Superfusion: Eliminating Intermediate Data Structures via Inductive Synthesis

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 best code generation model below 50m parameters [AAAI20]
- GrammarT5: the best code generation model below 500m parameters [ICSE24]
- Deepseek-Coder: the best open source code generation model [TechReport24]

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]

Simplicity vs. Efficiency

- Task: maximum tail-segment sum (mts)
  - mts [1, -2, 3, -1, 2] = 4

Short,
Easy to write,
Easy to understand

```
mts xs = maximum (map sum (tails xs))
```

$O(n^2)$

Long,
Difficult to write
Difficult to understand

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

$O(n)$

Simplicity vs. Efficiency

- Task: maximum tail-segment sum (mts)
  - mts [1, -2, 3, -1, 2] = 4 tmts=0 tsum=0

Short,
Easy to write,
Easy to understand

```
mts xs = maximum (map sum (tails xs))
```

$O(n^2)$

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Long,
Difficult to write
Difficult to understand

$O(n)$

Simplicity vs. Efficiency

- Task: maximum tail-segment sum (mts)
  - mts [1, -2, 3, -1, 2] = 4

[1, -2, 3, -1, 2]	tmts=0	tsum=0
[1, -2, 3, -1, 2]	tmts=2	tsum=2

Short,
Easy to write,
Easy to understand

```
mts xs = maximum (map sum (tails xs))
```

$O(n^2)$

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Long,
Difficult to write
Difficult to understand

$O(n)$

Simplicity vs. Efficiency

[1, -2, 3, -1, 2]	tmts=0	tsum=0
[1, -2, 3, -1, 2]	tmts=2	tsum=2
[1, -2, 3, -1, 2]	tmts=2	tsum=1

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Simplicity vs. Efficiency

[1, -2, 3, -1, 2]	tmts=0	tsum=0
[1, -2, 3, -1, 2]	tmts=2	tsum=2
[1, -2, 3, -1, 2]	tmts=2	tsum=1
[1, -2, 3, -1, 2]	tmts=3	tsum=4

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Simplicity vs. Efficiency

- Task: maximum tail-segment sum (mts)
  - mts [1, -2, 3, -1, 2] = 4

Short,
Easy to write,
Easy to understand

```
mts xs = maximum (map sum (tails xs))
```

$O(n^2)$

Long,
Difficult to write
Difficult to understand

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

$O(n)$

Research Goal: Automatic Optimization at Algorithm Level

- Task: maximum tail-segment sum (mts)
  - mts [1, -2, 3, -1, 2] = 4

Short,
Easy to write,
Easy to understand

```
mts xs = maximum (map sum (tails xs))
```

$O(n^2)$

Long,
Difficult to write
Difficult to understand

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

$O(n)$

Previous Progress

- Automatically applying D&C-like algorithm paradigms [TOPLAS24, WG24-MTG67]
  - D&C (Parallelization)
  - Incrementalization
  - Streaming Algorithms
  - Segment Trees
  - Longest segment problems
- Automatically applying the dynamic programming paradigm [OOPSLA23]
- Problem:
  - [Scalability] A program has to be optimized as a whole
  - [Generalizability] Different algorithm paradigms, different approaches

Contribution [PLDI'24]

- A new approach that automates the application of fusion
  - Fusion: a functional algorithmic paradigm that eliminates intermediate data structure
- [Scalability] Automatically finding code pieces to rewrite
  - given a lightweight annotation
- [Generalizability] Capturing more algorithm paradigms
- [Onward] A connecting to abstract interpretation

Attempt 1

Using syntax-guided inductive synthesis

```
mts xs = maximum (map sum (tails xs))
```

Synthesize an equivalent program in O(n)

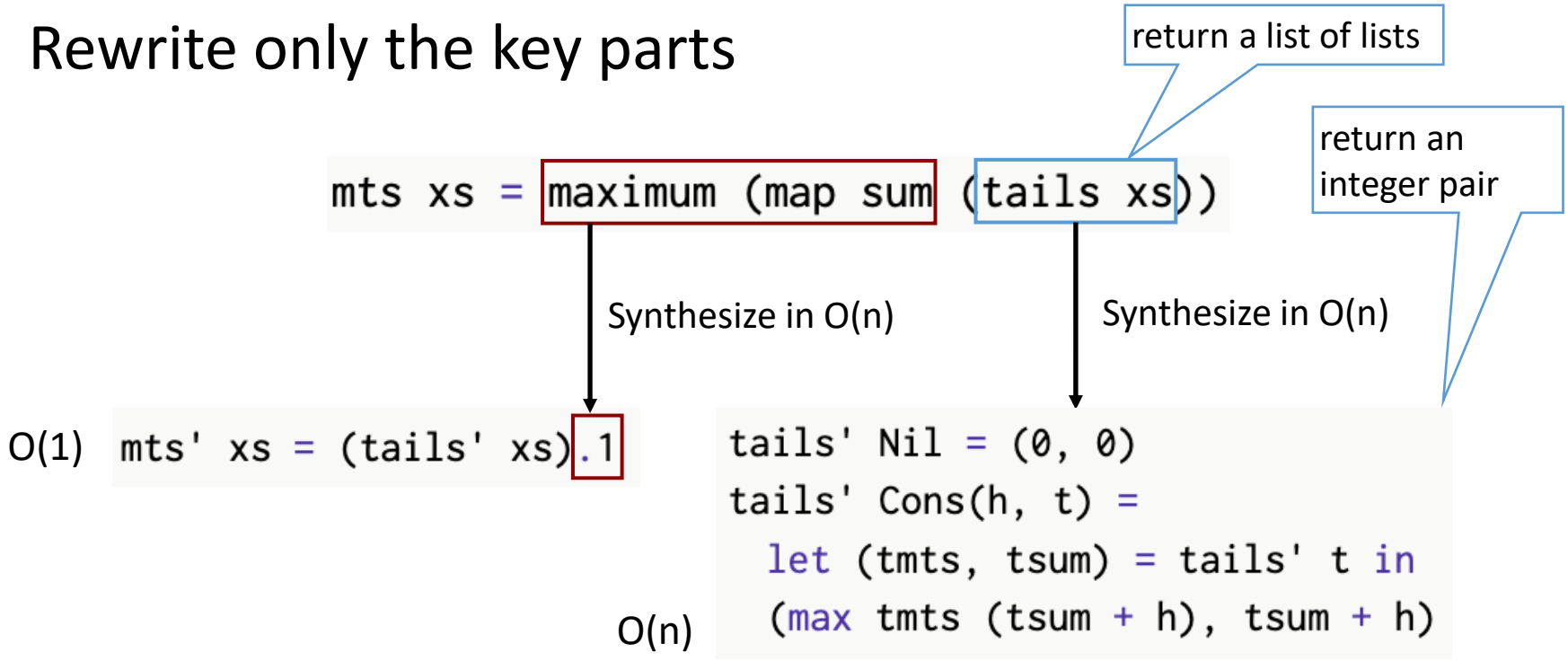
O(n) can be enforced by grammar

```
mts' xs = (tails' xs).1
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

Problem: the target program may be too large to synthesize

Attempt 2

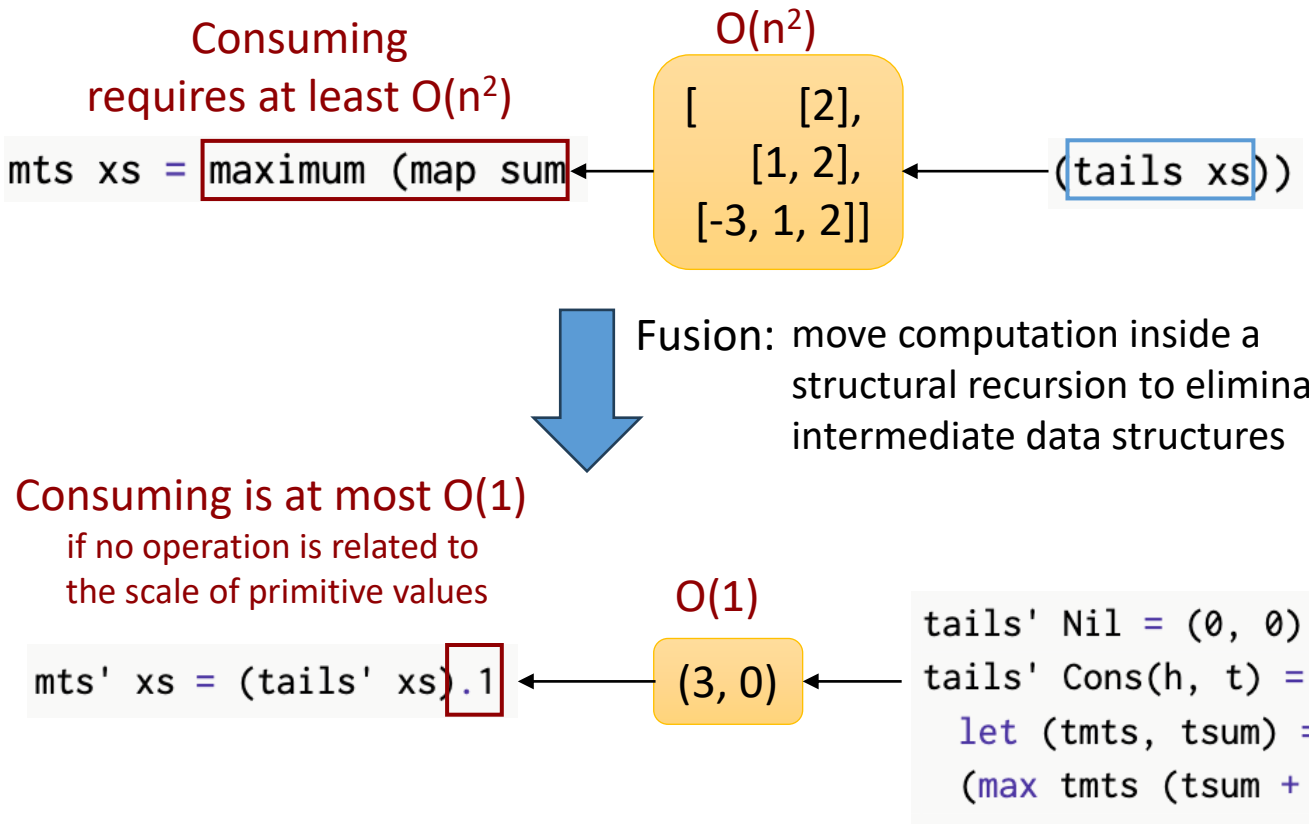
Rewrite only the key parts



Problem 1: what key parts need to be rewritten?

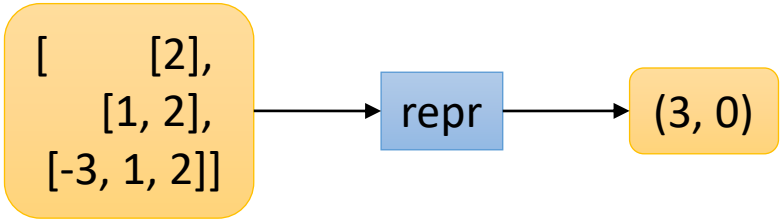
Problem 2: different parts cannot be individually synthesized.

Intermediate Data Structure



SuFu: SuperFusion

- Synthesize *repr* function, that converts between intermediate values
  - $repr: ret\ of\ tails \rightarrow ret\ of\ tails'$
- Different parts can be independently synthesized.




```
mts xs = maximum (map sum (tails xs))
```

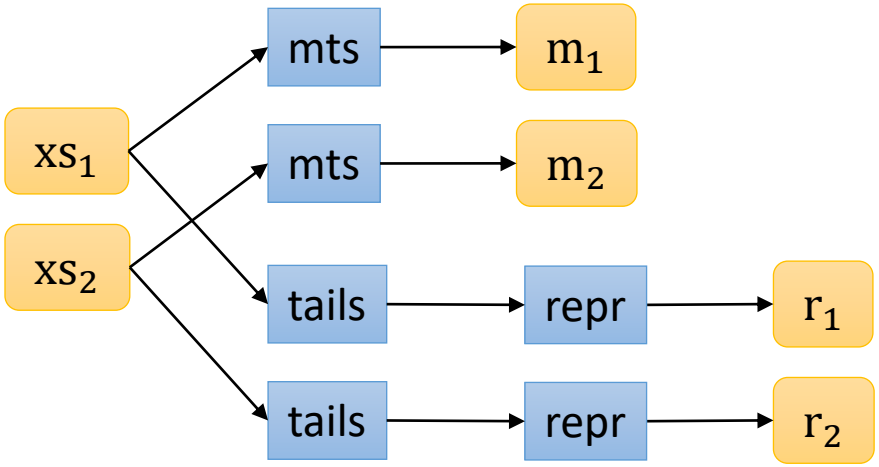
```
mts' xs = (tails' xs).1
```

```
tails' Nil = (0, 0)
tails' Cons(h, t) =
  let (tmts, tsum) = tails' t in
  (max tmts (tsum + h), tsum + h)
```

How to synthesize *repr*?

repr provides enough information for the final result

```
mts xs = maximum (map sum (tails xs))
```



If $m_1 \neq m_2$

then $r_1 \neq r_2$

Find a small *repr* that satisfy the specification

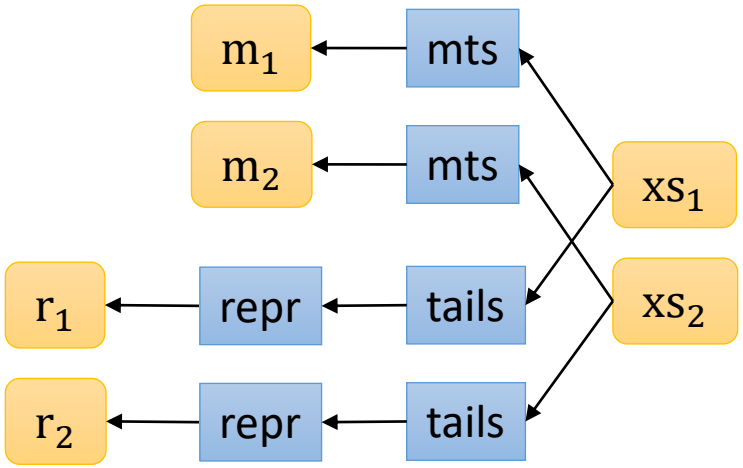
How to synthesize *repr*?

repr provides enough information for the final result

```
mts xs = maximum (map sum (tails xs))
```

If $m_1 \neq m_2$

then $r_1 \neq r_2$



Find a small *repr* that satisfy the specification

What parts need to be rewritten?

- The user marks the intermediate data structure to be eliminated
- The system infers the expressions reading/writing the data structure

```
tails :: List -> Packed NList
mts xs = maximum (map sum (tails
      xs))
```



```
tails Nil =
  NCons(Nil, NNil)
tails Cons(_, t)@xs =
  let ts = tails t in
  NCons(xs, ts)


mts xs =
  let ts = tails xs in
  maximum (map sum ts)
```

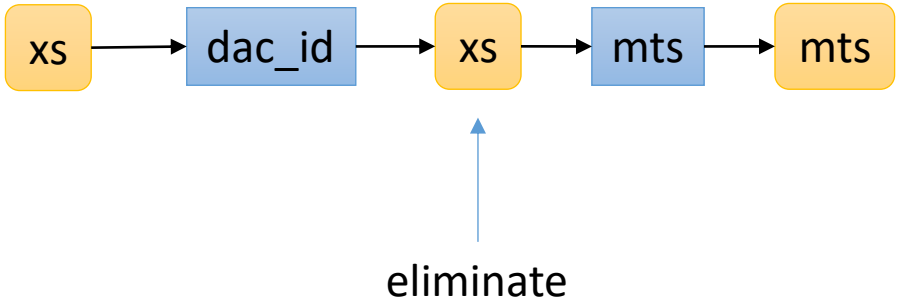
Application: supporting other algorithmic paradigms

- Many algorithmic paradigms define the ways of traversing the input data structure
- Capture them as templates

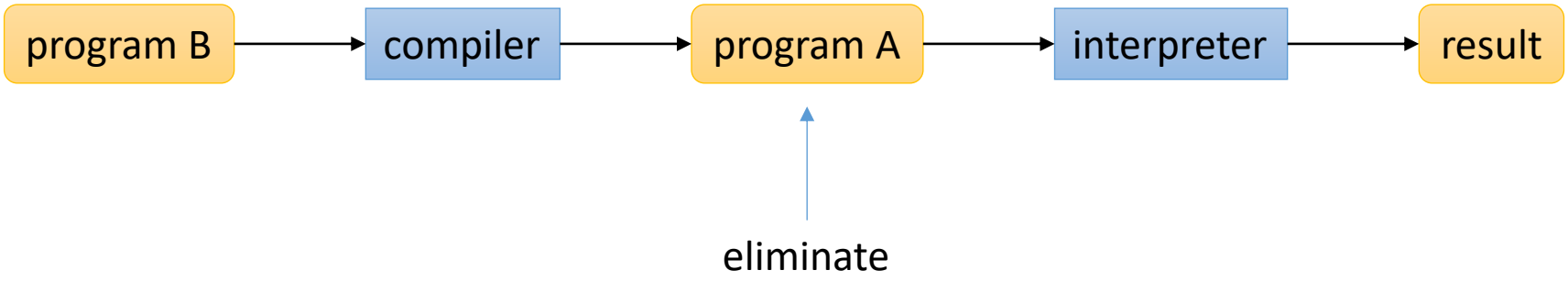
```
dac_id Elt(_)@xs = xs
dac_id Cons(_, _)@xs =
  let (ls, rs) = split xs in
  concat (dac_id ls) (dac_id rs)
```

```
mts xs = maximum (map sum (tails
      xs))
```

```
dac_id :: List -> Packed List
dac_mts xs = mts (dac_id xs)
```



Application: synthesize an interpreter



Evaluation: Benchmarks

Table 4. The profile of our dataset

Source	#Task	Size	#Packed
Fusion	16	126.5	1.250
Recursion	178	157.4	1.101
D&C	96	252.2	1.010
Total	290	187.1	1.079

[Bird 1989; Bird and de Moor 1997]

Synduce [Farzan et al. PLDI 2022]

AutoLifter [Ji et al. TOPLAS 2024]

Evaluation: Results

Table 6. Details on the performance of SuFu.

Source	Sketch Generation		Sketch Solving			
	Time	$S_{rewrite}$	Time	$S_{compress}$	$S_{extract}$	S_{holes}
Fusion	0.015	16.00	14.95	9.071	4.643	21.79
Recursion	0.018	17.65	24.73	7.876	6.759	30.41
D&C	0.037	16.39	48.98	11.64	6.563	84.66
Total	0.021	17.14	38.06	9.080	6.587	46.70

Evaluation:
Comparing to specialized solvers

Source	Tool	#Solved	Time
Recursion	SuFu	**170**	11.4
	Synduce	125	**1.7**

Source	Tool	#Solved	Time
D&C	SuFu	80	46.8
	AutoLifter	**82**	**15.6**

Demo

http://8.140.207.65/new-demo

SuperFusion

Eliminating Intermediate Data Structures via Inductive Synthesis

Reference Program

01Knapsack

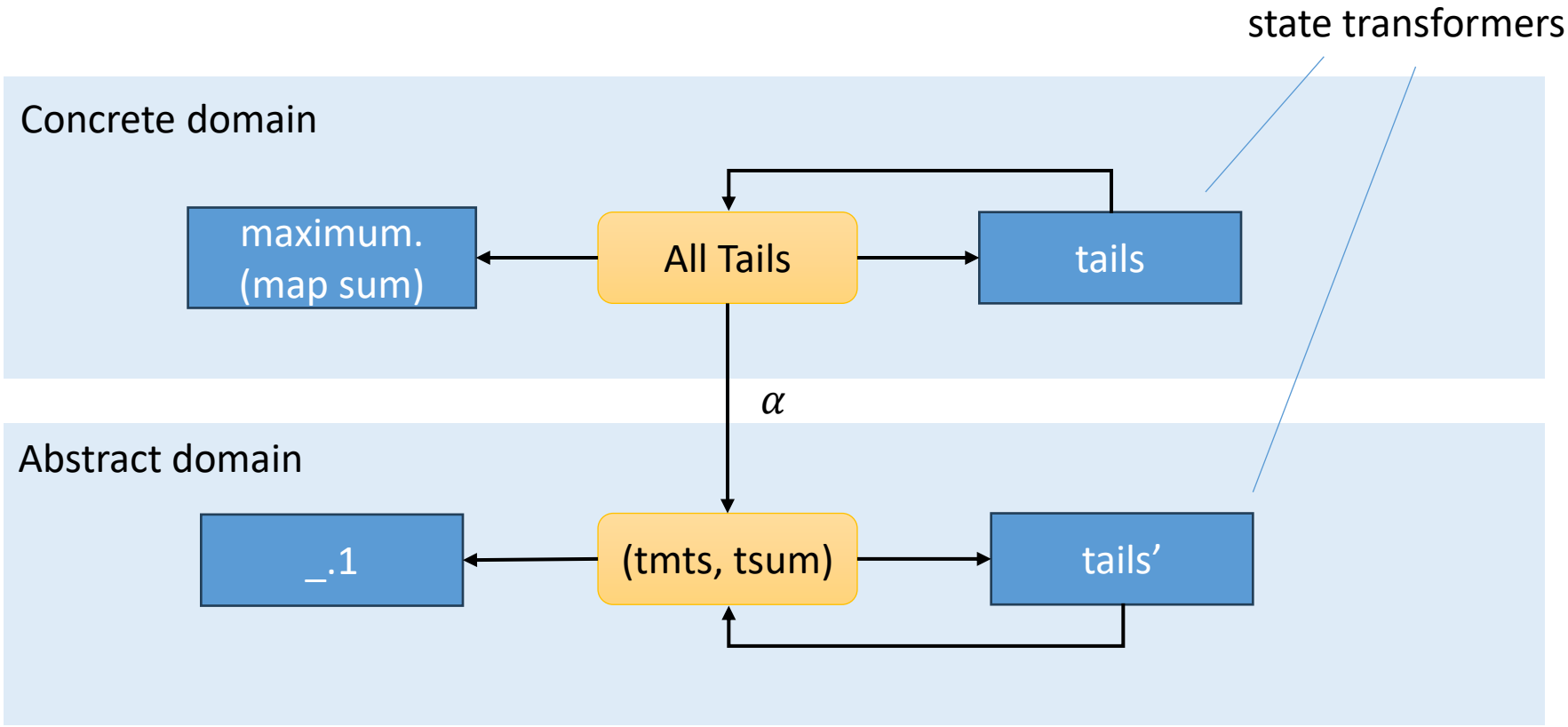
AutoLabel ☑ NonScalar ☐ Optimize

Optimized Program

```
1  import "compress";
2  import "list";
3
4  Item = Int * Int;
5  ItemList = List (Item);
6  Plan = Reframe (ItemList);
7
8  fun sumw items = sum (map fst items);
9  fun sumv items = sum (map snd items);
10
11 step :: Int -> Item -> Plan -> List (Plan);
12 fun step lim item plan =
13   if sumw plan + fst item <= lim then
14     Cons {Cons {item, plan}, Cons {plan, Nil}}
15   else Cons {plan, Nil};
16
17 gen :: Int -> ItemList -> List (Plan);
18 fun gen lim = function
```

```
1  import "list";
2
3  fun sumw items = sum (map fst items);
4
5  fun sumv items = sum (map snd items);
6
7  step :: Int -> (Int * Int) -> (Int * Int) -> List (Int * Int);
8  fun step lim item plan =
9    if (fst plan) + fst item <= lim
10   then Cons {
11       (let c0 = (fst item) in
12       let c1 = (snd item) in
13       {(fst plan) + c0, (snd plan) + c1}),
14       Cons {plan, Nil unit}
15     }
16   else Cons {plan, Nil unit};
17
18 gen :: Int -> (List (Int * Int)) -> List (Int * Int);
```

Connecting to Abstract Interpretation [Ongoing]



Question:

1. Is this perspective new?
2. What approaches for automatically designing abstract interpretation exist?
   - CEGAR, symbolic abstraction, what else?

Applications to Program Equivalence Verification [Ongoing]

- Proving program equivalence
  - forall (xs:List). sum (rev xs) = sum (sort xs)
- Challenge: inductive data structure and recursion
  - Direct induction on this proposition will get stuck
- Some propositions are easier to prove using induction
  - One structural recursion call in one hand side
- Synthesize a structural recursion function f
  - f xs = sum (sort xs)
- And change the original proposition into two
  - sum (rev xs) = f x
  - f xs = sum (sort xs)

Applications to Program Equivalence Verification [Ongoing]

	#Solved (Standard)	#Solved (Extension)	#Solved (Total)	#Fails (Timeout)	AvgTime
AutoProof	**140** (↑ **16.67%**)	**21** (↑ **600%**)	**161** (↑ **30.89%**)	**109**	**3.64s** (↓ **95.47%**)
Cvc4Ind	120	3	123	147	80.36s

Yican Sun, Ruyi Ji, Jian Fang, Xuanlin Jiang, Mingshuai Chen, Yingfei Xiong (2024). Proving Functional Program Equivalence via Directed Lemma Synthesis. Preprint. https://boyvolcano.github.io/publication/manuscript2/manuscript2.pdf

Conclusion

- Intermediate data structures
  - are bottlenecks for inefficient algorithms
- Eliminating intermediate data structures
  - leads to efficient programs
  - can be automated by program synthesis
  - has a connection to abstract interpretation
- Sufu: eliminating intermediate data structures
  - by synthesizing a *repr* function first
  - has been used to verify the correctness of the synthesized programs

Thank you for your attention!