Deductive Verification: Reasoning Across Abstraction Boundaries

May 8, 2024

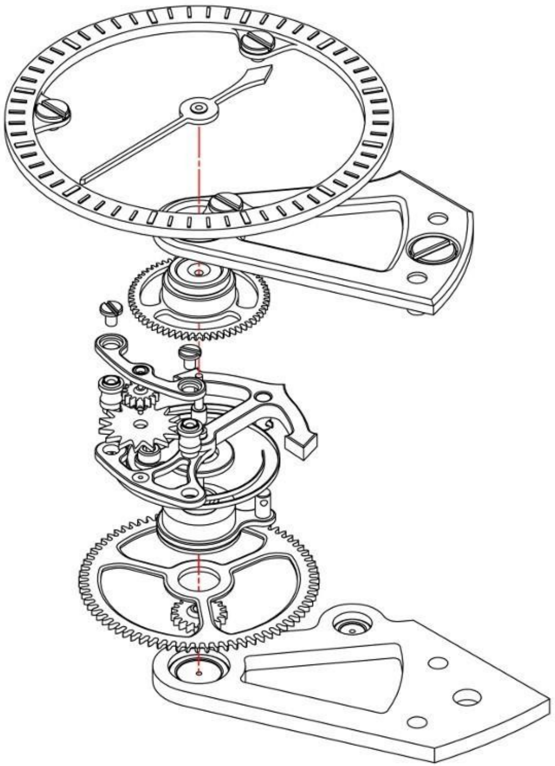
ConVeY Seminar

Gidon Ernst, LMU Munich
sosy-lab.org/people/ernst

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN
SoSy-Lab Software Systems

Abstraction is Essential for Trust in Technical Systems

technical system



laws of physics

specification

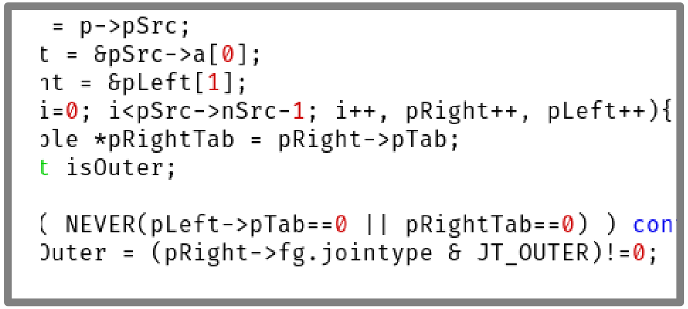
$$\frac{d\,hour}{d\,minute} = \frac{1}{60}$$

$$d\,minute = \frac{2\,\pi}{60\,s}$$

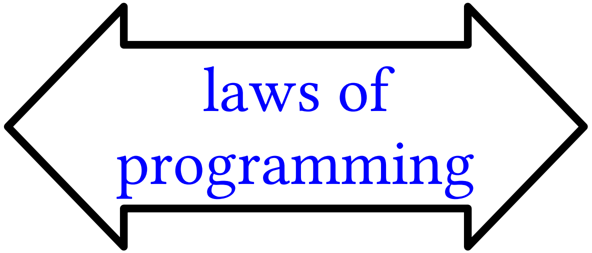
abstract yet precise!

Software Verification

software system



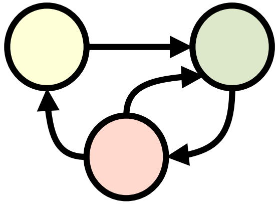
```
 = p->pSrc;
t = &pSrc->a[0];
nt = &pLeft[1];
i=0; i<pSrc->nSrc-1; i++, pRight++, pLeft++){
ble *pRightTab = pRight->pTab;
t isOuter;

( NEVER(pLeft->pTab==0 || pRightTab==0) ) con
Outer = (pRight->fg.jointype & JT_OUTER)!=0;
```



laws of programming

specification

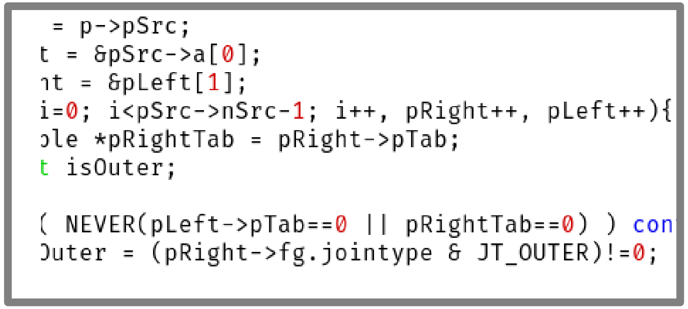
read(write(x)) = x



$O(n)$

Software Verification

software system

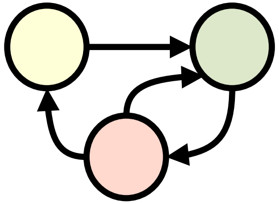


```
 = p->pSrc;
t = &pSrc->a[0];
nt = &pLeft[1];
i=0; i<pSrc->nSrc-1; i++, pRight++, pLeft++){
ble *pRightTab = pRight->pTab;
t isOuter;

( NEVER(pLeft->pTab==0 || pRightTab==0) ) con
Outer = (pRight->fg.jointype & JT_OUTER)!=0;
```

laws of programming

specification

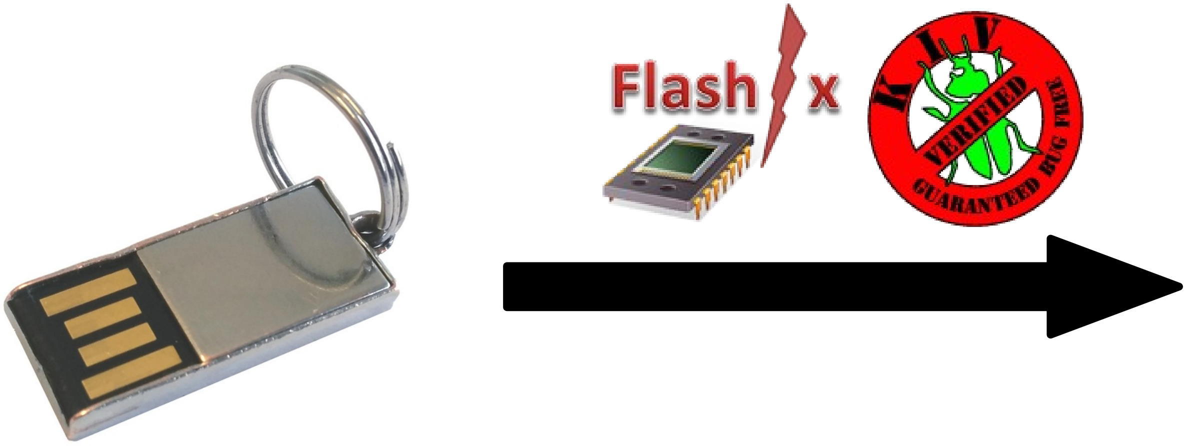
`read(write(x)) = x`



$O(n)$

deductive verification: establish correspondence by automated mathematical proof

Flashix: a fully verified Flash file system (PhD project)

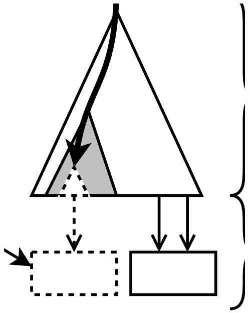


8 items, Free space: 78.7 GiB

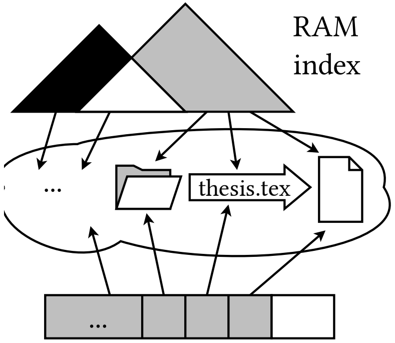
Related: Argosys, FSCQ, Yggdrasil, BilbyFS, ...

Flashix: a fully verified Flash file system (PhD project)

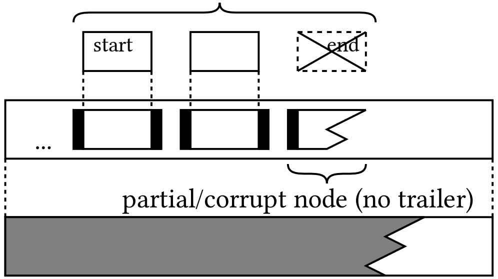
[ABZ 14, iFM SCP 16]



abstraction gap



abstraction gap



[SSV 12, VSTTE 13]

[VSTTE 15]

[HVC 13]

Flashix: a fully verified Flash file system (PhD project)

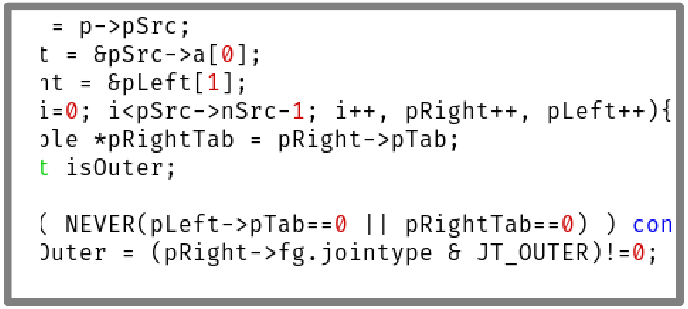
Flash x
KIV
VERIFIED
GUARANTEED BUG FREE

Challenge = bridging different abstraction levels: modeling, data structures, algorithms, refinement proofs

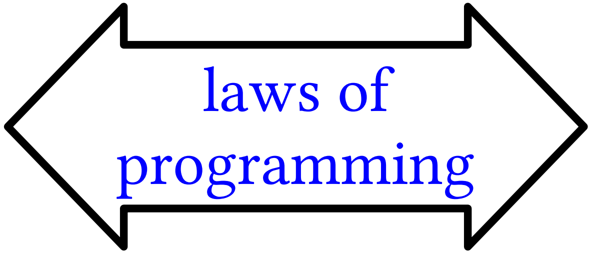
Technical Difficulties from unbounded state space and recursion, quantifiers, second-order

My Research Goal: Automation ⊕ Expressiveness

software system



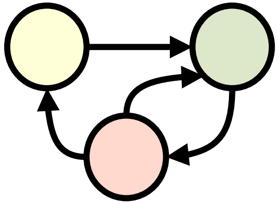
```
 = p->pSrc;
t = &pSrc->a[0];
ɿt = &pLeft[1];
i=0; i<pSrc->nSrc-1; i++, pRight++, pLeft++){
ɔle *pRightTab = pRight->pTab;
t isOuter;

( NEVER(pLeft->pTab==0 || pRightTab==0) ) con
Ↄuter = (pRight->fg.jointype & JT_OUTER)!=0;
```



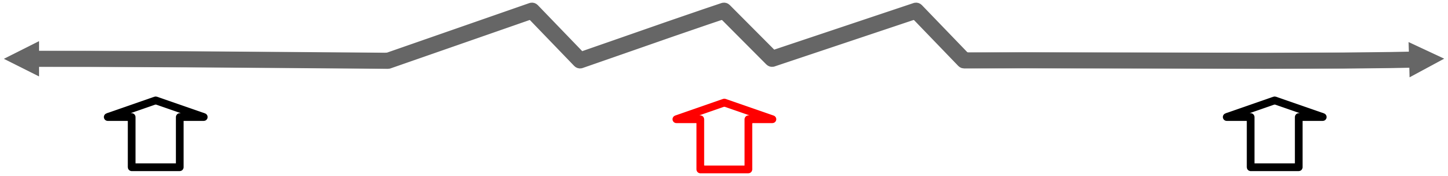
laws of programming

specification

`read(write(x)) = x`



$O(n)$



fully automatic
simpler properties
(SV-COMP)

open challenge!

human-guided
expressive specs
(VerifyThis)

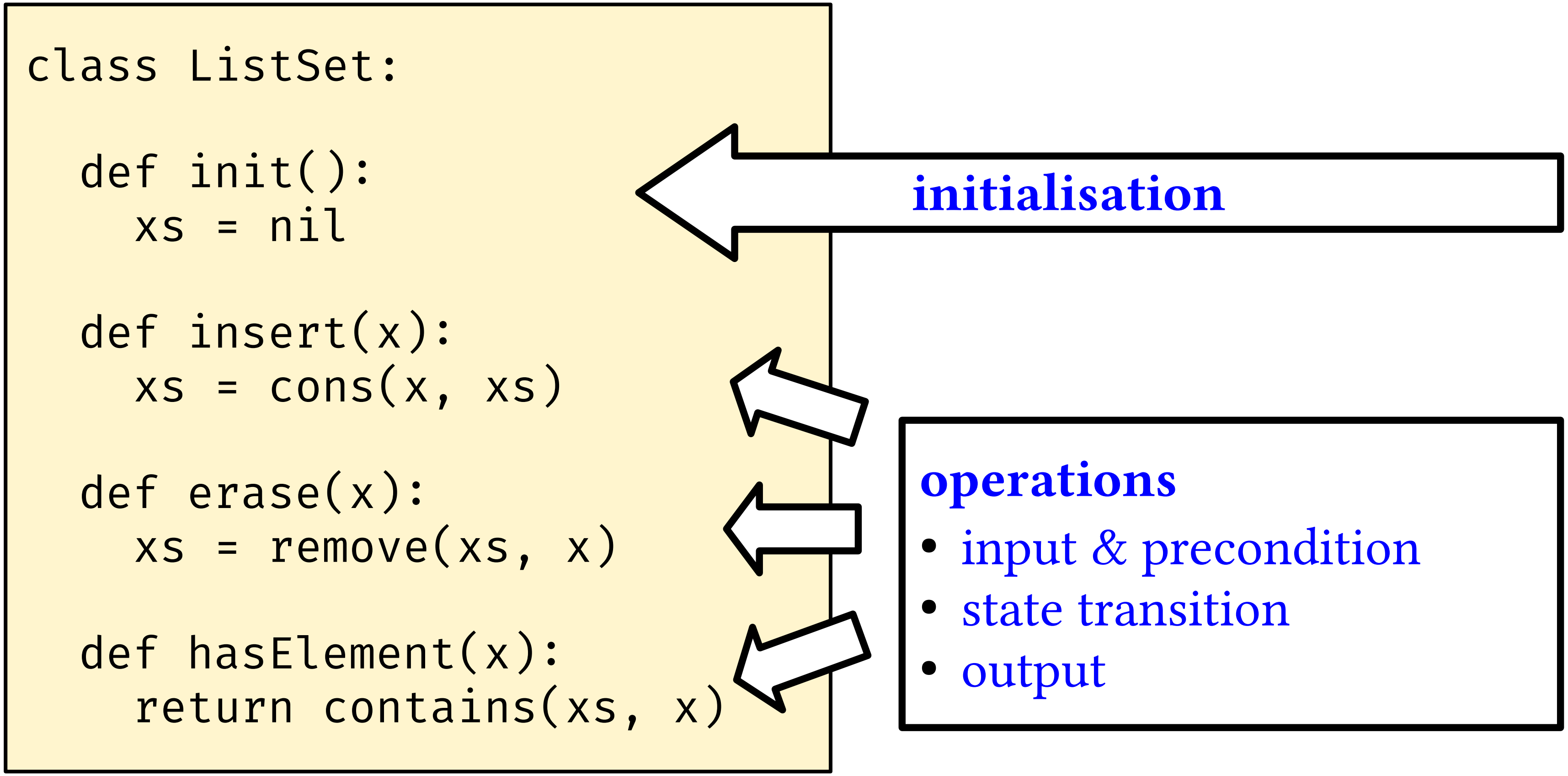
Example: Behavioral Component Models

Model (functional lists)

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    xs = cons(x, xs)

  def erase(x):
    xs = remove(xs, x)

  def hasElement(x):
    return contains(xs, x)
```

Example: Behavioral Component Models (think: B and Event-B)

Model (functional lists)



```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    xs = cons(x, xs)

  def erase(x):
    xs = remove(xs, x)

  def hasElement(x):
    return contains(xs, x)
```

initialisation

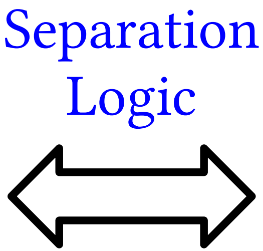
operations

- input & precondition
- state transition
- output

Teaser: Model ≃ Implementation

Model (functional lists)

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    xs = cons(x, xs)

  def erase(x):
    xs = remove(xs, x)

  def hasElement(x):
    return contains(xs, x)
```



Source Code (pointers)

```
  struct list {
    int             head;
    struct list *next;
  }
```

automatic connection via shape analysis
e.g. [Calcagno et al 09]

Example: Intuitive specification (= trust)

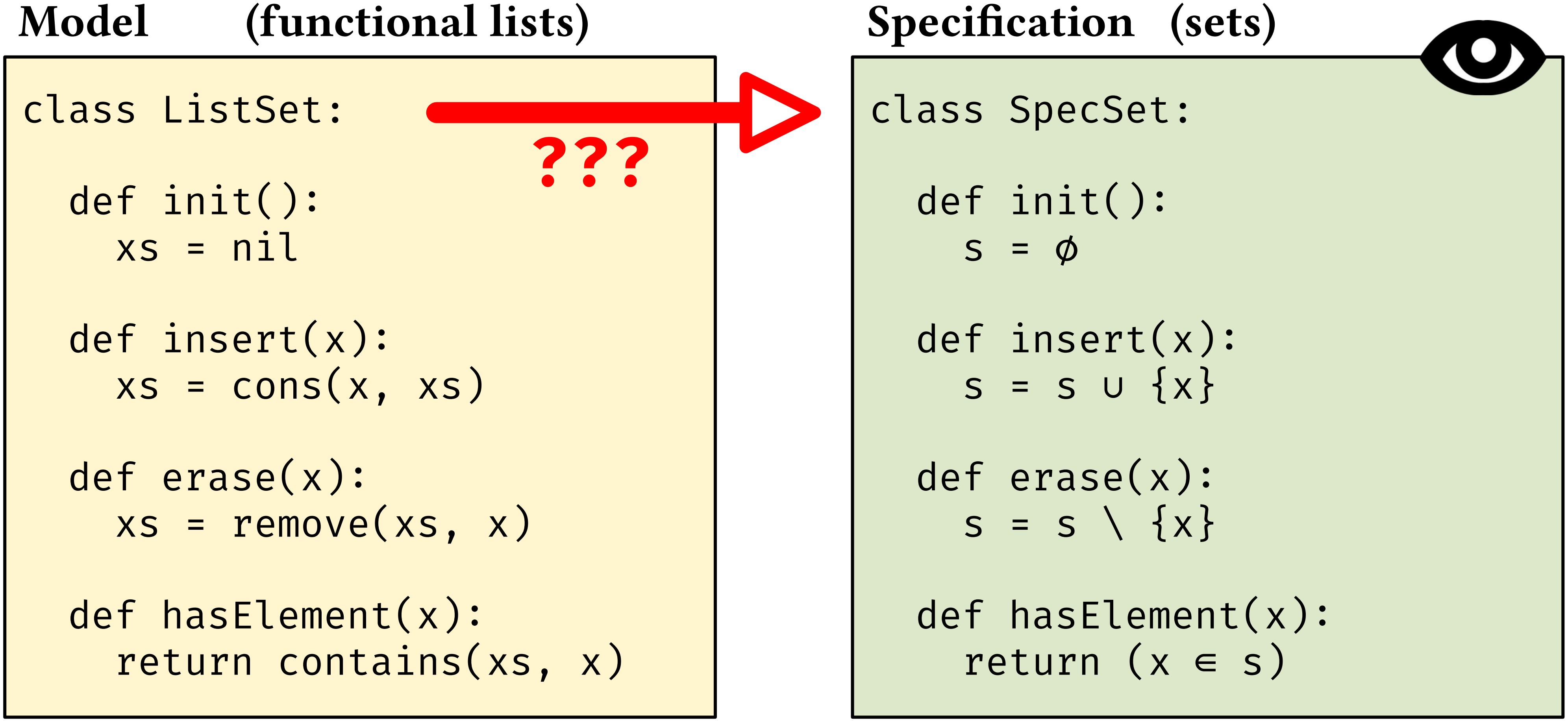
Model (functional lists)

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    xs = cons(x, xs)

  def erase(x):
    xs = remove(xs, x)

  def hasElement(x):
    return contains(xs, x)
```

Specification (sets)

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}

  def erase(x):
    s = s \ {x}

  def hasElement(x):
    return (x ∈ s)
```

Goal: Design ≃ Specification



Model (functional lists)

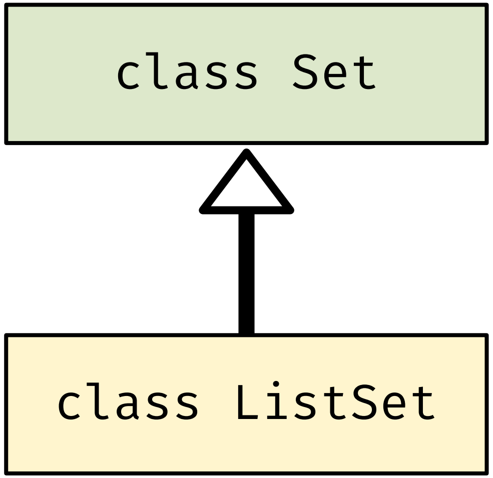
```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    xs = cons(x, xs)

  def erase(x):
    xs = remove(xs, x)

  def hasElement(x):
    return contains(xs, x)
```

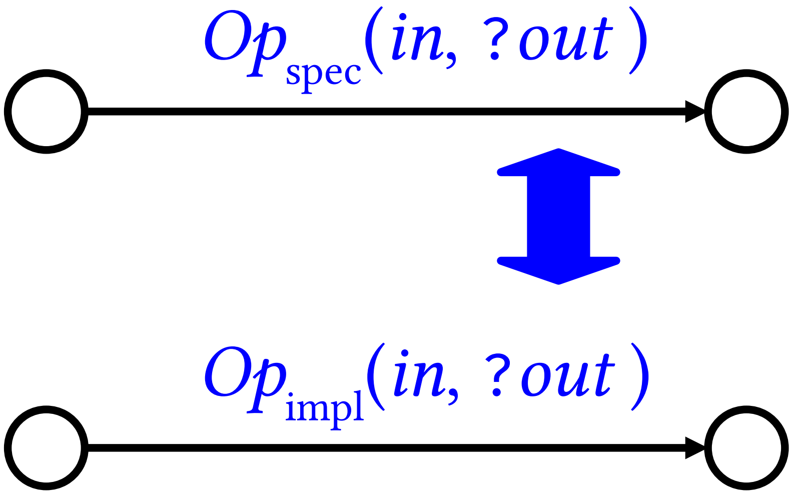
Specification (sets)

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}

  def erase(x):
    s = s \ {x}

  def hasElement(x):
    return (x ∈ s)
```

Proof: Design ≃ Specification

e.g. [He, Hoare, Sanders 86] [Liskov & Wing 94]

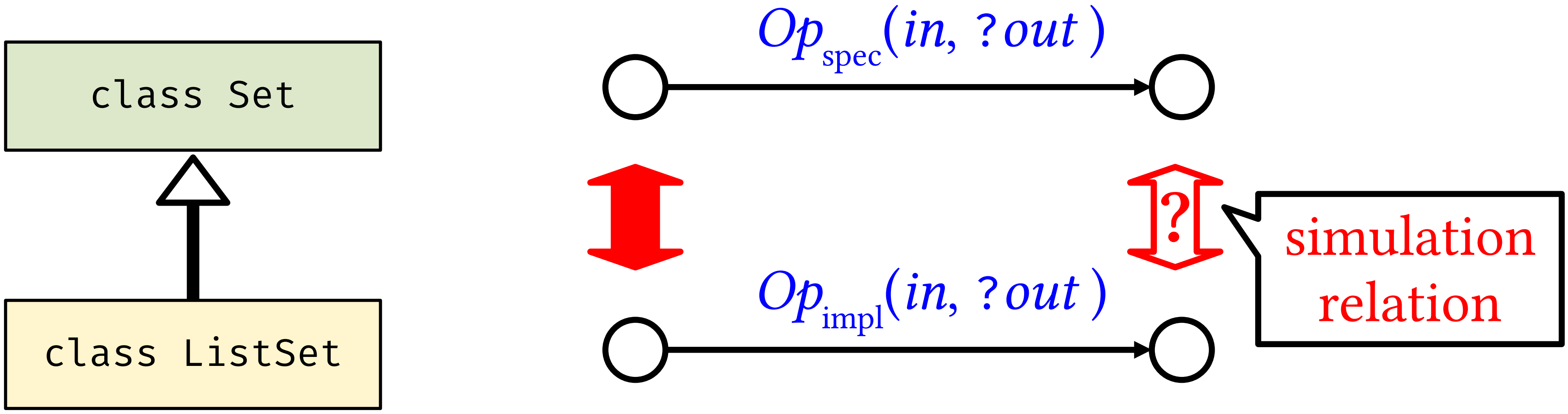




desired guarantee: same input/output of operations

Proof: Design ≃ Specification via Simulation Proofs

e.g. [He, Hoare, Sanders 86] [Liskov & Wing 94]

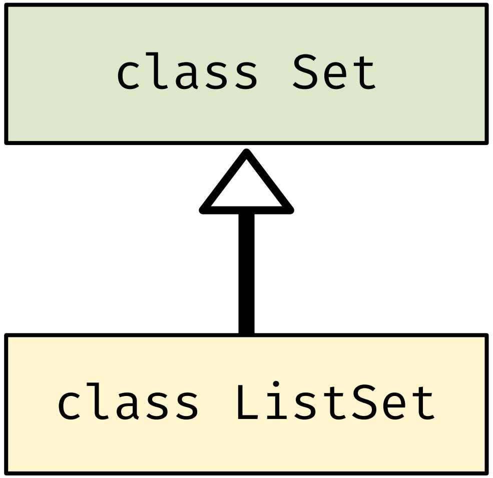


desired guarantee: same input/output of operations

inductive proof: correspondence of states

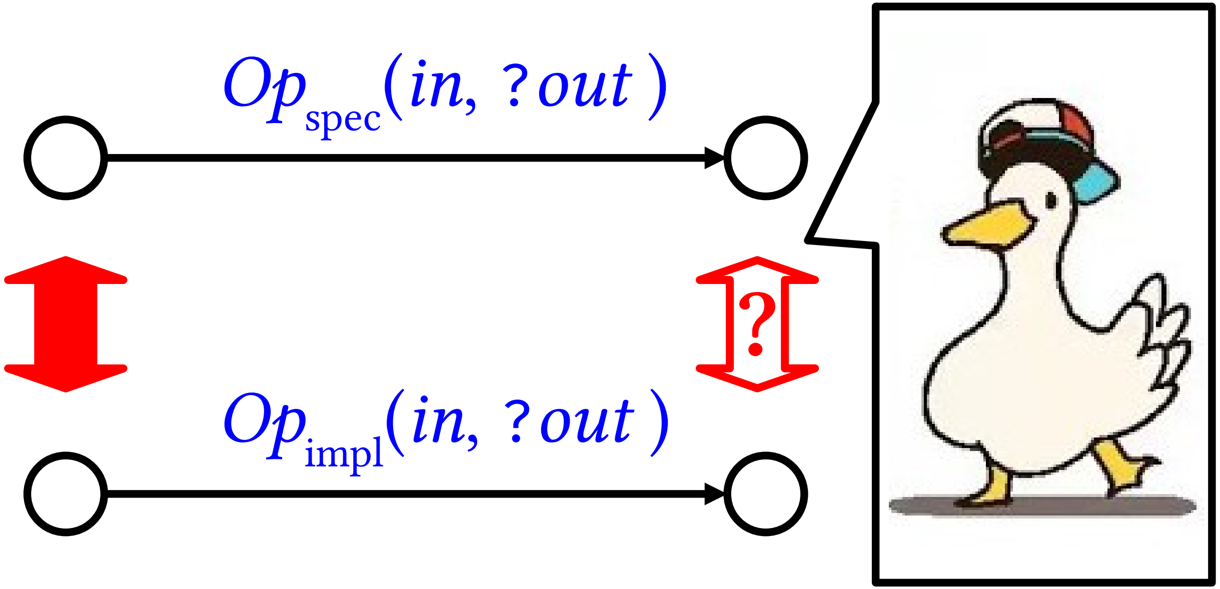
Proof: Design ≃ Specification via Simulation Proofs

e.g. [He, Hoare, Sanders 86] [Liskov & Wing 94]



```
class Set
```

```
class ListSet
```



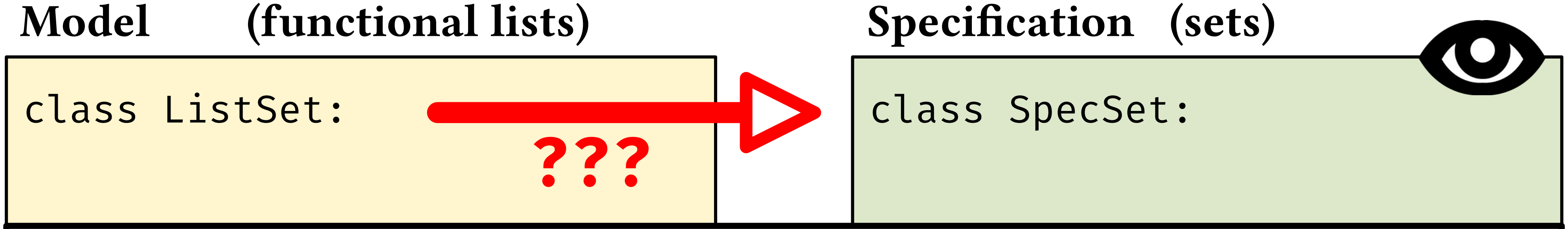
$Op_{\text{spec}}(in, ?out)$

$Op_{\text{impl}}(in, ?out)$

desired guarantee: same input/output of operations

inductive proof: correspondence of states

Goal: Design ≃ Specification (Example again)



Model **(functional lists)**

```
class ListSet:
```

???

Specification **(sets)** 👁

```
class SpecSet:
```

Interactive Demo

Deductive Verification in Dafny

Comparing one-step of input/output behavior

```
class ListSet:

  …

  def hasElement(x):
    return contains(xs, x)
```

```
class SpecSet:

  …

  def hasElement(x):
    return (x ∈ s)
```

Solution

```
R(xs, s)  :=  ∀ x. (contains(xs, x) ⟺ x ∈ s)
```

Construction: collecting observerations

```
class ListSet:

  ...

  def hasElement(x):
    return contains(xs, x)
```

```
class SpecSet:

  ...

  def hasElement(x):
    return (x ∈ s)
```

Construction: quantify over possible observations via operations

$$\mathtt{R(xs,\ s)} \quad := \boxed{\forall\ \mathtt{x}.}\ (\mathtt{contains(xs,\ x)} \iff \mathtt{x} \in \mathtt{s})$$

Construction: collecting observerations

```
class ListSet:

  …

  def hasElement(x):
    return contains(xs, x)
```

```
class SpecSet:

  …

  def hasElement(x):
    return (x ∈ s)
```

Construction: require equivalence of outputs

$$R(xs,\ s) := \boxed{\forall\ x.}\ (\underline{contains(xs,\ x)} \iff \underline{x \in s})$$ ✓

Remark: corresponds to first frame in Property-Directed Reachability (PDR)

Challenges: Quantifier, Sets (i.e. SMT Arrays), Lists (ADT)

Discussion: Observer Equivalence as Starting Point

```
R(xs, s)  :=  ∀ x. (contains(xs, x) ⟺ x ∈ s)
```

✓ no creativity needed + automation via SMT

(✓) usually require additional system invariants (abduction, AI)

? usually require standard & application-specific lemmas, e.g.

```
contains(remove(xs, x), x) ⟺ false
```

(?) refinement to source code: loops, pointers, etc, …

(?) where does the reference/specification come from in the first place?

LemmaCalc: Quick Theory Exploration for Algebraic Data Types via Program Transformations [ongoing]

Gidon Ernst, Robin Sögtrop, LMU Munich
Grigory Fedyukovich, FSU

! Goal: find lemmas automatically that support automatic proofs

$$\texttt{contains(remove(xs, x), x)} \iff \texttt{false}$$

$$\underbrace{\texttt{length(}}_{\text{spec}}\underbrace{\texttt{elems(tree)}}_{\text{abstraction}}\texttt{)} = \underbrace{\texttt{size(tree)}}_{\text{impl}}$$

Related Work

Lemmas from Stuck Proofs

Rippling [Bundy], Term Induction
ACL2, AdtInd, …

- ✓ effective
- ✓ like humans conduct proofs
- ? carries current proof context
- ? requires inductive generalizations

Theory Exploration

HipSpec [Buchberger, Johansson]
TheSy [Singher & Itzhaky], …

syntax-guided enumeration

+ inductive proof to check

- ✓ effective & “complete” wrt. oracle
- ? vast unstructured search space
- ? reports ”free-form” tautologies

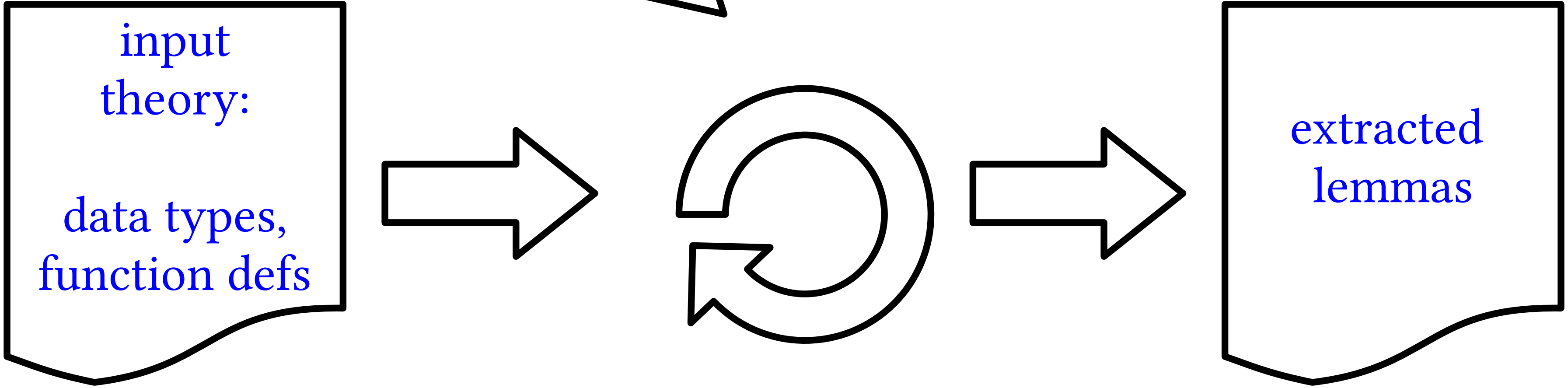
LemmaCalc

Key Idea: Combine structural function transformations

fusion: $\texttt{f(x, g(y))} = \texttt{fg(x, y)}$

accumulator removal: $\texttt{h(x,y)} = \texttt{h'(x)} \oplus \texttt{e(y)}$

(conditional) equivalence: $\texttt{pre(x, y)} \Longrightarrow \texttt{f(x)} = \texttt{g(y)}$



input theory:

data types, function defs

extracted lemmas

Key Idea: Combine Structural Function Transformations
[Bird, Burstall & Darlington, …]

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

Key Idea: Combine Structural Function Transformations

[Bird, Burstall & Darlington, …]

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

fixpoint fusion

Wadler, SPJ, Turchin

⇩

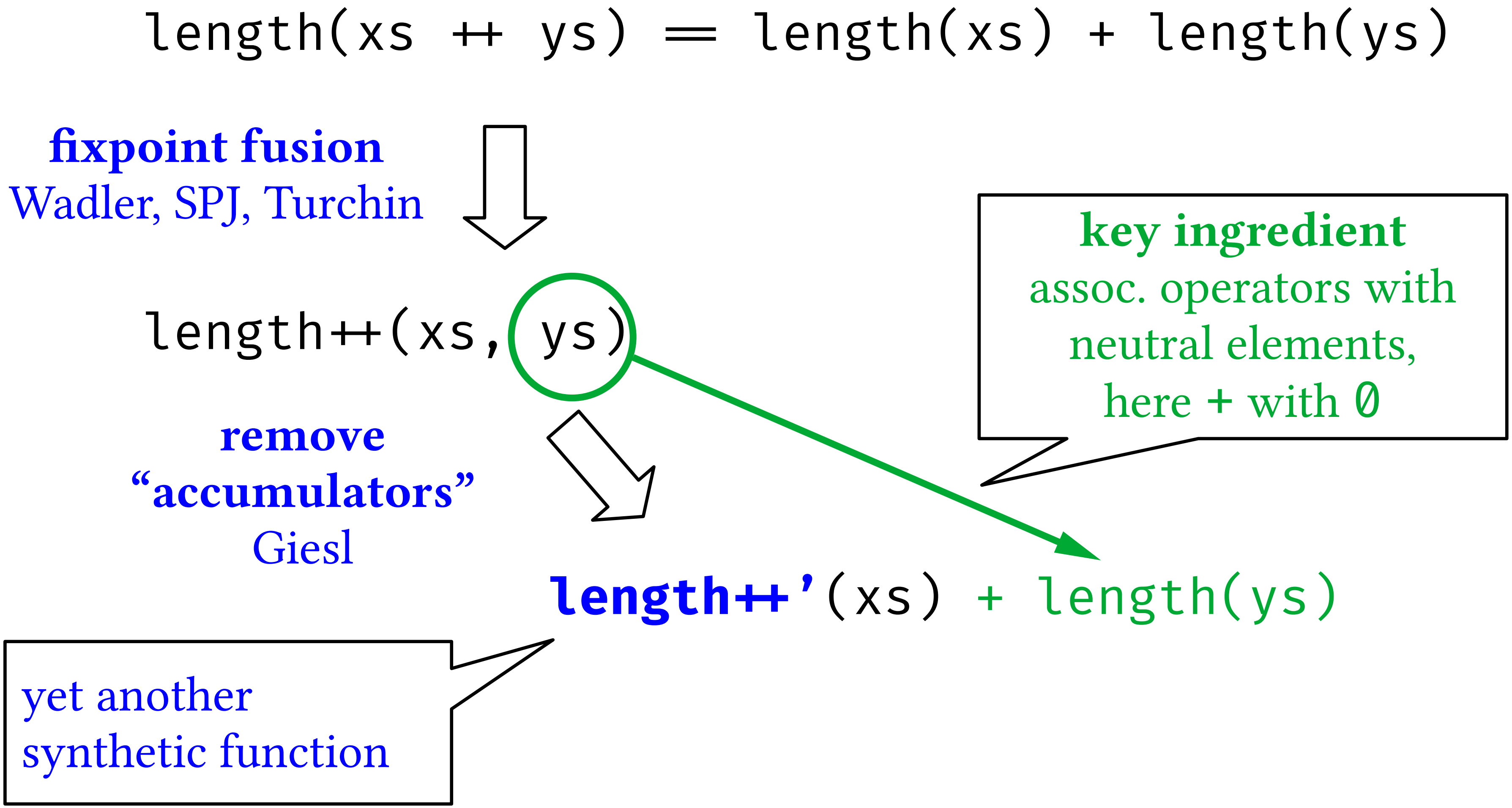
```
length++(xs, ys)
```

synthetic function

as compiler optimization:

- eliminates intermediate list
- only one recursive traversal

Key Idea: Combine Structural Function Transformations [Bird, Burstall & Darlington, …]



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

fixpoint fusion
Wadler, SPJ, Turchin

```
length++(xs, ys)
```

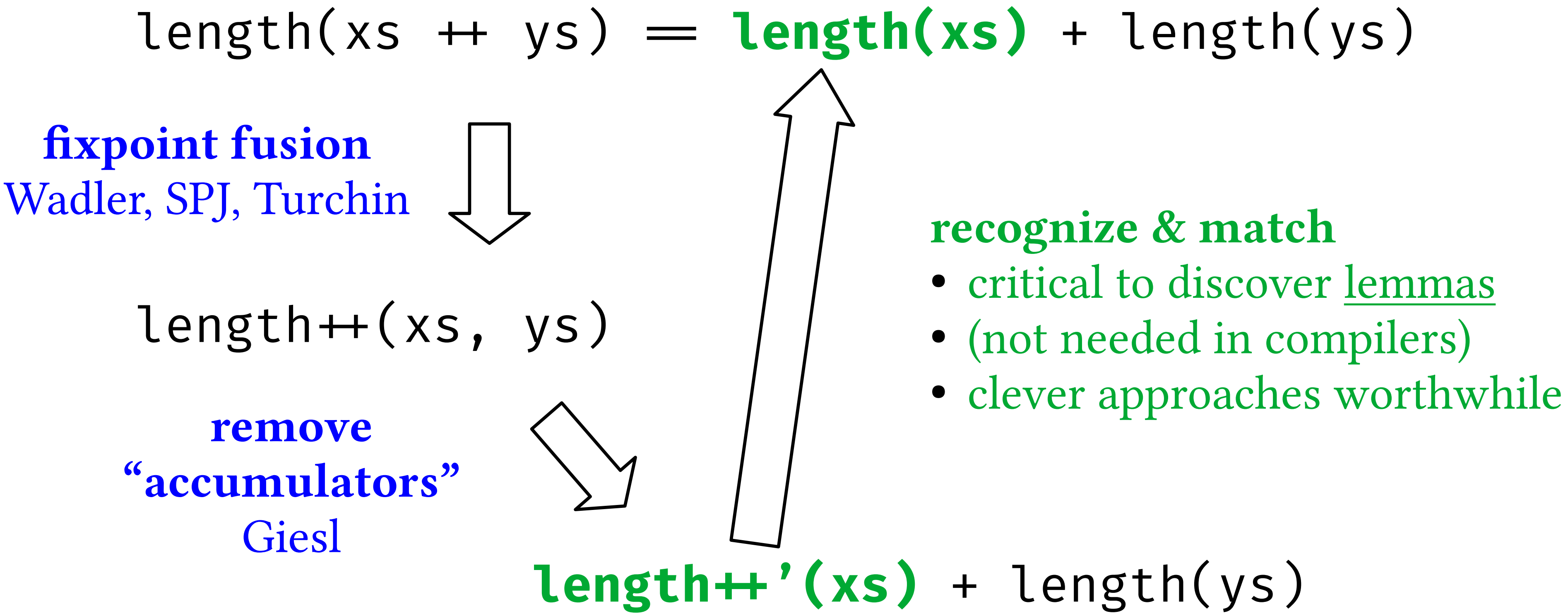
key ingredient
assoc. operators with
neutral elements,
here + with `0`

remove
“accumulators”
Giesl

`length++'``(xs) + length(ys)`

yet another
synthetic function

Key Idea: Combine Structural Function Transformations
[Bird, Burstall & Darlington, …]



`length(xs ++ ys) = `**`length(xs)`**` + length(ys)`

fixpoint fusion
Wadler, SPJ, Turchin

`length++(xs, ys)`

remove “accumulators”
Giesl

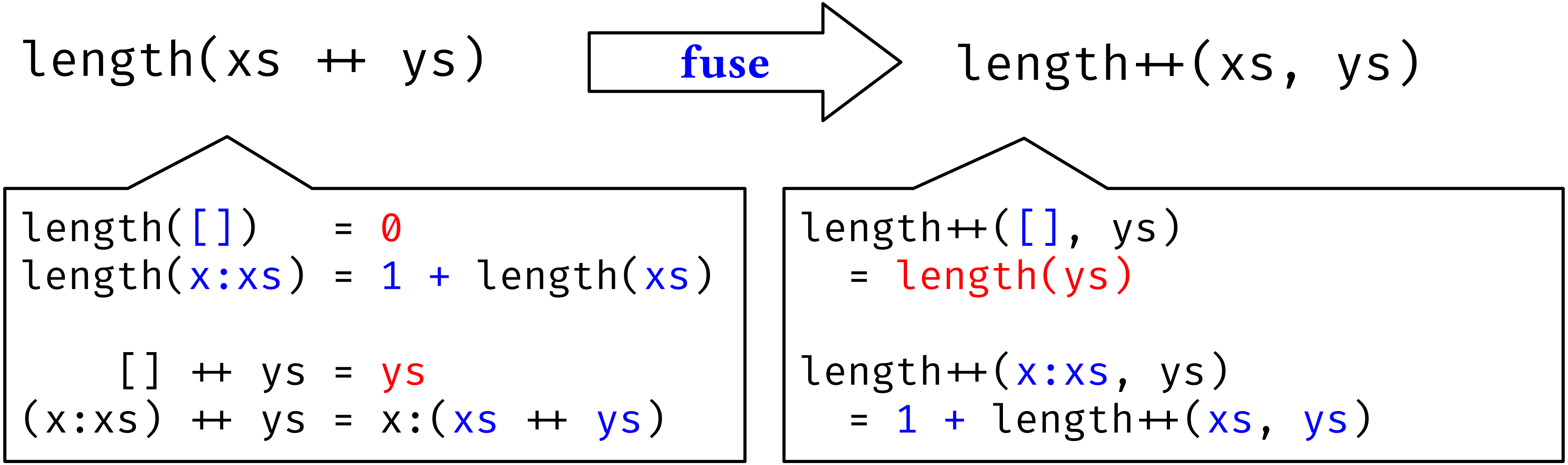
`length++'(xs)`` + length(ys)`

recognize & match

- critical to discover lemmas
- (not needed in compilers)
- clever approaches worthwhile

Example: Fusing `length++`

deforestation, supercompilation, partial evaluation, …



```
length(xs ++ ys)
```

fuse

```
length++(xs, ys)
```

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

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

```
length++([], ys)
  = length(ys)

length++(x:xs, ys)
  = 1 + length++(xs, ys)
```

skeleton of the recurrence is maintained

<u>typically</u> differences manifest in the base cases

Recognizing functions

$\neg\ \texttt{contains(x, xs)} \implies \texttt{remove(x, xs)} = \texttt{id(xs)}$

Recognizing functions

$$\neg\ \texttt{contains(x, xs)} \implies \texttt{remove(x, xs)} = \texttt{id(xs)}$$

```
remove(x, []) = []

remove(x, y:ys)
  = if x ≠ y
    then x : remove(x,ys)
    else     remove(x,ys)
```

$\sqcap$

```
id([]) = []


id(y:ys)
  = x : id(ys)
```

Recognizing functions

$\neg$ `contains(x, xs)` $\implies$ `remove(x, xs)` $=$ `id(xs)`

```
remove(x, []) = []

remove(x, y:ys)
  = if x ≠ y
    then x : remove(x,ys)
    else     remove(x,ys)
```

$\sqcap$

```
id([]) = []


id(y:ys)
  = x : id(ys)
```

$=$

```
pre(x, [])    = ???
pre(x, y:ys) = ???
```

Recognizing functions

```
¬ contains(x, xs)  ⟹  remove(x, xs) == id(xs)
```

```
remove(x, []) = []

remove(x, y:ys)
  = if x ≠ y
    then x : remove(x,ys)
    ~~else      remove(x,ys)~~
```

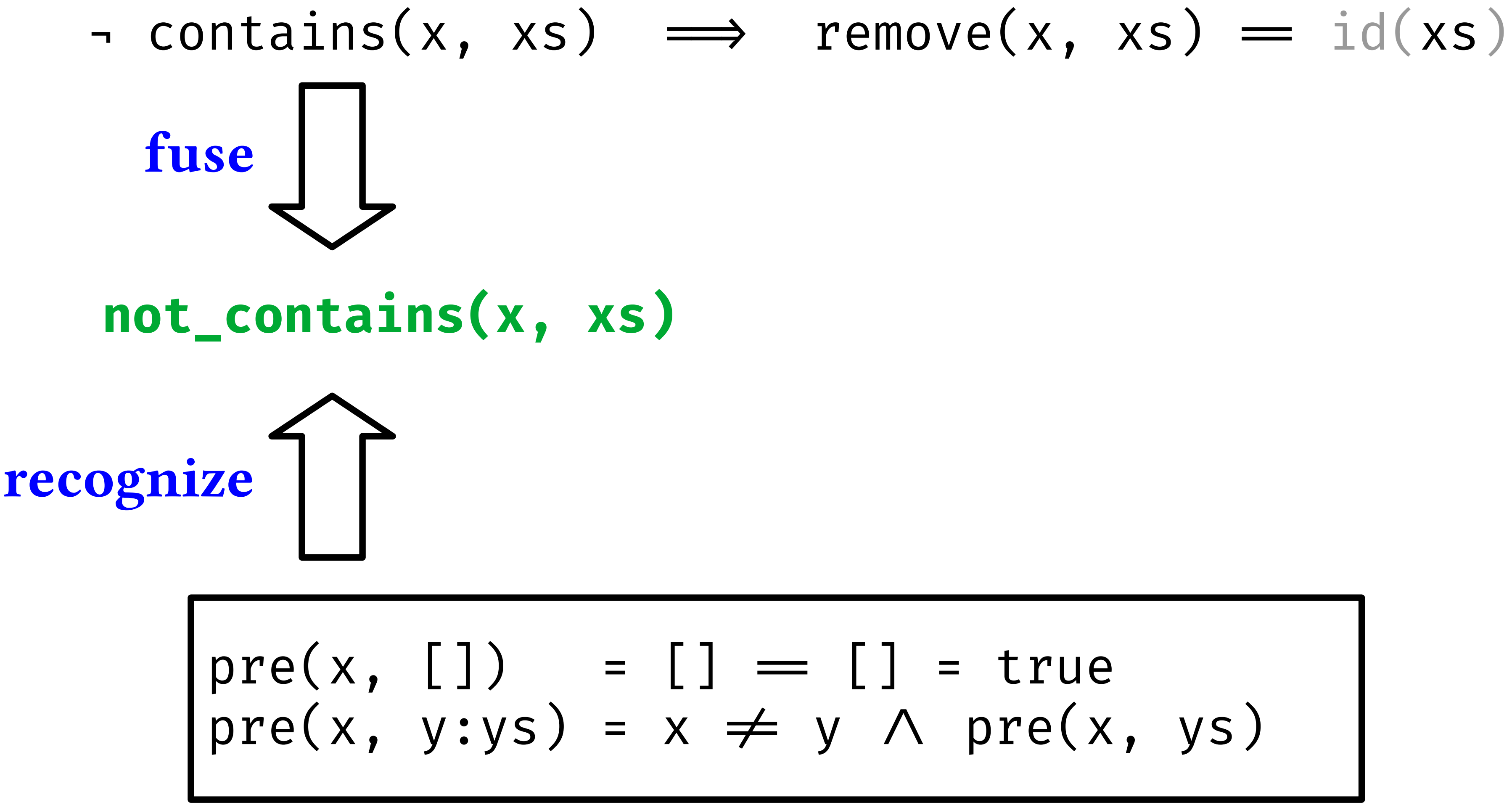
⊓

```
id([]) = []


id(y:ys)
  = x : id(ys)
```

=

```
pre(x, [])    = [] == []
pre(x, y:ys) = x ≠ y ∧ pre(x, ys)
```

Recognizing functions



¬ contains(x, xs) ⟹ remove(x, xs) == id(xs)

fuse

not_contains(x, xs)

recognize

```
pre(x, [])    = [] == [] = true
pre(x, y:ys) = x ≠ y ∧ pre(x, ys)
```

Evaluation

Research Questions: compare LemmaCalc against enumerate & check

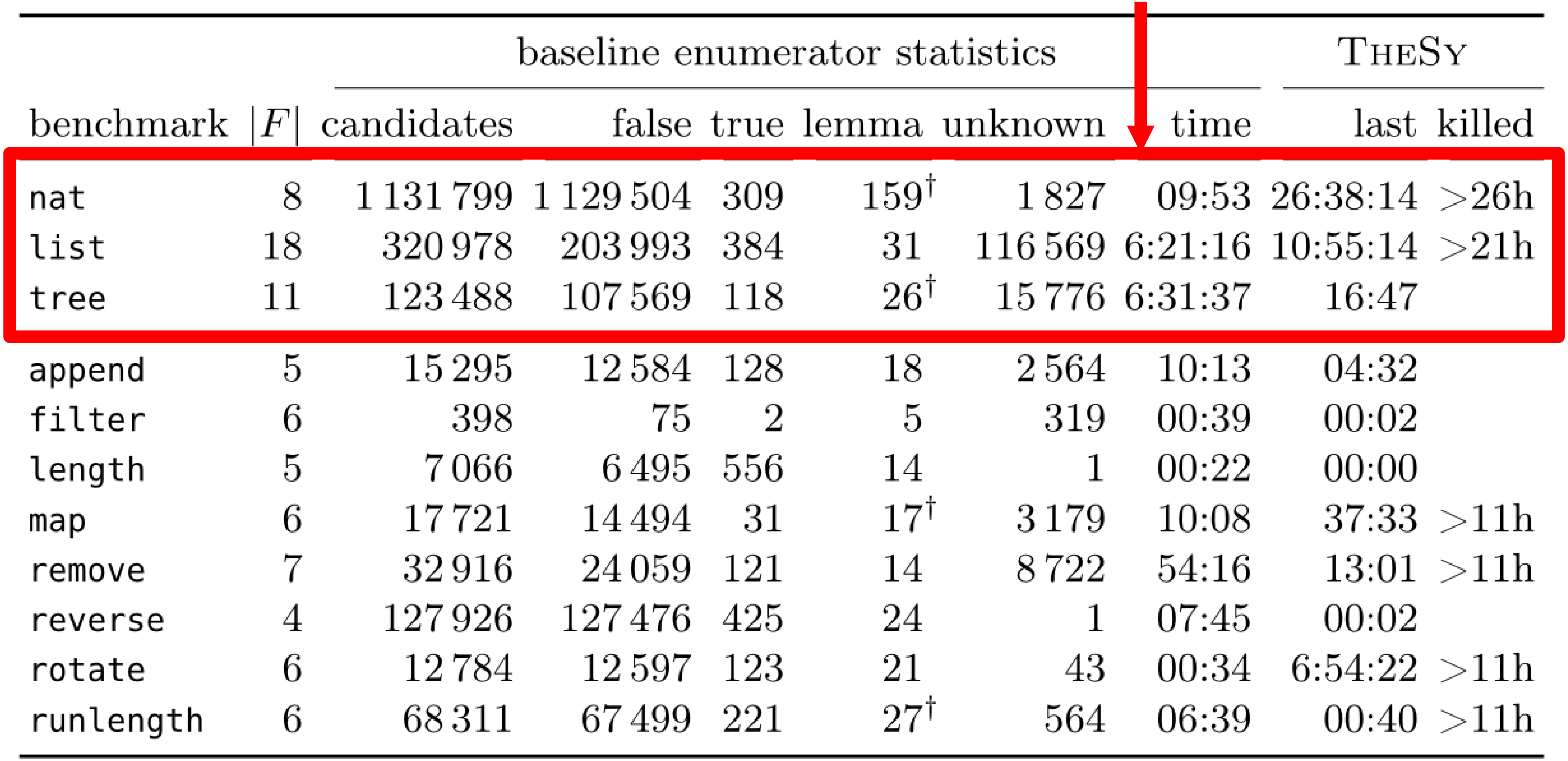
1. what is the scalability to large(r) theories?
2. what proportion of lemmas can be found?

Experimental Setup

- 8 specific benchmarks: combinations of functions with "interesting" lemmas
- three full theries: nat, list, tree with
- **LemmaCalc** (with/without conditional lemmas)
- own baseline enumerator (just lemmas `f(x, g(y)) = ???`)
- TheSy [Singher & Itzhaky, CAV 2021]

Scalability on large(r) theories?

cost of exploring an exponential search space



		baseline enumerator statistics						TheSy	
benchmark	$\|F\|$	candidates	false	true	lemma	unknown	time	last	killed
`nat`	8	1 131 799	1 129 504	309	159†	1 827	09:53	26:38:14	>26h
`list`	18	320 978	203 993	384	31	116 569	6:21:16	10:55:14	>21h
`tree`	11	123 488	107 569	118	26†	15 776	6:31:37	16:47	
`append`	5	15 295	12 584	128	18	2 564	10:13	04:32	
`filter`	6	398	75	2	5	319	00:39	00:02	
`length`	5	7 066	6 495	556	14	1	00:22	00:00	
`map`	6	17 721	14 494	31	17†	3 179	10:08	37:33	>11h
`remove`	7	32 916	24 059	121	14	8 722	54:16	13:01	>11h
`reverse`	4	127 926	127 476	425	24	1	07:45	00:02	
`rotate`	6	12 784	12 597	123	21	43	00:34	6:54:22	>11h
`runlength`	6	68 311	67 499	221	27†	564	06:39	00:40	>11h

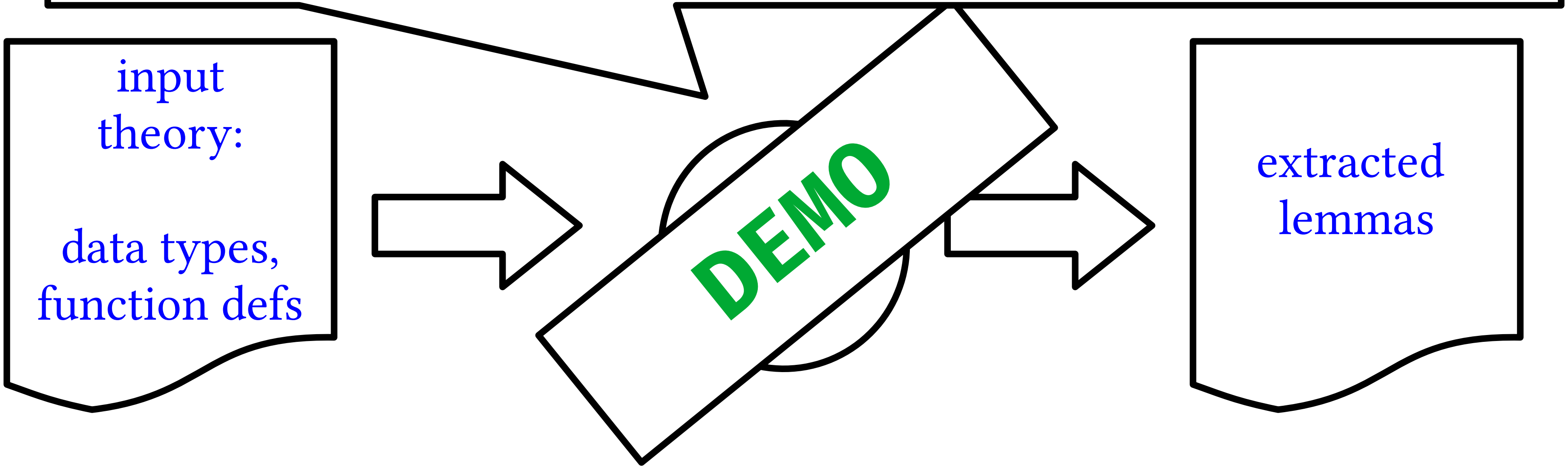
LemmaCalc

Contribution: lemma synthesis by combining structural function transformations

fusion: $f(x, g(y)) = fg(x, y)$

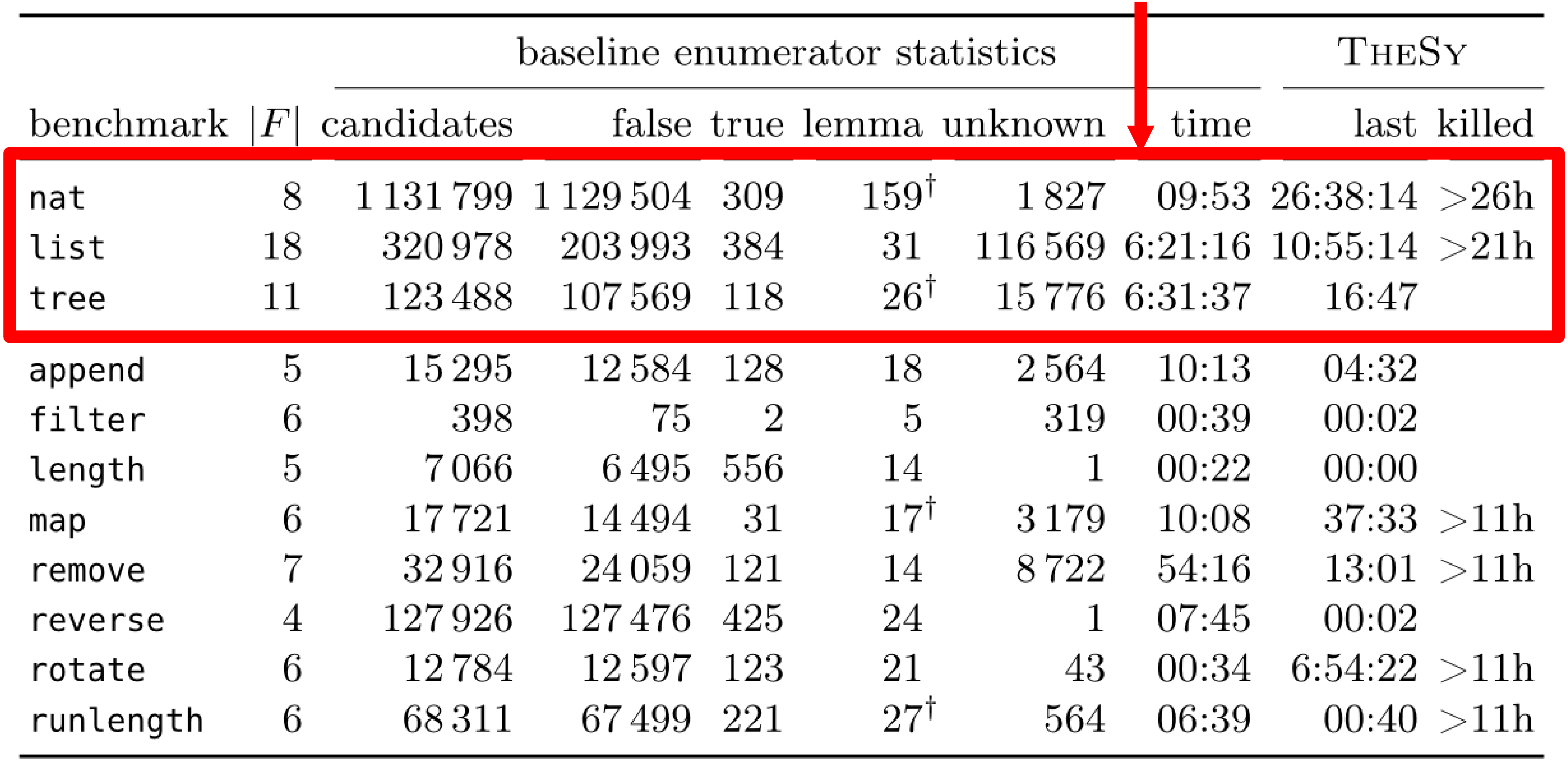
accumulator removal: $h(x,y) = h'(x) \oplus e(y)$

(conditional) equivalence: $pre(x, y) \implies f(x) = g(y)$



Scalability on large(r) theories?

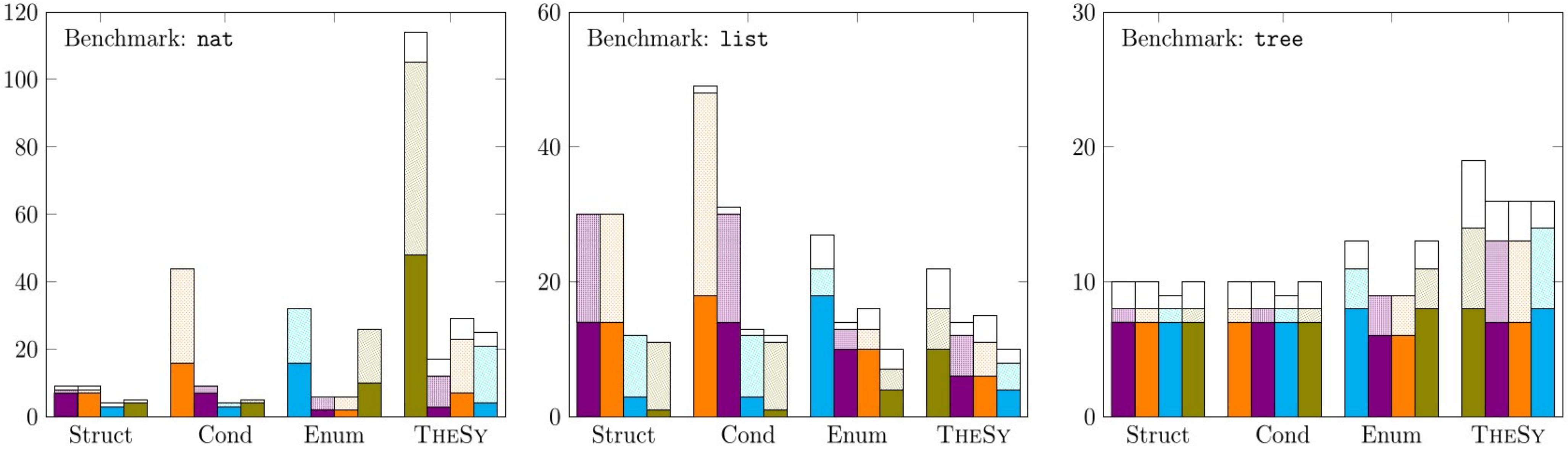
cost of exploring an exponential search space



		baseline enumerator statistics						TheSy	
benchmark	$\|F\|$	candidates	false	true	lemma	unknown	time	last	killed
`nat`	8	1 131 799	1 129 504	309	$159^\dagger$	1 827	09:53	26:38:14	>26h
`list`	18	320 978	203 993	384	31	116 569	6:21:16	10:55:14	>21h
`tree`	11	123 488	107 569	118	$26^\dagger$	15 776	6:31:37	16:47	
`append`	5	15 295	12 584	128	18	2 564	10:13	04:32	
`filter`	6	398	75	2	5	319	00:39	00:02	
`length`	5	7 066	6 495	556	14	1	00:22	00:00	
`map`	6	17 721	14 494	31	$17^\dagger$	3 179	10:08	37:33	>11h
`remove`	7	32 916	24 059	121	14	8 722	54:16	13:01	>11h
`reverse`	4	127 926	127 476	425	24	1	07:45	00:02	
`rotate`	6	12 784	12 597	123	21	43	00:34	6:54:22	>11h
`runlength`	6	68 311	67 499	221	$27^\dagger$	564	06:39	00:40	>11h

LemmaCalc takes 1–5 <u>seconds</u> on each, resp. 14 seconds on `list`

Proportion of lemmas found



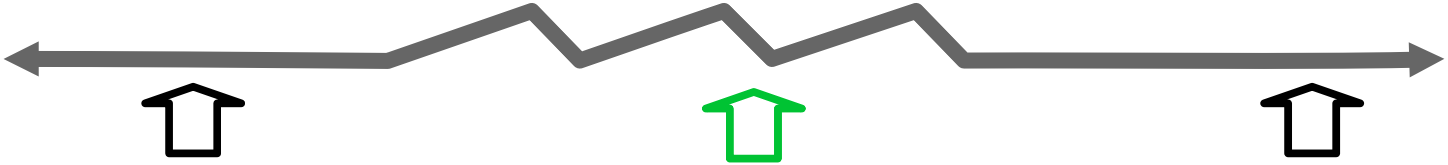
Findings

- approaches and implementations have complementary strenths and weaknesses
- each finds unique lemmas
- many redundant lemmas (different reasons!)
- LemmaCalc generates nice rewrite rules

Color		Pattern	
🟪	Struct	□	trivial
🟧	Cond	▨	implied
🟦	Enum	■	reduced
🟫	THESY		

Summary and Outlook

- **Goal: automation ⊕ expressiveness**



fully automatic **Goal: tackle this** expressive specs

- **Automating equivalence proofs**
  - Goal directed reasoning from candidates [TACAS 21]
  - LemmaCalc: quickly and automatically inferring helper lemmas sosy-lab.org/research/pub/2023-Draft.LemmaCalc.pdf

Example: Duplicate-free Representation

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)

  def erase(x):
    xs = removefirst(xs, x)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}


  def erase(x):
    s = s \ {x}
```

Intuitive approach: additional class invariant `no-duplicates(xs)`

```
contains(removefirst(xs, x), x) ⟺ false
```

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Solution

```
         R(nil, s) :=  s = ∅
R(cons(x,xs), s) :=  x ∈ s ∧ R(xs, s \ {x})
```

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Template for recursion over lists

```
         R(nil, s) :=  s = ∅
R(cons(x,xs), s) :=  x ∈ s ∧ R(xs, s \ {x})
```

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Construction: base case via initialization

```
        R(nil, s)  :=  s = ∅
R(cons(x,xs), s)  :=  x ∈ s ∧ R(xs, s \ {x})
```

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Construction: which operation matches the recursive case?

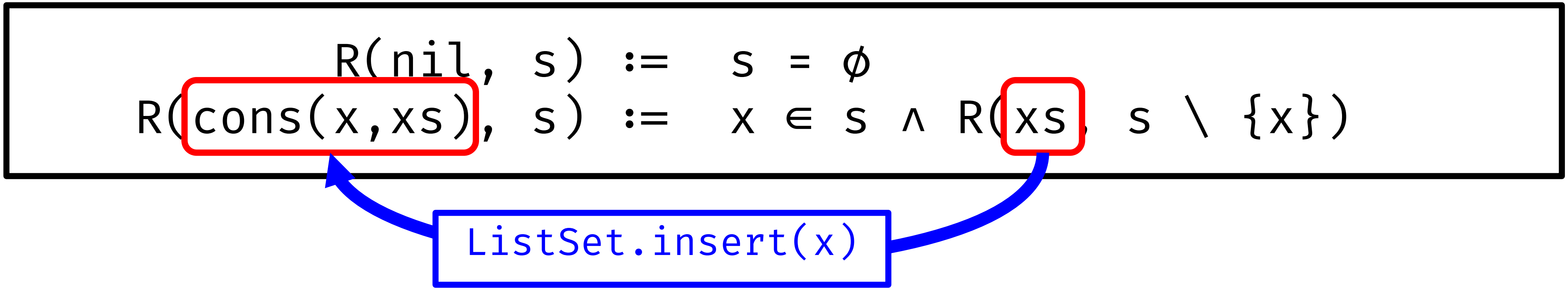
$$\begin{aligned} R(\mathtt{nil},\ s) &:= \quad s = \emptyset \\ R(\boxed{\mathtt{cons}(x,xs)},\ s) &:= \quad x \in s \wedge R(\boxed{xs},\ s \setminus \{x\}) \end{aligned}$$

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Construction: which operation matches the recursive case?



```
      R(nil, s)  :=  s = ∅
R(cons(x,xs), s) :=  x ∈ s ∧ R(xs, s \ {x})
```

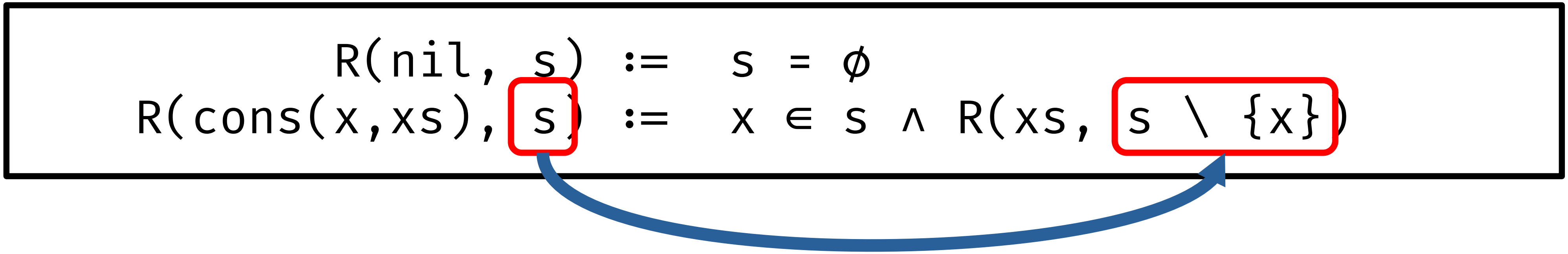
Approach: “Producer”-operation explains the recurrence of R

Approach 2: recursively defined simulation relations

```
class ListSet:

  def init():
    xs = nil

  def insert(x):
    if not contains(xs, x):
      xs = cons(x, xs)
```

```
class SpecSet:

  def init():
    s = ∅

  def insert(x):
    s = s ∪ {x}
```

Construction: which operation matches the recursive case?



```
          R(nil, s) :=  s = ∅
R(cons(x,xs), s) :=  x ∈ s ∧ R(xs, s \ {x})
```

- difference in observations via `hasElement` (somewhat involved)
- duplication-freedom is implied (via inductive lemma)

Proof Obligations of Forward Simulation (well known)

Given two System Models as Data Types

$$A = (AState,\ AInit,\ AOp_1,\ \ldots,\ AOp_n)$$

$$C = (CState,\ CInit,\ COp_1,\ \ldots,\ COp_n)$$

Refinement $B \leqslant A$ as Horn clauses (here deterministic systems), find R:

$$\left. as = AInit() \land cs = CInit() \implies R(as,\ cs) \right\} \text{initialization}$$

$$R(as,cs) \land (as',out) = AOp_i(in,as) \land (cs',out') = COp_i(in,cs)$$

$$\implies \underbrace{out = \text{out‘}}_{\text{output equivalence}} \land \underbrace{R(as',cs')}_{\text{inductive preservation}}$$

Recent Research:

- explore design space
- understand relative strengths of methods
- experiment with novel angles of attack

No runtime errors

- Legion: Coverage-guided Testing (with D. Liu+) [ASE 20]
- Korn: C verification with Horn clauses [VMCAI 22]

Automating functional correctness proofs

- Inference of data abstractions (with G. Fedyukovich) [TACAS 21]
- Synthesis of lemmas (submitted, with G. Fedyukovich)
- Cuvée: An SMT-LIB engineering Toolkit (ongoing)

High-level security for low-level C code

- Security Concurrent Separation Logic (SecCSL) [Ernst & Murray, CAV 19]
- Declassification policies [T. Murray, M. Tiwari, D. Naumann, CCS 23], Amazon grant

Falsification of Hybrid Systems

- FalStar: adaptive "Las-Vegas" tree search [Ernst+, TOMACS 21]
- Extending rapidly-exploring random trees to trajectories (ongoing, with J. Fejlek, S. Ratschan)

Competitions and Challenges

- VerifyThis [iFM 23, TACAS 20], SV-COMP, Test-Comp, ARCH-COMP