Algorithm Synthesis

Synthesizing Efficient Programs by Automatically Applying Algorithmic Paradigms

Yingfei Xiong

Peking University

About Me

- Associate Professor at Peking University
- Ph.D. at the University of Tokyo, 2009
- Postdoc at University of Waterloo, 2009-2011

Data-Driven Program Synthesis

- L2S: a general framework for data-driven enumerative program synthesis [TOSEM]
- TreeGen: the first transformer-based program synthesis work [AAAI20]

Data-Driven Program Repair

- ACS: the first program repair approach whose precision > 70% [ICSE17]
- Recoder: the first neural approach outperforming traditional approaches [FSE21]

Probabilistic Fault Localization

- ProbDD: delta-debugging guided by a Bayesian model [FSE21]
- SmartFL: test-based fault localization guided by a Beyesian model [ICSE22]

Quora Questions

What are the 5 most important CS courses that every computer science student must take?

Algorithms are nominated in **all** answer with ≥ 10 votes

1. **Data structures and algorithms:** Every company asks questions from this class in coding interviews and you need to know the basics to build good software.

1. Data Structures and Algorithms - actually everyone should have two or three courses on this subject because it is the core knowledge for every type of software developer.

3. **Something mathematical.** Logic, set theory, algorithms.

What is the hardest CS undergrad course?

Algorithms are also frequently nominated.

students who are really adept with code or hardware may find classes in algorithms or cryptography to be challenging because of the math involved.

However, theory courses such as Algorithms can be really tough if you're in a challenging version of the course. It is difficult if you do not have a background in mathematical proofs.

Honorable mentions are Data Structures and Algorithms,

Why Difficult: An Example.

- Maximum segment sum (mss) problem
  - Given an integer list
  - Select a contiguous segment from the list
  - Maximize the sum of elements in the segment
  - [1, -2, 3, -2, 3] → 4

- An exhaustive program in Python
  - Time complexity: $O(n^3)$
  - Any optimization?

```python
mss = -INF
for i in range(len(xs)):
  for j in range(i, len(xs)):
    mss = max(mss, sum(xs[i: j + 1]))
return mss
```

Why Difficult: An Example.

- Apply divide-and-conquer (D&C), a well-known paradigm.

```
def dac(xs, l, r):
  if l + 1 <= r
    return (xs[l], (xs[l], xs[l], xs[l]))
  mid = (l + r) // 2
  parallel:
    resL = dac(x, l, mid)
    resR = dac(x, mid, r)
  return comb(resL, resR)
return dac(xs, 0, len)[0]
```

```
def comb(resL, resR):
  (mssL, (mpsL, mtsL, sumL)) = resL
  (mssR, (mpsR, mtsR, sumR)) = resR
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

- Time complexity: O(n/m) on m ≤ n/logn processors
- Much longer and much more difficult to write
  - D&C only suggests to divide the problem into sub-problems.
  - How to combine the sub-results remains unknown.

Research Goal: Automatic Application of Paradigms

```
mss = -INF
for i in range(len(xs)):
  for j in range(i, len(xs)):
    mss = max(mss, sum(xs[i: j + 1]))
return mss
```

Apply D&C

```
def dac(xs, l, r):
  if l + 1 <= r
    return (xs[l], (xs[l], xs[l], xs[l]))
  mid = (l + r) // 2
  parallel:
    resL = dac(x, l, mid)
    resR = dac(x, mid, r)
  return comb(resL, resR)
return dac(xs, 0, len)[0]
```

```
def comb(resL, resR):
  (mssL, (mpsL, mtsL, sumL)) = resL
  (mssR, (mpsR, mtsR, sumR)) = resR
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

Does LLM solve this problem?

Can you optimize this program as a parallel program using D&C? The expected time complexity is O(n/p), where n is the length of the input list x, and p is the number of CPU cores.

```
mss = -INF
for i in range(len(x)):
  for j in range(i, len(x)):
    mss = min(mss, sum(x[i: j+1]))
return mss
```

min instead of max

```
import threading

# Define a global variable for infinity
INF = float('inf')

# Define a helper function that returns min(a,b)
def min(a,b):
  if a < b:
    return a
  else:
    return b

# Define a recursive function that returns
#the minimum sum subarray of x[low:high+1]
def mss(x, low, high):
  # Base case: only one element
  if low == high:
    return x[low]
```

```
# Find middle index
mid = (low + high) // 2

# Create two threads for left and right halves
t1 = threading.Thread(target=mss,args=(x, low,mid))
t2 = threading.Thread(target=mss,args=(x,mid+1 ,high))

# Start both threads
t1.start()
t2.start()

# Wait for both threads to finish
t1.join()
t2.join()

# Get their results
left_mss = t1.result
right_mss = t2.result
```

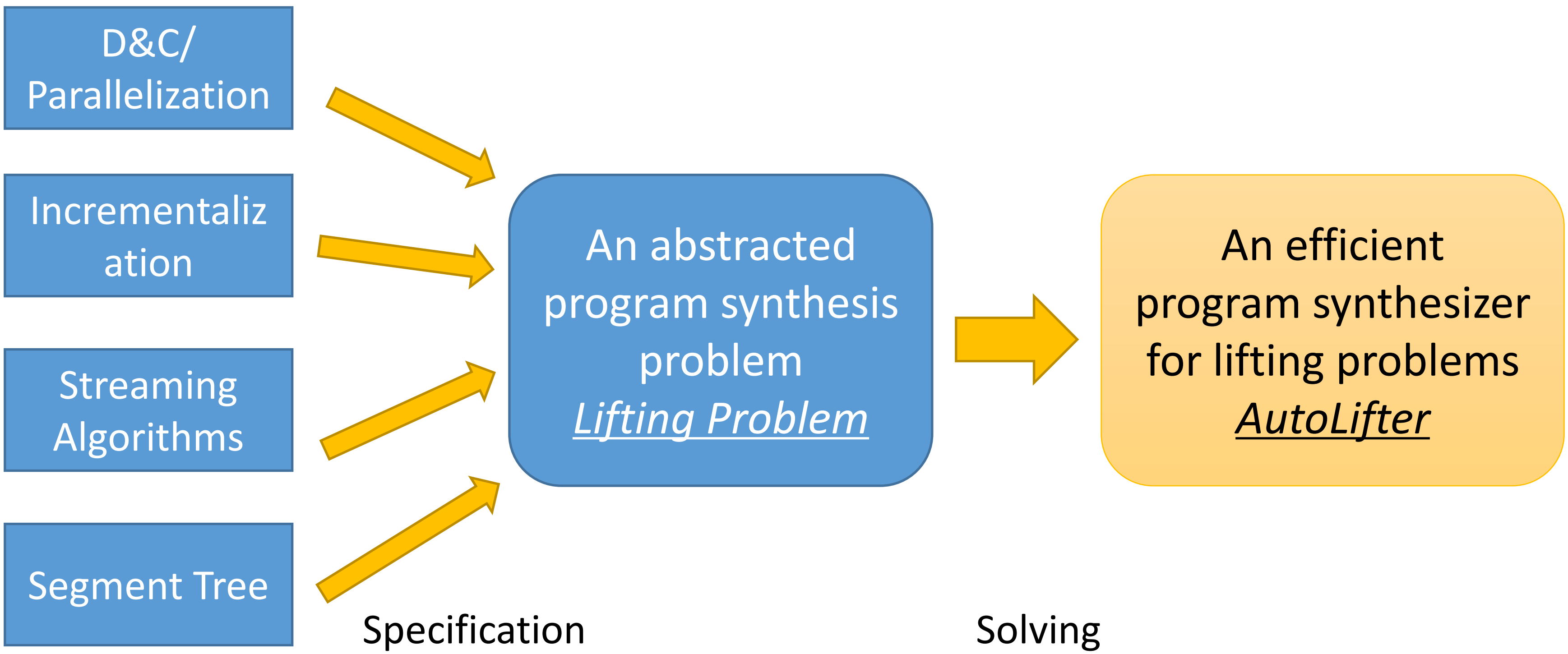
1 of 15

Our Current Progress

- A general automatic approach for applying D&C-like paradigms
  - D&C (parallelization), incremenalization, streaming algorithms, segment trees, algorithms for longest segment problems, etc.
  - Can be instantiated to different paradigms.

- Example: A synthesizer for O(n/m)-time D&C programs on lists
  - Input: An executable that produces a value from a list
    - Can be implemented using any language in any way.
  - Output: An O(n/m)-time D&C program
    - Keep the same input-output behavior as the input

Framework Overview



Manual Application of D&C

- Second minimum (sndmin) problem.
  - Given an integer list
  - Calculate the second minimum in the list

- Input Program

```
return sorted(xs)[1]
```

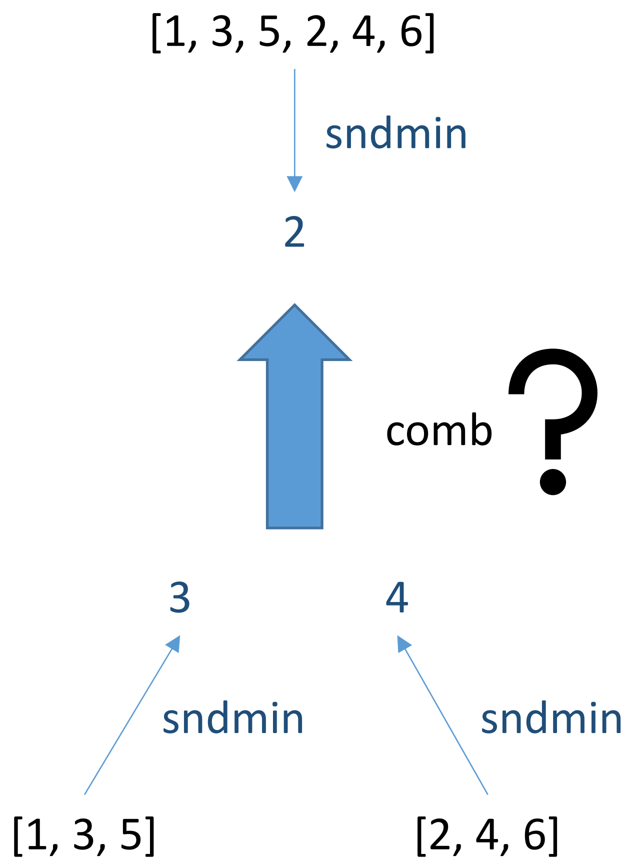
  - Time complexity: O(nlogn)

- Goal: Apply D&C to this program
  - Get an O(n/m)-time parallel program

Manual Application of D&C

- Three steps of D&C:
  1. Divide the input list xs into two halves $xs_L + xs_R$
  2. Recursively calculate the *sndmin* of the two halves
  3. Combine the sub-results to the *sndmin* of xs

- However, such a combinator does not exist.



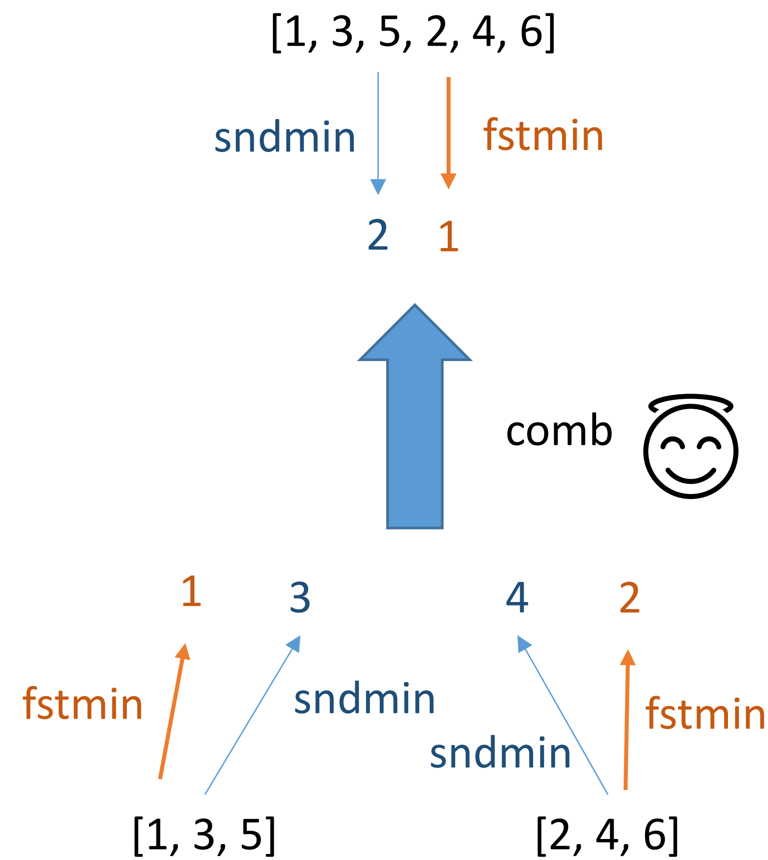
Manual Application of D&C

- Produce auxiliary values from the lists

$$aux(xs) = \min(xs)$$

- Find a combinator for both the original result and the auxiliary values

$$comb\left((snd_L, fst_L), (snd_R, fst_R)\right) \\ = \left(\min(snd_L, snd_R, \max(fst_L, fst_R))\right), \\ \min(fst_L, fst_R))$$

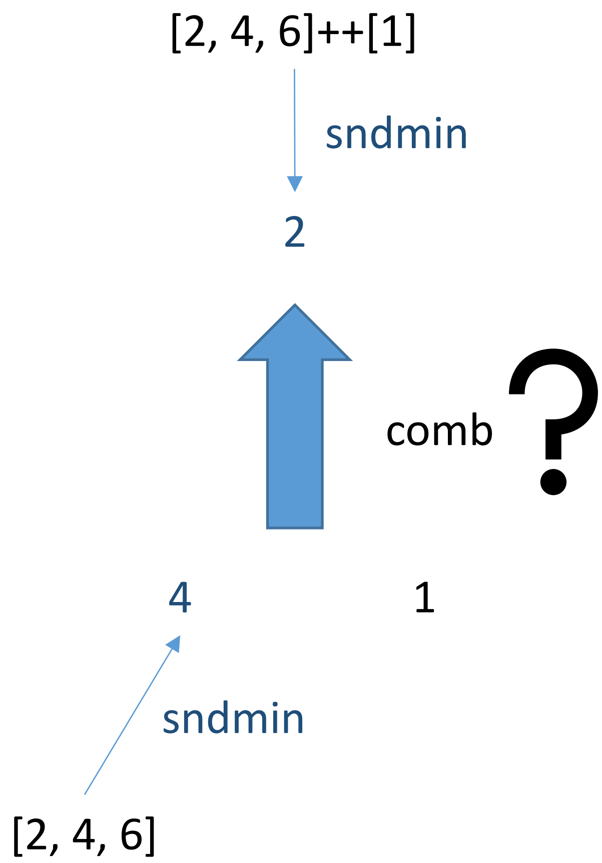


Find **auxiliary values** and a **combinator** for merging the sub-results on half lists into the whole results.

Manual Application of Incrementalization

- Problem:
  - Given a long list whose second minimum is known.
  - Now an integer is appended to this list.
  - How to efficiently update the second minimum?

- Similar to the D&C case, such a combinator does not exist.
  - The incremenalization paradigm suggests extra values from the original input to enable efficient update.



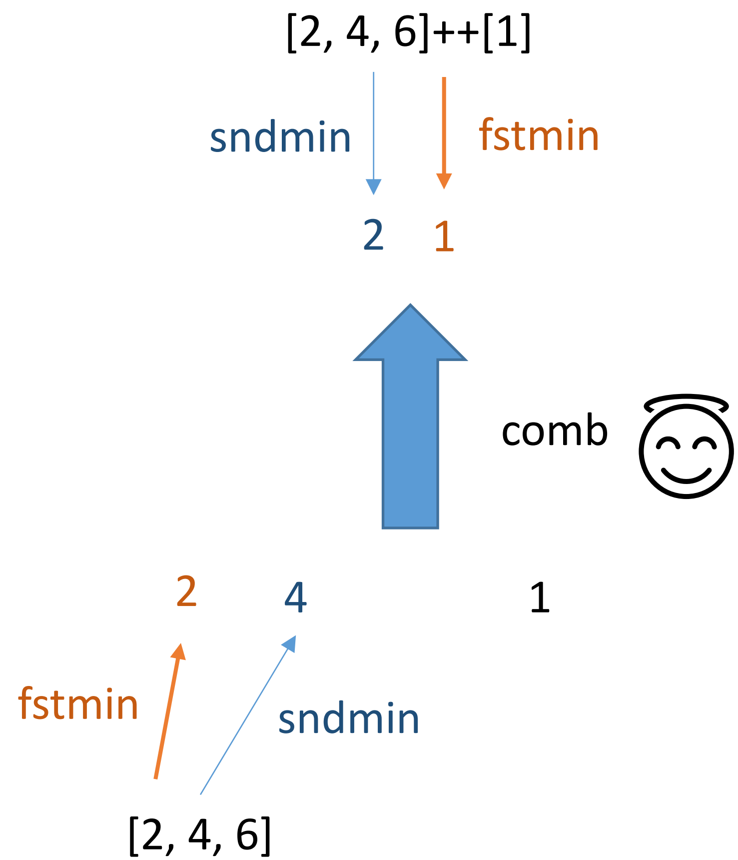
Manual Application of Incrementalization

- Produce auxiliary values from the lists

$$aux(xs) = \min(xs)$$

- Find a combinator for both the original result and the auxiliary values

$$comb\big((snd, fst), v\big) \\ = (\min(snd, \max(fst, v)), \min(fst, v))$$



Find **auxiliary values** and a **combinator** for updating the results using the previous results and the new integer.

Lifting Problem

- Input:
  1. a: instances of a data structure
  2. c: extra values
  3. orig: calculating results from the data structure
  4. op: constructing a new data structure from existing ones
- Goal: find an auxiliary program aux and a combinator $comb$ such that
  - $orig'\big(op(c, a_1, \ldots, a_n)\big) = comb\big(c, orig'(a_1), \ldots, orig'(a_n)\big)$ $where\ orig'(a) = (orig(a), aux(a))$

Lifting Problem

- Input:
  1. a: instances of a data structure
  2. c: extra values
  3. orig: calculating results from the data structure
  4. op: constructing a new data structure from existing ones
- Goal: find an auxiliary program aux and a combinator $comb$ such that
  - $orig'\big(op(c, a_1, \ldots, a_n)\big) = comb\big(c, orig'(a_1), \ldots, orig'(a_n)\big)$
    $where\ orig'(a) = (orig(a), aux(a))$

D&C on lists

$$c = ()$$
$$a_1 = [1, 3, 5]$$
$$a_2 = [2, 4, 6]$$
$$op: a_1 + a_2$$

$orig: sndmin$

Lifting Problem

- Input:
  1. a: instances of a data structure
  2. c: extra values
  3. orig: calculating results from the data structure
  4. op: constructing a new data structure from existing ones
- Goal: find an auxiliary program aux and a combinator $comb$ such that
  - $orig'\big(op(c, a_1, \ldots, a_n)\big) = comb\big(c, orig'(a_1), \ldots, orig'(a_n)\big)$
    $where\ orig'(a) = (orig(a), aux(a))$

Incrementalization

$c = 1$

$a_1 = [2, 4, 6]$

$op: a_1 + [c]$

$orig: sndmin$

Lifting Problem

- Applications of multiple algorithmic paradigms are instances of the lifting problem
  - D&C (Parallelization)
  - Incrementalization
  - Streaming Algorithms
  - Segment Trees
  - Longest segment problems
  - ……

Solving Lifting Problems as Program Synthesis Tasks

- Input
  - Specification:
    - $orig'\big(op(c, a_1, \ldots, a_n)\big) = comb\big(c, orig'(a_1), \ldots, orig'(a_n)\big)$
      $where\ orig'(a) = (orig(a), aux(a))$
  - Grammars of aux and $comb$
    - Including only O(1) operators to ensure the efficiency of the result
- Output
  - The implementation of aux and $comb$

Challenge: Scalability

- The scale of the solutions can be large.

$$aux(xs) = min(xs)$$
$$comb\big((snd_1, fst_1), (snd_2, fst_2)\big)$$
$$= (\min(snd_1, snd_2, \max(fst_1, fst_2)), \min(fst_1, fst_2))$$

sndmin

```
def aux(x):
  mps = max([sum(x[:i+1]) for i in range(lens(x))])
  mts = max([sum(x[i:]) for i in range(lens(x))])
  sumx = sum(x)
  return (mps, mts, sumx)
def comb(L, R):
  (mssL, (mpsL, mtsL, sumL)) = L
  (mssR, (mpsr, mtsR, sumR)) = R
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

Minimum segment sum

Addressing Scalability

- Divide and conquer the lifting problem
  - Derive a specification on a subpart of the target program
  - Synthesize this subpart
  - Synthesize the rest based on the subpart

- Two techniques
  - **Variable Elimination**
  - Component Elimination

$$\boxed{aux(xs) = min(xs)}$$

$$comb\big((snd_1, fst_1), (snd_2, fst_2)\big)$$
$$= (\min(snd_1, snd_2, \max(fst_1, fst_2)), \min(fst_1, fst_2))$$

Variable Elimination

$$sndmin\ (xs_L + xs_R) = comb\ (sndmin'(xs_L), sndmin'(xs_R))$$
$$\text{where } sndmin'(xs) = (sndmin\ xs, aux\ xs)$$

- In the specification, aux and $comb$ are mixed.
- Can we derive a specification only on aux?

Variable Elimination

$$sndmin\ (xs_L + xs_R) = comb\ (sndmin'(xs_L), sndmin'(xs_R))$$
$$\text{where } sndmin'(xs) = (sndmin\ xs, aux\ xs)$$

- $comb$ is a function
  - The same input leads to the same output

$$sndmin'\ (xs_L) = sndmin'(xs_L') \wedge sndmin'(xs_R) = sndmin'(xs_R')$$
$$\rightarrow sndmin(xs_L + xs_R) = sndmin(xs_L' + xs_R')$$

  - which equals

$$sndmin\ (xs_L + xs_R) \neq sndmin\ (xs_L' + xs_R')$$
$$\wedge\, sndmin\ (xs_L) = sndmin\ (xs_L') \wedge sndmin\ (xs_R) = sndmin\ (xs_R')$$
$$\Longrightarrow aux(xs_L) \neq aux(xs_L') \vee aux(xs_R) \neq aux(xs_R')$$

  - The specification includes only aux and can be solved by traditional synthesis techniques

Properties of Variable Elimination

- The decomposed specification is an approximation
  - It only ensures the existence of function $comb$
  - The function may not be implementable in the target program space
  - In such a case, the synthesis of $comb$ would fail, and backtracking is needed
- We prove that such failure is rare
  - Backtracking is seldomly needed

Current Result

- 96 tasks collected from existing datasets, existing publications for formulating algorithms, and codeforces.com.
  - maximum segment sum
  - conversion from strings to integers
  - parenthesis matching
  - a problem in 2020-2021 Winter Petrozavodsk Camp (solved only by 26 out of 243 participating teams)

An online platform for competitive programming

Problem	D&C	Single-pass	Longest Segment	Segment Tree	Total
#Task	36	39	8	13	96

- AutoLifter solves 82 out of 96 tasks with an average time cost of 6.53 tasks.
  - Significantly outperforms previous related approaches.

Summary

- Applying algorithmic paradigms is important
- Applying algorithmic paradigms is difficult
  - A paradigm only provides a high-level template.
  - The application depends on the ability of programmers.
- Is it possible to automate the application of paradigms?
  - Yes, at least for a class of paradigms similar to D&C
  - A general problem: the lifting problem.
  - The scalability challenge can be addressed by divide-and-conquer and proper approximate specifications.
- Reference
  - https://arxiv.org/abs/2202.12193

Thank you for your attention!