Velvet: A Multi-Modal Verifier for Imperative Programs in Lean

Ilya Sergey

ilyasergey.net

Get Velvet Now

github.com/verse-lab/velvet

branch **leanlang-demo**

lake exe cache get; lake build

Floyd-Hoare Logic

For predicates P and Q and program S, the *Hoare triple*

$$\{ P \}\ S\ \{ Q \}$$

states the following:

If S is started in any state that satisfies P,
then S will not crash,
and, if it terminates, it finishes in some state satisfying Q

Examples:

$$\{ x = 1 \}\ x := 20\ \{ x = 20 \}$$
$$\{ x < 18 \}\ y := 18 - x\ \{ y >= 0 \}$$
$$\{ x < 18 \}\ y := 5\ \{ y >= 0 \}$$

Non-example: $\{ x < 18 \}\ x := y\ \{ y >= 0 \}$

Reducing an Imperative Program to a Formula

Programming Languages
T.A. Standish Editor

Guarded Commands, Nondeterminacy and Formal Derivation of Programs

Edsger W. Dijkstra
Burroughs Corporation

So-called "guarded commands" are introduced as a building block for alternative and repetitive constructs that allow nondeterministic program components for which at least the activity evoked, but possibly even the final state, is not necessarily uniquely determined by the initial state. For the formal derivation of programs expressed in terms of these constructs, a calculus will be be shown.

Key Words and Phrases: programming languages, sequencing primitives, program semantics, programming language semantics, nondeterminacy, case-construction, repetition, termination, correctness proof, derivation of programs, programming methodology
CR Categories: 4.20, 4.22

1. Introduction

In Section 2, two statements, an alternative construct and a repetitive construct, are introduced, together with an intuitive (mechanistic) definition of their semantics. The basic building block for both of them is the so-called "guarded command," a statement list prefixed by a boolean expression: only when this boolean expression is initially true, is the statement list eligible for execution. The potential nondeterminacy allows us to map otherwise (trivially) different programs on the same program text, a circumstance that seems largely responsible for the fact that programs can now be derived in a manner more systematic than before.

In Section 3, after a prelude defining the notation, a formal definition of the semantics of the two constructs is given, together with two theorems for each of the constructs (without proof).

In Section 4, it is shown how, based upon the above, a formal calculus for the derivation of programs can be founded. We would like to stress that we do not present "an algorithm" for the derivation of programs: we have used the term "a calculus" for a formal discipline—a set of rules—such that, if applied successfully: (1) it will have derived a correct program; and (2) it will tell us that we have reached such a goal. (We use the term as in "integral calculus.")

2. Two Statements Made from Guarded Commands

Commun. of ACM,1975

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {

    res := true

  } else {

    res := false

  }

} else {

  res := true

}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {

    res := true

  } else {

    res := false

  }
{ res ⇔ (a ⇒ b) }
} else {

  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {

    res := true

  } else {

    res := false

  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {

    res := true
{ res ⇔ (a ⇒ b) }
  } else {

    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {

    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {

  if (b) {
{ true ⇔ (a ⇒ b) }
    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)

if (a) {
{ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b))) }
  if (b) {
{ true ⇔ (a ⇒ b) }
    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Reducing an Imperative Program to a Formula

```
{ true }
// Check that this implication is valid (negation is unsatisfiable)
{ (a ⇒ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b)))) ∧ (¬a ⇒ (true ⇔ (a ⇒ b)))  }
if (a) {
{ (b ⇒ (true ⇔ (a ⇒ b))) ∧ (¬b ⇒ (false ⇔ (a ⇒ b))) }
  if (b) {
{ true ⇔ (a ⇒ b) }
    res := true
{ res ⇔ (a ⇒ b) }
  } else {
{ false ⇔ (a ⇒ b) }
    res := false
{ res ⇔ (a ⇒ b) }
  }
{ res ⇔ (a ⇒ b) }
} else {
{ true ⇔ (a ⇒ b) }
  res := true
{ res ⇔ (a ⇒ b) }
}
{ res ⇔ (a ⇒ b) }
```

Automated Program Verifiers

gobra

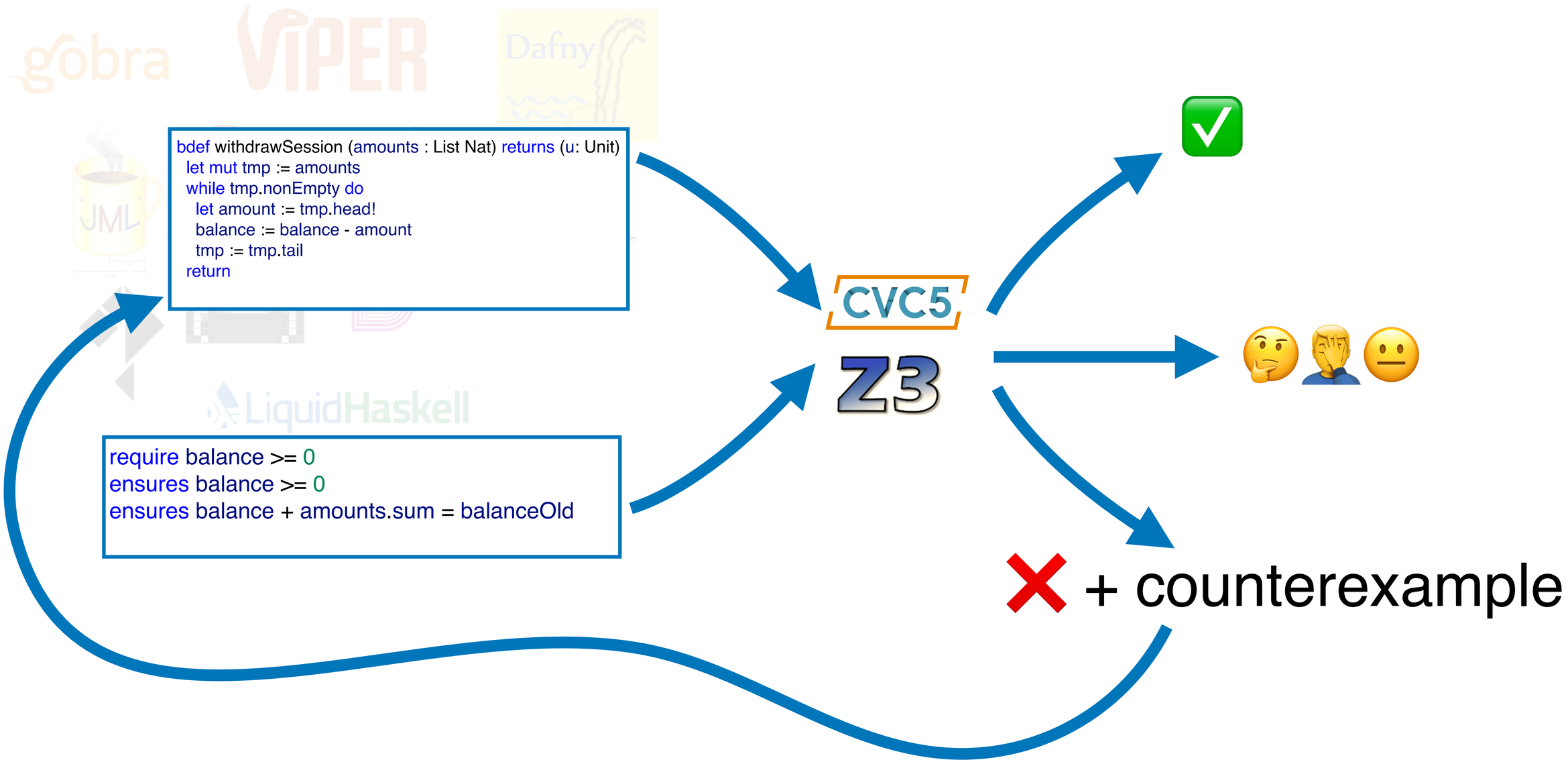
VIPER

Dafny

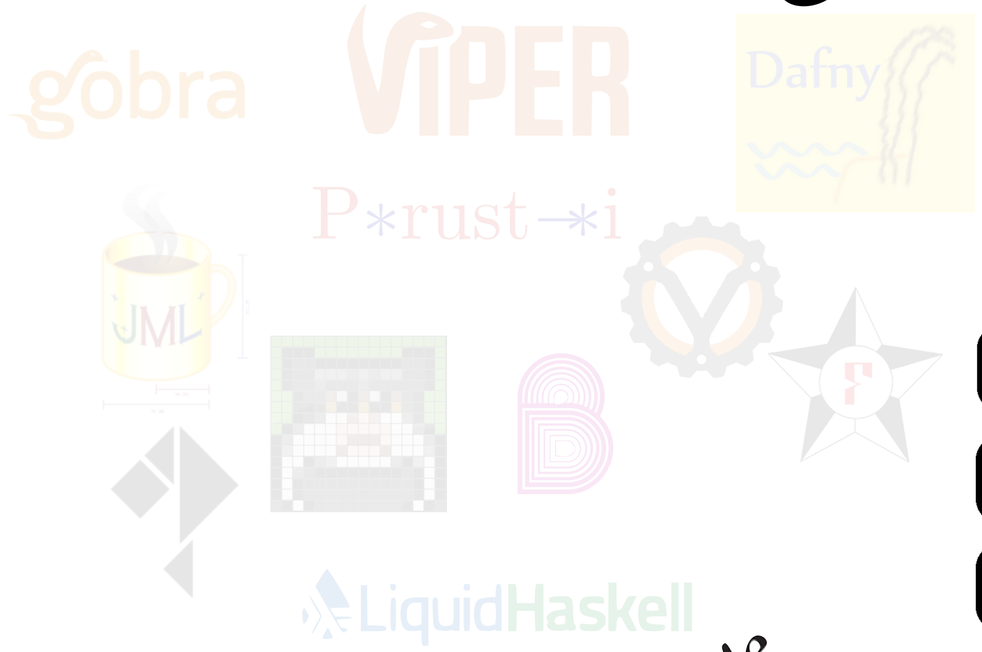
P*rust→*i

LiquidHaskell


gobra
VIPER
Dafny
P∗rust−∗i
JML
LiquidHaskell


bdef withdrawSession (amounts : List Nat) returns (u: Unit)
let mut tmp := amounts
while tmp.nonEmpty do
let amount := tmp.head!
balance := balance - amount
tmp := tmp.tail
return
require balance >= 0
ensures balance >= 0
ensures balance + amounts.sum = balanceOld
CVC5
Z3
+ counterexample

Interactive Program Verifiers



HTT


gobra

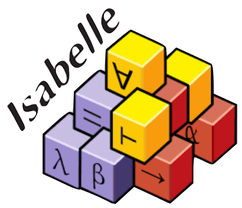

VIPER


Dafny


P*rust→*i


JML


LiquidHaskell

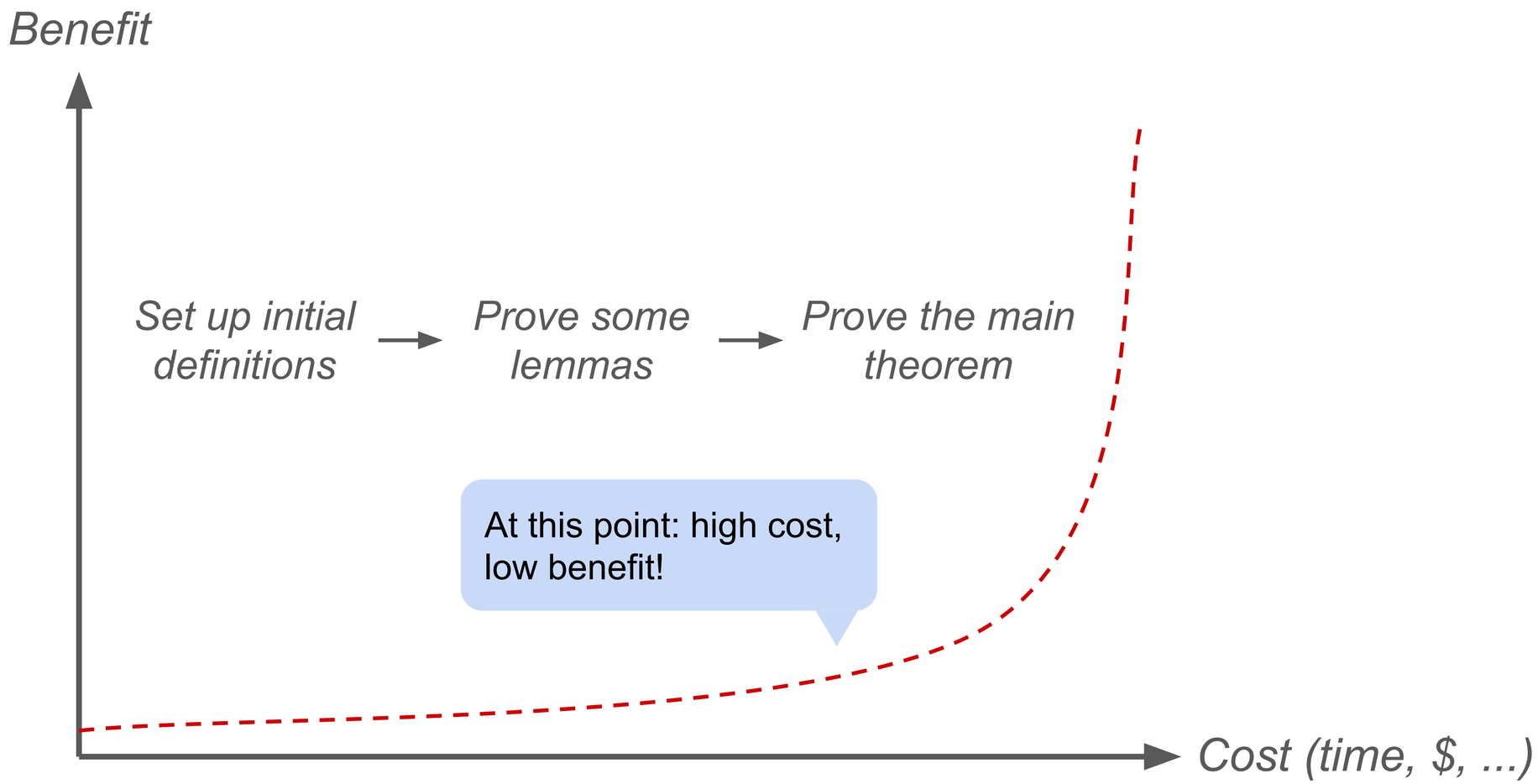

Isabelle


Verified
Software
Toolchain


Iris


ROCQ
HTT

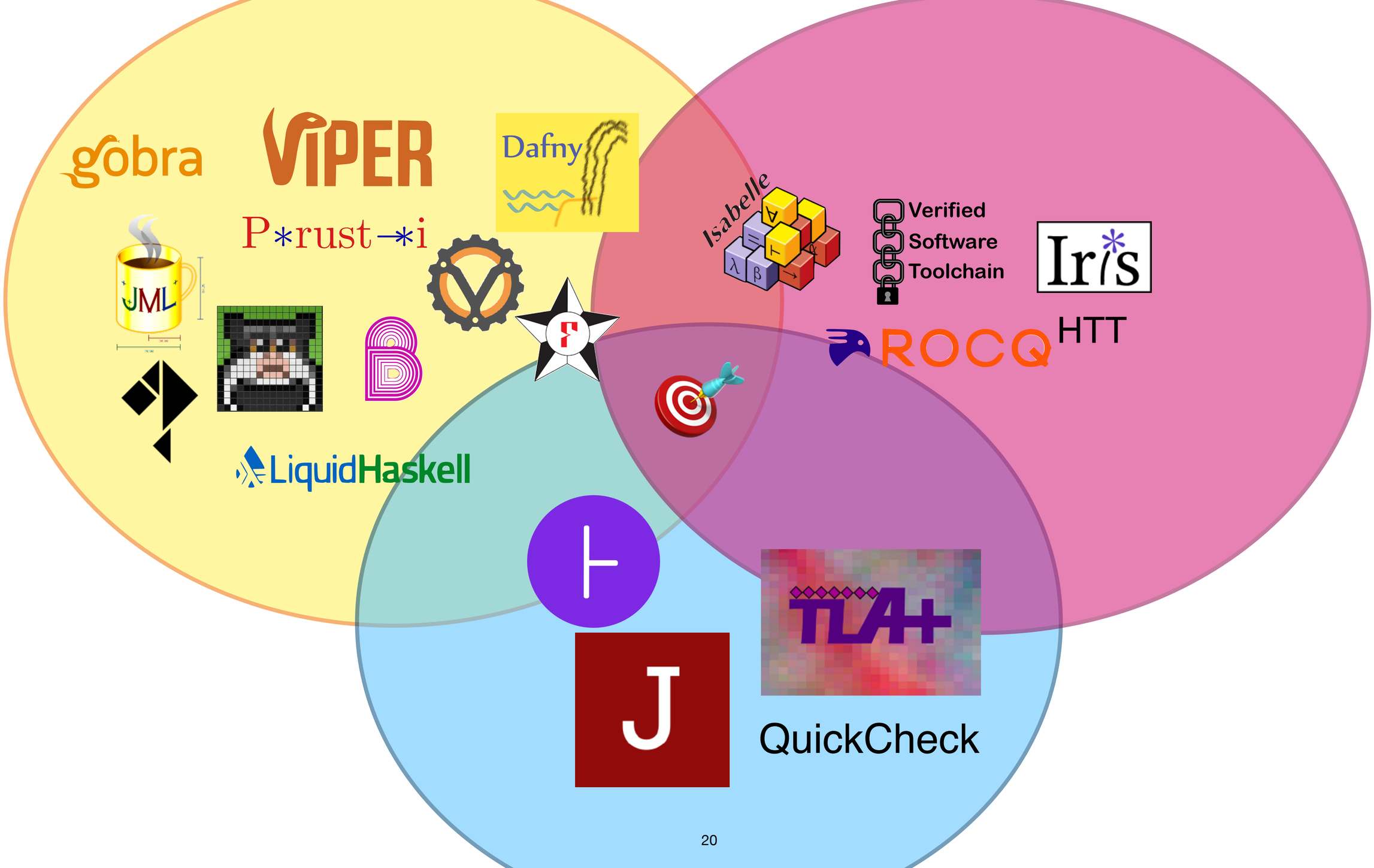
A bad cost/benefit curve

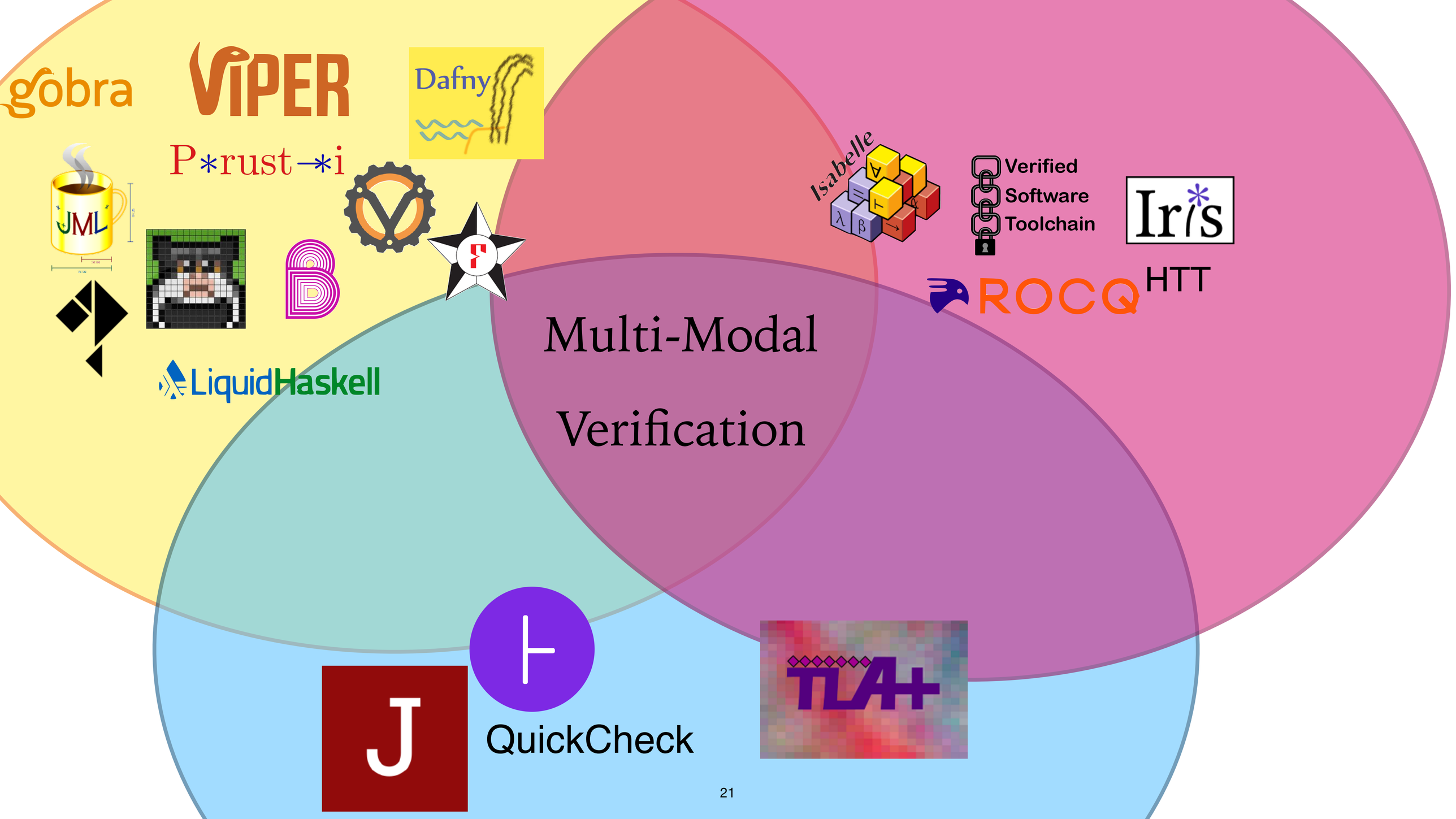


N things I learned trying to do formal methods in industry, Mike Dodds., Big Spec Workshop'24

Bug Finding Tools

QuickCheck


gobra
VIPER
Dafny
P*rust→*i
JML
LiquidHaskell
Isabelle
Verified
Software
Toolchain
Iris
ROCQ
HTT
TLA+
J
QuickCheck


gobra
VIPER
Dafny
P*rust→*i
JML
Isabelle
Verified
Software
Toolchain
Iris
ROCQ
HTT
LiquidHaskell
Multi-Modal
Verification
QuickCheck
TLA+

Verification Framework Design Aspects

- State
- Termination
- Bugs
- Non-determinism

Verification Framework Design Aspects

- State: Symbolic
- Termination: Total
- Bugs: Absence
- Non-determinism: ∀

Verification Framework Design Aspects

- State: Symbolic
- Termination: Total vs Partial
- Bugs: Absence
- Non-determinism: ∀

Verification Framework Design Aspects

- State: Symbolic
- Termination: Total vs Partial
- Bugs: Absence vs Presence
- Non-determinism: ∀ vs ∃

Verification Framework Design Aspects

- State: Symbolic vs Concrete
- Termination: Total vs Partial
- Bugs: Absence vs Presence
- Non-determinism: ∀ vs ∃

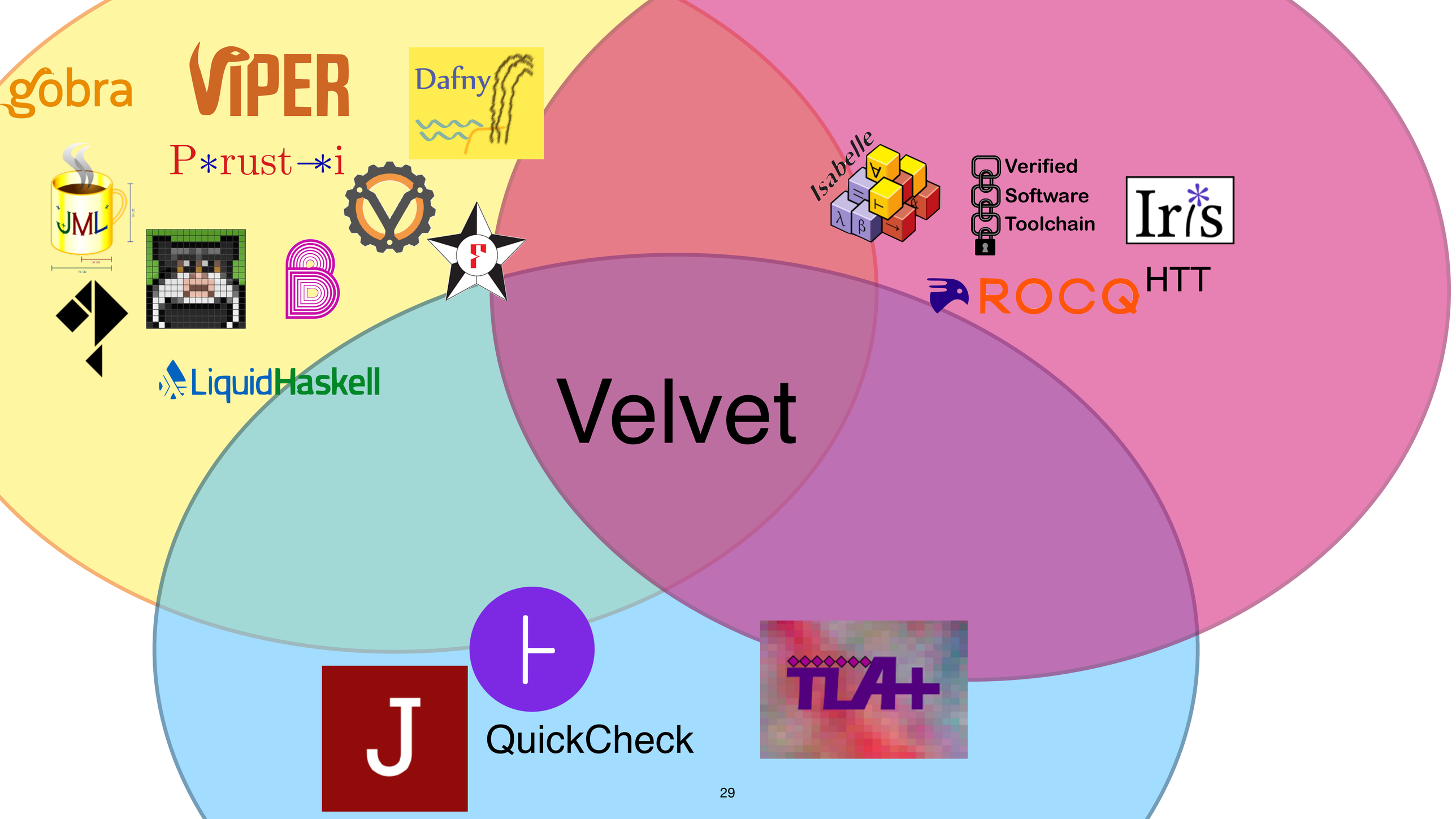
QuickCheck

We present **Velvet**

We present **Velvet**

Verification framework combining

+  ROCQ +  + QuickCheck


gobra
VIPER
Dafny
P*rust→*i
JML
Isabelle
Verified
Software
Toolchain
Iris
ROCQ
HTT
LiquidHaskell
Velvet
J
QuickCheck
TLA+

Velvet

Interactive Proofs

SMT
grind, aesop

Velvet Demo

- Stateful Programs
- Handling Termination
- Absence and Presence of Bugs
- Non-deterministic choice

Combining Correctness Proofs

```
lemma partial_total_split {α} : ∀ (c₁ c₂ : VelvetM α) (P : Prop) (Q : α → Prop),
eraseEq c₁ c₂ → triplePartial P c₁ Q → tripleTotal P c₂ (λ _, True) → tripleTotal P c₁ Q
```

same program

functional correctness

termination

total correctness

Demo

(Functional and Total correctness of InsertSort)

Combining program verification and math proofs

$$\{\mathbf{P}\}\ \mathrm{c}(args)\ \{\lambda\, res.\ \varphi(args, res)\}$$

Combining program verification and math proofs

$\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ \varphi(args, res)\}$

Two-layer verification:

1. Prove the Hoare triple for a program c

   $\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ res = f(args)\}$

   for some mathematical function f

2. Prove that the result of $f(x)$ satisfies the property $\varphi(x, f(x))$

3. Specification weakening gets us:

   $\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ \varphi(args, res)\}$

Combining program verification and math proofs

$\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ \varphi(args, res)\}$

Two-layer verification:

1. Prove the Hoare triple for a program c
   $\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ res = f(args)\}$
   for some mathematical function f

Often can be done via SMT

2. Prove that the result of $f(x)$ satisfies the property $\varphi(x, f(x))$

interactive via **mathlib**

3. Specification weakening gets us:
   $\{\mathbf{P}\}$ c(*args*) $\{\lambda\, res.\ \varphi(args, res)\}$

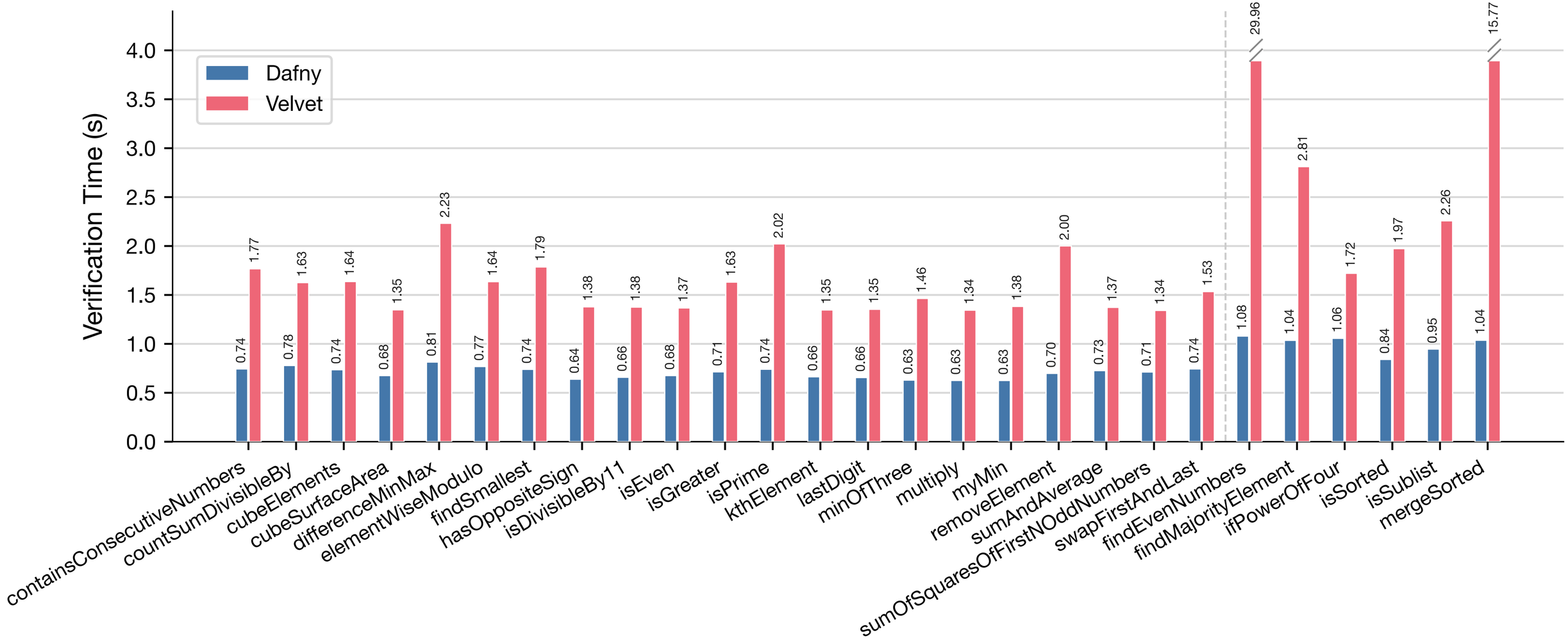
for free

Case Study: Sparse Matrix/Vector Multiplication

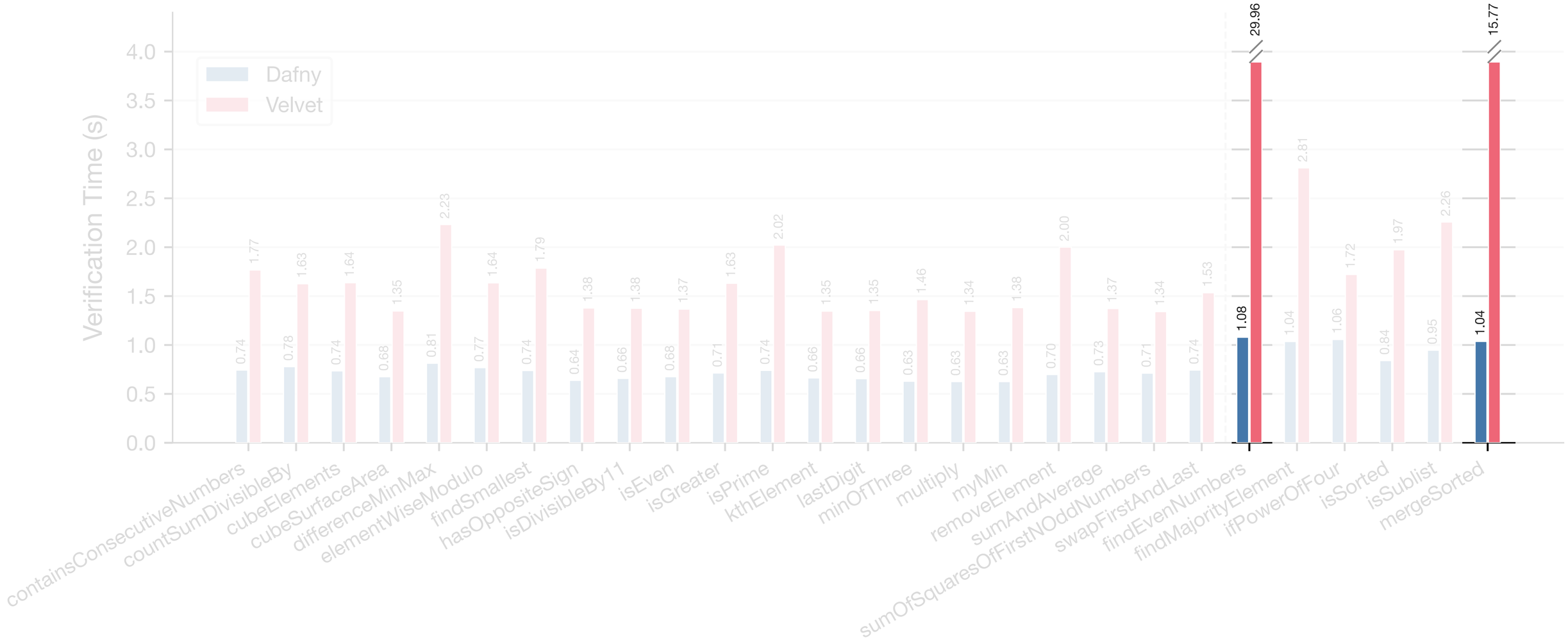
- Efficient multiplication of two compressed tensor representations
- Both **spm** and **spv** are collections of arrays compressing the tensors

```
method SpMSpV
  (mut out: arrVal)
  (spm: arrSpV)
  (spv: SpVAbs) return (u: Unit)
  ensures size outNew = size spm
  ensures ∀ i < size spm, outNew[i] = spv_dot spm[i] spv 0 0
  do
    out := val_inst.replicate (size spm) 0
    let mut spmInd := ind_inst.replicate (size spm) 0
    let mut spvInd := ind_inst.replicate (size spm) 0
    while_some i :| i < (size spm) ∧ TArray.get i spmInd < SpVT.size (TArray.get i spm) ∧
                   TArray.get i spvInd < SpVT.size spv
    do
      let ind_m := spmInd[i]
      let ind_v := spvInd[i]
      if TArray.get ind_m (SpVT.ind spm[i]) = TArray.get ind_v (SpVT.ind spv) then
        out[i] += TArray.get ind_m (SpVT.val spm[i]) * TArray.get ind_v (SpVT.val spv)
        spmInd[i] += 1
        spvInd[i] += 1
      else
        if (SpVT.ind spm[i])[ind_m] < (SpVT.ind spv)[ind_v] then
          spmInd[i] += 1
        else
          spvInd[i] += 1
    return
```

How does Velvet compare to state-of-the-art?

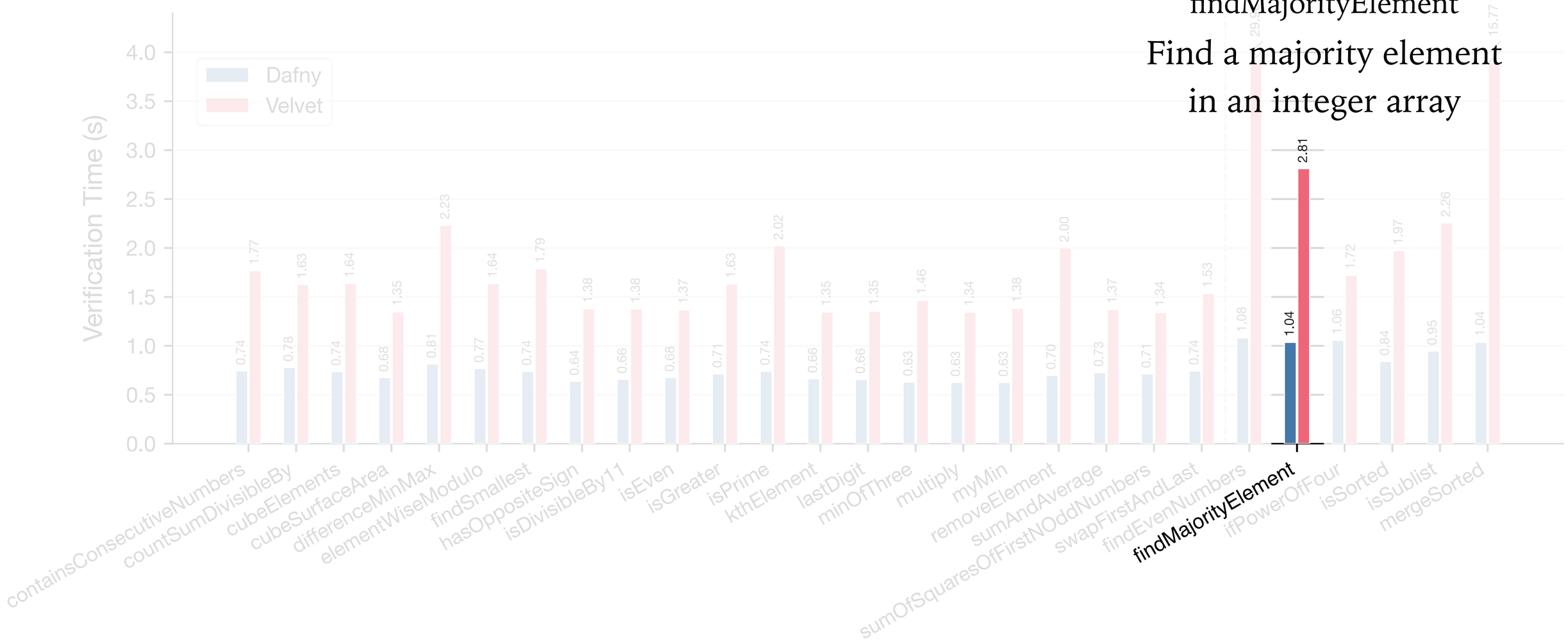

Verification Time (s)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Dafny
Velvet
0.74
1.77
0.78
1.63
0.74
1.64
0.68
1.35
0.81
2.23
0.77
1.64
0.74
1.79
0.64
1.38
0.66
1.38
0.68
1.37
0.71
1.63
0.74
2.02
0.66
1.35
0.66
1.35
0.63
1.46
0.63
1.34
0.63
1.38
0.70
2.00
0.73
1.37
0.71
1.34
0.74
1.53
1.08
29.96
1.04
2.81
1.06
1.72
0.84
1.97
0.95
2.26
1.04
15.77
containsConsecutiveNumbers
countSumDivisibleBy
cubeElements
cubeSurfaceArea
differenceMinMax
elementWiseModulo
findSmallest
hasOppositeSign
isDivisibleBy11
isEven
isGreater
isPrime
kthElement
lastDigit
minOfThree
multiply
myMin
removeElement
sumAndAverage
sumOfSquaresOfFirstNOddNumbers
swapFirstAndLast
findEvenNumbers
findMajorityElement
ifPowerOfFour
isSorted
isSublist
mergeSorted

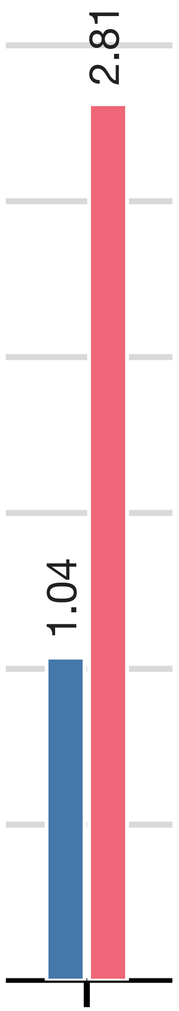
grind problems



findMajorityElement

Find a majority element in an integer array

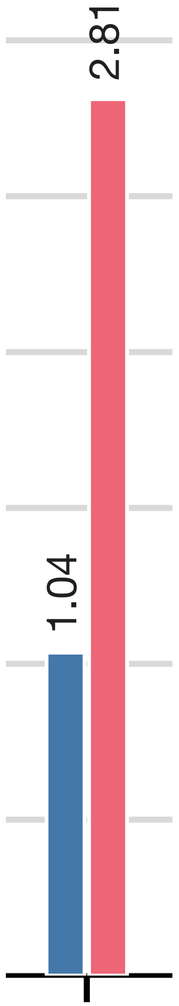




findMajorityElement

```
import CaseStudies.Velvet.Std

set_option loom.semantics.termination "partial"
set_option loom.semantics.choice "demonic"

…

method findMajorityElement (lst : List Int)
  return (result : Int)
  ensures lst.hasMajorityElement → (result ∈ lst ∧ lst.isMajorityElement result)
  ensures ¬lst.hasMajorityElement → result = -1
  do
  let n := lst.length
  let threshold := n / 2
  let mut i := 0
  let mut found := false
  let mut candidate : Int := -1

  while i < n
  invariant "i_bounds" 0 ≤ i ∧ i ≤ n
  invariant "found_implies_majority" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
  invariant "not_found_checked" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
  do
    let elem := lst[i]!
    let mut count := 0
    let mut j := 0

    while j < n
    invariant "j_bounds" 0 ≤ j ∧ j ≤ n
    invariant "count_correct" count = (lst.take j).count elem
    invariant "found_preserved" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
    invariant "i_preserved" 0 ≤ i ∧ i < n
    invariant "elem_in_list" i < lst.length → elem = lst[i]!
    invariant "not_found_checked_inner" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
    do
      if lst[j]! = elem then
        count := count + 1
      j := j + 1

    if count > threshold then
      found := true
      candidate := elem

    i := i + 1

  if found then
    return candidate
  else
    return -1
```

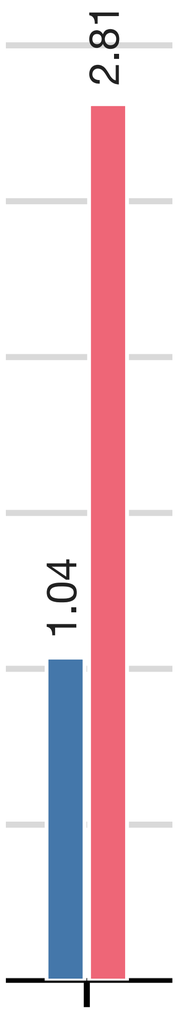
```
attribute [grind] List.take_succ_eq_append_getElem


prove_correct findMajorityElement by
  loom_vc_gen <;>


  loom_solver
```



findMajorityElement

```
// Lemma: count in prefix increases by 1 when element matches
lemma CountPrefixIncrement(lst: seq<int>, x: int, j: nat)
  requires 0 <= j < |lst|
  requires lst[j] == x
  ensures countPrefix(lst, x, j + 1) == countPrefix(lst, x, j) + 1
{
  assert lst[..j+1] == lst[..j] + [lst[j]];
  CountConcat(lst[..j], [lst[j]], x);
}

// Lemma: count in prefix stays same when element doesn't match
lemma CountPrefixSame(lst: seq<int>, x: int, j: nat)
  requires 0 <= j < |lst|
  requires lst[j] != x
  ensures countPrefix(lst, x, j + 1) == countPrefix(lst, x, j)
{
  assert lst[..j+1] == lst[..j] + [lst[j]];
  CountConcat(lst[..j], [lst[j]], x);
}

// Lemma: full prefix equals the whole list
lemma CountPrefixFull(lst: seq<int>, x: int)
  ensures countPrefix(lst, x, |lst|) == count(lst, x)
{
  assert lst[..|lst|] == lst;
}
```

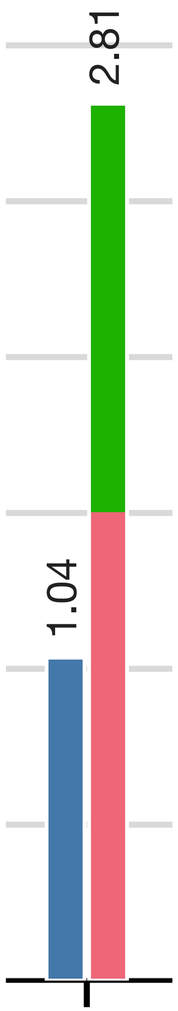
```
var cnt := 0;
var j := 0;

while j < n
  invariant 0 <= j <= n
  invariant cnt == countPrefix(lst, elem, j)
  invariant found == true ==> (candidate in lst && isMajorityElement(lst, candidate))
  invariant 0 <= i < n
  invariant elem == lst[i]
  invariant found == false ==> (forall k :: 0 <= k < i ==> !isMajorityElement(lst, lst[k]))
  decreases n - j
{
  if lst[j] == elem {
    CountPrefixIncrement(lst, elem, j);
    cnt := cnt + 1;
  } else {
    CountPrefixSame(lst, elem, j);
  }
  j := j + 1;
}

CountPrefixFull(lst, elem);

if cnt > threshold {
```



findMajorityElement

```
import CaseStudies.Velvet.Std

set_option loom.semantics.termination "partial"
set_option loom.semantics.choice "demonic"

...

method findMajorityElement (lst : List Int)
  return (result : Int)
  ensures lst.hasMajorityElement → (result ∈ lst ∧ lst.isMajorityElement result)
  ensures ¬lst.hasMajorityElement → result = -1
  do
  let n := lst.length
  let threshold := n / 2
  let mut i := 0
  let mut found := false
  let mut candidate : Int := -1

  while i < n
  invariant "i_bounds" 0 ≤ i ∧ i ≤ n
  invariant "found_implies_majority" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
  invariant "not_found_checked" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
  do
    let elem := lst[i]!
    let mut count := 0
    let mut j := 0

    while j < n
    invariant "j_bounds" 0 ≤ j ∧ j ≤ n
    invariant "count_correct" count = (lst.take j).count elem
    invariant "found_preserved" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
    invariant "i_preserved" 0 ≤ i ∧ i < n
    invariant "elem_in_list" i < lst.length → elem = lst[i]!
    invariant "not_found_checked_inner" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
    do
      if lst[j]! = elem then
        count := count + 1
      j := j + 1

    if count > threshold then
      found := true
      candidate := elem

    i := i + 1

  if found then
    return candidate
  else
    return -1
```

```
attribute [grind] List.take_succ_eq_append_getElem


prove_correct findMajorityElement by
  loom_vc_gen <;>


  loom_solver
```



findMajorityElement

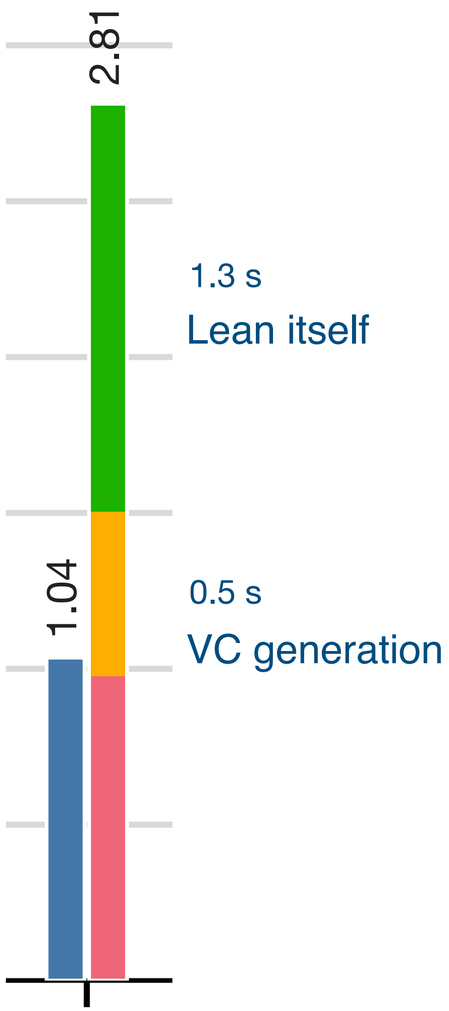
1.3 s

Lean itself

```
import CaseStudies.Velvet.Std

set_option loom.semantics.termination "partial"
set_option loom.semantics.choice "demonic"

...

method findMajorityElement (lst : List Int)
  return (result : Int)
  ensures lst.hasMajorityElement → (result ∈ lst ∧ lst.isMajorityElement result)
  ensures ¬lst.hasMajorityElement → result = -1
  do
  let n := lst.length
  let threshold := n / 2
  let mut i := 0
  let mut found := false
  let mut candidate : Int := -1

  while i < n
  invariant "i_bounds" 0 ≤ i ∧ i ≤ n
  invariant "found_implies_majority" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
  invariant "not_found_checked" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
  do
    let elem := lst[i]!
    let mut count := 0
    let mut j := 0

    while j < n
    invariant "j_bounds" 0 ≤ j ∧ j ≤ n
    invariant "count_correct" count = (lst.take j).count elem
    invariant "found_preserved" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
    invariant "i_preserved" 0 ≤ i ∧ i < n
    invariant "elem_in_list" i < lst.length → elem = lst[i]!
    invariant "not_found_checked_inner" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
    do
      if lst[j]! = elem then
        count := count + 1
      j := j + 1

    if count > threshold then
      found := true
      candidate := elem

    i := i + 1

  if found then
    return candidate
  else
    return -1
```

attribute [grind] List.take_succ_eq_append_getElem

```
prove_correct findMajorityElement by
  loom_vc_gen <;>


  loom_solver
```



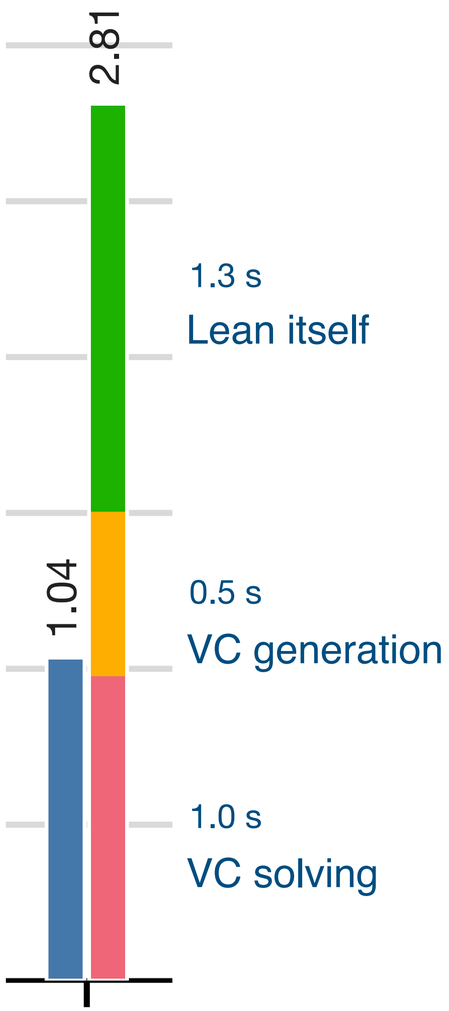
findMajorityElement

```
import CaseStudies.Velvet.Std

set_option loom.semantics.termination "partial"
set_option loom.semantics.choice "demonic"

...

method findMajorityElement (lst : List Int)
  return (result : Int)
  ensures lst.hasMajorityElement → (result ∈ lst ∧ lst.isMajorityElement result)
  ensures ¬lst.hasMajorityElement → result = -1
  do
  let n := lst.length
  let threshold := n / 2
  let mut i := 0
  let mut found := false
  let mut candidate : Int := -1

  while i < n
  invariant "i_bounds" 0 ≤ i ∧ i ≤ n
  invariant "found_implies_majority" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
  invariant "not_found_checked" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
  do
    let elem := lst[i]!
    let mut count := 0
    let mut j := 0

    while j < n
    invariant "j_bounds" 0 ≤ j ∧ j ≤ n
    invariant "count_correct" count = (lst.take j).count elem
    invariant "found_preserved" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
    invariant "i_preserved" 0 ≤ i ∧ i < n
    invariant "elem_in_list" i < lst.length → elem = lst[i]!
    invariant "not_found_checked_inner" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
    do
      if lst[j]! = elem then
        count := count + 1
      j := j + 1

    if count > threshold then
      found := true
      candidate := elem

    i := i + 1

  if found then
    return candidate
  else
    return -1
```

attribute [grind] List.take_succ_eq_append_getElem

```
prove_correct findMajorityElement by
  loom_vc_gen <;>
```

```
  loom_solver
```



findMajorityElement

```
import CaseStudies.Velvet.Std

set_option loom.semantics.termination "partial"
set_option loom.semantics.choice "demonic"

...

method findMajorityElement (lst : List Int)
  return (result : Int)
  ensures lst.hasMajorityElement → (result ∈ lst ∧ lst.isMajorityElement result)
  ensures ¬lst.hasMajorityElement → result = -1
  do
  let n := lst.length
  let threshold := n / 2
  let mut i := 0
  let mut found := false
  let mut candidate : Int := -1

  while i < n
  invariant "i_bounds" 0 ≤ i ∧ i ≤ n
  invariant "found_implies_majority" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
  invariant "not_found_checked" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
  do
    let elem := lst[i]!
    let mut count := 0
    let mut j := 0

    while j < n
    invariant "j_bounds" 0 ≤ j ∧ j ≤ n
    invariant "count_correct" count = (lst.take j).count elem
    invariant "found_preserved" found = true → (candidate ∈ lst ∧ lst.isMajorityElement candidate)
    invariant "i_preserved" 0 ≤ i ∧ i < n
    invariant "elem_in_list" i < lst.length → elem = lst[i]!
    invariant "not_found_checked_inner" found = false → (∀ k : Nat, k < i → ¬lst.isMajorityElement lst[k]!)
    do
      if lst[j]! = elem then
        count := count + 1
      j := j + 1

    if count > threshold then
      found := true
      candidate := elem

    i := i + 1

  if found then
    return candidate
  else
    return -1
```

```
attribute [grind] List.take_succ_eq_append_getElem
```

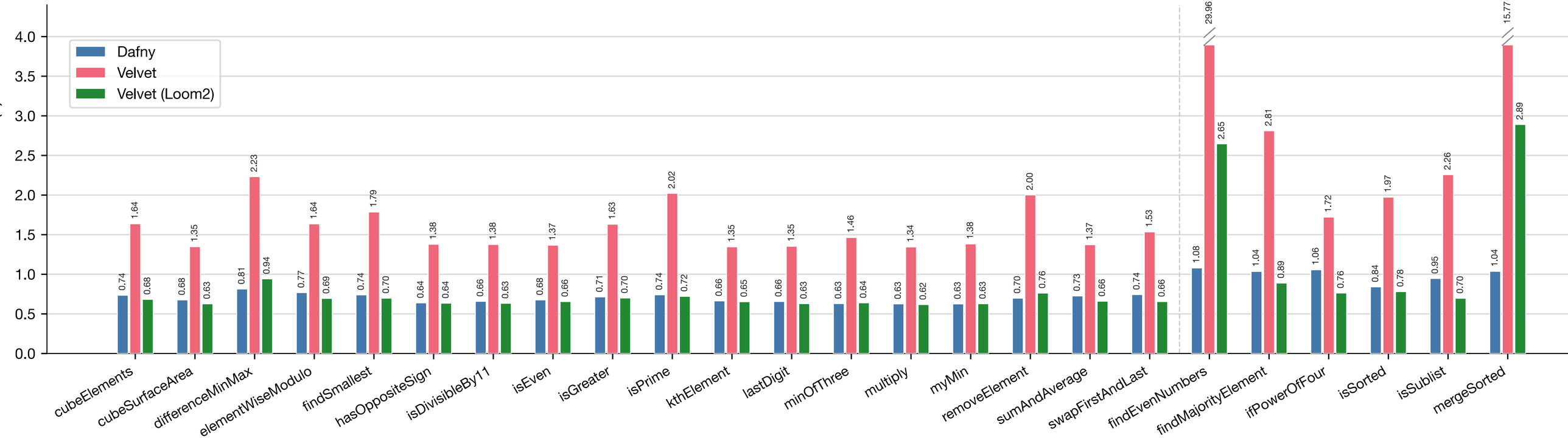
```
prove_correct findMajorityElement by
  loom_vc_gen <;>
```

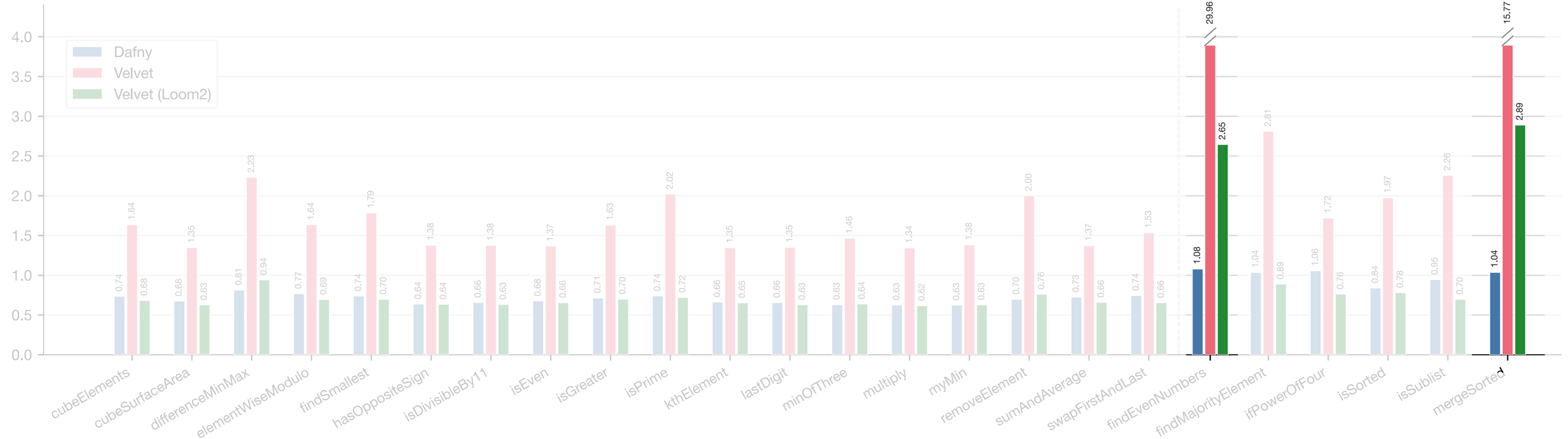
```
loom_solver
```

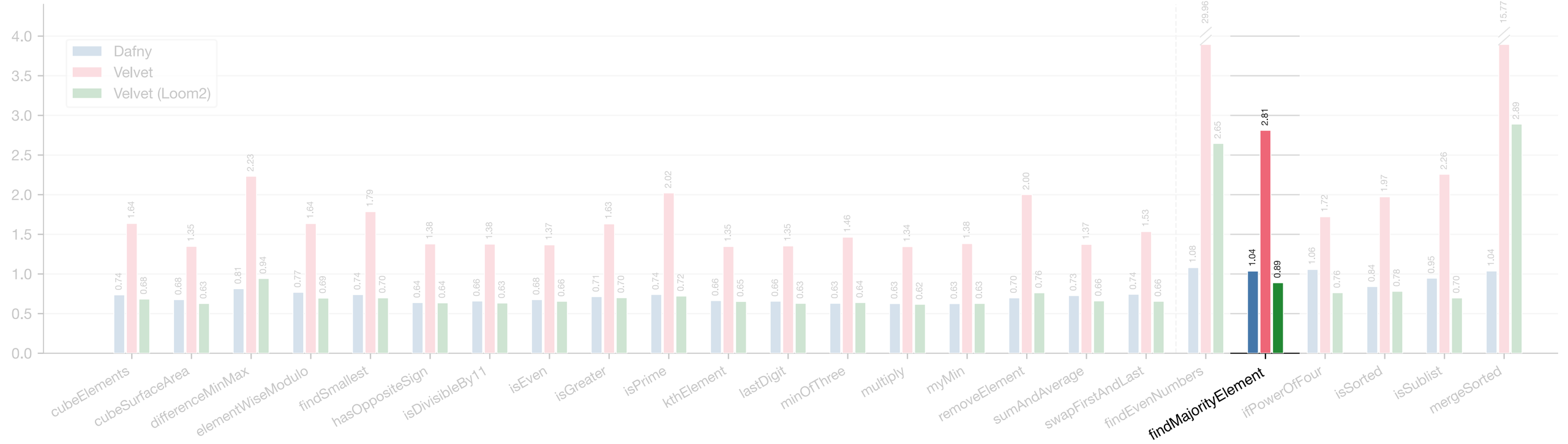
Velvet 2.0 Announce

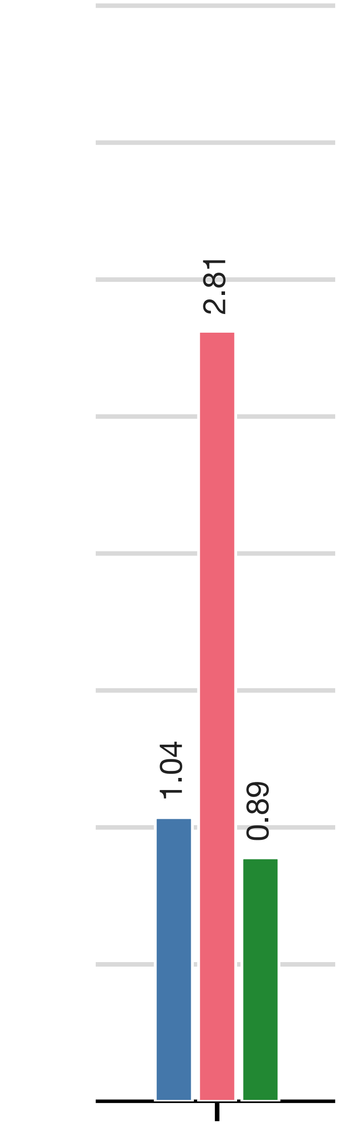
Velvet 2.0

- Lean got much faster itself!
  - New complier, new do-notation…
- SymM framework for efficient metaprogramming
  - Canonising terms, hash-consing, cashing…
- Incremental grind and simp context
  - grind and simp do not have to re-process context every time for each VCs from scratch


Dafny
Velvet
Velvet (Loom2)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.74 1.64 0.68
0.68 1.35 0.63
0.81 2.23 0.94
0.77 1.64 0.69
0.74 1.79 0.70
0.64 1.38 0.64
0.66 1.38 0.63
0.68 1.37 0.66
0.71 1.63 0.70
0.74 2.02 0.72
0.66 1.35 0.65
0.66 1.35 0.63
0.63 1.46 0.64
0.63 1.34 0.62
0.63 1.38 0.63
0.70 2.00 0.76
0.73 1.37 0.66
0.74 1.53 0.66
1.08 29.96 2.65
1.04 2.81 0.89
1.06 1.72 0.76
0.84 1.97 0.78
0.95 2.26 0.70
1.04 15.77 2.89
cubeElements
cubeSurfaceArea
differenceMinMax
elementWiseModulo
findSmallest
hasOppositeSign
isDivisibleBy11
isEven
isGreater
isPrime
kthElement
lastDigit
minOfThree
multiply
myMin
removeElement
sumAndAverage
swapFirstAndLast
findEvenNumbers
findMajorityElement
ifPowerOfFour
isSorted
isSublist
mergeSorted


Dafny
Velvet
Velvet (Loom2)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.74 1.64 0.68
0.68 1.35 0.63
0.81 2.23 0.94
0.77 1.64 0.69
0.74 1.79 0.70
0.64 1.38 0.64
0.66 1.38 0.63
0.68 1.37 0.66
0.71 1.63 0.70
0.74 2.02 0.72
0.66 1.35 0.65
0.66 1.35 0.63
0.63 1.46 0.64
0.63 1.34 0.62
0.63 1.38 0.63
0.70 2.00 0.76
0.73 1.37 0.66
0.74 1.53 0.66
1.08 29.96 2.65
1.04 2.81 0.89
1.06 1.72 0.76
0.84 1.97 0.78
0.95 2.26 0.70
1.04 15.77 2.89
cubeElements
cubeSurfaceArea
differenceMinMax
elementWiseModulo
findSmallest
hasOppositeSign
isDivisibleBy11
isEven
isGreater
isPrime
kthElement
lastDigit
minOfThree
multiply
myMin
removeElement
sumAndAverage
swapFirstAndLast
findEvenNumbers
findMajorityElement
ifPowerOfFour
isSorted
isSublist
mergeSorted


Dafny
Velvet
Velvet (Loom2)
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0.74
1.64
0.68
0.68
1.35
0.63
0.81
2.23
0.94
0.77
1.64
0.69
0.74
1.79
0.70
0.64
1.38
0.64
0.66
1.38
0.63
0.68
1.37
0.66
0.71
1.63
0.70
0.74
2.02
0.72
0.66
1.35
0.65
0.66
1.35
0.63
0.63
1.46
0.64
0.63
1.34
0.62
0.63
1.38
0.63
0.70
2.00
0.76
0.73
1.37
0.66
0.74
1.53
0.66
1.08
29.96
2.65
1.04
2.81
0.89
1.06
1.72
0.76
0.84
1.97
0.78
0.95
2.26
0.70
1.04
15.77
2.89
cubeElements
cubeSurfaceArea
differenceMinMax
elementWiseModulo
findSmallest
hasOppositeSign
isDivisibleBy11
isEven
isGreater
isPrime
kthElement
lastDigit
minOfThree
multiply
myMin
removeElement
sumAndAverage
swapFirstAndLast
findEvenNumbers
findMajorityElement
ifPowerOfFour
isSorted
isSublist
mergeSorted



findMajorityElement

Lean itself 1.3 s ~2x 0.60s

VC generation 0.5 s ~6x 0.08 s

VC solving 1.0 s ~6x 0.17 s

To Take Away

- **Velvet** is a *multi-modal* program verifier written in Lean
- Velvet is *foundational*: each Velvet proof is checked with Lean kernel
- Velvet is *fast*: its performance is comparable to Dafny
- Velvet connects AI4Math progress to program verification

github.com/verse-lab/velvet

Velvet: A Foundational Multi-Modal Verifier for Imperative Programs in Lean, CAV'26

Foundational Multi-Modal Program Verifiers, POPL'26

Certified Program Synthesis with a Multi-Modal Verifier, ASE'26

Thanks!