`LemmaCalc`:

Quick Theory Exploration for Algebraic Data Types via Program Transformations

Gidon Ernst, Augsburg University, LMU Munich
Grigory Fedyukovich, Florida State University

https://doi.org/10.5281/zenodo.16932462

iFM 2025, Paris
November 20

LemmaCalc:

Quick Theory Exploration for Algebraic Data Types via Program Transformations

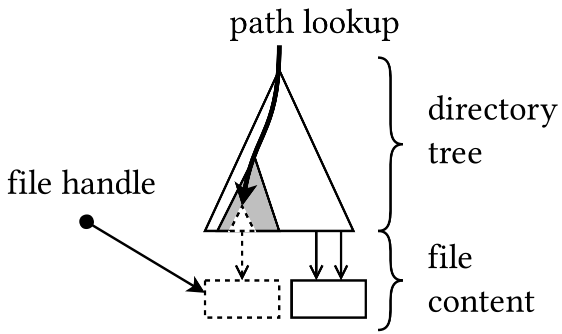
Gidon Ernst, Augsburg University, LMU Munich
Grigory Fedyukovich, Florida State University

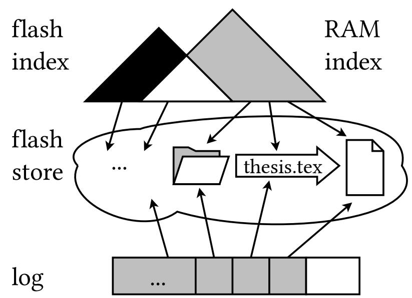
https://doi.org/10.5281/zenodo.16932462

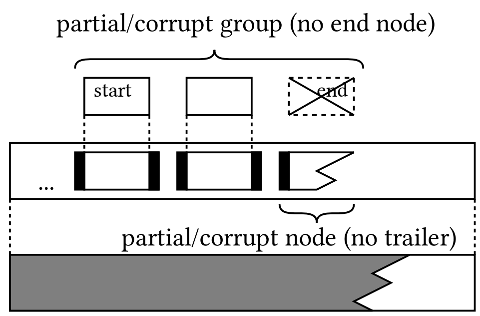
Motivation: Automated, Interactive Proofs

A typical verification case-study

- custom data types & function definitions
- expressivene logic (recursion, quantifiers)
- incremental development







Flashix [Ernst+]

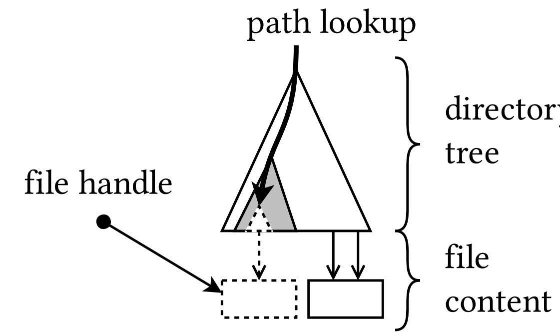
Motivation: Automated, Interactive Proofs

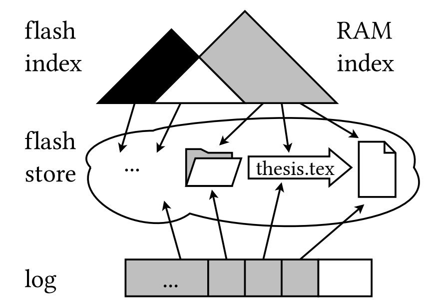
A typical verification case-study

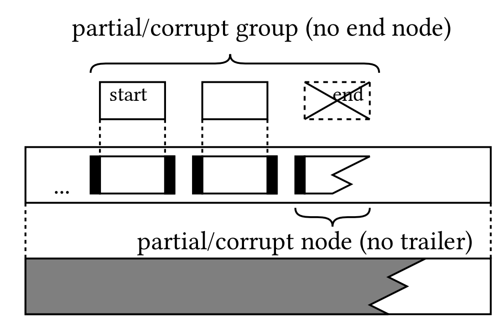
- custom data types & function definitions
- expressivene logic (recursion, quantifiers)
- **incremental development**

Bottleneck: formulating & proving lemmas

- break down proof obligations
- automate recurring proof steps







Flashix [Ernst+]

Goal: Theory Exploration

Input

```
data Nat        :=  0 | 1+ Nat
data List<a>    := [] | a :: List<a>

length([])      := 0
length(x::xs)   := 1 + length(xs)

     [] ++ ys := ys
(x::xs) ++ ys := x::(xs ++ ys)
```

inductive types

recursive functions

Goal: Theory Exploration

Input

```
data Nat        :=  0 | 1+ Nat
data List<a>    := [] | a :: List<a>

length([])      := 0
length(x :: xs) := 1 + length(xs)

     [] ++ ys := ys
(x :: xs) ++ ys := x :: (xs ++ ys)
```

inductive types

recursive functions

Output

```
lemma length(xs ++ ys)
          == length(xs) + length(ys)
 ...
```

“useful/ intersting” lemmas

Classic Approach: Term Enumeration

[Claessen, Hughes, Johansson+, Singher & Itzhaky, …]

Algorithm

```
repeat
    sample phi: Bool with |phi| < N
    if theory ∪ lemmas proves phi
        lemmas := lemmas ∪ {phi}
```

"complete" relative to search space and prover

In practice

- further restrictions: shape, few duplicate vars, …
- relevance filter: requires induction, …
- testing (QuickCheck) vs. inductive proofs (HipSpec, TheSy)

Combinatorial Search Spaces (Table 1)

		baseline enumerator statistics					TheSy	
benchmark	$\|F\|$	candidates	true	$\|\Lambda\|$	?	time	last	killed
`nat`	8	1 131 799	501	32	1759	6:50:00	26:38:14	>26h
`list`	18	319 019	408	32	522	1:48:22	10:55:14	>21h
`tree`	11	123 178	130	20	38	11:25	16:47	
`append`	5	15 058	133	22	5	02:03	04:32	
`filter`	6	398	2	5	16	02:11	00:02	
`length`	5	7 066	558	12	1	01:59	00:00	
`map`	8	34 726	103	13	35	07:18	37:33	>11h
`remove`	7	32 302	117	14	13	22:11	13:01	>11h
`reverse`	4	127 926	427	22	1	03:29	00:02	
`rotate`	6	12 784	124	20	43	08:50	6:54:22	>11h
`runlength`	7	68 311	182	23	847	†1:12:12	00:40	>11h

long runtimes

exponential in |definitions|

“interesting” lemmas are rare

productivity is very sporadic

Example Lemmas over Lists and Trees

```
length(filter(p, xs)) == countif(p, xs)
take(n, map(f, xs))   == map(f, take(n, xs))
rev(rev(xs))          == xs
sum(ys ++ zs)         == sum(ys) + sum(zs)
length(elems(t))      == size(t)
 ...
```

Example Lemmas over Lists and Trees

```
length(filter(p, xs)) = countif(p, xs)
take(n, map(f, xs))   = map(f, take(n, xs))
rev(rev(xs))          = xs
sum(ys ++ zs)         = sum(ys) + sum(zs)
length(elems(t))      = size(t)
...
```

```
...
rev(rev(xs))
  = take(length(xs), snoc(xs, zero))
```

lemmas from term enumeration are not necessarily “useful”

Contribution

LemmaCalc: an approach for theory exploration

- **quickly** explores a **structured** search space
- lemmas are **intuitive** for humans
- lemmas tend to be **useful** for automation

10.5281/zenodo.16932462

Focus: **equations** (associative/distributive laws)

Main method: integrate **program transformations** and **deduction**

Evaluation against enumerative theory exploration

- RQ1: relative explanatory strength of the sets of lemmas generated?
- RQ2: impact of the search space on lemma synthesis time?

What's a Lemma?

one way to present
a computation

$$
\begin{aligned}
(x - y)^2 &= (x - y)(x - y) \\
&= x^2 - xy - yx + y^2 \\
&= x^2 - xy - xy + y^2 \\
&= x^2 - 2xy + y^2
\end{aligned}
$$

another way to present
a computation with the same result

What's a Lemma?

$$
\begin{aligned}
&(x - y)^2 \\
&\quad = (x - y)(x - y) && \text{definition of } (\_)^2 \\
&\quad = x^2 - xy - yx + y^2 \\
&\quad = x^2 - xy - xy + y^2 && \text{use lemma } xy = yx \\
&\quad = x^2 - 2xy + y^2
\end{aligned}
$$

What's a Lemma?

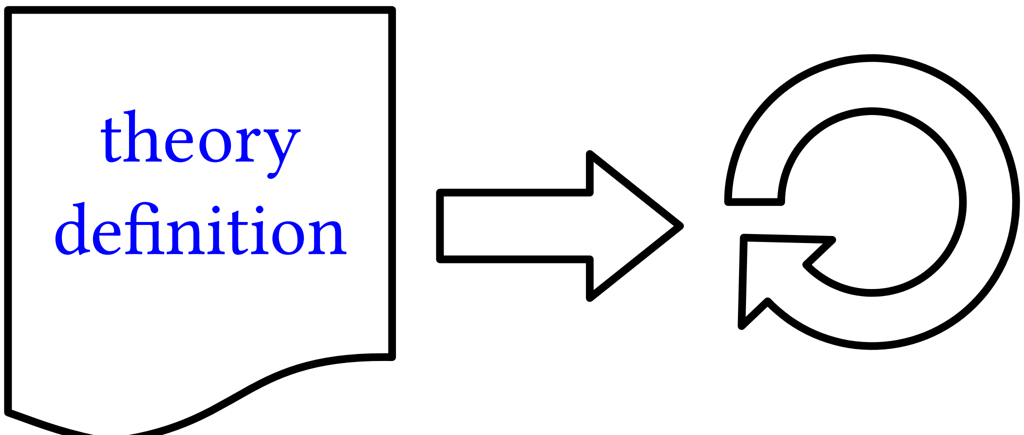
one way to present
a computation

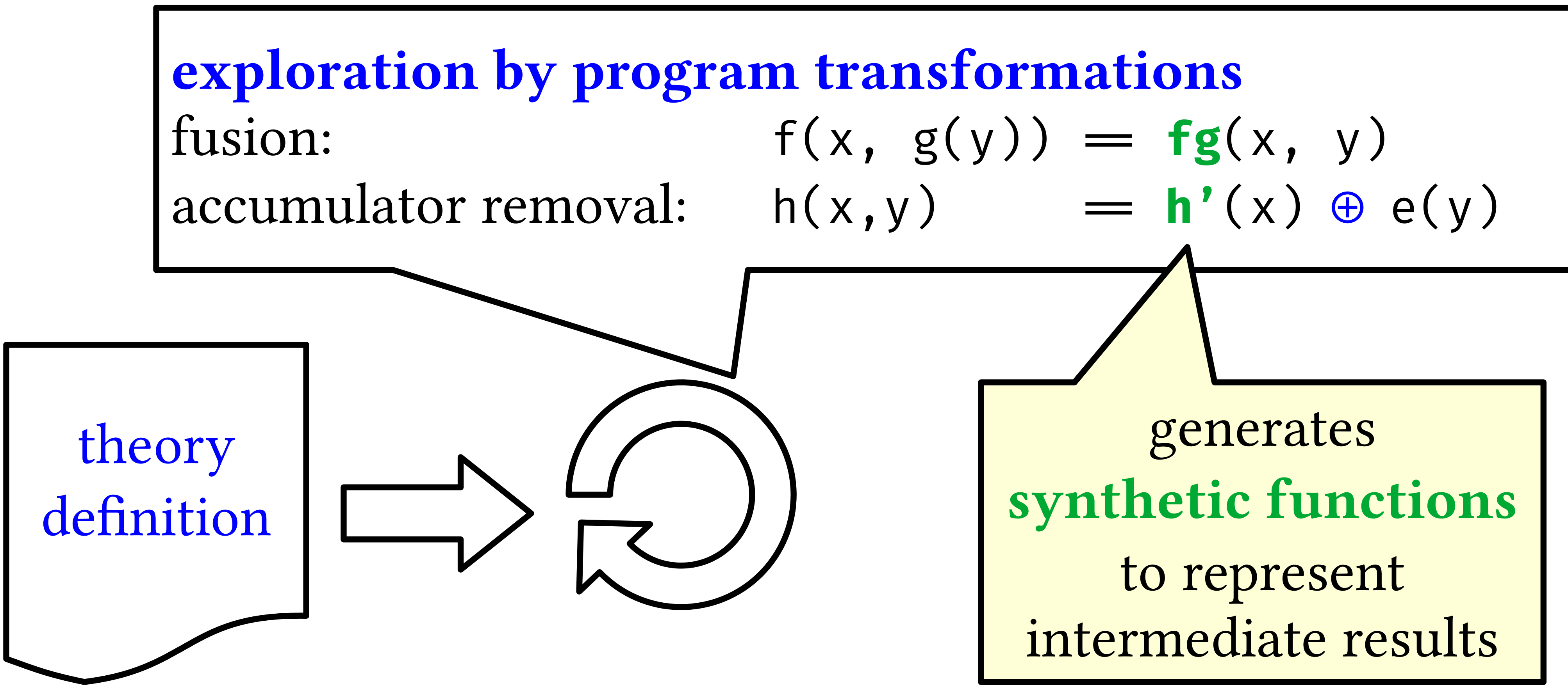
$$\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2)$$

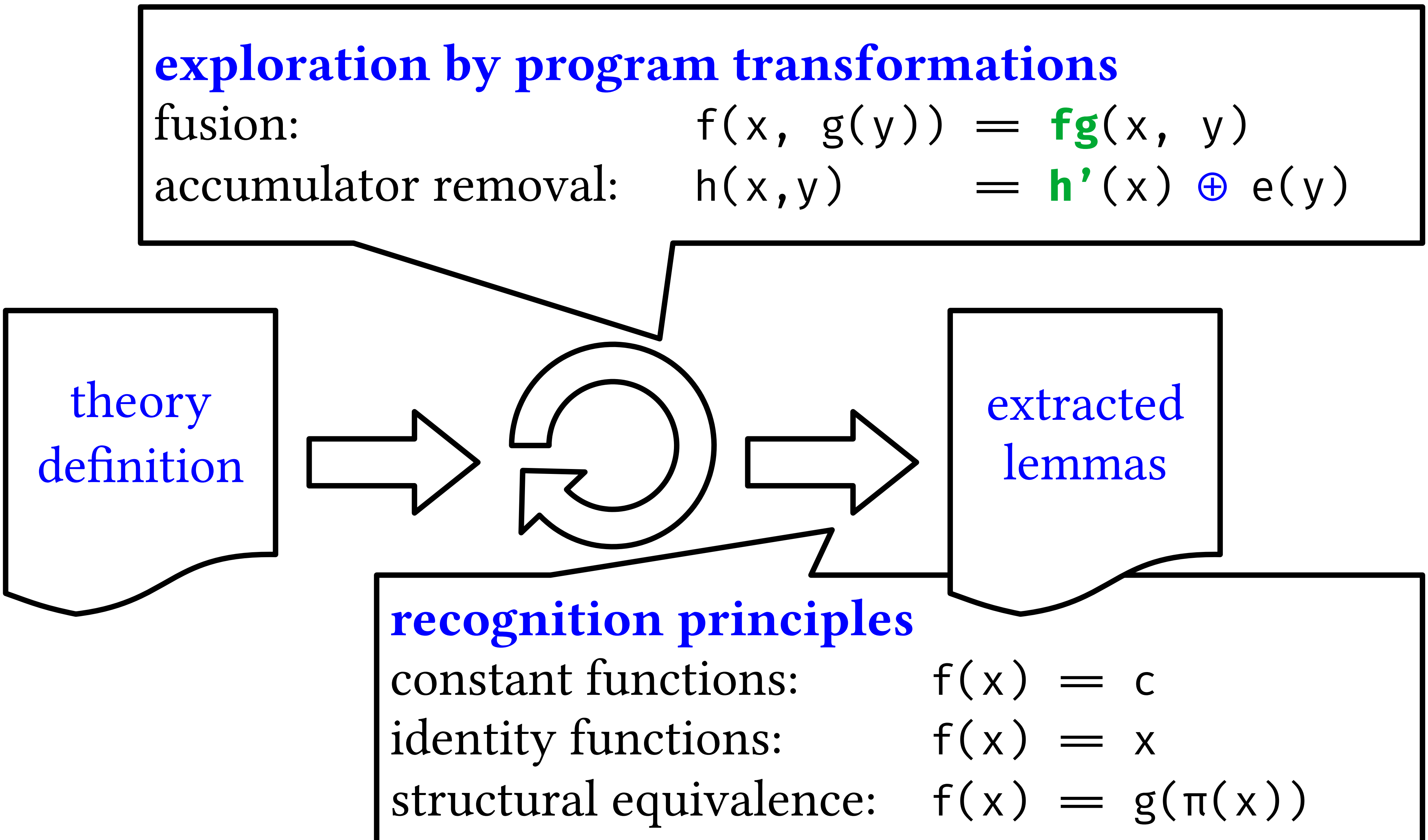
$$= \ ???$$

$$= \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev}(\mathtt{xs}_1)$$

another way to present
a computation with the same result


theory
definition


exploration by program transformations
fusion: f(x, g(y)) = fg(x, y)
accumulator removal: h(x,y) = h'(x) ⊕ e(y)
theory definition
generates synthetic functions to represent intermediate results


exploration by program transformations
fusion: f(x, g(y)) = fg(x, y)
accumulator removal: h(x,y) = h'(x) ⊕ e(y)
theory definition
extracted lemmas
recognition principles
constant functions: f(x) = c
identity functions: f(x) = x
structural equivalence: f(x) = g(π(x))

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, …]

$$
\begin{aligned}
&\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2)\\
&\qquad = \mathtt{rev{+\!\!+}}(\mathtt{xs}_1,\ \mathtt{xs}_2)\\
&\qquad = \mathtt{rev{+\!\!+}'}(\mathtt{xs}_1) \oplus^{?} \mathtt{e}^{?}(\mathtt{xs}_2)\\
&\qquad = \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev{+\!\!+}'}(\mathtt{xs}_1)\\
&\qquad = \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev}(\mathtt{xs}_1)
\end{aligned}
$$

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, …]

$$
\begin{aligned}
& \mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2) \\
& \quad = \mathtt{rev{+\!\!+}}(\mathtt{xs}_1, \mathtt{xs}_2) \\
& \quad = \mathtt{rev{+\!\!+}'}(\mathtt{xs}_1) \oplus^{?} \mathtt{e}^{?}(\mathtt{xs}_2) \\
& \quad = \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev{+\!\!+}'}(\mathtt{xs}_1) \\
& \quad = \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev}(\mathtt{xs}_1)
\end{aligned}
$$

unblocks

$$
\begin{aligned}
& \mathtt{rev}(\mathtt{rev}(\mathtt{xs})) \\
& \quad = \ldots = \mathtt{id}(\mathtt{xs}) = \mathtt{xs}
\end{aligned}
$$

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, ...]

$$\begin{aligned}&\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2)\\ &\quad = \mathbf{rev{+\!\!+}}(\mathtt{xs}_1, \mathtt{xs}_2)\end{aligned}$$

fusion

[Wadler, SPJ, Turchin, ...]

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, ...]

$$\begin{aligned}
&\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2) \\
&\quad = \mathtt{rev{+\!\!+}}(\mathtt{xs}_1, \mathtt{xs}_2) \\
&\quad = \mathbf{rev{+\!\!+}'}(\mathbf{xs}_1) \oplus^{?} e^{?}(\mathtt{xs}_2)
\end{aligned}$$

fusion

template for accumulator removal

$e^{?}(\_) := ???$

$z_1 \oplus^{?} z_2 := ???$

[Giesl]

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, ...]

fusion

$$
\begin{aligned}
&\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2)\\
&\quad = \mathtt{rev}{+\!\!+}(\mathtt{xs}_1, \mathtt{xs}_2)\\
&\quad = \mathtt{rev}{+\!\!+}'(\mathtt{xs}_1) \oplus^{?} e^{?}(\mathtt{xs}_2)\\
&\quad = \mathbf{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev}{+\!\!+}'(\mathtt{xs}_1)
\end{aligned}
$$

solution for accumulator removal

$$
\begin{aligned}
e^{?}(\_) &:= \mathbf{rev}(\_)\\
z_1 \oplus^{?} z_2 &:= \mathbf{z_2} \mathbin{+\!\!+} \mathbf{z_1}
\end{aligned}
$$

Calculating with Functions

[Bird, Burstall & Darlington, Meijer, ...]

```
rev(xs₁ ++ xs₂)
    = rev++(xs₁, xs₂)
    = rev++'(xs₁) ⊕? e?(xs₂)
    = rev(xs₂) ++ rev++'(xs₁)
    = rev(xs₂) ++ rev(xs₁)
```

recognize that

`rev++' ≡ rev`

Why it works

Fusion **regularizes** computations

- optimize away intermediate results (original goal)
- `fg` like `f` but extra parameters and more complex base case
- `fg` represents a “canned” (amortized) inductive proof

Fused function rev++(xs_1, xs_2)

base case: additional computation

```
rev++(xs₁, xs₂) = match xs₁
                  | case []    → rev(xs₂)
                  | case y::ys → rev++(ys, xs₂) ++ [y]
```

like rev

extra parameter

Why it works: A.R. picks up what is left by fusion

Fusion **regularizes** computations

- optimize away intermediate results (original goal)
- `fg` like `f` but extra parameters and more complex base case
- `fg` represents a “canned” (amortized) **inductive proof**

Accumulator removal **factors** computations

- push computation fragments out of functions
- template _ $\oplus^?$ $e^?$(_) represents **inductive generalizations**
- requires choice, but strictly more powerful than fusion! [Zhu]

Accumulator Removal for $\mathtt{rev{+\!\!+}}(\mathtt{xs}_1,\ \mathtt{xs}_2)$

$$\mathtt{rev{+\!\!+}}(\mathtt{xs}_1,\ \mathtt{xs}_2) = \mathtt{rev{+\!\!+}'}(\mathtt{xs}_1) \oplus^{?} e^{?}(\mathtt{xs}_2)$$

avoid accumulator in recursion

compensation operator

compensation term

Accumulator Removal for $\mathbf{rev{+\!\!+}(xs_1,\ xs_2)}$

$$\mathtt{rev{+\!\!+}(xs_1,\ xs_2)} = \mathbf{rev{+\!\!+}'}\mathtt{(xs_1)}\ \oplus^?\ \mathbf{e}^?\mathbf{(}\mathtt{xs_2}\mathbf{)}$$

$$
\begin{array}{l}
\mathbf{(}\mathtt{match\ xs_1} \\
\quad \mathtt{|\ case\ []} \quad\ \ \rightarrow \mathtt{rev(xs_2)} \\
\quad \mathtt{|\ case\ y::ys} \rightarrow \mathtt{rev{+\!\!+}(ys,\ xs_2)\ {+\!\!+}\ [y]}\mathbf{)} \\
\qquad = \\
\mathbf{(}\mathtt{match\ xs_1} \\
\qquad \mathtt{|\ case\ []} \quad\ \ \rightarrow \mathbf{base}^? \\
\qquad \mathtt{|\ case\ y::ys} \rightarrow \mathbf{rec}^?\mathtt{(rev{+\!\!+}'(ys),\ y)}\mathbf{)}\ \oplus^?\ \mathbf{e}^?\mathbf{(}\mathtt{xs_2}\mathbf{)}
\end{array}
$$

Accumulator Removal for **rev++(xs₁, xs₂)**

```
rev++(xs₁, xs₂) = rev++'(xs₁) ⊕? e?(xs₂)

(match xs₁
 | case []     → rev(xs₂)
 | case y::ys  → rev++(ys, xs₂) ++ [y])
   =
(match xs₁
  | case []     → base?
  | case y::ys  → rec?(rev++'(ys), y)) ⊕? e?(xs₂)
```

additional structure to instantiate _ ⊕? e?(_)

Accumulator Removal for $\texttt{rev++}(\texttt{xs}_1,\ \texttt{xs}_2)$

$\texttt{rev++}(\texttt{xs}_1,\ \texttt{xs}_2) = \texttt{rev++'}(\texttt{xs}_1) \oplus^{?} e^{?}(\texttt{xs}_2)$

$\texttt{(match xs}_1$

$\quad \texttt{| case []} \quad \to \texttt{rev}(\texttt{xs}_2)$

$\quad \texttt{| case y::ys} \to \texttt{rev++(ys,}$

$\qquad =$

$\texttt{rev}(\texttt{xs}_2)\ \texttt{++}\ \texttt{(match xs}_1$

$\qquad \texttt{| case []} \quad \to \texttt{[]}$

$\qquad \texttt{| case y::ys} \to \texttt{rev++'(ys) ++ p[y])}$

$e^{?}(\_) := \texttt{rev}(\_)$

$z_1 \oplus^{?} z_2 := z_2\ \texttt{++}\ z_1$

$\texttt{base} := \texttt{[]}$

$\texttt{rec} :=$ **(original body)**

Heuristics for Accumulator Removal

$$\texttt{f}(\texttt{x}, \texttt{z}) = \mathbf{f'}(\texttt{x}) \oplus^{?} \mathbf{e}^{?}(\texttt{z})$$

avoid accumulator in recursion

compensation operator

compensation term

Heuristics for Accumulator Removal

$$\texttt{f(x, z)} = \mathbf{f'}\texttt{(x)} \oplus^{?} \mathbf{e}^{?}(\texttt{z})$$

avoid accumulator in recursion

compensation operator

compensation term

- Choose new base case of **`f'`** and compensation operator from functions with neutral elements (e.g. **`++`** with **`[]`** left/right)

Heuristics for Accumulator Removal

$$\texttt{f(x, z)} = \texttt{f'(x)} \oplus^{?} \texttt{e}^{?}\texttt{(z)}$$

avoid accumulator in recursion

compensation operator

compensation term

- Choose new base case of **f'** and compensation operator from functions with neutral elements (e.g. **++** with **[]** left/right)

```
               (..(rev(xs₂) ++ [y₁]) .. ++ [yₙ₋₁]) ++ [yₙ]
= rev(xs₂) ++ (..([]        ++ [y₁]) .. ++ [yₙ₋₁]) ++ [yₙ]
```

Heuristics for Accumulator Removal

`f(x, z) = f'(x) ⊕? e?(z)`

avoid accumulator in recursion

compensation operator

compensation term

- Choose new base case of `f'` and compensation operator from functions with neutral elements (e.g. `++` with `[]` left/right)
- Choose compensation term as original base case

(side conditions apply)

- Choose new recursive case(s) of `f'` unchanged

(strong limitation: misses some results)

Heuristics for Accumulator Removal

```
f(x, z) = f'(x) ⊕? e?(z)
```

avoid accumulator in recursion

compensation operator

compensation term

- Choose new base case of `f'` and compensation operator from functions with neutral elements (e.g. `++` with `[]` left/right)
- Choose compensation term as original base case

(side conditions apply)

- Choose new recursive case(s) of `f'` unchanged

(strong limitation: misses some results)

Evaluation

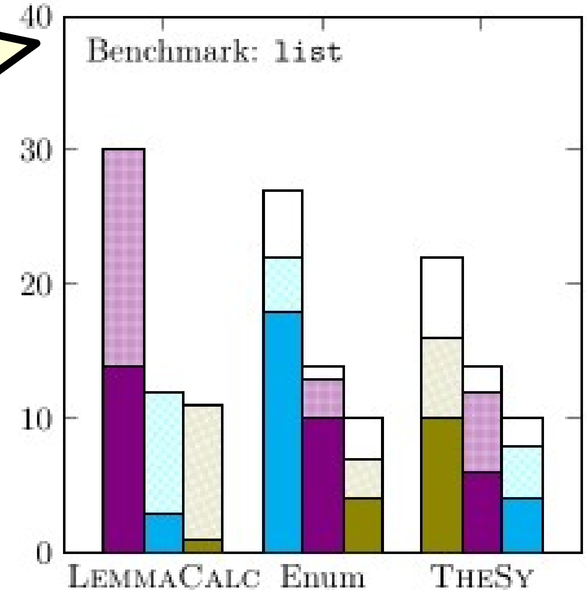
RQ1: What is the **relative explanatory strength** of the sets of lemmas generated by the different methods?

RQ2: What is the impact of the search space on lemma **synthesis time**?

Results: Number of Lemmas

LemmaCalc
Baseline Enumerator
TheSy [Singher, Itzhaky]

“large” theory:
18 functions

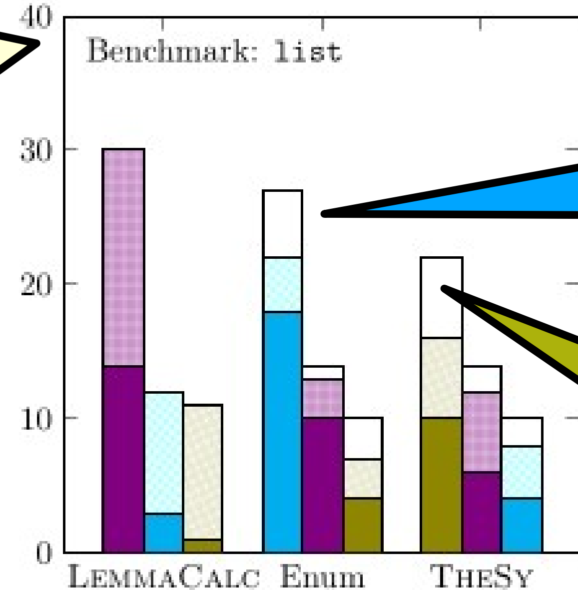


~10s >24h each

Results: Number of Lemmas

LemmaCalc
Baseline Enumerator
TheSy [Singher, Itzhaky]

"large" theory:
18 functions



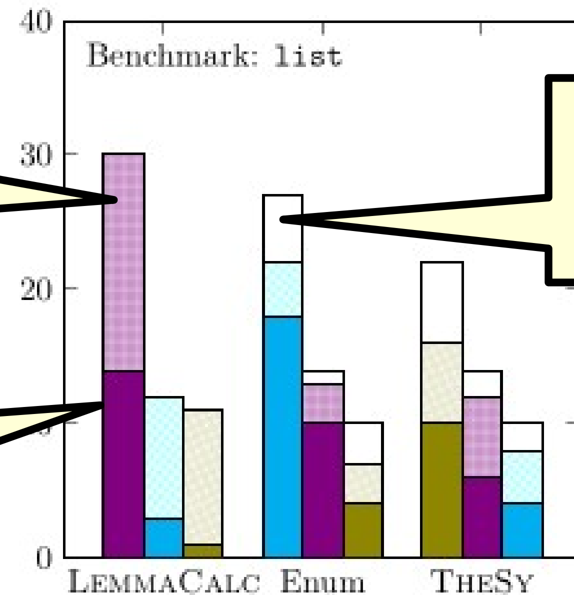
> 300K candidates
`f(x, g(y)) == ?`

with conditional lemmas

~10s
>24h each

Results: Number of Lemmas

LemmaCalc
Baseline Enumerator
TheSy [Singher, Itzhaky]



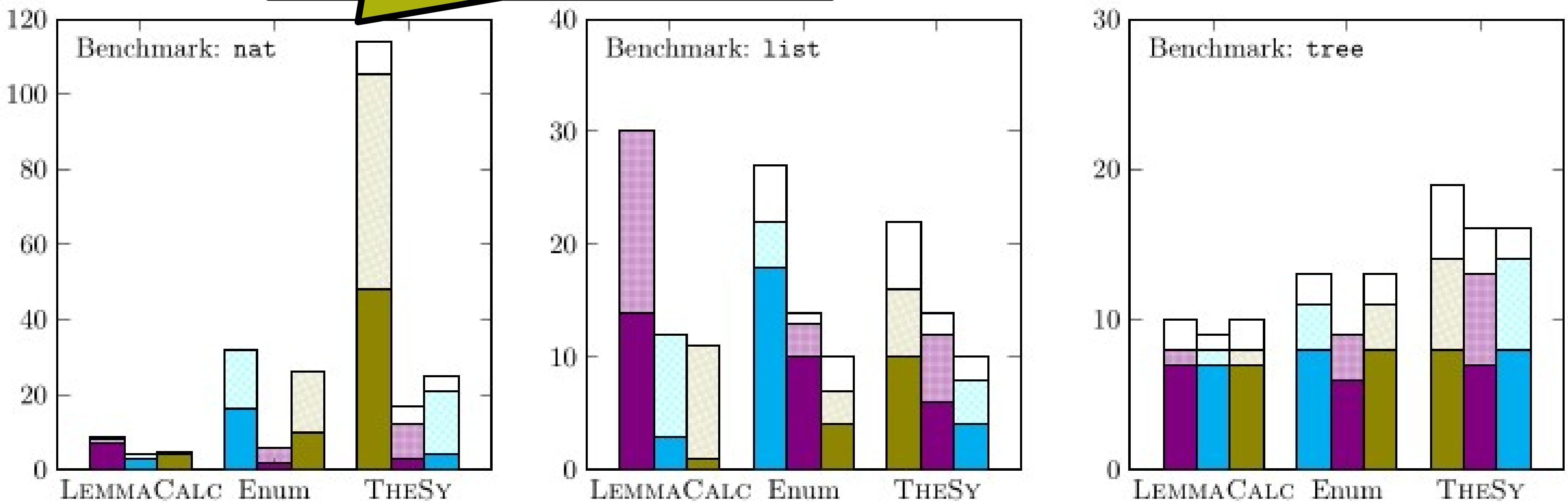
redundant

interesting

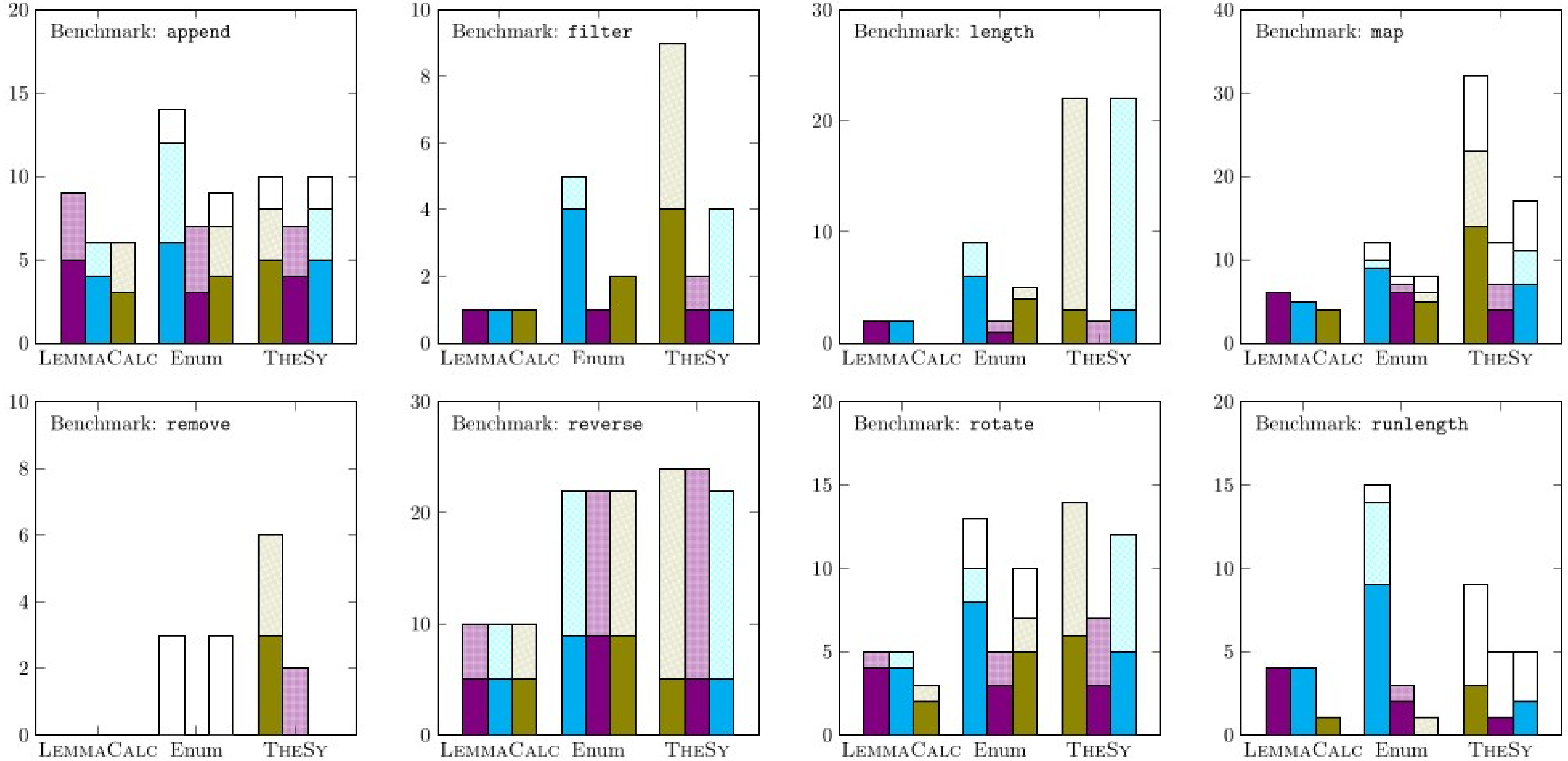
trivial
(no induction)

% covered by
competitor

many
conditional lemmas



(almost) identical
non-redundant sets



Remark: it is expected that baseline Enum always dominates LemmaCalc (modulo prover)

Results and Discussion

- RQ1: methods have comparable explanatory power but complementary strengths and weaknesses
- RQ2: LemmaCalc is much faster but misses some results

- Key advantage of LemmaCalc over (naive) enumeration
  - **structured** search space
  - potential to **amortize** parts of proofs
  - **short-cuts** via intermittend deduction
- Main current limitation: accumulator-transformations are uninformed and independent of recognition algorithms

Related Work

- Buchberger, Claessen, Hughes, Johansson
  pioneered theory exploration (tools & larger case studies)
- Sonnex, deAngelis
  program transformations make effective proof methods
- Hamilton & Poítin, Klyuchnikov & Romanenko
  suggest fusion for lemma inference (= low hanging fruits)
- Giesl: deaccumulation schemes (not implemented)
- lots of work in the PL community (see paper)

Contribution: LemmaCalc

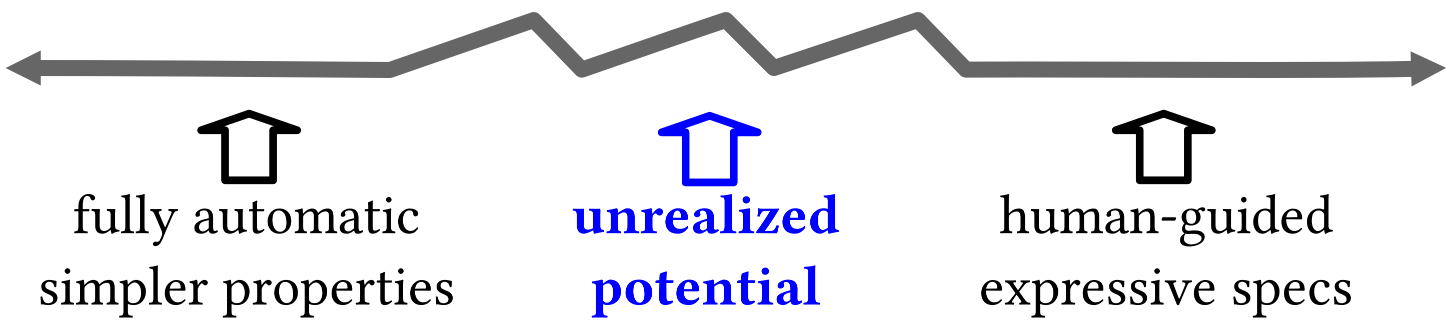
- **quickly** explores a productive search space
- lemmas tend to be **intuitive** and **useful**
- integrate **transformations** and **deduction**

10.5281/zenodo.16932462

Contribution: LemmaCalc

10.5281/zenodo.16932462

- **quickly** explores a productive search space
- lemmas tend to be **intuitive** and **useful**
- integrate **transformations** and **deduction**
- this implementation: proof of concept;
  current limitations hit "hard ceiling"
- outlook: exploit further short-cuts accumulator transformations
  online integration with enumeration



Appendix

Fusing rev(xs₁ ++ xs₂)

```
rev++(xs₁, xs₂) := rev(xs₁ ++ xs₂)
        = rev(match xs₁
                | case []     → xs₂
                | case y::ys → y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(ys ++ xs₂) ++ [y]
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev++(ys, xs₂) ++ [y]
```

Define and Unfold [Bird, Burstall & Darlington]

```
rev++(xs₁, xs₂) := rev(xs₁ ++ xs₂)
        = rev(match xs₁
                | case []     → xs₂
                | case y::ys  → y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev(y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev(ys ++ xs₂) ++ [y]
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev++(ys, xs₂) ++ [y]
```

Shift Function Application [Turchin, ...]

```
rev++(xs₁, xs₂) := rev(xs₁ ++ xs₂)
        = rev(match xs₁
                | case []     → xs₂
                | case y::ys  → y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev(y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev(ys ++ xs₂) ++ [y]
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys  → rev++(ys, xs₂) ++ [y]
```

Apply **Definitions** and known Lemmas

[Gill, Launchbury, SPJ]

```
rev++(xs₁, xs₂) := rev(xs₁ ++ xs₂)
        = rev(match xs₁
                | case []     → xs₂
                | case y::ys → y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(ys ++ xs₂) ++ [y]
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev++(ys, xs₂) ++ [y]
```

Simplify and Fold [Bird, Burstall & Darlington]

```
rev++(xs₁, xs₂) := rev(xs₁ ++ xs₂)
        = rev(match xs₁
                | case []     → xs₂
                | case y::ys → y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(y :: (ys ++ xs₂))
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev(ys ++ xs₂) ++ [y]
        = match xs₁
            | case []     → rev(xs₂)
            | case y::ys → rev++(ys, xs₂) ++ [y]
```

Fusing **rev(rev(xs))**

```
rev2(xs) := rev(rev(xs))
         = rev(match xs
                | case [] → []
                | case y :: ys → rev(ys) ++ [y])
         = match xs
            | case [] → rev([])
            | case y :: ys → rev(rev(ys) ++ [y])
         = match xs
            | case [] → rev([])
            | case y :: ys → rev([y]) ++ rev(rev(ys))
         = match xs
            | case [] → []
            | case y :: ys → y :: rev2(ys)
         = id(xs) = xs
```

Final Lemma

```
rev2(xs) := rev(rev(xs))
      = rev(match xs
            | case [] → []
            | case y::ys → rev(ys) ++ [y])
      = match xs
         | case [] → rev([])
         | case y::ys → rev(rev(ys) ++ [y])
      = match xs
         | case [] → rev([])
         | case y::ys → rev([y]) ++ rev(rev(ys))
      = match xs
         | case [] → []
         | case y::ys → y :: rev2(ys)
      = id(xs) = xs
```

Define and Unfold [Bird, Burstall & Darlington]

```
rev2(xs) := rev(rev(xs))
        = rev(match xs
                | case [] → []
                | case y::ys → rev(ys) ++ [y])
        = match xs
            | case [] → rev([])
            | case y::ys → rev(rev(ys) ++ [y])
        = match xs
            | case [] → rev([])
            | case y::ys → rev([y]) ++ rev(rev(ys))
        = match xs
            | case [] → []
            | case y::ys → y :: rev2(ys)
        = id(xs) = xs
```

Shift Function Application [Turchin, ...]

```
rev2(xs) := rev(rev(xs))
        = rev(match xs
                | case [] → []
                | case y::ys → rev(ys) ++ [y])
        = match xs
            | case [] → rev([])
            | case y::ys → rev(rev(ys) ++ [y])
        = match xs
            | case [] → rev([])
            | case y::ys → rev([y]) ++ rev(rev(ys))
        = match xs
            | case [] → []
            | case y::ys → y :: rev2(ys)
        = id(xs) = xs
```

Intermittent Deduction using Prior Lemmas

[Gill, Launchbury, SPJ]

```
rev2(xs) := rev(rev(xs))
      = rev(match xs
              | case [] → []
              | case y::ys → rev(ys) ++ [y])
      = match xs
          | case [] → rev([])
          | case y::ys → rev(rev(ys) ++ [y])
      = match xs
          | case [] → rev([])
          | case y::ys → rev([y]) ++ rev(rev(ys))
      = match xs
          | case [] → []
          | case y::ys → y :: rev2(ys)
      = id(xs) = xs
```

Intermittent Deduction using **Prior Lemmas**

[Gill, Launchbury, SPJ]

```
rev2(xs) := rev(rev(xs))
         = rev(match xs
                 | case [] → []
                 | case y::ys → rev(ys) ++ [y])
         = match xs
             | case [] → rev([])
             | case y::ys → rev(rev(ys) ++ [y])
         = match xs
             | case [] → rev([])
             | case y::ys → rev([y]) ++ rev(rev(ys))
```

most difficult step, requires to know already

$\mathtt{rev}(\mathtt{xs}_1 \mathbin{+\!\!+} \mathtt{xs}_2) = \mathtt{rev}(\mathtt{xs}_2) \mathbin{+\!\!+} \mathtt{rev}(\mathtt{xs}_1)$

Simplify and Fold [Bird, Burstall & Darlington]

```
rev2(xs) := rev(rev(xs))
       = rev(match xs
              | case [] → []
              | case y::ys → rev(ys) ++ [y])
       = match xs
          | case [] → rev([])
          | case y::ys → rev(rev(ys) ++ [y])
       = match xs
          | case [] → rev([])
          | case y::ys → rev([y]) ++ rev(rev(ys))
       = match xs
          | case [] → []
          | case y::ys → y :: rev2(ys)
       = id(xs) = xs
```

Recognize identity, constant, existing Functions

```
rev2(xs) := rev(rev(xs))
         = rev(match xs
                | case [] → []
                | case y::ys → rev(ys) ++ [y])
         = match xs
            | case [] → rev([])
            | case y::ys → rev(rev(ys) ++ [y])
         = match xs
            | case [] → rev([])
            | case y::ys → rev([y]) ++ rev(rev(ys))
         = match xs
            | case [] → []
            | case y::ys → y :: rev2(ys)
         = id(xs) = xs
```