Carnegie Mellon University

Gradual Verification: Assuring Software Incrementally

Jonathan Aldrich

October 2023

Work done with Jenna (Wise) DiVincenzo, Ian McCormack, Mona Zhang, Jacob Gorenburg, Hemant Gouni, Conrad Zimmerman, Joshua Sunshine, and Éric Tanter. Sponsored by the US National Science Foundation.

S3D Software and Societal Systems Department

Naïve Verification Attempt: Dynamic Verification

```
int findMax(Node l)
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL) {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }
  return m;
}
```

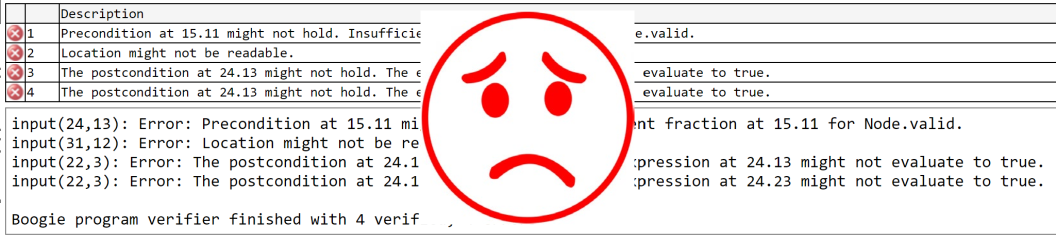
Naïve Verification Attempt: Dynamic Verification

```
int findMax(Node l)
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL) {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }
  assert max(m,l) && contains(m,l);
  return m;
}
```

Challenges:
- Would like to ensure spec for all executions
- Cost of dynamic checking may be significant

Naïve Verification Attempt: Static Verification

```
int findMax(Node l)
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l
  while(curr !=
    if(curr->va
      m = curr-
    }
    curr = curr->next;
  }
  return m;
}
```



S3D

Naïve Verification Attempt: Static Verification

```
int findMax(Node l)
  requires l != NULL
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
    FOLDS/UNFOLDS
  while(curr != NULL)  LOOP INVARIANTS  {
    if(curr->val > m) { m = curr->val; }
    curr = curr->next;
      FOLDS/UNFOLDS
         LEMMAS
  }
    FOLDS/UNFOLDS
  return m;
}
```

S3D

Gradual Verification to the Rescue

```
int findMax(Node l)
  requires  ?
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL)  ?  {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }

  return m;
}
```

Gradual Verification to the Rescue

```
int findMax(Node l)
  requires ? && l != NULL
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL) ? {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }

  return m;
}
```

Gradual Verification to the Rescue

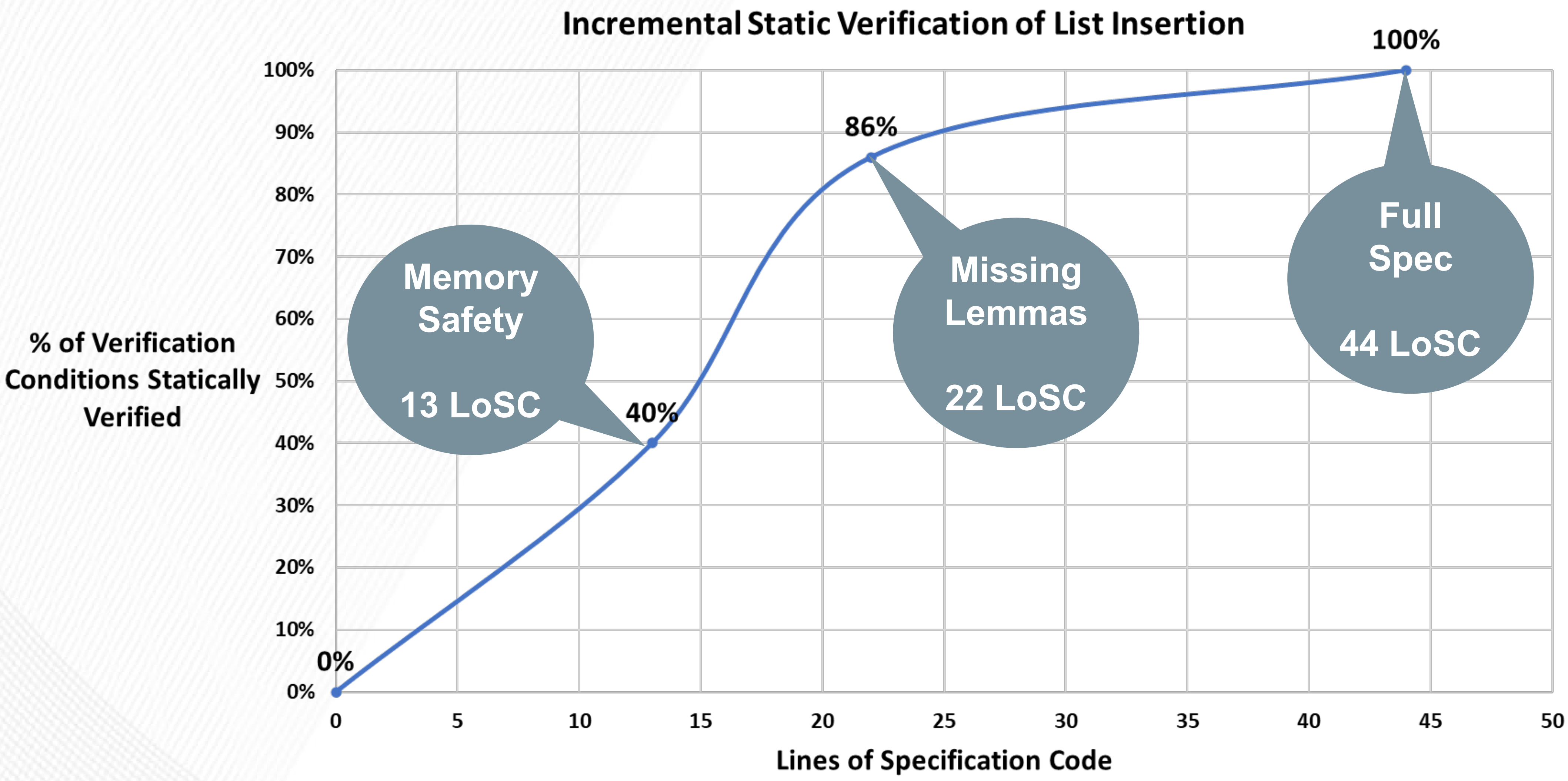
```
int findMax(Node l)
  requires ? && l != NULL
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL)  ? && LOOP INVARIANTS  {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }

  return m;
}
```

Gradual Verification to the Rescue

```
int findMax(Node l)
  requires l != NULL
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  FOLDS/UNFOLDS
  while(curr != NULL)  LOOP INVARIANTS  {
    if(curr->val > m) { m = curr->val; }
    curr = curr->next;
    FOLDS/UNFOLDS
    LEMMAS
  }
  FOLDS/UNFOLDS
  return m;
}
```

Stop Specification Anytime with Gradual Verification



Summary: The Problem

- Dynamic verification is low-cost to the programmer, but:
  - No feedback at compile time (and no static guarantees)
  - Can slow the program—a lot!

- Static verification pays off only after a ton of work
  - Static verification can be 10x as costly as writing the program (sel4, CompCert)
  - Requires an inductively complete specification
    - Many “false positive” warnings when spec is incomplete
    - No feedback on incorrect specs until there’s a static inconsistency

- What we need:
  - Incremental payoff for incremental specification work
    - Ability to focus on most important properties of most important components
  - Early feedback on mistakes – both compile time checking & running incomplete specs
  - Properties – soundness, conservative extension, gradual guarantee, pay as you go

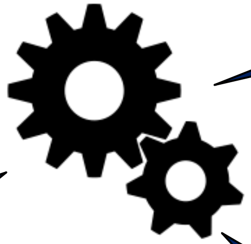
Properties of Gradual Verification

All specification violations are caught either statically or at run time

Soundness

If a specification doesn't have ?, and it verifies, the code is correct

Conservative extension



Run time checking cost
- increases as you add specs
- decreases as specs become statically verifiable

Pay as you go

If your specs are a subset of a correct spec, you won't get any errors

Gradual Guarantee
[Siek et al. 2015]

First 3 proved for initial models of gradual verification in [Bader et al. '18], [Wise et al. '20]

How does gradual verification work?

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

```
predicate acyclic(Node root) =
    (root == NULL) ?
     true
    :
     acc(root->val) * acc(root->next)
      * acyclic(root->next)
```

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

```
predicate acyclic(Node root) =
    (root == NULL) ?
      true
     :
      acc(root->val) * acc(root->next)
       * acyclic(root->next)
```

Accessibility Predicate - permission to access a heap location

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

```
predicate acyclic(Node root) =
    (root == NULL) ?
      true
     :
      acc(root->val) * acc(root->next)
      * acyclic(root->next)
```

Separating Conjunction - predicates refer to different heap locations

S3D

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

```
predicate acyclic(Node root) =
    (root == NULL) ?
      true
     :
      acc(root->val) * acc(root->next)
       * acyclic(root->next)
```

Preliminaries

```
{ acyclic(l) }
l = new Node(3,l);
assert acyclic(l);
```

```
predicate acyclic(Node root) =
    (root == NULL) ?
      true
     :
      acc(root->val) * acc(root->next)
       * acyclic(root->next)
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);



assert acyclic(l);
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }


assert acyclic(l);
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }


assert acyclic(l);
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }



assert acyclic(l);
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }



assert acyclic(l);
```

```
predicate acyclic(Node l) =
    (l == NULL) ? true :
       acc(l->val) * acc(l->next)
        * acyclic(l->next)
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }
fold acyclic(l);


assert acyclic(l);
```

Static Verification of Daisy's List Insertion Program

```
{ acyclic(l) }
l = new Node(3,l);
{ l != NULL * acc(l->val) * acc(l->next)
  * acyclic(l->next) }
fold acyclic(l);
{ l != NULL * acyclic(l) }
assert acyclic(l);
```

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);


fold acyclic(l);


assert acyclic(l);
```

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);


assert acyclic(l);
```

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);


assert acyclic(l);
```

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);


assert acyclic(l);
```

```
predicate acyclic(Node l) =
    (l == NULL) ? true :
      acc(l->val) * acc(l->next)
       * acyclic(l->next)
```


S3D

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);


assert acyclic(l);
```

```
predicate acyclic(Node l) =
    (l == NULL) ? true :
        acc(l->val) * acc(l->next)
         * acyclic(l->next)
```

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);

assert acyclic(l);
```

? optimistically provides acyclic(l->next) for the fold

Gradual Verification of Daisy's List Insertion Program

```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);
{ l != NULL * acyclic(l) }
assert acyclic(l);
```

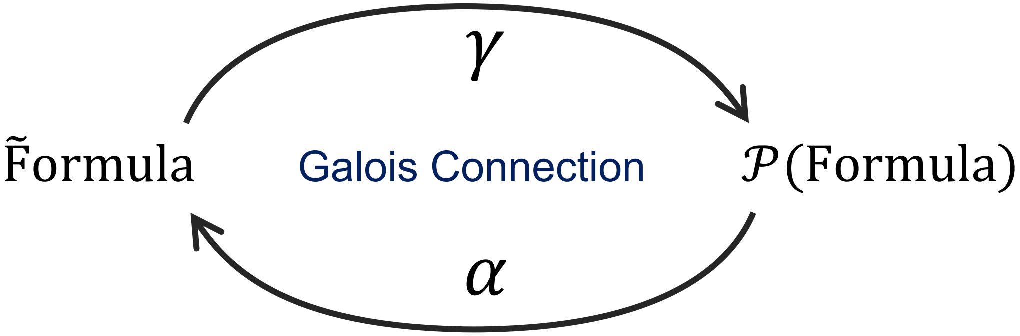
? optimistically provides acyclic(l->next) for the fold

S3D

Semantics of Gradual Formulas

What does a gradual formula mean?

$$\tilde{\phi} \;::=\; \phi \;\mid\; \phi \wedge ?$$



$$\gamma(\phi) = \{\, \phi \,\}$$

$$\gamma(\phi \;\wedge\; ?) = \{\; \text{satisfiable}\, \phi' \;\mid\; \phi' \Rightarrow \phi \;\}$$

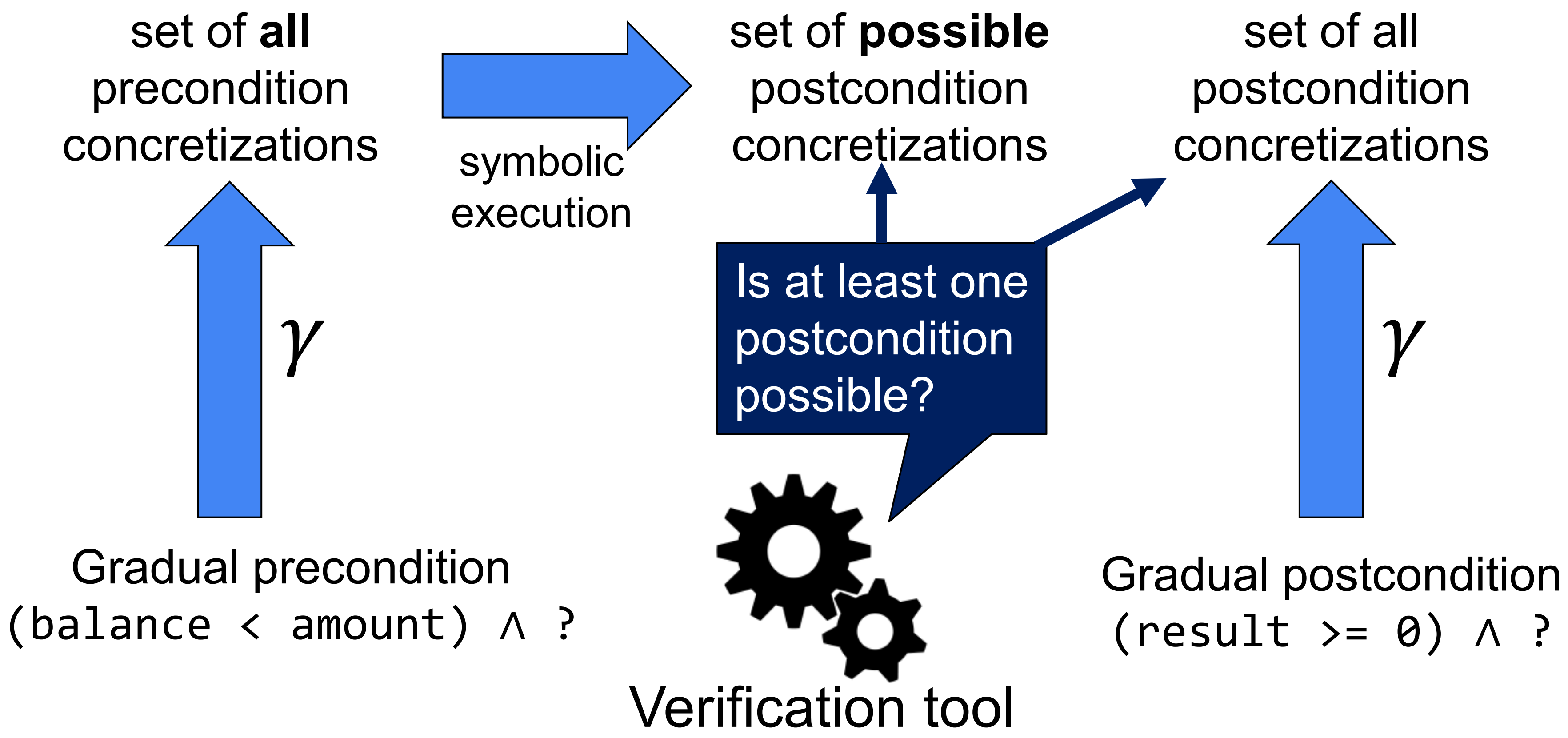
Must be **satisfiable** so we don't accept a procedure by making the precondition **false**

```
int withdraw(int balance, int amount)
    requires (balance >= amount) ∧ ?
    ensures  (result >= 0) ∧ ?
{
  return balance - amount;
}
```

```
result >= 0
result >= 1
result == balance – amount
...
```

Checking approach, conceptually

- Adapts the Abstracting Gradual Typing methodology
  [Garcia et al. 2016]



Checking approach, concretely

set of **all** precondition concretizations → (symbolic execution) → set of **possible** postcondition concretizations | set of all postcondition concretizations

Is at least one postcondition possible?

Verification tool

- In practice, no tool can deal with (possibly infinite) concretization sets
- Our approach:
  - Underapproximate what we definitely know
  - Statically overapproximate what postconditions can be satisfied by what we know in combination with ?
    - In practice: warn about contradictions
  - Use the difference to generate dynamic checks
    - “assert any conjuncts you can’t prove statically”

Ensuring all specifications are executable

- acc(x.f)
  - Keep track of what the currently executing method owns - a set of (object, field) pairs
  - Verify we own this field
  - Ensure owned state on both sides of a * does not overlap
- Disjunction: support “if cond then X else Y” *instead of* “X or Y”
  - checking X or Y is exponential in practice – must try all combinations to see if ownership works
- Quantification – not supported yet
  - Future: support some kind of finite quantification
- Recursive predicates
  - Executed as functions
  - Must terminate
    - Our approach: each recursive call must assert ownership of at least one heap cell

Example: producing dynamic checks

```
{ ? }
l := new Node(3,l);
{ ? * l != null * acc(l.val) * acc(l.next) }
fold acyclic(l);

assert acyclic(l);
```

```
predicate acyclic(l) =
  acc(l.val) * acc(l.next) *
    acyclic(l.next)
```

Dynamically Verifying Predicates

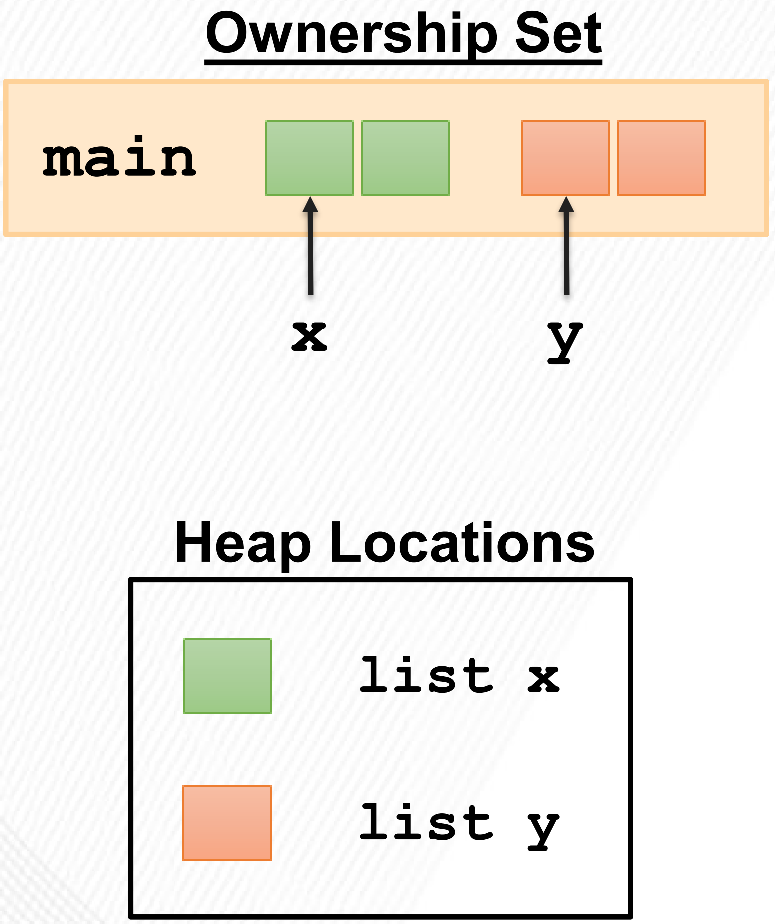
```
{ ? }
l := new Node(3,l);
{ ? * l != null * acc(l.val) * acc(l.next) }
fold acyclic(l);
{ ? * l != null * acyclic(   }
assert acyclic(l);
```

Runtime check:
acyclic(l.next)

Equi-recursive

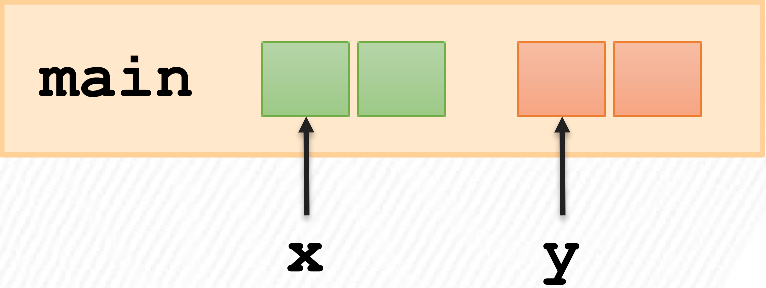
S3D

Dynamically Verifying Accessibility Predicates



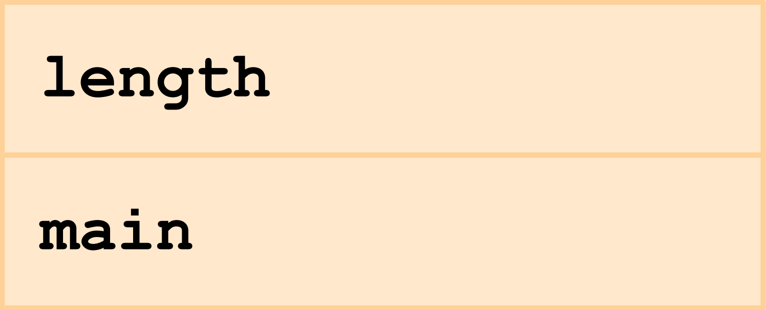
Dynamically Verifying Accessibility Predicates

Ownership Set

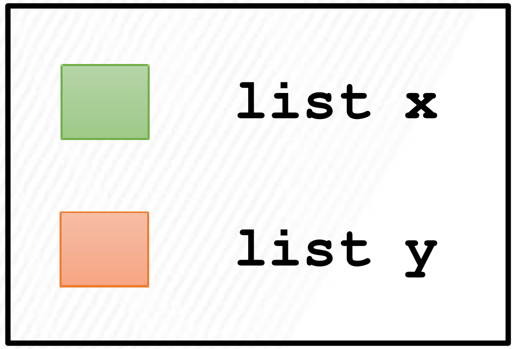



```
length(Node x)
  requires
    acyclic(x)
```

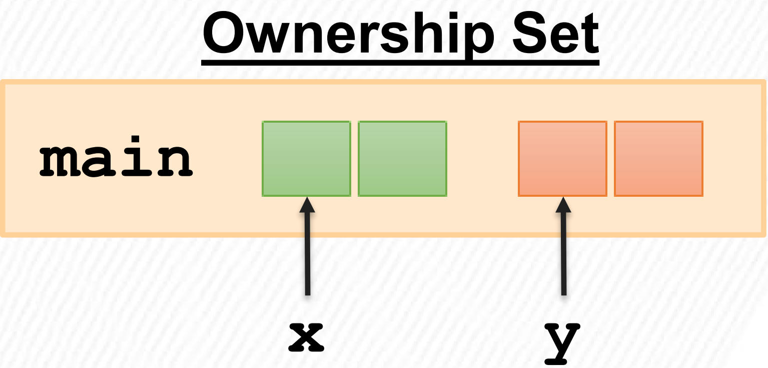
Ownership Sets



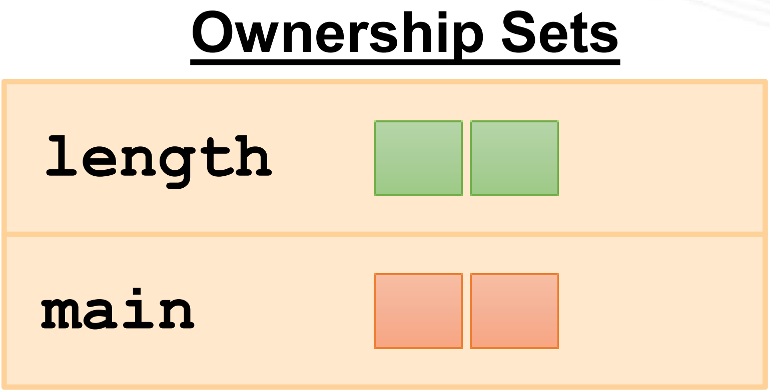
Heap Locations

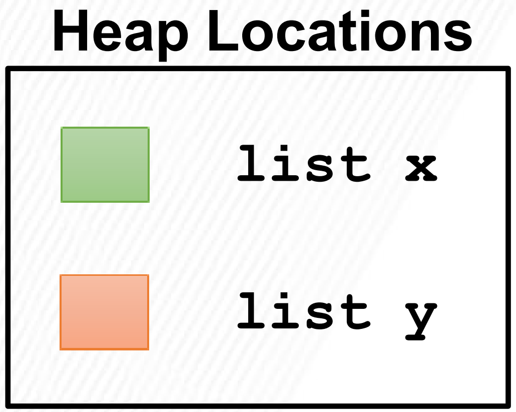


Dynamically Verifying Accessibility Predicates

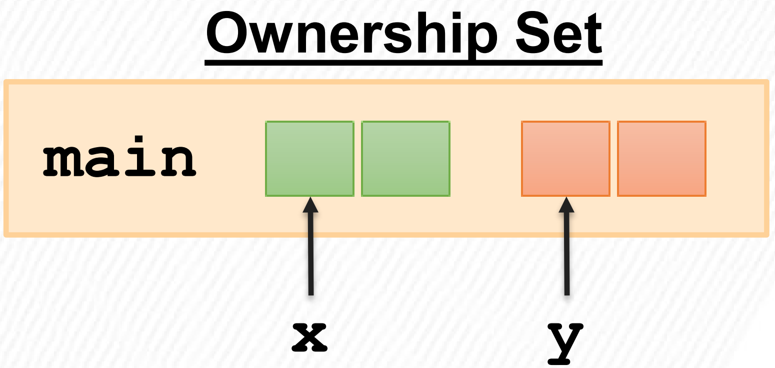



```
length(Node x)
  requires
    acyclic(x)
```

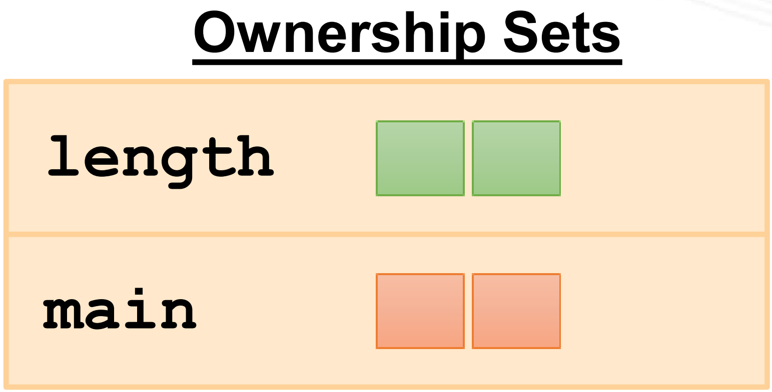


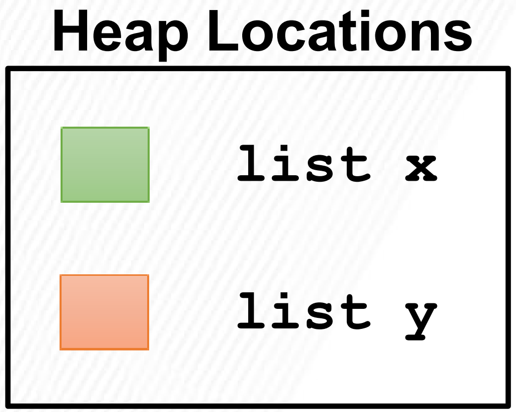


Dynamically Verifying Accessibility Predicates

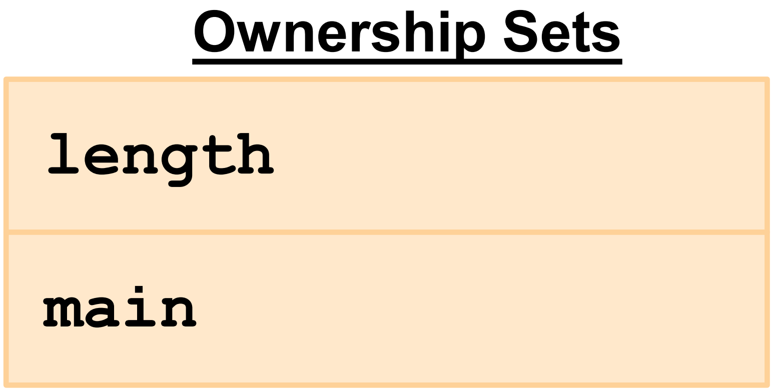



```
length(Node x)
  requires
    acyclic(x)
```



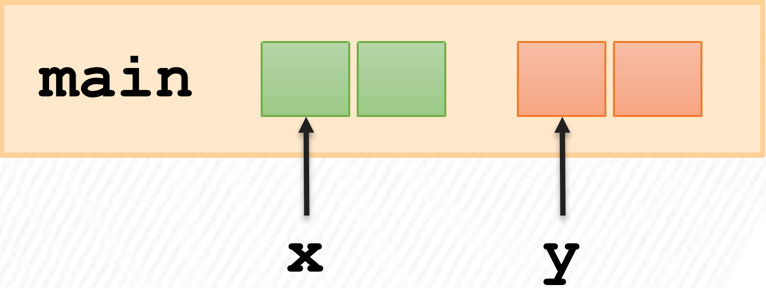



```
length(Node x)
  requires ?
```



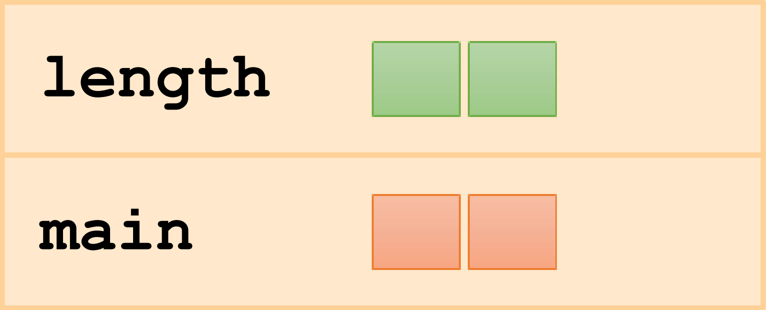
Dynamically Verifying Accessibility Predicates

Ownership Set

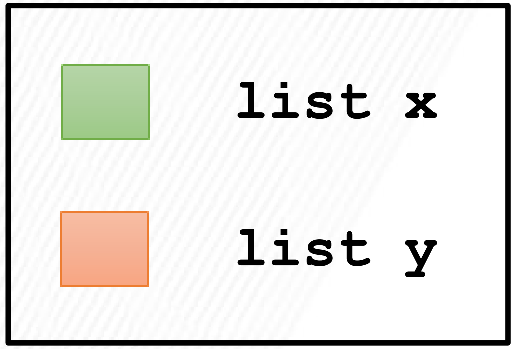



```
length(Node x)
   requires
      acyclic(x)
```

Ownership Sets

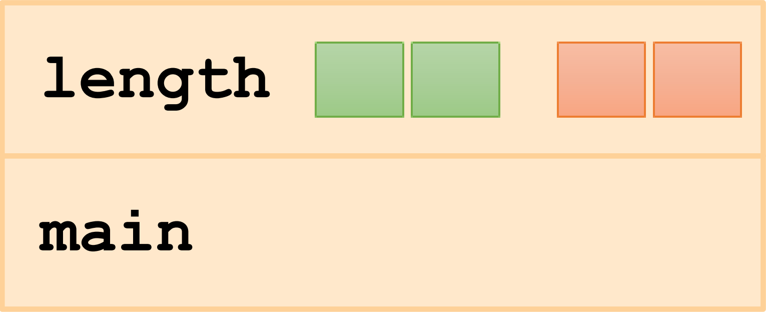


Heap Locations

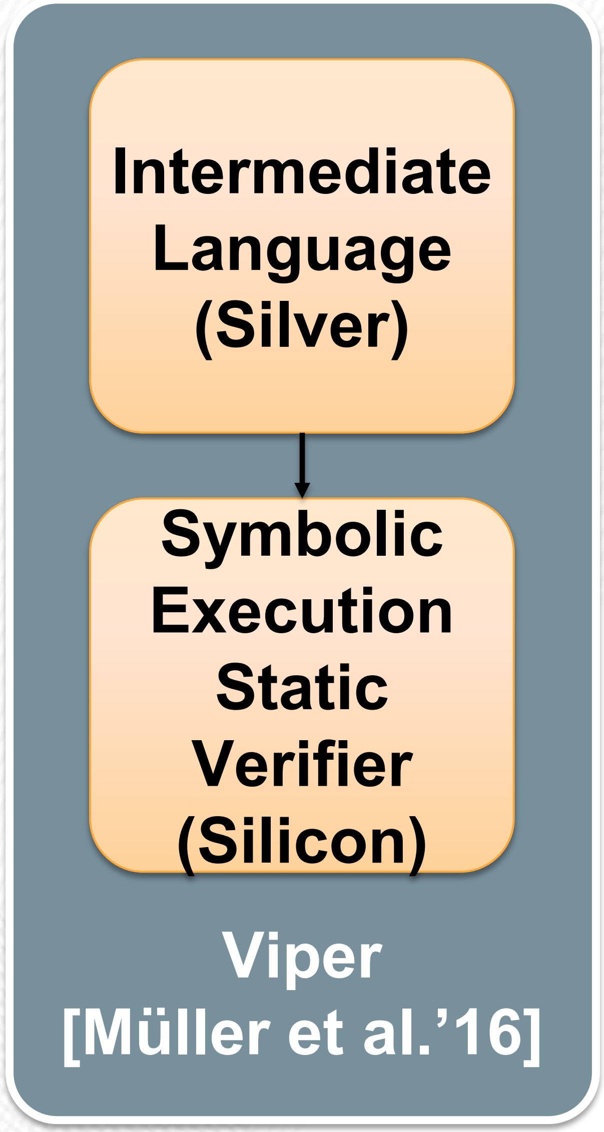



```
length(Node x)
   requires ?
```

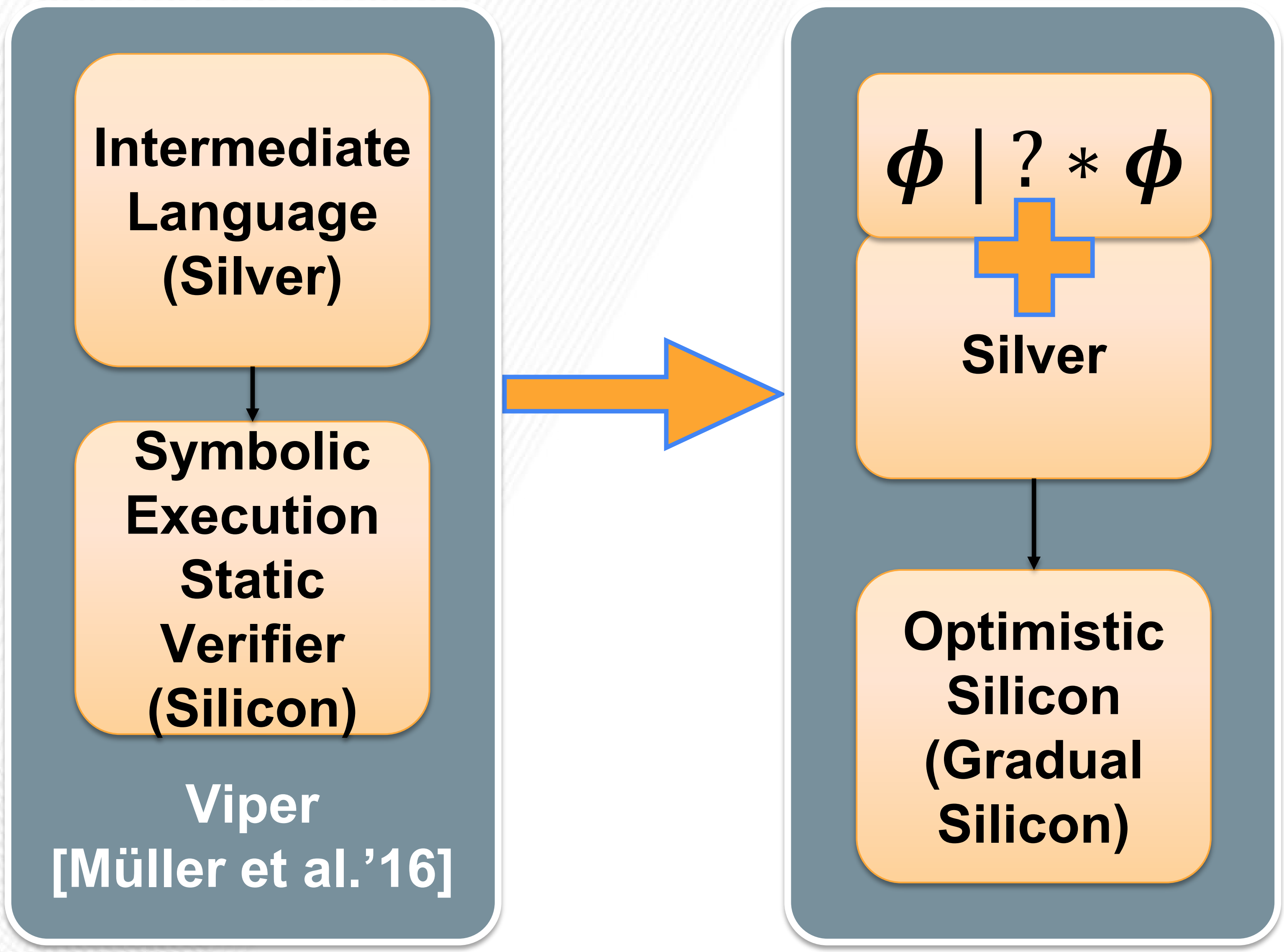
Ownership Sets



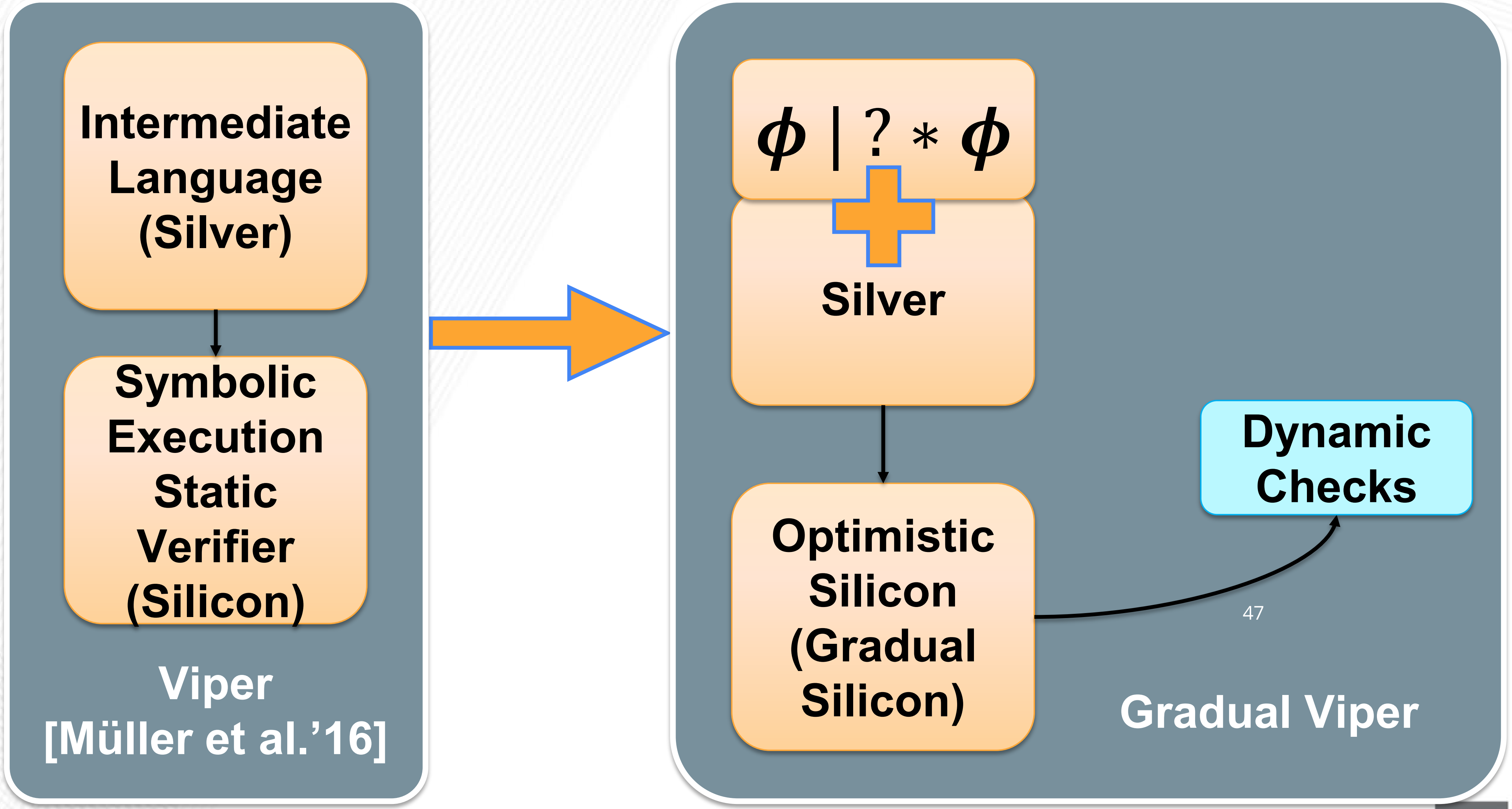
Gradual Viper: Prototype Design & Implementation



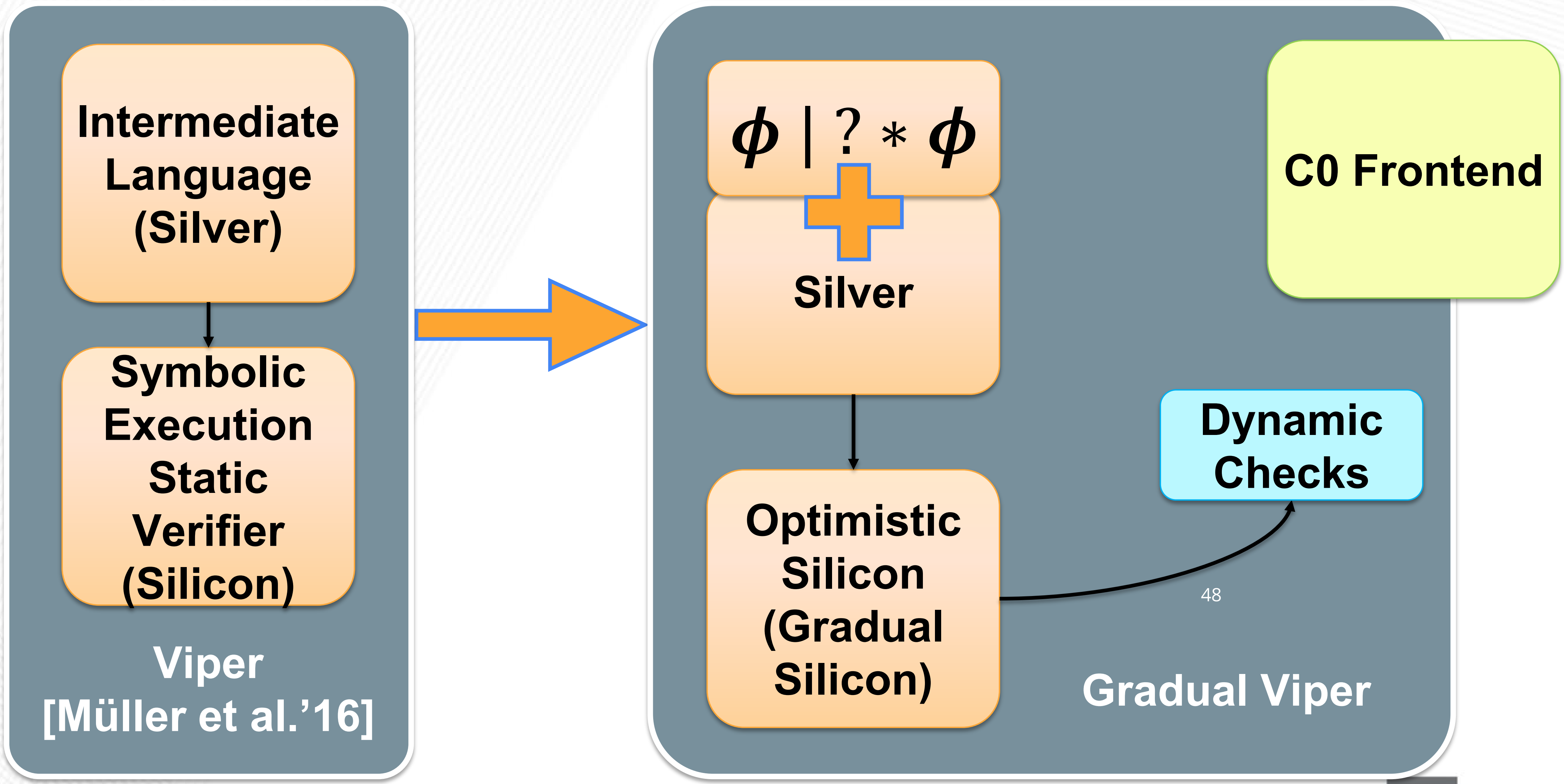
Gradual Viper: Prototype Design & Implementation



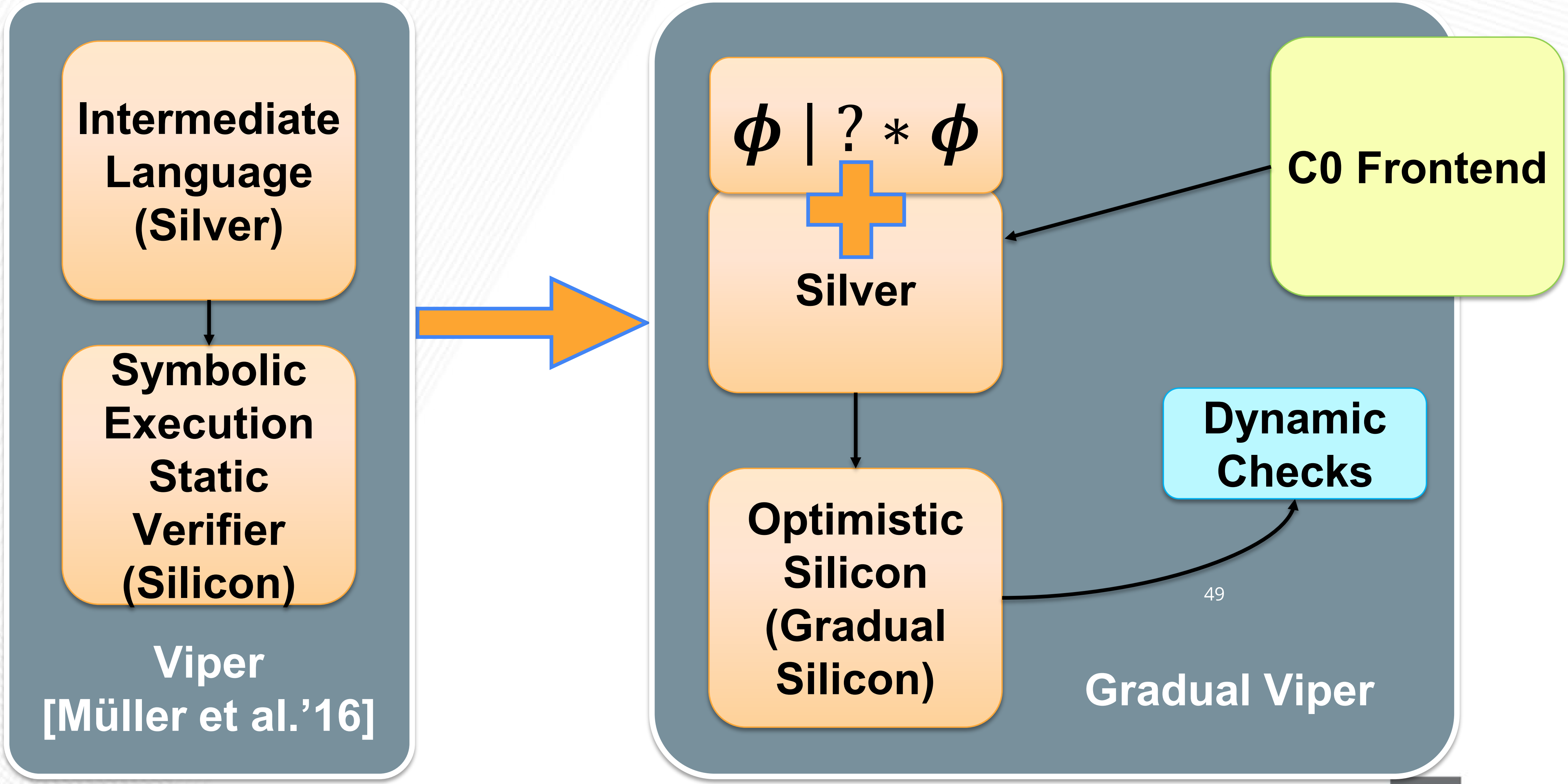
Gradual Viper: Prototype Design & Implementation



Gradual Viper: Prototype Design & Implementation

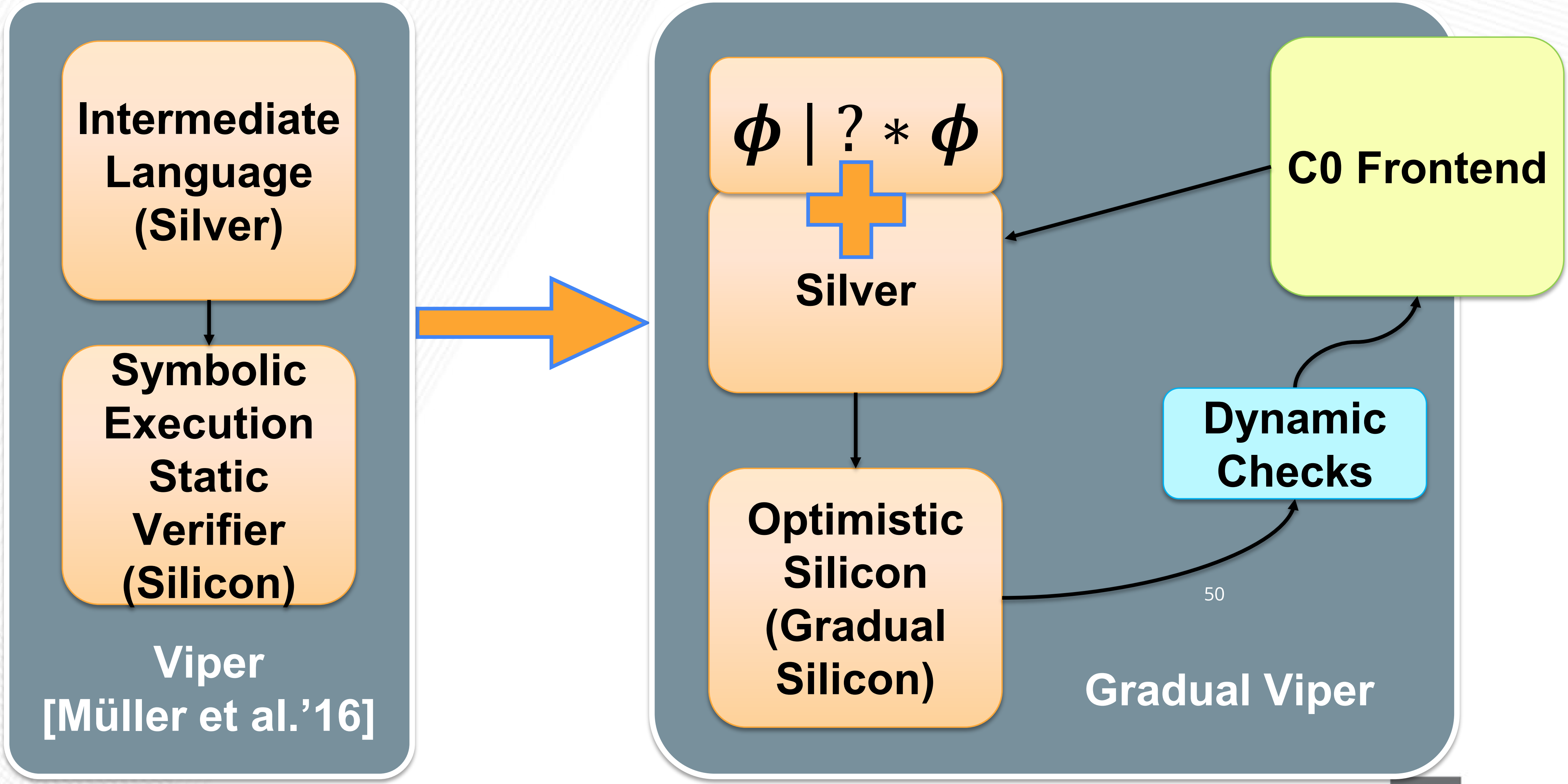


Gradual Viper: Prototype Design & Implementation



S3D

Gradual Viper: Prototype Design & Implementation



Research Questions

[RQ1] Qualitatively, is gradual verification helpful in specifying code?

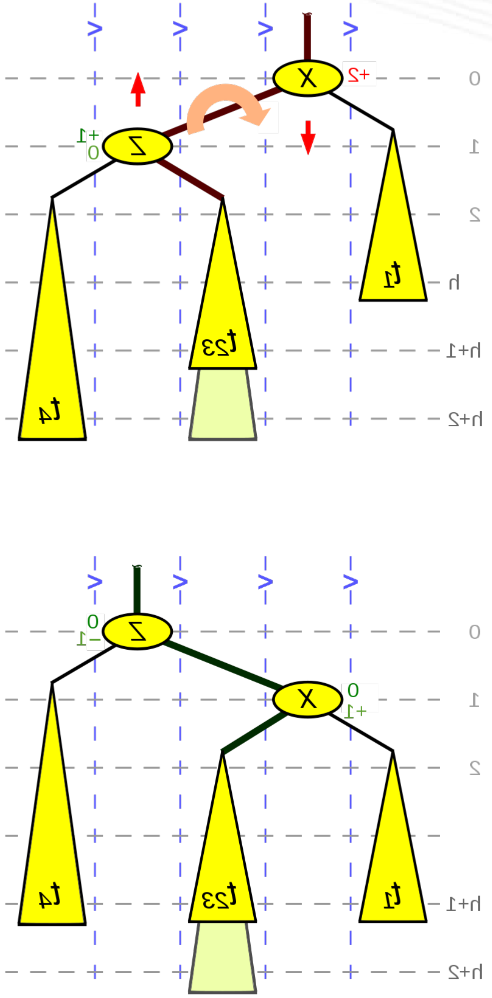
[RQ2] As specifications are made more precise, can more verification conditions be eliminated statically?

[RQ3] Does gradual verification result in less run-time overhead than a fully dynamic approach?

[RQ4] Are there types of specification constructs that significantly impact run-time performance?

[RQ1] Can Gradual Verification Help with Specifying Code?

- Case study: verifying AVL trees
- Found an implementation of AVL trees in C
- Started with ? everywhere
- Added specifications incrementally
  - “Natural” order: specify data structure invariant, then “rotate” helper functions
  - Wikipedia helpfully provides a diagram expressing the pre- and post-conditions of rotateLeft

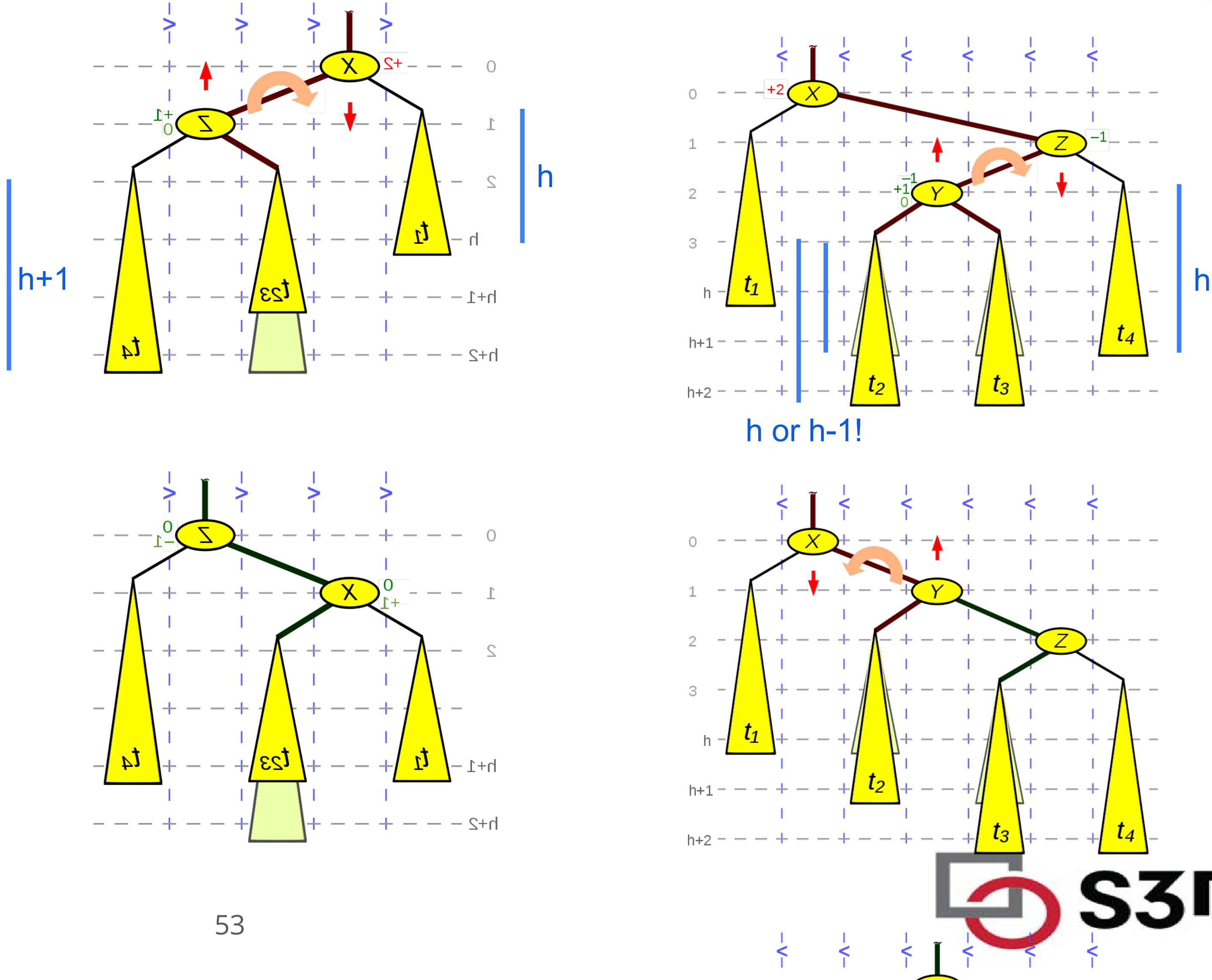


- Demo time!
  - run avlja-demo.c0
  - run -x avlja-demo.c0

Oops! rotateRight is used twice. Compare:

- Our original spec only considered the first use of rotateRight
- The second use is part of a double rotation
- A more generic precondition is required!

- Demo!
  - run -x avlja.c0



Observations

- Our initial spec wasn't general enough
  - But it was sufficient to statically verify rotateRight()
  - Notice: no annoying ("false positive") warnings because the spec is incomplete

- The ability to run the spec demonstrated an error
  - The precondition was violated on some calls to rotateRight()

- Delayed identification of the error could be costly
  - Might have verified getBalance(), rotateLeft() & much of insert() before finding the problem
  - Then, we'd have to modify these proofs after fixing the rotateRight() spec

- Old story: finding errors early is good!
- New story: running your spec can help find errors early!

[RQ2] Does making specs more precise enable discharging more VCs statically?

- “Performance Lattice” study methodology adapted from Takikawa et al., “Is Sound Gradual Typing Dead?”

- We follow a path from no specifications (all ?) to full specifications (no ?)
  - Each step adds a specification conjunct, or removes ? from a spec that is complete

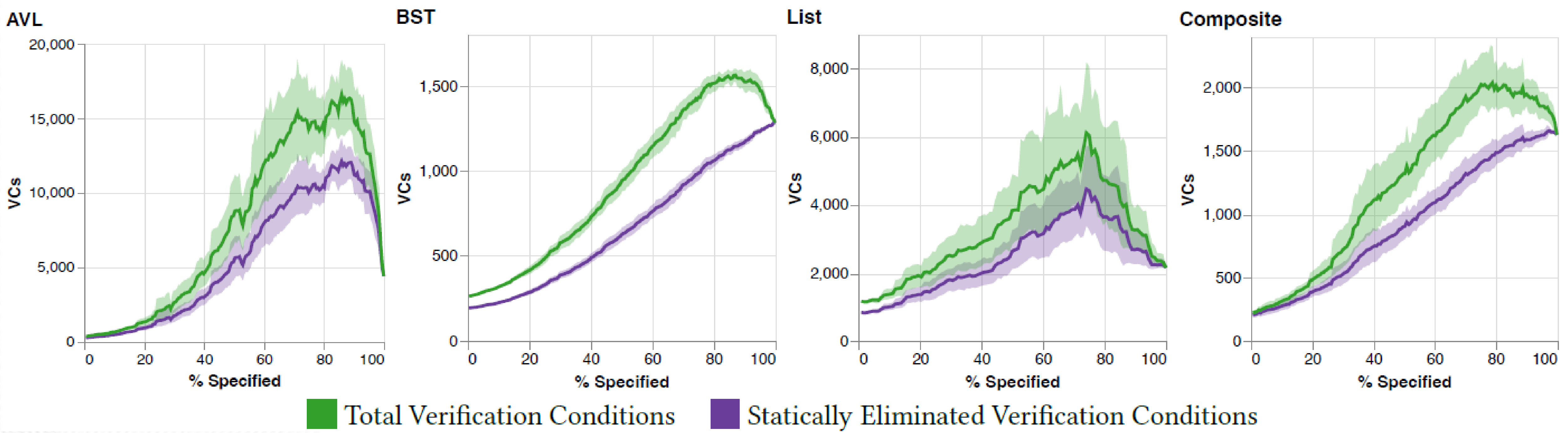
- Takikawa explored the complete lattice
  - But our lattices have ~2^{100} elements, so we sample paths

S3D

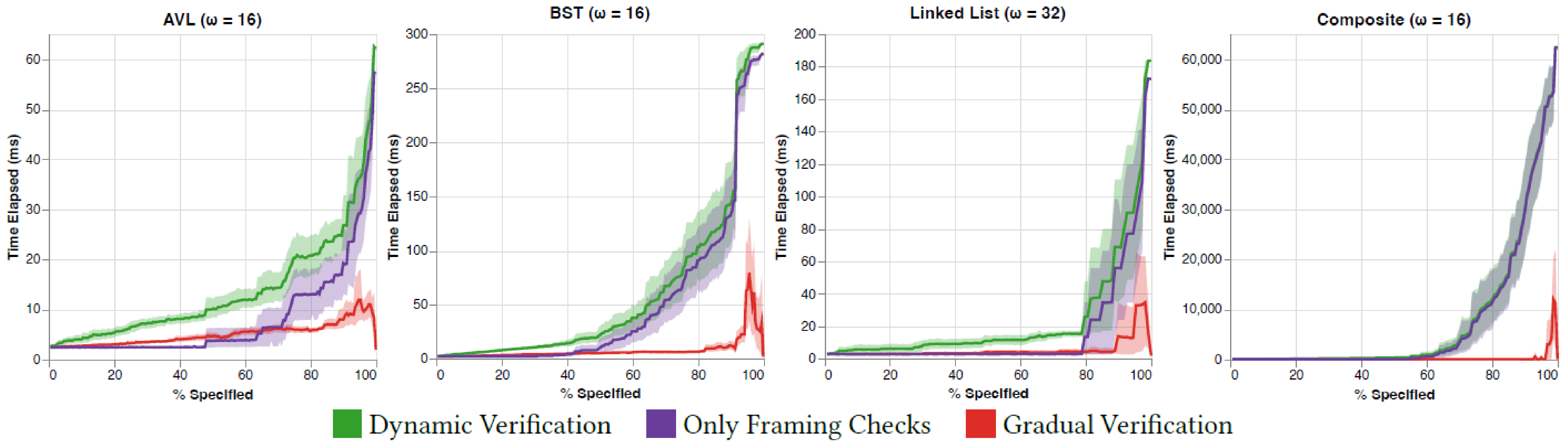
Thousands of Partial Specifications Evaluated

Benchmark	# of Sampled Partial Specifications
Linked List	1728
Binary Search Tree	3344
Composite Tree	2577
AVL Tree	3056

[RQ2] Does making specs more precise enable discharging more VCs statically?



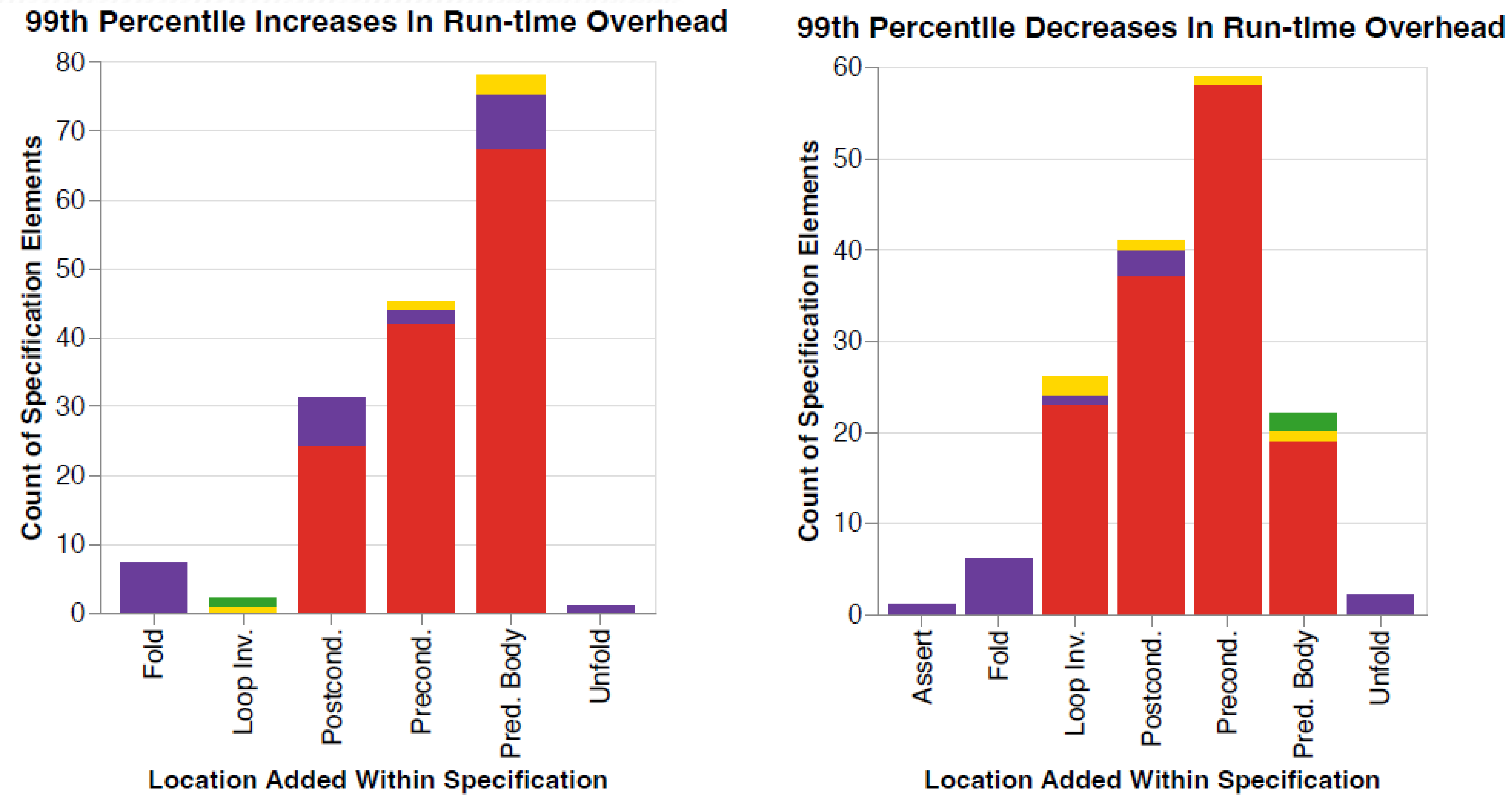
[RQ3] Does gradual verification reduce run-time overhead, compared to dynamic analysis?



Summary: Performance

- Costs are greatly reduced by gradual verification!
- Costs can still be high, though!
- Our paths are randomly chosen, but you can be smart
  - avoid high-cost dynamic checks in hot code
  - avoid transitioning between statically and dynamically-checked components in hot code when there's a substantial footprint

[RQ4] What changes cause execution time to jump?



Thanks to my Awesome Collaborators!

Jenna (Wise) DiVincenzo (CMU)

Ian McCormack (CMU)

Éric Tanter (University of Chile)

Joshua Sunshine (CMU)

Mona Zhang (Columbia University)

Jacob Gorenburg (Haverford College)

Hemant Gouni (University of Minnesota)

Conrad Zimmerman (Brown University)

Gradual Verification Helps Bring Engineering to Verification

- Makes partial / missing specs explicit with ?
- Checks specs statically where possible and dynamically where necessary
- Interesting theory
  - Soundness, conservative extension, gradual guarantee, pay as you go
  - Connection between static iso-recursive checking and dynamic equi-recursive checking
- Interesting implementation
  - Representations and algorithms for optimized run-time checking
- Lots more research to do!
  - More powerful specifications (higher-order, quantification, concurrency, …)
  - Case studies, human subjects experiments to evaluate practical value
  - Further optimization in implementation