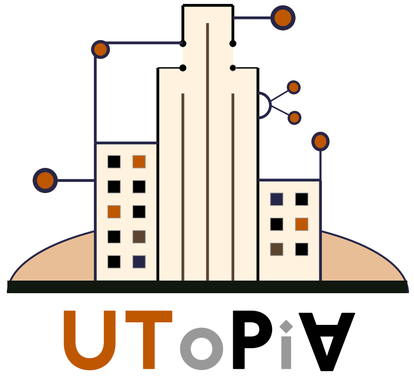
Homomorphism Calculus for User-Defined Aggregations

Ziteng Wang, Ruijie Fang, Linus Zheng, Dixin Tang, Isil Dillig

University of Texas at Austin



TEXAS

The University of Texas at Austin

data processing frameworks + udafs

background

Apache Spark™ Flink trino ClickHouse

built-in aggregations

handy but fixed standard operations

count, sum, avg, min, ...

user-defined aggregations

domain-specific logic:

- custom statistics
- weighted averages
- telemetry summaries

User-defined aggregate functions (UDAF) are everywhere in large-scale analytics

how to implement custom aggregation logic?

background

1. `init`

The starting state `I`
(zeros, empty collections, . . .)

2. `accumulate` $f : \tau_r \times \tau \to \tau_r$

Folds the next row into the state

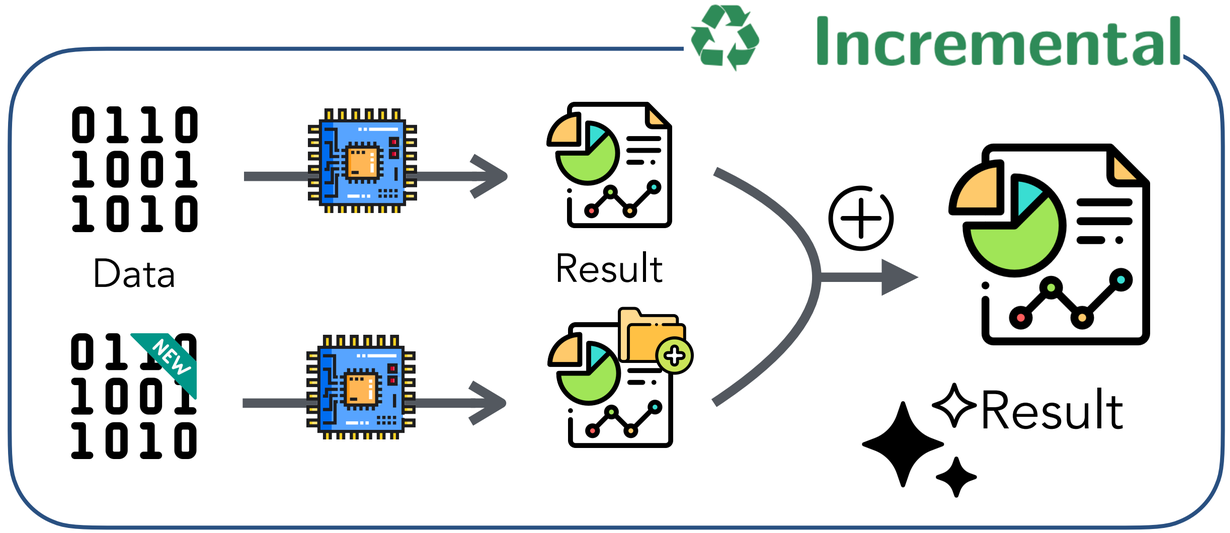
$$P(D) = \texttt{fold}(f,\ \texttt{I},\ D) = f(\ldots f(f(\texttt{I}, r_1), r_2) \ldots, r_n)$$

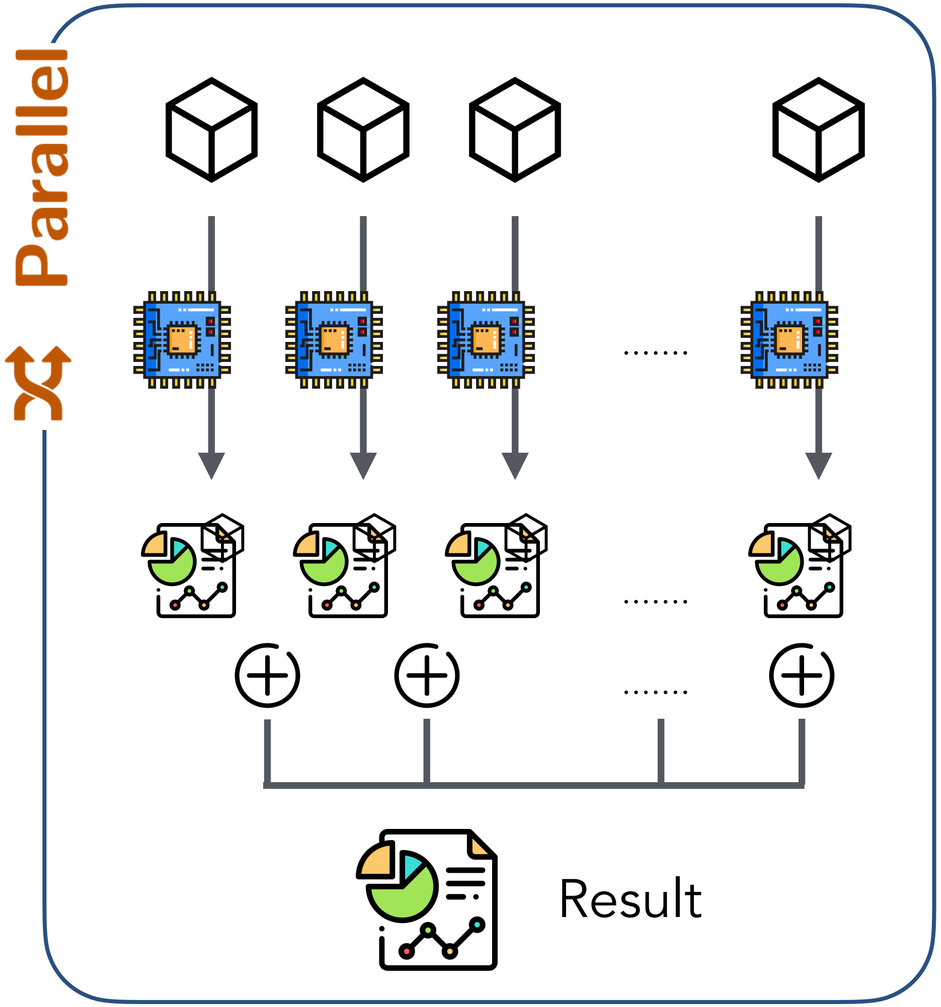
merge function

background

a **merge operator** $\oplus$ combines two partial aggregation buffers

$$b_1 \oplus b_2$$





correctness defined

background

DataFrame Homomorphism

P is a dataframe homomorphism if there exists $\oplus$ such that

$$\forall \mathcal{D}_1, \mathcal{D}_2.\ \mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2) = \mathcal{P}(\mathcal{D}_1) \oplus \mathcal{P}(\mathcal{D}_2)$$

concatenates dataframes

merges partial results

wrong merge function corrupts result

background

the merge output depends on how the data was partitioned and merged?

non-deterministic, incorrect output in distributed execution

developers write merge by hand; easy to get wrong

found bugs in

9 out of 45

sampled UDAFs

does merge always exist?

background

```
// clickstream: tracking per-user activity
def zero = Aggregate(userId = 0, eventCount = 0, checkoutCount = 0, depts = {})

def reduce(acc, e) = {
  if (acc.userId == 0) acc.userId = e.userid
  if (e.type nonEmpty) { acc.eventCount++; acc.depts.add(e.type) }
  if (e.evt == "checkout") acc.checkoutCount = acc.eventCount
}
```

can a merge function exist here?

does merge always exist?

background

```
// clickstream: tracking per-user activity
def zero = Aggregate(userId = 0, eventCount = 0, checkoutCount = 0, depts = {})

def reduce(acc, e) = {
  if (acc.userId == 0) acc.userId = e.userid
  if (e.type nonEmpty) { acc.eventCount++; acc.depts.add(e.type) }
  if (e.evt == "checkout") acc.checkoutCount = acc.eventCount
}
```

checkoutCount snapshots **eventCount** at checkout event.
Past **checkoutCount** is lost on overwriting.

searching a merge is futile!

refined problem statement

background

DataFrame Homomorphism

P is a dataframe homomorphism if there exists $\oplus$ such that

$$\forall \mathcal{D}_1, \mathcal{D}_2.\ \mathcal{P}(\mathcal{D}_1 \boxplus \mathcal{D}_2) = \mathcal{P}(\mathcal{D}_1) \oplus \mathcal{P}(\mathcal{D}_2)$$

concatenates dataframes

merges partial results

Synthesize

construct a merge operator

correct by construction

Refute

prove that no merge function exists

warn that any $\oplus$ is unsafe

refined problem statement

background

either way, developer gets a trustworthy answer

Synthesize

construct a merge operator

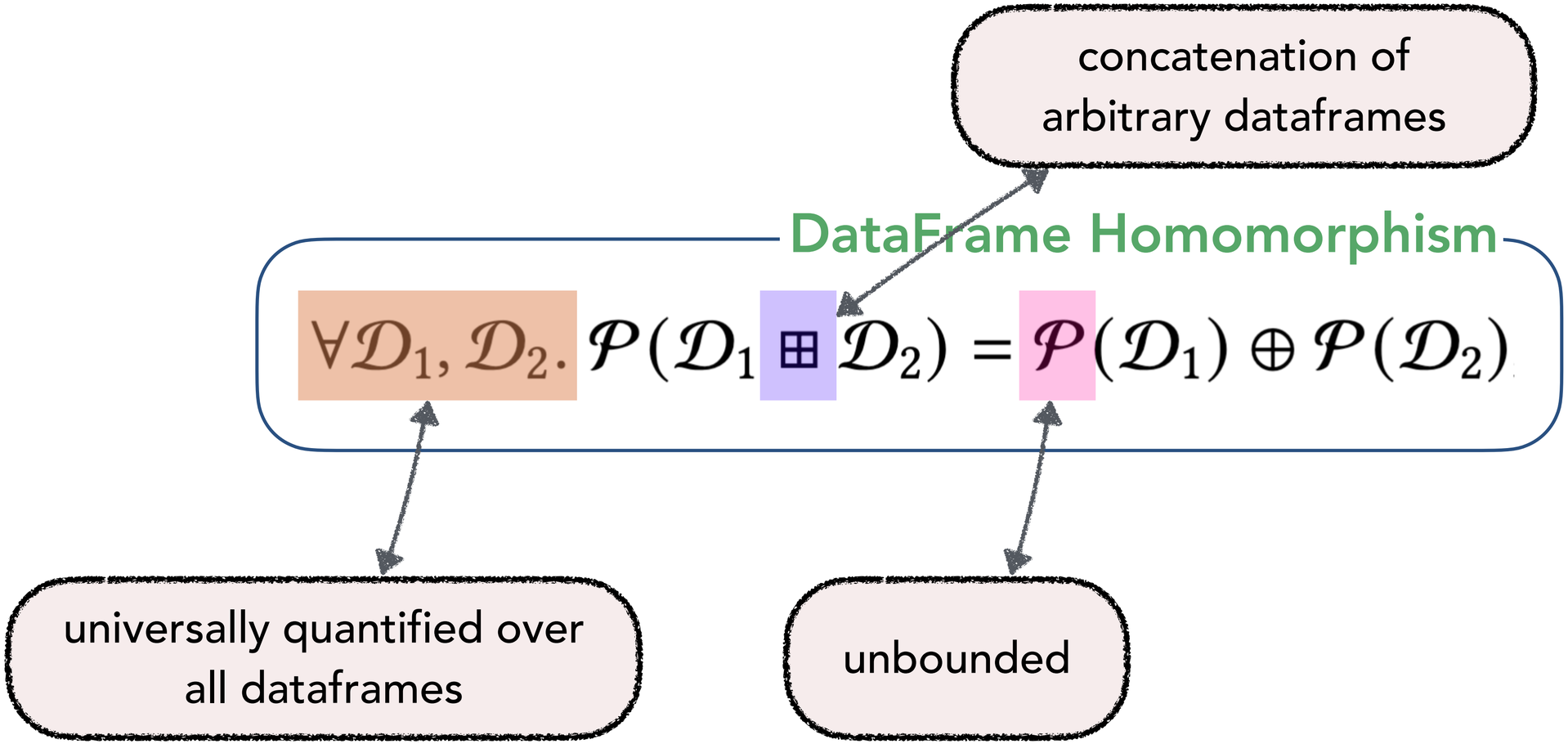
correct by construction

Refute

prove that no merge function exists

warn that any $\oplus$ is unsafe

what is the challenge?



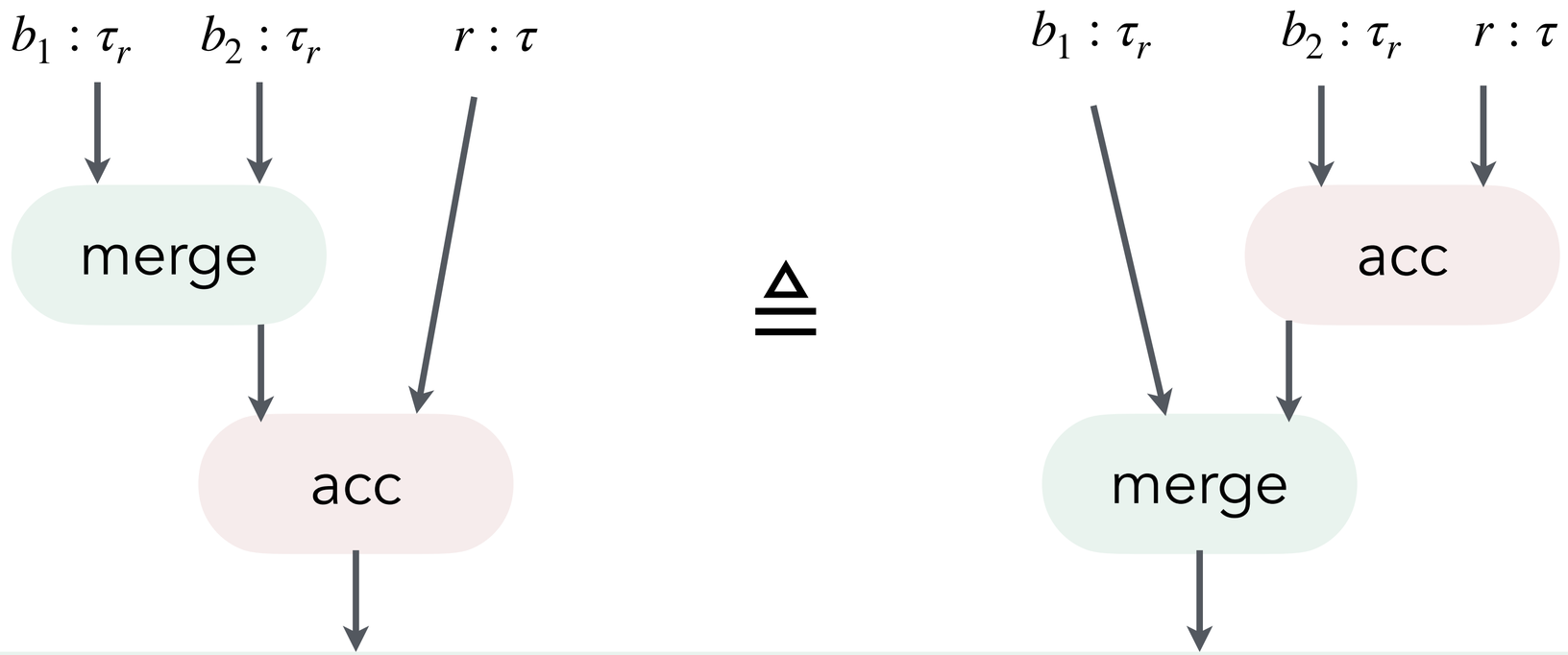
solution: normalizer

reduce verification to a **local per-row property**

accumulate function f needs to satisfy generalized commutativity condition w.r.t. merge operator $\oplus$

idea 1: the normalizer

generalized commutativity



Fold (f, I) is homomorphic iff f satisfies these properties

from verification to synthesis

(1) $\forall b_1, b_2, r.\ \texttt{reduce}(\texttt{merge}(b_1, b_2), r) = \texttt{merge}(b_1, \texttt{reduce}(b_2, r))$
(2) $\forall b.\ \texttt{merge}(b, \texttt{zero}) = b$

a UDAF is a homomorphism *iff* its accumulator f has a normalizer h, and that normalizer is the merge operator

Find a normalizer function h of the accumulator function f

SyGuS for normalizer synthesis?

SyGuS explodes once state has many fields and nested collections

```
// auction-bids UDAF
def init = Buffer(Float.MinValue, 0, {})

def reduce(buf, row) = Buffer(
    math.max(buf.maxBid, row.price),                    // highest bid
    if (row.price > 1000) buf.hi + 1 else buf.hi,        // # bids > $1000
    buf.itemCnt + (row.item → cnt + 1) )                 // per-item Map ← non-scalar
```

searching for a non-existent merge operator never terminates

tractable, per-row synthesis spec

ink

workflow

⊘ 1. Refutation rules

Necessary conditions that prove no normalizer exists — so we **never search** for a merge that cannot be found.

2. Type-directed decomposition

Break synthesis into simple subproblems by *type*; solve scalar leaves with SyGuS, **deductively combine** the rest.

idea 1: proof rules for refutation

reject hopeless problems before synthesis with a necessary condition

state doesn't change on I

state does change on s

$$\frac{\exists x, s.\ f(\mathcal{I}, x) = \mathcal{I} \wedge f(s, x) \neq s}{(f, \mathcal{I}) \sim \bot}\ (\textsc{Norm-Refute-1})$$

inconsistent; no normalizer

but synthesis is still hard

SyGuS explodes once state has many fields and nested collections

```
// auction-bids UDAF
def init = Buffer(Float.MinValue, 0, {})

def reduce(buf, row) = Buffer(
    math.max(buf.maxBid, row.price),                  // highest bid
    if (row.price > 1000) buf.hi + 1 else buf.hi,      // # bids > $1000
    buf.itemCnt + (row.item → cnt + 1) )               // per-item Map ← non-scalar
```

reject hopeless problems before search

tractable, per-row synthesis spec

idea 2: type-directed decomposition

UDAFs are often made of multiple smaller operations combined

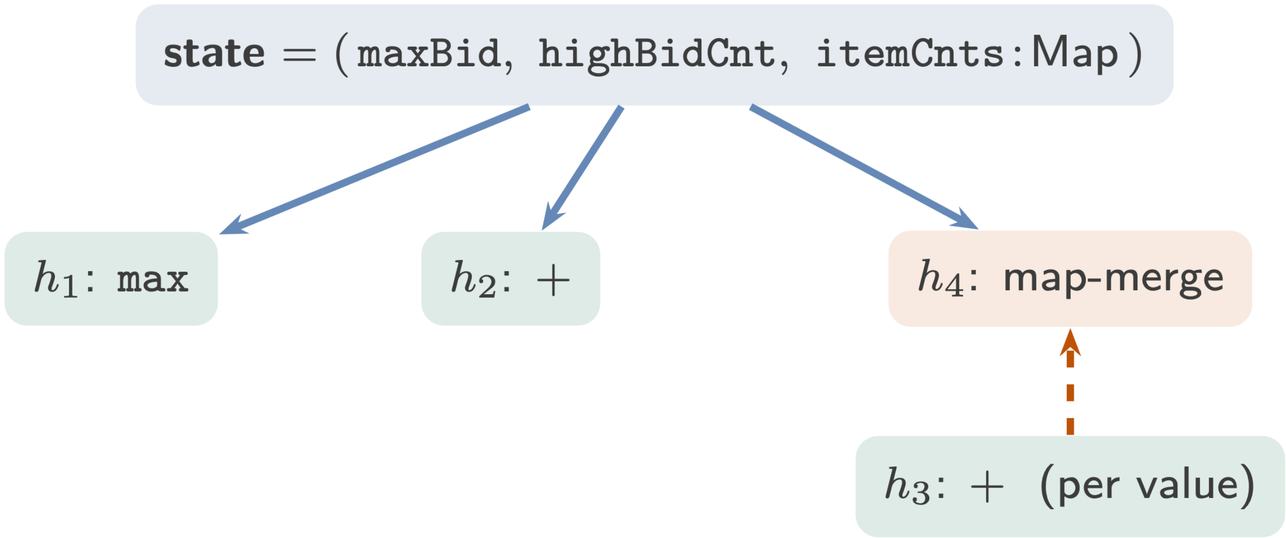
A normalizer for a composite type splits into smaller normalizers

Smaller normalizers are independent to each other

Decompose normalizer synthesis to independently solvable subproblems

idea 2: type-directed decomposition

A normalizer for a **composite type** splits into independent normalizers for its parts:



SyGuS solves normalizers for scalar leaves

Inductive (SyGuS)

derivation rules instantiate h_4 from h_3

Deductive

search only the scalar leaves; derive everything structural

recap

full dataframe property to a local per-row property

proof rules for refuting homomorphism

type-directed decomposition

how well does it work?

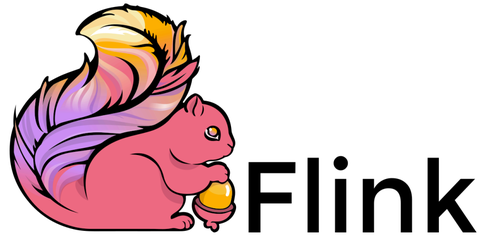
benchmark

50 tasks
collected

45
homomorphic

5
non-homomorphic

non-trivial
conditionals, compound types + collections



baseline comparison

evaluation

synthesized

43 out of 45

merge operators

refuted

5 out of 5

non-homomorphic benchmarks

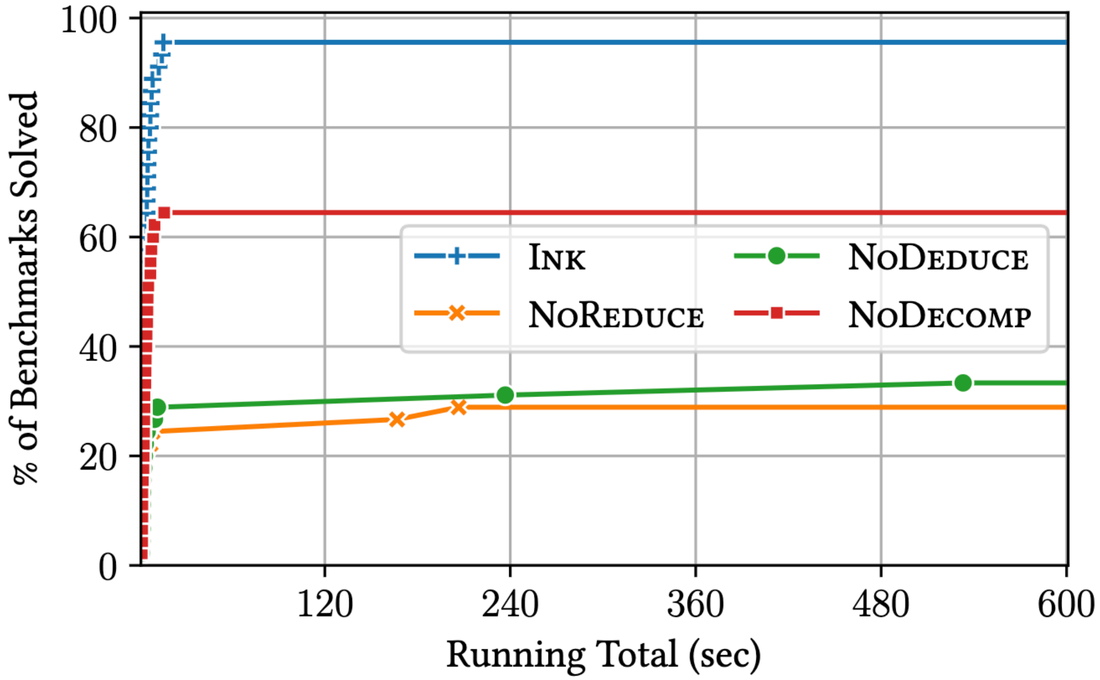
Ink synthesizes

2.3× more than ParSynt

3.4× more than CVC5

ablation study

evaluation



refuted only

2 out of 5

non-homomorphic benchmarks

Merge synthesis ablations

NoReduce

normalizer spec disabled

NoDeduce

deductive synthesis disabled

NoDecomp

no type-directed decomposition

Refutation ablations

NoRefute

refutation rules disabled

ink

Problem

Is a UDAF a homomorphism? If so, construct a correct $\oplus$ for parallel & incremental execution.

Method

Reduce to a *local* normalizer · type-directed decomposition · inductive + deductive synthesis · refutation.

Results

43/45 synthesized, 5/5 refuted, beats CVC5 & ParSynt — and **finds real bugs** in hand-written merges.