# For a Few Dollars More

## Verified Fine-Grained Algorithm Analysis Down to LLVM

Maximilian P. L. Haslbeck [1], Peter Lammich [2]

[1]Technical University of Munich

*haslbema@in.tum.de*

[2]University of Twente

*p.lammich@utwente.nl*

April, 2021

## C++ Standard

### 25.4.1 Sorting [alg.sort]

#### 25.4.1.1 sort [sort]

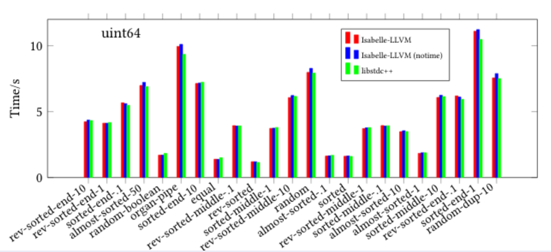
```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

*Effects:* Sorts the elements in the range `[first,last)`.

*Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

*Complexity:* $\mathcal{O}(N \log(N))$ (where $N$ `== last - first`) comparisons.

# Competitive




# C++ Standard

**25.4.1 Sorting** [alg.sort]

**25.4.1.1 sort** [sort]

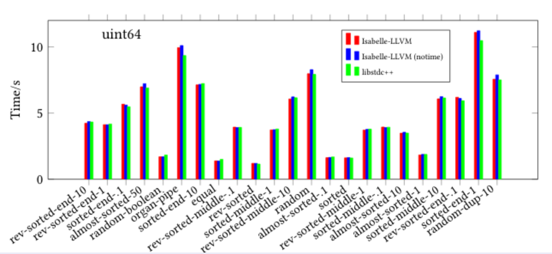
```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

*Effects:* Sorts the elements in the range `[first,last)`.

*Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

*Complexity:* $\mathcal{O}(N \log(N))$ (where $N$ `== last - first`) comparisons.

# Competitive




# C++ Standard

**25.4.1 Sorting** [alg.sort]

**25.4.1.1** `sort` [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

*Effects:* Sorts the elements in the range `[first,last)`.

*Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

*Complexity:* $\mathcal{O}(N \log(N))$ (where $N$ `== last - first`) comparisons.

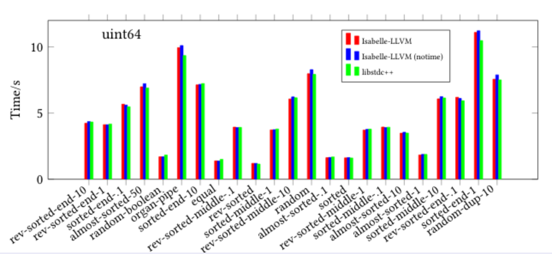
# Verified

$$\{\mathsf{array}_A\ p\ xs_0 \star \mathsf{snat}_A\ l_\dagger\ l \star \mathsf{snat}_A\ h_\dagger\ h \star \uparrow(l \le h \wedge h < |xs_0|) \star \$(\mathsf{introsort\_impl}_{cost}\ (h - l))\}$$

$$\quad \mathsf{introsort\_impl}\ p\ l_\dagger\ h_\dagger$$

$$\{\lambda r.\ \exists_A xs.\ \mathsf{array}_A\ r\ xs \star \uparrow(\mathsf{slice\_sort\_aux}\ xs_0\ l\ h\ xs) \star \mathsf{snat}_A\ l_\dagger\ l \star \mathsf{snat}_A\ h_\dagger\ h\}$$

## Competitive




## C++ Standard

**25.4.1 Sorting** [alg.sort]

**25.4.1.1 sort** [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

*Effects:* Sorts the elements in the range `[first,last)`.

*Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

*Complexity:* $\mathcal{O}(N \log(N))$ (where $N$ `== last - first`) comparisons.

## Verified

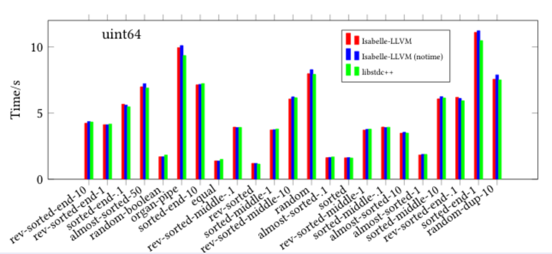
$$\{\text{array}_A\ \text{p}\ \text{xs}_0 \star \text{snat}_A\ \text{l}_\dagger\ \text{l} \star \text{snat}_A\ \text{h}_\dagger\ \text{h} \star \uparrow(\text{l} \le \text{h} \wedge \text{h} < |\text{xs}_0|) \star \$(\text{introsort\_impl}_{cost}\ (\text{h}-\text{l}))\}$$

$$\text{introsort\_impl p } \text{l}_\dagger\ \text{h}_\dagger$$

$$\{\lambda \text{r}.\ \exists_A \text{xs}.\ \text{array}_A\ \text{r xs} \star \uparrow(\text{slice\_sort\_aux xs}_0\ \text{l h xs}) \star \text{snat}_A\ \text{l}_\dagger\ \text{l} \star \text{snat}_A\ \text{h}_\dagger\ \text{h}\}$$

$$\text{introsort\_impl}_{cost} \in O(n \log n)$$

## Competitive




## C++ Standard

**25.4.1 Sorting** [alg.sort]

**25.4.1.1** `sort` [sort]

```
template<class RandomAccessIterator>
  void sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
  void sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

*Effects:* Sorts the elements in the range `[first,last)`.

*Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (17.6.3.2). The type of `*first` shall satisfy the requirements of `MoveConstructible` (Table 20) and of `MoveAssignable` (Table 22).

*Complexity:* $\mathcal{O}(N \log(N))$ (where $N$ `== last - first`) comparisons.

## Verified

$\{\mathrm{array}_A\ p\ xs_0 \star \mathrm{snat}_A\ l_\dagger\ l \star \mathrm{snat}_A\ h_\dagger\ h \star \uparrow(l \le h \wedge h < |xs_0|) \star \$(\mathrm{introsort\_impl}_{cost}\ (h-l))\}$

$\quad \mathrm{introsort\_impl}\ p\ l_\dagger\ h_\dagger$

$\{\lambda r.\ \exists_A xs.\ \mathrm{array}_A\ r\ xs \star \uparrow(\mathrm{slice\_sort\_aux}\ xs_0\ l\ h\ xs) \star \mathrm{snat}_A\ l_\dagger\ l \star \mathrm{snat}_A\ h_\dagger\ h\}$

$(\lambda n.\ \Sigma c.\ \mathrm{introsort\_impl}_{cost}\ n\ c) \in O(n \log n)$

# Top-Down Approach

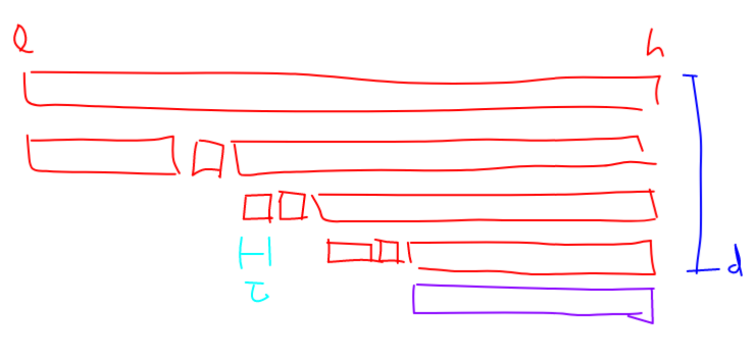
- First verification of a competitive implementation of Introsort with Time Bound
- Stepwise Refinement Calculus with Resource Currencies
- Correctness-and-Time-Bound-Preserving Synthesis Mechanism
- LLVM Semantics with Cost Model
- Basic Reasoning Infrastructure (SL + TC)

# Top-Down Approach

- First verification of a competitive implementation of Introsort with Time Bound
- Stepwise Refinement Calculus with **Resource Currencies**
- Correctness-and-Time-Bound-Preserving Synthesis Mechanism
- LLVM Semantics with Cost Model
- Basic Reasoning Infrastructure (SL + TC)

# Introsort

## Quicksort Scheme

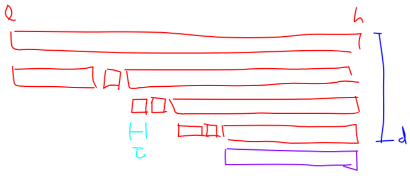


# Introsort

## Musser's Pseudocode

```
Algorithm Introsort(A, f, b)
                              Inputs: A, a random access data structure containing the sequence
                                 of data to be sorted, in positions A[f], ..., A[b − 1];
                                f, the first position of the sequence
                                b, the first position beyond the end of the sequence
                              Output: A is permuted so that A[f] ≤ A[f+1] ≤ ... ≤ A[b − 1]
    Introsort_loop(A, f, b, 2 * Floor_lg(b − f))
    Insertion_sort(A, f, b)

Algorithm Introsort_loop(A, f, b, depth_limit)
                              Inputs: A, f, b as in Introsort;
                                depth_limit, a nonnegative integer
                              Output: A is permuted so that A[i] ≤ A[j]
                                for all i, j: f ≤ i < j < b and size_threshold < j − i
    while b − f > size_threshold
        do if depth_limit = 0
                then Heapsort(A, f, b)
                    return
            depth_limit := depth_limit − 1
            p := Partition(A, f, b, Median_of_3(A[f], A[f+(b−f)/2], A[b−1]))
            Introsort_loop(A, p, b, depth_limit)
            b := p
```

## Quicksort Scheme



```
introsort xs l h =
  n ← return h−l;                   ($_sub)
  if_c n > 1 then                   ($_lt)
    xs ← almost_sort_spec xs l h;   ($_almost_sort)
    xs ← final_sort_spec xs l h;    ($_final_sort)
    return xs
  else return xs
```

```
introsort xs l h =
  n ← return h−l;                      ($_sub)
  if_c n > 1 then                      ($_lt)
    xs ← almost_sort_spec xs l h;      ($_almost_sort)
    xs ← final_sort_spec xs l h;       ($_final_sort)
    return xs
  else return xs
```

- $introsort \le slice\_sort_{spec}\ \$_{slice\_sort}$

```
introsort xs l h =
  n ← return h−l;                           ($_sub)
  if_c n > 1 then                           ($_lt)
    xs ← almost_sort_spec xs l h;            ($_almost_sort)
    xs ← final_sort_spec xs l h              ($_final_sort)
    return xs
  else return xs
```

- $\mathit{introsort} \leq \Downarrow_C E_1\ (\mathit{slice\_sort}_{\mathit{spec}}\ \$_{\mathit{slice\_sort}})$
- $E_1 :: \mathit{currency} \rightarrow \mathit{currency} \rightarrow \mathbb{N}$

```
introsort xs l h =
  n ← return h−l;                          ($_sub)
  if_c n > 1 then                          ($_lt)
    xs ← almost_sort_spec xs l h;          ($_almost_sort)
    xs ← final_sort_spec xs l h            ($_final_sort)
    return xs
  else return xs
```

- $\mathit{introsort} \leq \Downarrow_C E_1 \; (\mathit{slice\_sort}_{spec} \; \$_{slice\_sort})$
- $E_1 :: \mathit{currency} \rightarrow \mathit{currency} \rightarrow \mathbb{N}$
- $E_1 \; \mathit{slice\_sort} = \$_{sub} + \$_{lt} + \$_{if} + \$_{almost\_sort} + \$_{final\_sort}$

```
introsort_rec xs l h d =
  assert (l ≤ h);
  n ← h − l;                                  ($_sub)
  if_c n > τ then                              ($_lt)
    if_c d = 0 then                            ($_eq)
      slice_sort_spec xs l h                   ($_sort_c (μ (h-l)))
    else
      (xs, m) ← partition_spec xs l h;         ($_partition_c (h-l))
      d′ ← d − 1;                              ($_sub)
      xs ← introsort_rec xs l m d′;
      xs ← introsort_rec xs m h d′;
      return xs
  else return xs
```

```
introsort_rec xs l h d =
  assert (l ≤ h);
  n ← h − l;                                ($_sub)
  if_c n > τ then                           ($_lt)
    if_c d = 0 then                         ($_eq)
      slice_sort_spec xs l h                ($_sort_c (μ (h-l)))
    else
      (xs, m) ← partition_spec xs l h;      ($_partition_c (h-l))
      d′ ← d − 1;                           ($_sub)
      xs ← introsort_rec xs l m d′;
      xs ← introsort_rec xs m h d′;
      return xs
  else return xs
```

- $\mu\ n = n \log n$

```
introsort_rec xs l h d =
  assert (l ≤ h);
  n ← h − l;                                 ($_sub)
  if_c n > τ then                            ($_lt)
    if_c d = 0 then                          ($_eq)
      slice_sort_spec xs l h                 ($_sort_c (μ (h-l)))
    else
      (xs, m) ← partition_spec xs l h;       ($_partition_c (h-l))
      d′ ← d − 1;                            ($_sub)
      xs ← introsort_rec xs l m d′;
      xs ← introsort_rec xs m h d′;
      return xs
  else return xs
```

- $\mu \; n = n \log n$
- $introsort\_rec \leq \Downarrow_C \; E_2 \; (almost\_sort_{spec} \; \$_{almost\_sort})$

```
introsort_rec xs l h d =
  assert (l ≤ h);
  n ← h − l;                              ($_sub)
  if_c n > τ then                         ($_lt)
    if_c d = 0 then                       ($_eq)
      slice_sort_spec xs l h              ($_sort_c (μ (h-l)))
    else
      (xs, m) ← partition_spec xs l h;    ($_partition_c (h-l))
      d′ ← d − 1;                         ($_sub)
      xs ← introsort_rec xs l m d′;
      xs ← introsort_rec xs m h d′;
      return xs
  else return xs
```

- $\mu\ n = n \log n$
- $introsort\_rec \le \Downarrow_C\ E_2\ (almost\_sort_{spec}\ \$_{almost\_sort})$
- $introsort\_rec_{cost}\ (n,\ d) =$
  $\$_{sort_c}\ (\mu\ n) + \$_{partition_c}\ d * n$
  $+ ((d{+}1){*}n){*}(\$_{if}\ 2 + \$_{call}\ 2 + \$_{eq} + \$_{lt} + \$_{sub}\ 2)$

```
introsort_rec xs l h d =
  assert (l ≤ h);
  n ← h − l;                                 ($_sub)
  if_c n > τ then                            ($_lt)
    if_c d = 0 then                          ($_eq)
      slice_sort_spec xs l h                 ($_sort_c (μ (h-l)))
    else
      (xs, m) ← partition_spec xs l h;       ($_partition_c (h-l))
      d′ ← d − 1;                            ($_sub)
      xs ← introsort_rec xs l m d′;
      xs ← introsort_rec xs m h d′;
      return xs
  else return xs
```

- $\mu\ n = n \log n$
- $introsort\_rec \leq \Downarrow_C\ E_2\ (almost\_sort_{spec}\ \$_{almost\_sort})$
- $introsort\_rec_{cost}\ (n,\ d) =$
  $\$_{sort_c}\ (\mu\ n) + \$_{partition_c}\ d * n$
  $+ ((d{+}1) * n) * (\$_{if}\ 2 + \$_{call}\ 2 + \$_{eq} + \$_{lt} + \$_{sub}\ 2)$
- $E_2\ almost\_sort = introsort\_rec_{cost}\ (h - l,\ d)$

# Stepwise Refinement

- Refine $\mathit{slice\_sort}_{spec}$ with HEAPSORT in $O(n \log n)$
- Refine $\mathit{partition}_{spec}$ in $O(n)$
- Refine $\mathit{final\_sort}_{spec}$ with INSERTIONSORT in $O(\tau\ n)$
  - Unguarded Optimization
- Synthesis to LLVM Code

# Final result

$$(introsort_3,\ slice\_sort_{spec}\ introsort\_impl_{cost}) \in Id \rightarrow Id \rightarrow Id \rightarrow Id$$

# Final result

$$(\mathit{introsort}_3,\ \mathit{slice\_sort}_{\mathit{spec}}\ \mathit{introsort\_impl}_{\mathit{cost}}) \in \mathit{Id} \rightarrow \mathit{Id} \rightarrow \mathit{Id} \rightarrow \mathit{Id}$$

$$(\mathit{introsort}_{\mathit{impl}},\ \mathit{introsort}_3) \in \mathit{array}_A \rightarrow \mathit{snat}_A \rightarrow \mathit{snat}_A \rightarrow \mathit{array}_A$$

# Final result

$$(\mathit{introsort}_3,\ \mathit{slice\_sort}_{spec}\ \mathit{introsort\_impl}_{cost}) \in \mathit{Id} \to \mathit{Id} \to \mathit{Id} \to \mathit{Id}$$

$$(\mathit{introsort}_{impl},\ \mathit{introsort}_3) \in \mathit{array}_A \to \mathit{snat}_A \to \mathit{snat}_A \to \mathit{array}_A$$

$$\{\mathsf{array}_A\ \mathsf{p}\ \mathsf{xs}_0 \star \mathsf{snat}_A\ \mathsf{l}_\dagger\ \mathsf{l} \star \mathsf{snat}_A\ \mathsf{h}_\dagger\ \mathsf{h} \star \uparrow(\mathsf{l} \leq \mathsf{h} \wedge \mathsf{h} < |\mathsf{xs}_0|) \star \$(\mathsf{introsort\_impl}_{cost}\ (\mathsf{h}-\mathsf{l}))\}$$
$$\quad \mathsf{introsort\_impl}\ \mathsf{p}\ \mathsf{l}_\dagger\ \mathsf{h}_\dagger$$
$$\{\lambda \mathsf{r}.\ \exists_A \mathsf{xs}.\ \mathsf{array}_A\ \mathsf{r}\ \mathsf{xs} \star \uparrow(\mathsf{slice\_sort\_aux}\ \mathsf{xs}_0\ \mathsf{l}\ \mathsf{h}\ \mathsf{xs}) \star \mathsf{snat}_A\ \mathsf{l}_\dagger\ \mathsf{l} \star \mathsf{snat}_A\ \mathsf{h}_\dagger\ \mathsf{h}\}$$

# Final result

$$(\mathit{introsort}_3,\ \mathit{slice\_sort}_{\mathit{spec}}\ \mathit{introsort\_impl}_{\mathit{cost}}) \in \mathit{Id} \to \mathit{Id} \to \mathit{Id} \to \mathit{Id}$$

$$(\mathit{introsort}_{\mathit{impl}},\ \mathit{introsort}_3) \in \mathit{array}_A \to \mathit{snat}_A \to \mathit{snat}_A \to \mathit{array}_A$$

$\{\mathrm{array}_A\ \mathrm{p}\ \mathrm{xs}_0 \star \mathrm{snat}_A\ \mathrm{l}_\dagger\ \mathrm{l} \star \mathrm{snat}_A\ \mathrm{h}_\dagger\ \mathrm{h} \star \uparrow(\mathrm{l} \leq \mathrm{h} \wedge \mathrm{h} < |\mathrm{xs}_0|) \star \$(\mathrm{introsort\_impl}_{cost}\ (\mathrm{h}-\mathrm{l}))\}$
$\quad \mathrm{introsort\_impl}\ \mathrm{p}\ \mathrm{l}_\dagger\ \mathrm{h}_\dagger$
$\{\lambda \mathrm{r}.\ \exists_A \mathrm{xs}.\ \mathrm{array}_A\ \mathrm{r}\ \mathrm{xs} \star \uparrow(\mathrm{slice\_sort\_aux}\ \mathrm{xs}_0\ \mathrm{l}\ \mathrm{h}\ \mathrm{xs}) \star \mathrm{snat}_A\ \mathrm{l}_\dagger\ \mathrm{l} \star \mathrm{snat}_A\ \mathrm{h}_\dagger\ \mathrm{h}\}$

$(\lambda \mathrm{n}.\ \Sigma \mathrm{c}.\ \mathrm{introsort\_impl}_{cost}\ \mathrm{n}\ \mathrm{c}) \in O(n \log n)$ $\qquad$ $(\lambda \mathrm{n}.\ \mathrm{introsort\_impl}_{cost}\ \mathrm{n}\ \mathrm{cmp}) \in O(n \log n)$

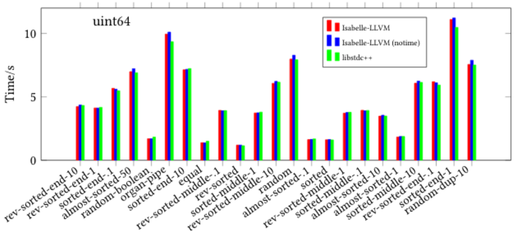
# Final result

$$(\mathit{introsort}_3,\ \mathit{slice\_sort}_{spec}\ \mathit{introsort\_impl}_{cost}) \in \mathit{Id} \to \mathit{Id} \to \mathit{Id} \to \mathit{Id}$$

$$(\mathit{introsort}_{impl},\ \mathit{introsort}_3) \in \mathit{array}_A \to \mathit{snat}_A \to \mathit{snat}_A \to \mathit{array}_A$$

$$\{\mathrm{array}_A\ \mathrm{p}\ \mathrm{xs}_0 \star \mathrm{snat}_A\ \mathrm{l}_\dagger\ \mathrm{l} \star \mathrm{snat}_A\ \mathrm{h}_\dagger\ \mathrm{h} \star \uparrow(\mathrm{l} \le \mathrm{h} \wedge \mathrm{h} < |\mathrm{xs}_0|) \star \$(\mathrm{introsort\_impl}_{cost}\ (\mathrm{h}-\mathrm{l}))\}$$
$$\mathrm{introsort\_impl}\ \mathrm{p}\ \mathrm{l}_\dagger\ \mathrm{h}_\dagger$$
$$\{\lambda \mathrm{r}.\ \exists_A \mathrm{xs}.\ \mathrm{array}_A\ \mathrm{r}\ \mathrm{xs} \star \uparrow(\mathrm{slice\_sort\_aux}\ \mathrm{xs}_0\ \mathrm{l}\ \mathrm{h}\ \mathrm{xs}) \star \mathrm{snat}_A\ \mathrm{l}_\dagger\ \mathrm{l} \star \mathrm{snat}_A\ \mathrm{h}_\dagger\ \mathrm{h}\}$$

$(\lambda \mathrm{n}.\ \Sigma \mathrm{c}.\ \mathrm{introsort\_impl}_{cost}\ \mathrm{n}\ \mathrm{c}) \in O(n \log n)$ $\quad$ $(\lambda \mathrm{n}.\ \mathrm{introsort\_impl}_{cost}\ \mathrm{n}\ \mathrm{cmp}) \in O(n \log n)$



# In the Paper

- Nondeterministic Result Monad with Time (NREST)
- Refinement Patterns + Automation
- Synthesis Mechanism (Connecting NREST with LLVM Monad)
- LLVM Semantics + Cost Model
    - Basic Reasoning Infrastructure

# Conclusion

- Verification of a State-of-the-Art Sorting Algorithm
  - Competitive and Verified
  - Stepwise Refinement with Resource Analysis
- Future Work
  - Improved Tooling
  - Further Case Studies
  - Other Resources: e.g. Stack Size

# Conclusion

- Verification of a State-of-the-Art Sorting Algorithm
  - Competitive and Verified
  - Stepwise Refinement with Resource Analysis
- Future Work
  - Improved Tooling
  - Further Case Studies
  - Other Resources: e.g. Stack Size

Formalization & Benchmarks & More:
https://www21.in.tum.de/~haslbema/llvm-time/

Thank you!