ≮ xs[l] then
            m ← PARTITION_LEFT(xs, l, h)
            assert m + 1 ≤ h
            PDQSORT_AUX(false, xs, m + 1, h, d)
        else
            (m, ap) ← PARTITION_RIGHT(xs, l, h)
            if m − l < ⌊(h − l)/8⌋ ∨ h − m − 1 < ⌊(h − l)/8⌋ then
                if −−d = 0 then HEAPSORT(xs,l,h); return
                SHUFFLE(xs,l,h,m)
            else if ap ∧ MAYBE_SORT(xs, l, m) ∧ MAYBE_SORT(xs, m + 1, h) then
                return
            PDQSORT_AUX(lm, xs, l, m, d)
            PDQSORT_AUX(false, xs, m + 1, h, d)
```

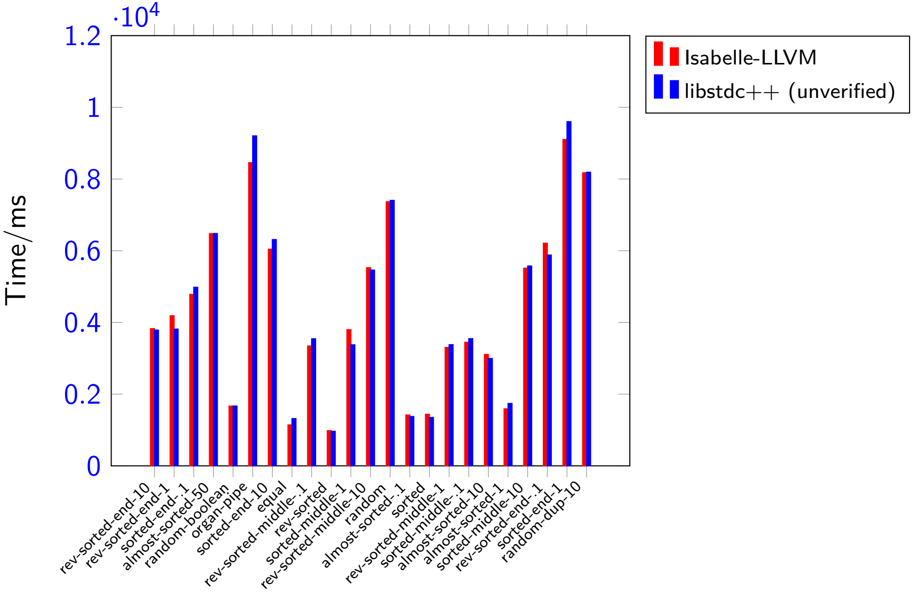
Pdqsort: Verification

- Similar to introsort, but
  - more complex
  - different depth-limit implementation (max #unbalanced partitions)
  - insort inside algorithm (rather than final insort)

Pdqsort: Verification

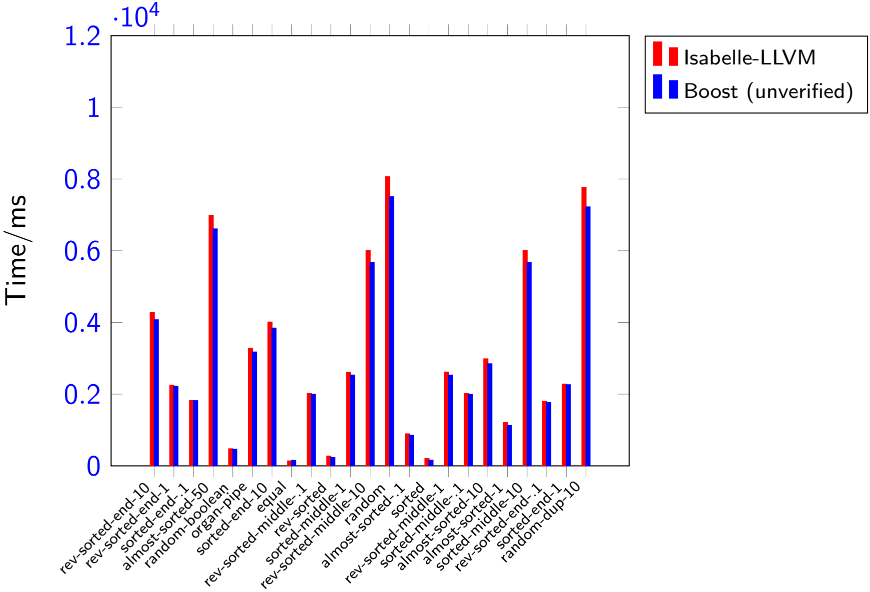
- Similar to introsort, but
  - more complex
  - different depth-limit implementation (max #unbalanced partitions)
  - insort inside algorithm (rather than final insort)
- Verification went mostly smoothly
  - heapsort, and parts of insort could be re-used
  - had learned our lessons from introsort verification
    - 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)



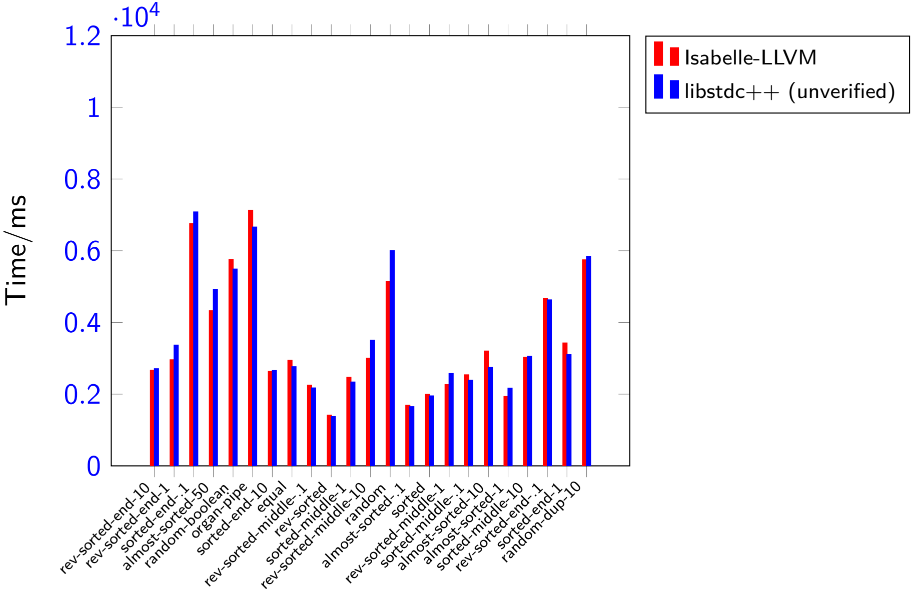
Sorting $100 \cdot 10^6$ `uint64s` on Intel Core i7-8665U CPU, 32GiB RAM

Benchmarks: Pdqsort (64 bit integers) (Intel laptop)



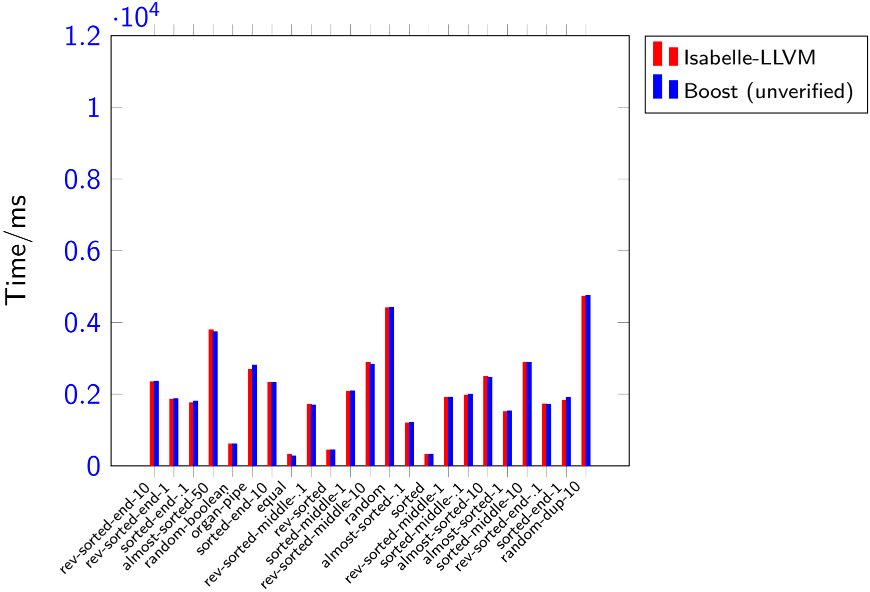
Sorting $100 \cdot 10^6$ `uint64s` on Intel Core i7-8665U CPU, 32GiB RAM

Benchmarks: Introsort (strings) (Intel laptop)



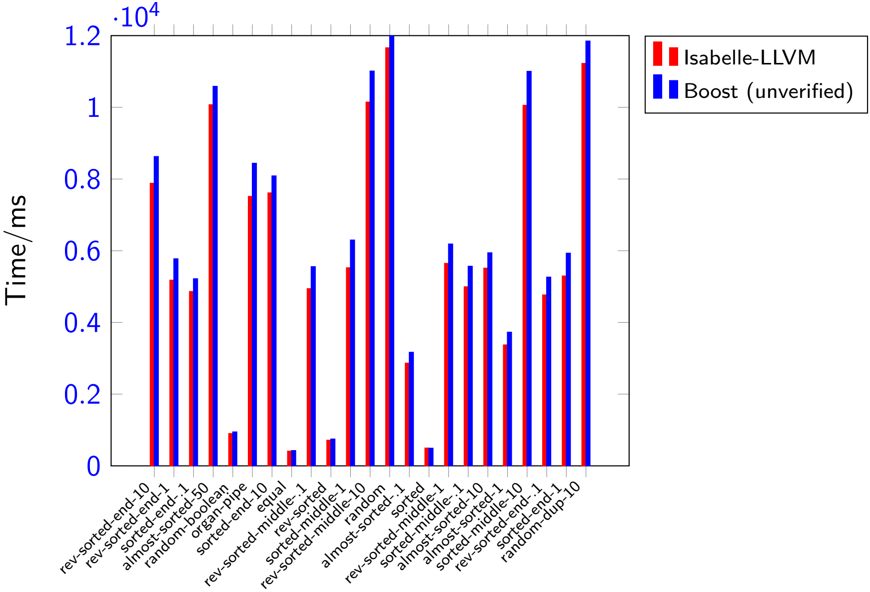
Sorting $10 \cdot 10^6$ strings on Intel Core i7-8665U CPU, 32GiB RAM

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