a formalized checker for size-optimal sorting networks

luís cruz-filipe
(joint work with peter schneider-kamp)

department of mathematics and computer science
university of southern denmark

types meeting
may 20th, 2015

outline

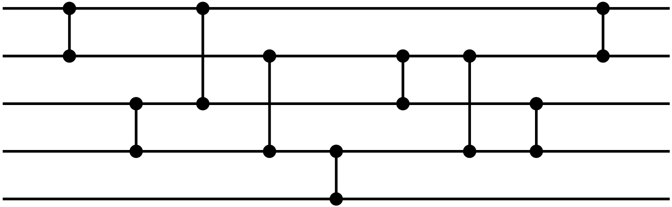
sorting networks in a nutshell

sorting networks, coq style

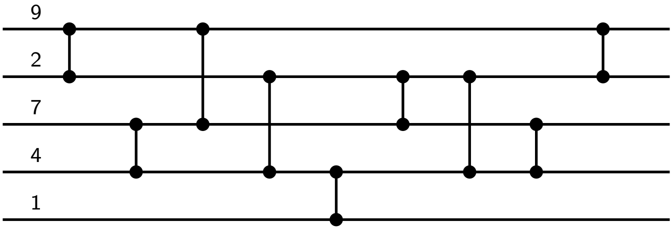
generate-and-prune

conclusions & future work

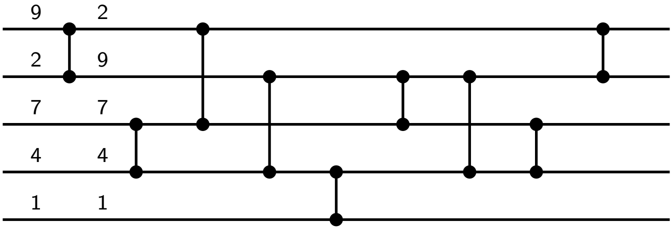
a sorting network



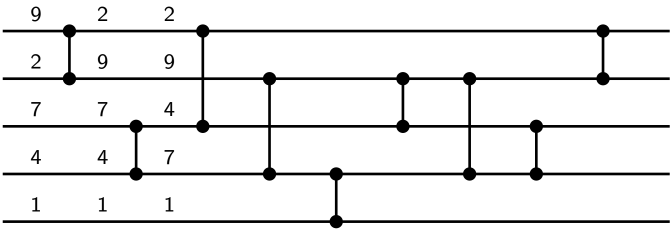
a sorting network



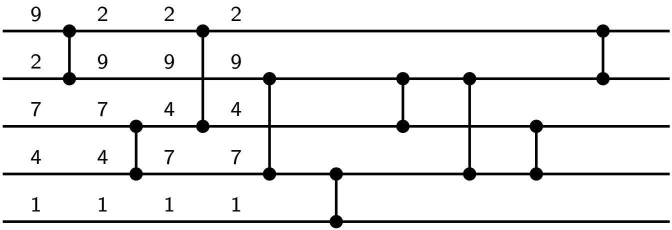
a sorting network



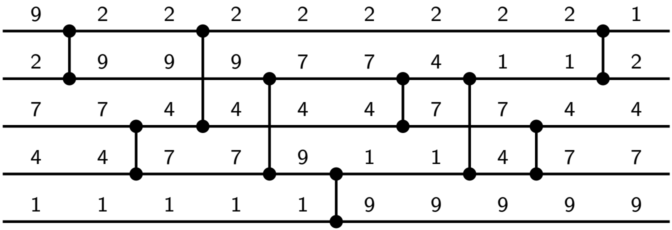
a sorting network



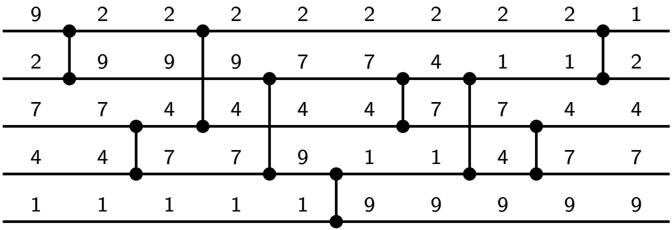
a sorting network



a sorting network

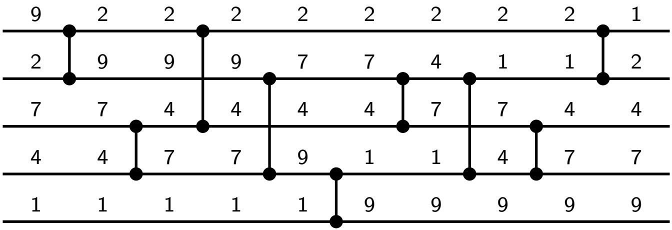


a sorting network



size this net has 5 *channels* and 9 *comparators*

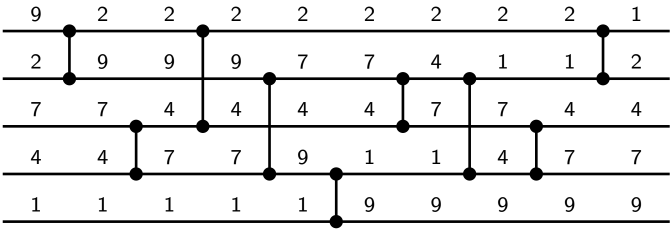
a sorting network



size this net has 5 *channels* and 9 *comparators*

more info see d.e. knuth, *the art of computer programming*, vol. 3

a sorting network



size this net has 5 *channels* and 9 *comparators*

more info see d.e. knuth, *the art of computer programming*, vol. 3

the optimal size problem what is the minimal number of *comparators* in a sorting network on n channels (s_n)?

history

optimal size

s_n: minimal number of *comparisons* to sort n inputs

knuth 1973

n	1	2	3	4	5	6	7	8	9	10
s_n	0	1	3	5	9	12	16	19	25 23	29 27

n	11	12	13	14	15	16	17
s_n	35 31	39 35	45 39	51 43	56 47	60 51	73 56

- values for $n \leq 4$ from information theory
- values for $n = 5$ and $n = 7$ by exhaustive case analysis

knuth $s_n \geq s_{n-1} + 3$ $\rightsquigarrow$ values for $n = 6, 8$

van voorhis $s_n \geq s_{n-1} + \lg(n)$ $\rightsquigarrow$ other lower bounds

history

optimal size

s_n: minimal number of *comparisons* to sort n inputs

yours truly 2014

n	1	2	3	4	5	6	7	8	9	10
s_n	0	1	3	5	9	12	16	19	**25**	**29**

n	11	12	13	14	15	16	17
s_n	35 **33**	39 **37**	45 **41**	51 **45**	56 **49**	60 **53**	73 **58**

- generate-and-prune algorithm
- intensive parallel computing
- $\sim$ 16 years of cpu time to compute s_9

history

optimal size

s_n: minimal number of *comparisons* to sort n inputs

yours truly 2014

n	1	2	3	4	5	6	7	8	9	10
s_n	0	1	3	5	9	12	16	19	**25**	**29**

n	11	12	13	14	15	16	17
s_n	35 **33**	39 **37**	45 **41**	51 **45**	56 **49**	60 **53**	73 **58**

- generate-and-prune algorithm
- intensive parallel computing
- $\sim$ 16 years of cpu time to compute s_9

- but how do we know that these results are correct?

outline

pros and cons

the easy stuff

- (very) constructive theory
- everything is decidable
- many proofs by exhaustive case analysis
- elementary definitions

pros and cons

the easy stuff

- (very) constructive theory
- everything is decidable
- many proofs by exhaustive case analysis
- elementary definitions

main challenges

- all finite domains (channels, inputs, . . .)
- reasoning about permutations (in proofs)
- very informal proofs ("trivial", "exercise", "clearly")

comparator network

sequence of *comparators* (i, j) with $1 \leq i \neq j \leq n$
n is the number of channels

```
Definition comparator : Set := (prod nat nat).
Definition comp_net : Set := list comparator.

Definition comp_channels (n:nat) (c:comparator) :=
  let (i,j) := c in (i<n) /\ (j<n) /\ (i<>j).

Definition channels (n:nat) (C:comp_net) :=
  forall c:comparator, (In c C) -> (comp_channels n c).
```

comparator networks

comparator network

sequence of *comparators* (i, j) with $1 \leq i \neq j \leq n$
n is the number of channels

```
Definition comparator : Set := (prod nat nat).
Definition comp_net : Set := list comparator.

Definition comp_channels (n:nat) (c:comparator) :=
  let (i,j) := c in (i<n) /\ (j<n) /\ (i<>j).

Definition channels (n:nat) (C:comp_net) :=
  forall c:comparator, (In c C) -> (comp_channels n c).
```

intuition

$(1, 3), (2, 4)$ is a comparator network on 4 channels, but also on 6 channels

comparator networks

comparator network

sequence of *comparators* (i, j) with $1 \leq i \neq j \leq n$
n is the number of channels

```
Definition comparator : Set := (prod nat nat).
Definition comp_net : Set := list comparator.

Definition comp_channels (n:nat) (c:comparator) :=
  let (i,j) := c in (i<n) /\ (j<n) /\ (i<>j).

Definition channels (n:nat) (C:comp_net) :=
  forall c:comparator, (In c C) -> (comp_channels n c).
```

standard

$i < j$ for all $(i, j) \in C$

```
Definition comp_standard (n:nat) (c:comparator) :=
  let (i,j) := c in (i<n) /\ (j<n) /\ (i<j).

Definition standard (n:nat) (C:comp_net) :=
  forall c:comparator, (In c C) -> (comp_standard n c).
```

sorting networks (i/iii)

0/1 lemma (knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$

sorting networks (i/iii)

0/1 lemma (knuth 1973)

C is a sorting network on n channels iff C sorts all inputs in $\{0,1\}^n$

```
Inductive bin_seq : nat -> Set :=
  | empty : bin_seq 0
  | zero : forall n:nat, bin_seq n -> bin_seq (S n)
  | one : forall n:nat, bin_seq n -> bin_seq (S n).

Fixpoint get n (s:bin_seq n) (i:nat) : nat := ...
Fixpoint set n (s:bin_seq n) (i:nat) (x:nat)
             : (bin_seq n) := ...
```

- similar to `Vector` from the standard library
- definition of `sorted` (property) and `sort` (operation)
- induction principles, exhaustive enumeration
- $\sim$ 70 lemmas in total

sorting networks (ii/iii)

output

$C(\vec{x})$ denotes the *output* of C on $\vec{x} = x_1 \ldots x_n$

```
Fixpoint apply (c:comparator) n (s:bin_seq n) : (bin_seq n) :=
  let (i,j):=c in let x:=(get s i) in let y:=(get s j) in
    match (le_lt_dec x y) with
    | left _ => s
    | right _ => set (set s j x) i y
    end.

Fixpoint full_apply (C:comp_net) n (s:bin_seq n)
  : (bin_seq n) :=
  match C with
  | nil => s
  | cons c C' => full_apply C' _ (apply c s)
  end.
```

output

$C(\vec{x})$ denotes the *output* of C on $\vec{x} = x_1 \ldots x_n$

```
Fixpoint apply (c:comparator) n (s:bin_seq n) : (bin_seq n).

Fixpoint full_apply (C:comp_net) n (s:bin_seq n) : (bin_seq n).
```

binary outputs

$\text{outputs}(C) = \{C(\vec{x}) \mid x \in \{0,1\}^n\}$

```
Definition outputs (C:comp_net) (n:nat) : (list (bin_seq n))
  := (map (full_apply C (n:=n)) (all_bin_seqs n)).
```

sorting networks (ii/iii)

output

$C(\vec{x})$ denotes the *output* of C on $\vec{x} = x_1 \ldots x_n$

```
Fixpoint apply (c:comparator) n (s:bin_seq n) : (bin_seq n).

Fixpoint full_apply (C:comp_net) n (s:bin_seq n) : (bin_seq n).
```

binary outputs

$\text{outputs}(C) = \{C(\vec{x}) \mid x \in \{0,1\}^n\}$

```
Definition outputs (C:comp_net) (n:nat) : (list (bin_seq n))
  := (map (full_apply C (n:=n)) (all_bin_seqs n)).
```

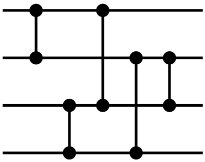
sorting network

$C(\vec{x})$ is sorted for every input $\vec{x}$

```
Definition sort_net (n:nat) (C:comp_net) :=
  (channels n C) /\
  forall s:bin_seq n, sorted (full_apply C s).

Theorem SN_char : forall C n, channels n C ->
       (forall s, In s (outputs C n) -> sorted s) ->
                 sort_net n C.
```

sorting networks (iii/iii)

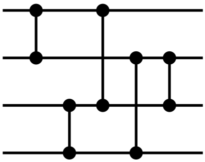
sanity check



```
Definition SN4 :=
  (0[<]1 :: 2[<]3 :: 0[<]2 ::
       1[<]3 :: 1[<]2 :: nil).

Theorem SN4_SN: sort_net 4 SN4.
```

sorting networks (iii/iii)

sanity check



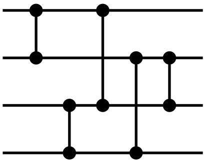
```
Definition SN4 :=
  (0[<]1 :: 2[<]3 :: 0[<]2 ::
       1[<]3 :: 1[<]2 :: nil).

Theorem SN4_SN: sort_net 4 SN4.
```

the bad news

does not scale for 9 channels

sorting networks (iii/iii)

sanity check



```
Definition SN4 :=
  (0[<]1 :: 2[<]3 :: 0[<]2 ::
       1[<]3 :: 1[<]2 :: nil).

Theorem SN4_SN: sort_net 4 SN4.
```

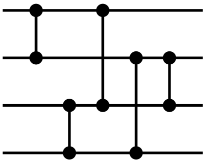
the bad news does not scale for 9 channels

the good news "C is a sorting network" is decidable

```
Lemma SN_dec : forall n C, channels n C ->
               {sort_net n C} + {~sort_net n C}.
```

sorting networks (iii/iii)

sanity check



```
Definition SN4 :=
  (0[<]1 :: 2[<]3 :: 0[<]2 ::
       1[<]3 :: 1[<]2 :: nil).

Theorem SN4_SN: sort_net 4 SN4.
```

the bad news does not scale for 9 channels

the good news "C is a sorting network" is decidable

```
Lemma SN_dec : forall n C, channels n C ->
               {sort_net n C} + {~sort_net n C}.
```

- program extraction $\leadsto$ haskell program (tests all inputs)
- nearly best possible algorithm (known result)
- short formalization ($\sim$ 35 lemmas)

the key result

output lemma (parberry 1991)

if outputs$(C) \subseteq$ outputs(C') and $C'; N$ is a sorting network, then $C; N$ is a sorting network

the key result

output lemma (parberry 1991)

if outputs(C) $\subseteq$ outputs(C') and $C'; N$ is a sorting network, then $C; N$ is a sorting network

permuted output lemma

if π(outputs(C)) $\subseteq$ outputs(C') for some permutation π and C' extends to a sorting network, then C extends to a sorting network

the key result

output lemma (parberry 1991)

if outputs(C) $\subseteq$ outputs(C') and $C'; N$ is a sorting network, then $C; N$ is a sorting network

permuted output lemma

if π(outputs(C)) $\subseteq$ outputs(C') for some permutation π and C' extends to a sorting network, then C extends to a sorting network

proof

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{C} & X & & \\ & & \downarrow \pi & & \\ \{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S \end{array}$$

the key result

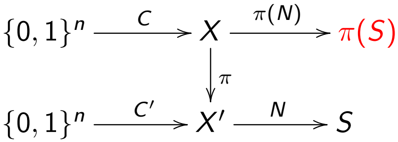
output lemma (parberry 1991)

if outputs(C) $\subseteq$ outputs(C') and $C'; N$ is a sorting network, then $C; N$ is a sorting network

permuted output lemma

if π(outputs(C)) $\subseteq$ outputs(C') for some permutation π and C' extends to a sorting network, then C extends to a sorting network

proof



the key result

output lemma (parberry 1991)

if outputs(C) $\subseteq$ outputs(C') and $C'; N$ is a sorting network, then $C; N$ is a sorting network

permuted output lemma

if π(outputs(C)) $\subseteq$ outputs(C') for some permutation π and C' extends to a sorting network, then C extends to a sorting network

proof

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{\;C\;} & X & \xrightarrow{st(\pi(N))} & S \\ & & \big\downarrow \pi & & \\ \{0,1\}^n & \xrightarrow{\;C'\;} & X' & \xrightarrow{\;N\;} & S \end{array}$$

the key result

output lemma (parberry 1991)

if $\mathrm{outputs}(C) \subseteq \mathrm{outputs}(C')$ and $C'; N$ is a sorting network, then $C; N$ is a sorting network

permuted output lemma

if $\pi(\mathrm{outputs}(C)) \subseteq \mathrm{outputs}(C')$ for some permutation π and C' extends to a sorting network, then C extends to a sorting network

proof

$$\begin{array}{ccccc} \{0,1\}^n & \xrightarrow{C} & X & \xrightarrow{st(\pi(N))} & S \\ & & \downarrow{\scriptstyle \pi} & & \\ \{0,1\}^n & \xrightarrow{C'} & X' & \xrightarrow{N} & S \end{array}$$

$\rightsquigarrow$ how do we formalize this?

standardization (i/ii)

standardization take the first non-standard comparator (i, j) and interchange i and j in all subsequent positions; repeat until network is standard

lemma if C is a sorting network, then so is $st(C)$

standardization (i/ii)

standardization take the first non-standard comparator (i, j) and interchange i and j in all subsequent positions; repeat until network is standard

lemma if C is a sorting network, then so is $st(C)$

proof the elements of outputs($st(C)$) are obtained by permuting all elements of outputs(C) in the same way; since $st(C)$ does not change sorted inputs, this permutation must be the identity

standardization (i/ii)

standardization take the first non-standard comparator (i, j) and interchange i and j in all subsequent positions; repeat until network is standard

lemma if C is a sorting network, then so is $st(C)$

proof the elements of outputs($st(C)$) are obtained by permuting all elements of outputs(C) in the same way; since $st(C)$ does not change sorted inputs, this permutation must be the identity
$\rightsquigarrow$ in our case: need a (simple?) generalization

standardization (ii/ii)

standardization

```
Function standardize (C:comp_net) {measure length C}
  : comp_net := match C with
  | nil => nil
  | cons c C' => let (x,y) := c in
      match (le_lt_dec x y) with
      | left _ => (x[<]y :: standardize C')
      | right _ => (y[<]x :: standardize (permute x y C'))
      end
  end.
```

standardization (ii/ii)

```
Function standardize (C:comp_net) {measure length C}
  : comp_net := match C with
  | nil => nil
  | cons c C' => let (x,y) := c in
      match (le_lt_dec x y) with
      | left _ => (x[<]y :: standardize C')
      | right _ => (y[<]x :: standardize (permute x y C'))
      end
  end.
```

- not structurally decreasing
- lots of implicit properties
  - preserves size and number of channels
  - preserves standard prefix
  - result is standard
  - idempotent

standardization (ii/ii)

standardization

```
Function standardize (C:comp_net) {measure length C}
  : comp_net := match C with
  | nil => nil
  | cons c C' => let (x,y) := c in
      match (le_lt_dec x y) with
      | left _ => (x[<]y :: standardize C')
      | right _ => (y[<]x :: standardize (permute x y C'))
      end
  end.
```

- not structurally decreasing
- lots of implicit properties

lemma

```
Theorem standardization_sort : forall C n,
        sort_net n C -> sort_net n (standardize C).
```

standardization (ii/ii)

standardization

```
Function standardize (C:comp_net) {measure length C}
  : comp_net := match C with
  | nil => nil
  | cons c C' => let (x,y) := c in
      match (le_lt_dec x y) with
      | left _ => (x[<]y :: standardize C')
      | right _ => (y[<]x :: standardize (permute x y C'))
      end
  end.
```

- not structurally decreasing
- lots of implicit properties

lemma

```
Theorem standardization_sort : forall C n,
        sort_net n C -> sort_net n (standardize C).
```

$\rightsquigarrow$ requires $\sim$ 60 lemmas about permutations

subsumption

definition $C \preceq_\pi C'$ if $\pi(\text{outputs}(C)) \subseteq \text{outputs}(C')$
$C \preceq C'$ if $C \preceq_\pi C'$ for some permutation π

$\rightsquigarrow$ subsumption is reflexive and transitive

subsumption

definition

$C \preceq_\pi C'$ if $\pi(\text{outputs}(C)) \subseteq \text{outputs}(C')$
$C \preceq C'$ if $C \preceq_\pi C'$ for some permutation π

```
Variable n:nat.
Variables C C':comp_net.
Variable P:permut.
Variable HP:permutation n P.

Definition subsumption :=
    forall s:bin_seq n, In s (outputs C n) ->
                        In (apply_perm P s) (outputs C' n).

Theorem BZ : standard n C -> subsumption ->
    sort_net n (C'++N) ->
    sort_net n (standardize (C ++ apply_perm_to_net P N)).

Lemma subsumption_dec : {subsumption} + {~subsumption}.
```

outline

the algorithm

init set $R_0^n = \{\emptyset\}$ and $k = 0$

repeat until $k > 1$ and $|R_k^n| = 1$

generate N_{k+1}^n extend each net in R_k^n by one comparator in all possible ways

prune to R_{k+1}^n keep only one element from each minimal equivalence class w.r.t. $\preceq^T$

step increase k

the algorithm

init set $R_0^n = \{\emptyset\}$ and $k = 0$

repeat until $k > 1$ and $|R_k^n| = 1$

generate N_{k+1}^n extend each net in R_k^n by one comparator in all possible ways

prune to R_{k+1}^n keep only one element from each minimal equivalence class w.r.t. $\preceq^T$

step increase k

pruning

- quadratic step
- inner loop searches among all permutations typically fails
- record successful subsumptions

the algorithm

init set $R_0^n = \{\emptyset\}$ and $k = 0$

repeat until $k > 1$ and $|R_k^n| = 1$

generate N_{k+1}^n extend each net in R_k^n by one comparator in all possible ways

prune to R_{k+1}^n keep only one element from each minimal equivalence class w.r.t. $\preceq^T$

step increase k

certified checker using recorded subsumptions as an oracle

- replace pruning cycle by oracle calls
- skeptic approach towards oracle
- use program extraction
- verifies all cases up to s_8, requires $\sim$18 years for s_9...

checker soundness

```
Definition Oracle := list (comp_net * comp_net * (list nat)).

Inductive Answer : Set :=
  | yes : nat -> nat -> Answer
  | no : forall n k:nat, forall R:list comp_net,
         NoDup R ->
        (forall C, In C R -> length C = k) ->
        (forall C, In C R -> standard n C) -> Answer
  | maybe : Answer.

Fixpoint Generate_and_Prune (m n:nat) (O:list Oracle) :
   Answer.

Theorem GP_no : forall m n O R HR0 HR1 HR2,
   Generate_and_Prune m n O = no m n R HR0 HR1 HR2 ->
   forall C, sort_net m C -> length C > n.

Theorem GP_yes : forall m n O k,
   Generate_and_Prune m n O = yes m k ->
  (forall C, sort_net m C -> length C >= k) /\
   exists C, sort_net m C /\ length C = k.
```

an offline oracle

typical approach

- call oracle to solve difficult tasks
- check result
- oracle is online, waiting for the next problem

an offline oracle

typical approach

- call oracle to solve difficult tasks
- check result
- oracle is online, waiting for the next problem

in our case

- oracle is pre-computed (offline)
- information from oracle guides algorithm
- potential for optimizations

improving the pruning step

old algorithm while oracle has a next subsumption $C \preceq_\pi C'$

1. check that $C \preceq_\pi C'$
2. check that C, C' are in the current set
3. remove C' from the current set

(laziness performs the last two steps together)

improving the pruning step

old algorithm

while oracle has a next subsumption $C \preceq_\pi C'$

1. check that $C \preceq_\pi C'$
2. check that C, C' are in the current set
3. remove C' from the current set

(laziness performs the last two steps together)

new algorithm

while oracle has a next subsumption $C \preceq_\pi C'$

1. check that $C \preceq_\pi C'$
2. store C
3. remove C' from the current set

after: check that all stored networks are in the final set

improving the pruning step

new algorithm while oracle has a next subsumption $C \preceq_\pi C'$

1. check that $C \preceq_\pi C'$
2. store C
3. remove C' from the current set

after: check that all stored networks are in the final set

requirement

- cannot have subsumption chains, e.g. $C_1 \preceq C_2 \preceq C_3$

pre-processing replace chains by endpoint subsumptions (e.g. $C_1 \preceq C_3$)
computing adequate permutation

$\rightsquigarrow$ don't care how, they will be checked anyway!

improving the pruning step

new algorithm

while oracle has a next subsumption $C \preceq_\pi C'$

1. check that $C \preceq_\pi C'$
2. store C
3. remove C' from the current set

after: check that all stored networks are in the final set

optimizations

- provide C's in the order they were generated (replaces quadratic step by linear)
- replace lists by search trees (improves performance)
- extract naturals to native integers (unfortunately necessary, but clearly sound)
- represent comparators as a single number (reduces memory consumption)

philosophical considerations

the good news checker verifies s_9 in around 6 days using “moderate” resources

moderate not-so-new commonplace cpu, 64 gb ram

philosophical considerations

the good news checker verifies s_9 in around 6 days using "moderate" resources

more good news (almost) no changes to the formalization

- relatively quick changes (a few hours each)
- mostly require proving that optimized version coincides with original version

philosophical considerations

the good news checker verifies s_9 in around 6 days using "moderate" resources

more good news (almost) no changes to the formalization

- relatively quick changes (a few hours each)
- mostly require proving that optimized version coincides with original version

offline oracles a new methodology?

outline

conclusions & future work

results

- formal verification of exact values of s_n for $n \leq 9$
- new methodology (offline oracles)
- able to deal with $\sim$ 27 gb of proof witnesses
- clean separation between formalization ("mathematics") and optimization of checker ("computer science")

next episodes

- formal proof of van voorhis' $s_n \geq s_{n-1} + \lg(n)$ to obtain s_{10}
- other problems in sorting networks
- application of this method to other search-intensive proofs

thank you!