Gradual Verification: Assuring Programs Incrementally

Dr. Jenna DiVincenzo

Assistant Professor
Purdue University

Special Thanks to my Awesome Co-authors!

Johannes Bader
(Jane Street)

Ian McCormack
(CMU)

Mona Zhang
(Columbia University)

Jacob Gorenburg
(Haverford College)

Hemant Gouni
(CMU)

Jonathan Aldrich
(CMU)

Éric Tanter
(University of Chile)

Joshua Sunshine
(CMU)

Conrad Zimmerman
(Northeastern University)

Cameron Wong
(Harvard University)

Jan-Paul Ramos-Dávila
(Cornell University)

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;
}
```

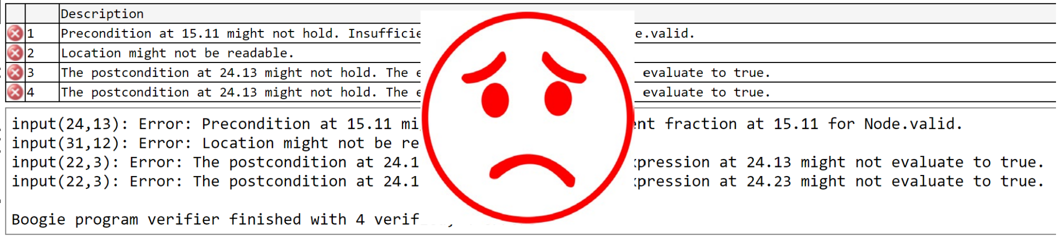
Naïve Verification Attempt: Dynamic Verification

```
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;
}
```

Dynamic Verifiers increase run-time overhead & provide assurances only for current execution path!

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;
}
```



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;
}
```

Static Verifiers have a large upfront specification cost & cannot support incremental verification!

Gradual Verification supports incremental verification
by applying
static verification where possible &
dynamic verification where necessary.

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

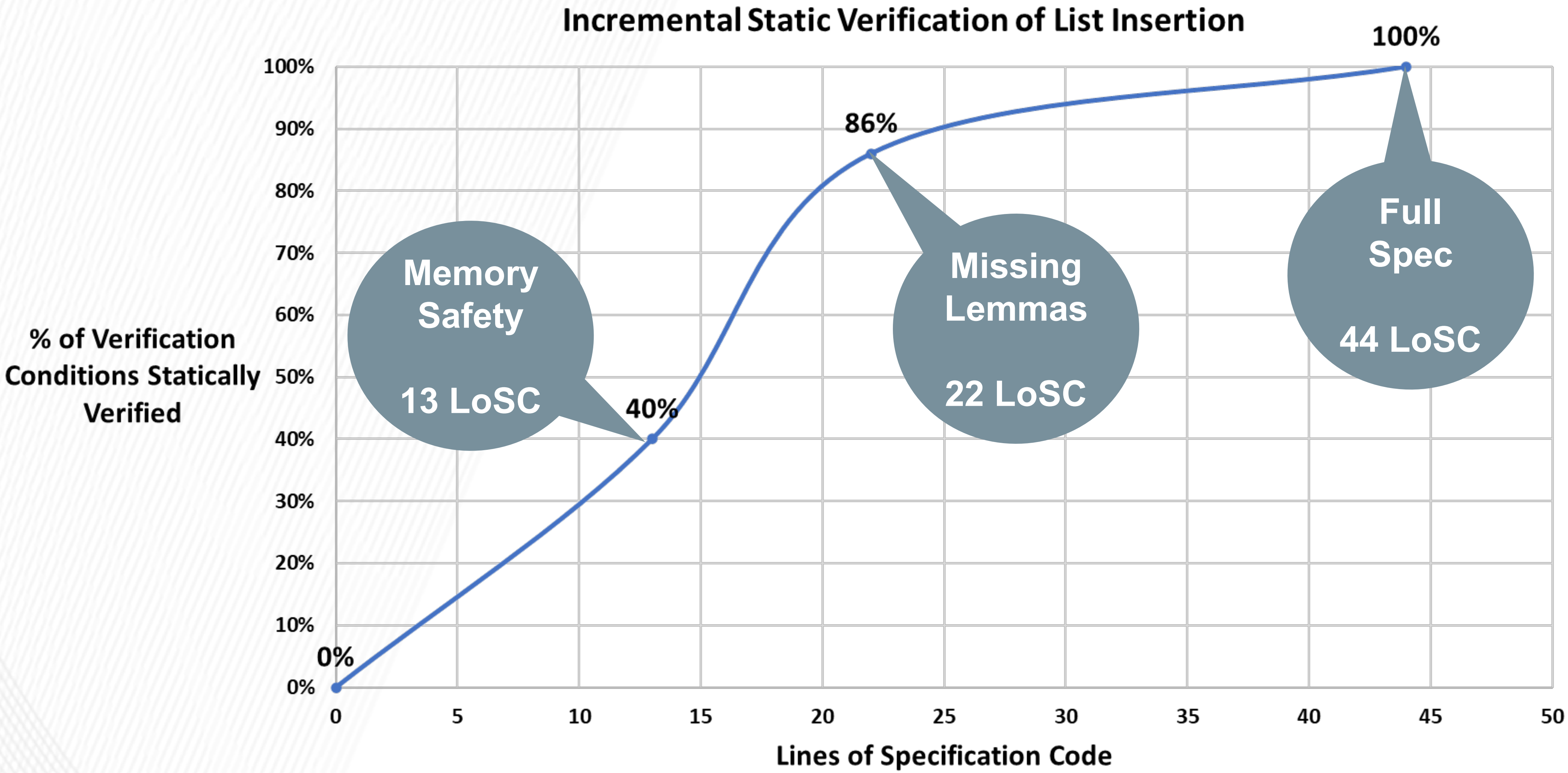
```
int findMax(Node l)
  requires ? && l != NULL
  ensures max(result,l) && contains(result,l)
{
  int m = l->val;
  Node curr = l->next;
  while(curr != NULL)   ? && LOOP INVARIANTS   {
    if(curr->val > m) {
      m = curr->val;
    }
    curr = curr->next;
  }

  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;
}
```

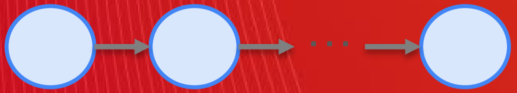
Stop Specification Anytime with Gradual Verification



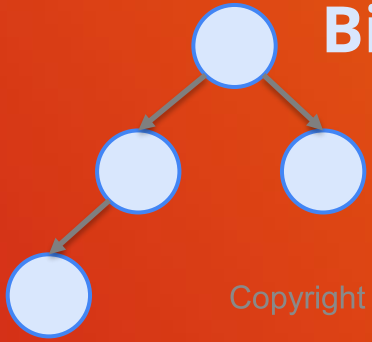
Current State-of-the-Art in Gradual Verification

Thesis:
Gradual Verification of
Recursive Heap Data Structures

Sorted List

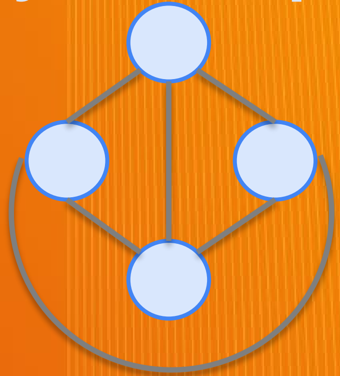


FindMax



Binary Search Tree

Cyclic Graph



Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

Theory of ? & Run-time Checking

Wise et al. 2020
Gradual Verification of Recursive Heap Data Structures

Tool Implementation: Gradual C0

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Zimmerman et al. 2024
Sound Gradual Verification with Symbolic Execution

Run-time Performance Study

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Gradual C0 Case Study: Gradually Verifying a C parser

Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

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

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)
```

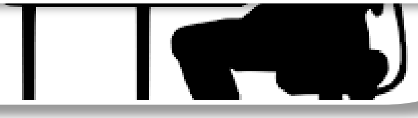
Static Verification of Daisy's List Insertion Program

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

assert acyclic(l)
```

Predicates treated *iso-recursively* in static verifiers

```
     te acyclic(Node l) =
       == NULL) ? true :
       cc(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)
```

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

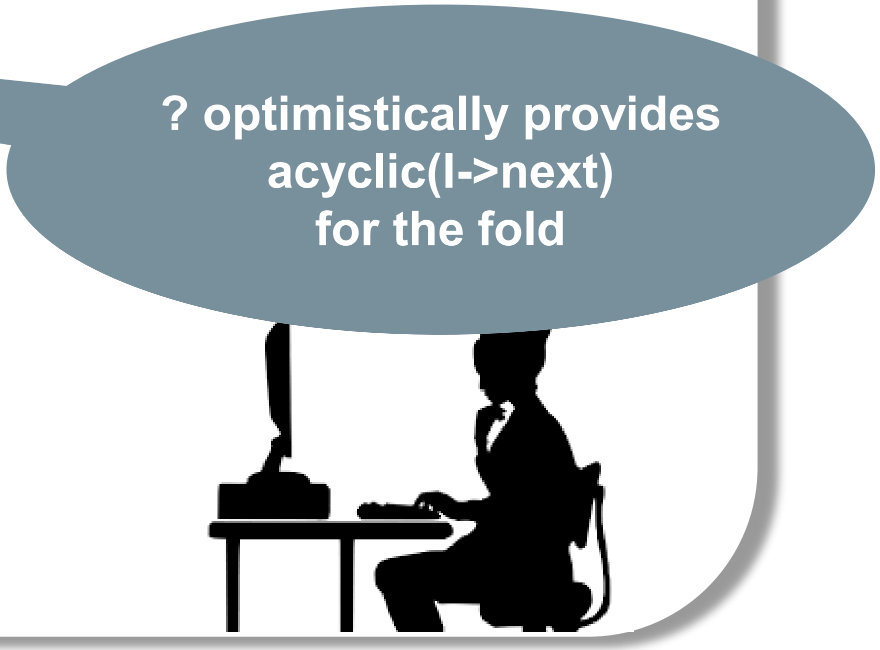
```
{ ? }
l = new Node(3,l);
{ ? * l != NULL * acc(l->val) * acc(l->next) }
fold acyclic(l);


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);
```



Gradual Verification of Daisy’s List Insertion Program

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

? can represent predicates & accessibility predicates

? 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);
```

Run-time check for acyclic(l->next) here

Gradual Verification of Daisy's List Insertion Program

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

Predicates checked ***equi-recursively*** at run time

Run-time check for acyclic(l->next) here

Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

Theory of ? & Run-time Checking


Gradual Verification of Recursive Heap Data Structures

CAMERON WONG, Jane Street, USA

JONATHAN ALDRICH, Carnegie Mellon University, USA

Additional Key Words and Phrases: gradual verification, separation logic, implicit dynamic frames, recursive predicates

ACM Reference Format:

Jenna Wise, Johannes Bader, Cameron Wong, Jonathan Aldrich, Éric Tanter, and Joshua Sunshine. 2020. Gradual Verification of Recursive Heap Data Structures. Proc. ACM Program. Lang. 4, OOPSLA, Article 228 (November 2020), 28 pages. https://doi.org/10.1145/3428296

Wise et al. 2020

Gradual Verification of Recursive Heap Data Structures

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val) * acc(l->next)
```

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val) * acc(l->next)
```

```
l != NULL * acc(l->val) * acc(l->next) *
acyclic(l->next)
```

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val) * acc(l->next)
```

```
l != NULL * acc(l->val) * acc(l->next) *
acyclic(l->next)
```

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val) * acc(l->next)
```

Imprecise formulas represent precise formulas that are …

- ☑ Self-framed (IDF logic)
- ☑ Satisfiable

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val) * acc(l->next)
```

Imprecise formulas represent precise formulas that are …

- ☑ Self-framed (IDF logic)
- ☑ Satisfiable
- ☑ Preserves (implies) static information

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val)
    * acc(l->next)
```

γ

```
l != NULL * acc(l->val) * acc(l->next)
    * acyclic(l->next)
```

```
l != NULL * acc(l->val) * acc(l->next)
    * true
```

...

UNSAT

```
l != NULL * acc(l->val) * acc(l->next)
    * l == NULL
```

```
l != NULL * true
```

MISSING STATIC INFO

Theory of ? and Imprecise Formulas [Wise et al. OOPSLA 2020]

```
? * l != NULL * acc(l->val)
* acc(l->next)
```

$\stackrel{\sim}{\Longrightarrow}$ ✅

```
l != NULL * acc(l->val) *
acc(l->next) * acyclic(l->next)
```

γ γ

Adaption of
Abstracting Gradual Typing
[Garcia et al. 2016]

```
l != NULL * acc(l->val) *
acc(l->next) * acyclic(l->next)
...
```

$\Longrightarrow$ ✅

```
l != NULL * acc(l->val) *
acc(l->next) * acyclic(l->next)
```

Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

Theory of ? & Run-time Checking

Wise et al. 2020
Gradual Verification of Recursive Heap Data Structures

Run-time Checking of Formulas [Wise et al. OOPSLA 2020]

```
l != NULL * acc(l->val) * acc(l->next) * acyclic(l->next)
```

```
SPECIAL STRATEGY
```

Accessibility Predicates

```
BOOLEAN FUNCTION
```

Predicates

```
assert l != NULL;
```

Boolean Expression

Run-time Checking of Predicates [Wise et al. OOPSLA 2020]

Recursive Predicate

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

Boolean Function

```
bool acyclic(Node root, OwnedFields of)
{
  if (root == NULL) {
    return true;
  } else {
    return assertAcc(root->val,of)  &&
           assertAcc(root->next,of) &&
           acyclic(root->next,of);
  }
}
```

Run-time Checking of Predicates [Wise et al. OOPSLA 2020]

Recursive Predicate

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

Boolean Function

```
bool acyclic(Node root, OwnedFields of)
{
  if (root == NULL) {
    return true;
  } else {
    return assertAcc(root->val,of)  &&
            assertAcc(root->next,of) &&
           acyclic(root->next,of);
  }
}
```

Run-time Checking of Predicates [Wise et al. OOPSLA 2020]

Recursive Predicate

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

Boolean Function

```
bool acyclic(Node root, OwnedFields of)
{
  if (root == NULL) {
    return true;
  } else {
    return assertAcc(root->val,of)  &&
           assertAcc(root->next,of) &&
           acyclic(root->next,of);
  }
}
```

Run-time Checking of Predicates [Wise et al. OOPSLA 2020]

Recursive Predicate

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

Boolean Function

```
bool acyclic(Node root, OwnedFields of)
{
  if (root == NULL) {
    return true;
  } else {
    return assertAcc(root->val,of)   &&
           assertAcc(root->next,of)  &&
           acyclic(root->next,of);
  }
}
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

$$\left(\text{Heap} \qquad \text{Stack} \qquad\qquad \right)$$

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

[Heap Stack **Owned Fields**]

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Owned Fields

$\{ \quad \}$

```
x = NULL;  l = NULL;
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

```
x = NULL;  l = NULL;

x = new Node(5,x);
```

Owned Fields

{ }

{ x->val x->next }

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Owned Fields

```
x = NULL;   l = NULL;
```

{ }

```
x = new Node(5,x);
```

{ x->val x->next }

```
l = new Node(3,l);
```

{ x->val x->next l->val l->next }

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Owned Fields

```
x = NULL;  l = NULL;
```

{ }

```
x = new Node(5,x);
```

{ x->val x->next }

```
l = new Node(3,l);
```

{ x->val x->next l->val l->next }

```
max = findMax(l);
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ `x->val` `x->next` `l->val` `l->next` }

```
max = findMax(l);
```

FindMax Owned Fields

{ }

```
int findMax(Node l)
  requires acyclic(l)
  ensures …
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ `x->val` `x->next` }

```
max = findMax(l);
```

FindMax Owned Fields

{ `l->val` `l->next` }

```
int findMax(Node l)
  requires acyclic(l)
  ensures …
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ `x->val` `x->next` `l->val` `l->next` }

```
max = findMax(l);
```

FindMax Owned Fields

{ }

```
int findMax(Node l)
  requires ?
  ensures …
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ }

```
max = findMax(l);
```

FindMax Owned Fields

{ x->val x->next l->val l->next }

```
int findMax(Node l)
  requires ?
  ensures …
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ }

```
max = findMax(l);
```

FindMax Owned Fields

{ x->val x->next l->val l->next }

```
int findMax(Node l)
  requires ?
  ensures …
```

{ x->val x->next l->val l->next }

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ }

```
max = findMax(l);
```

FindMax Owned Fields

{ x->val x->next l->val l->next }

```
int findMax(Node l)
  requires …
  ensures ?
```

{ x->val x->next l->val l->next }

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

Main Owned Fields

{ }

```
max = findMax(l);
```

{ x->val x->next l->val l->next }

FindMax Owned Fields

{ x->val x->next l->val l->next }

```
int findMax(Node l)
  requires …
  ensures ?
```

{ }

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

```
acc(l->val)
```

→ Owned Fields

```
bool assertAcc(l->val,of)
{
    if (l->val in of) {
      remove(l->val,of);
      return true;
    } else {
      return false;
    }
}
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

```
acc(l->val)
```

Owned Fields

```
bool assertAcc(l->val,of)
{
    if (l->val in of) {
      remove(l->val,of);
      return true;
    } else {
      return false;
    }
}
```

Run-time Checking of Accessibility Predicates [Wise et al. OOPSLA 2020]

```
acc(l->val)
```

→ **Owned Fields**

```
bool assertAcc(l->val,of)
{
    if (l->val in of) {
      remove(l->val,of);
      return true;
    } else {
      return false;
    }
}
```

Important Gradual Properties

Soundness
All specification violations caught statically or at run time

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

Gradual Guarantee
If your specifications are a subset of a correct specification, you won't get any errors

[Siek et al. 2015]

Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

Theory of ? & Run-time Checking

Wise et al. 2020

Gradual Verification of Recursive Heap Data Structures

DiVincenzo et al. 2025

Gradual C0: Symbolic Execution for Gradual Verification

Tool Implementation: Gradual C0

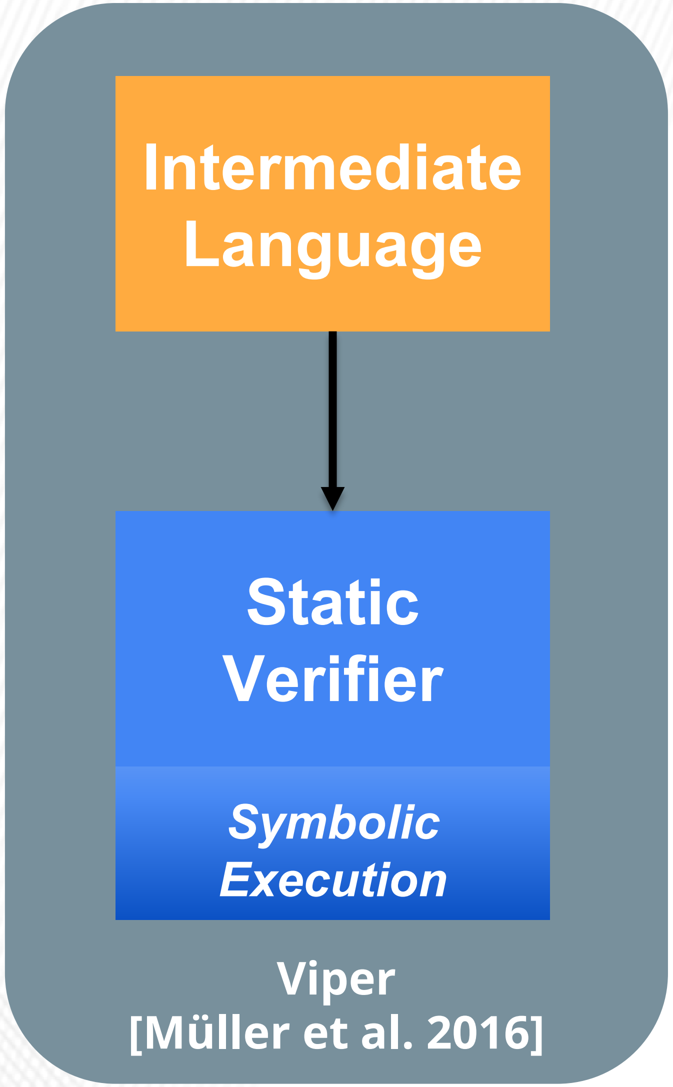
Zimmerman et al. 2024

Sound Gradual Verification with Symbolic Execution

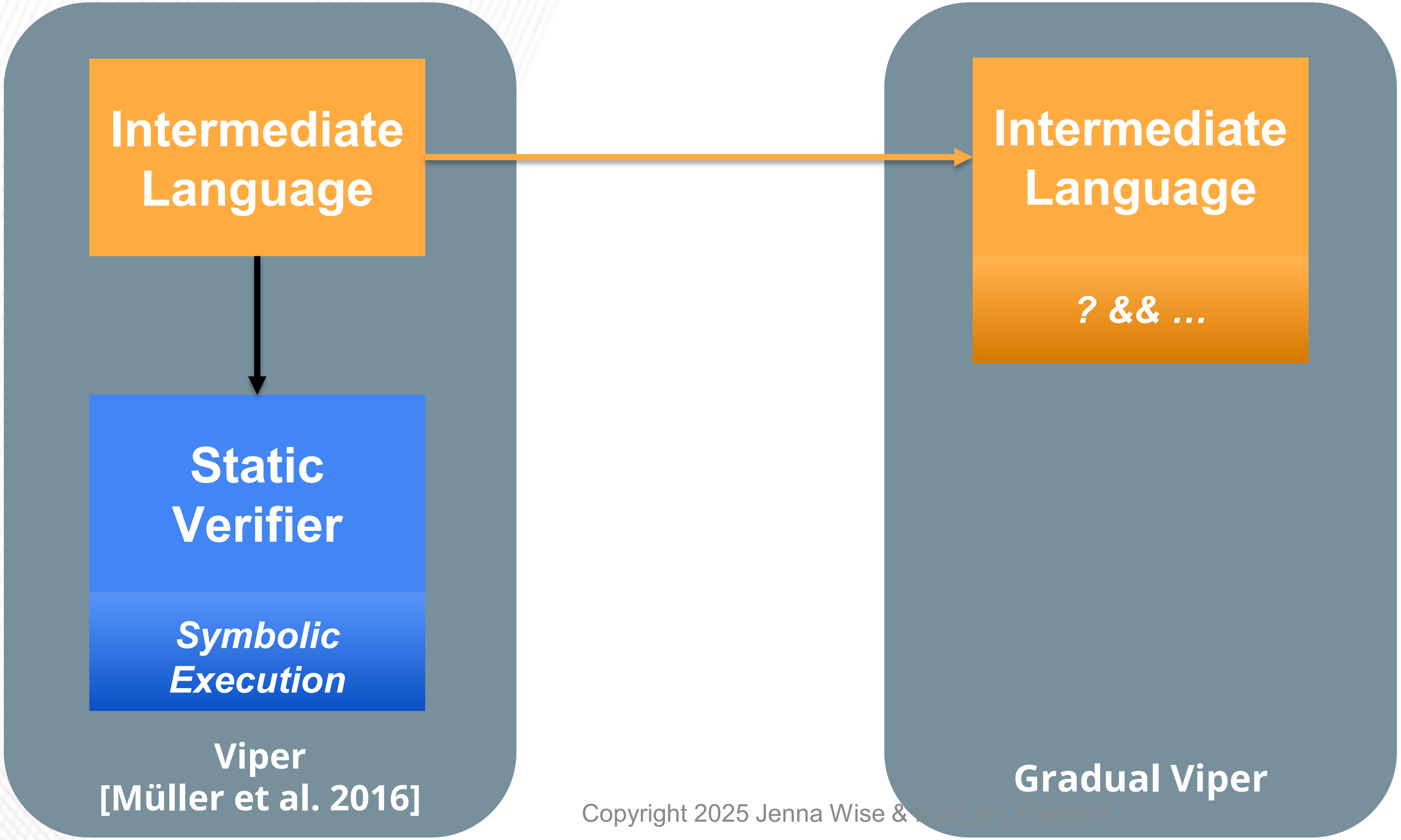
[DiVincenzo et al. 2025]

[Zimmerman et al. 2024]

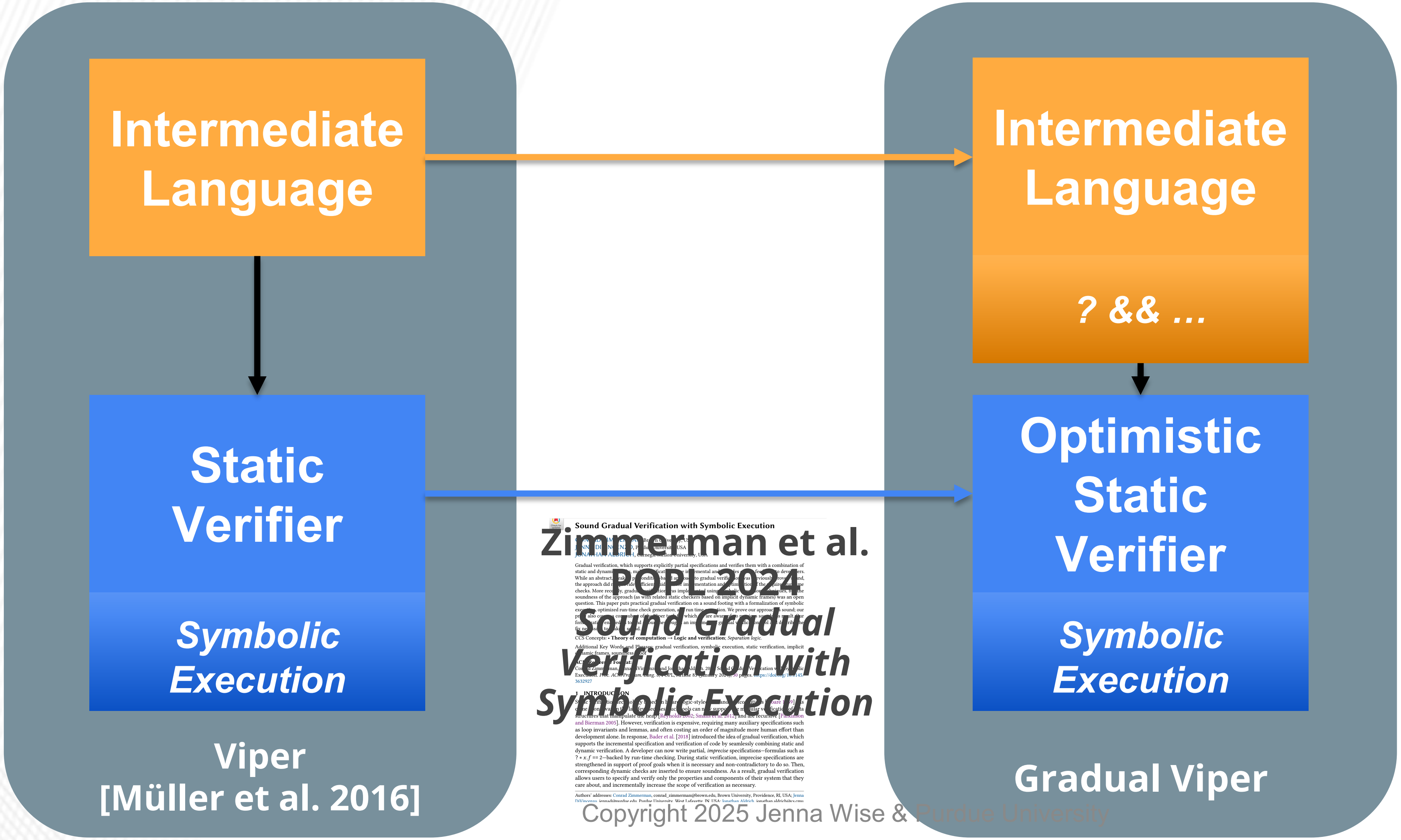
System Design of Gradual C0 [DiVincenzo et al. TOPLAS 2025]:
The First Gradual Verification Tool



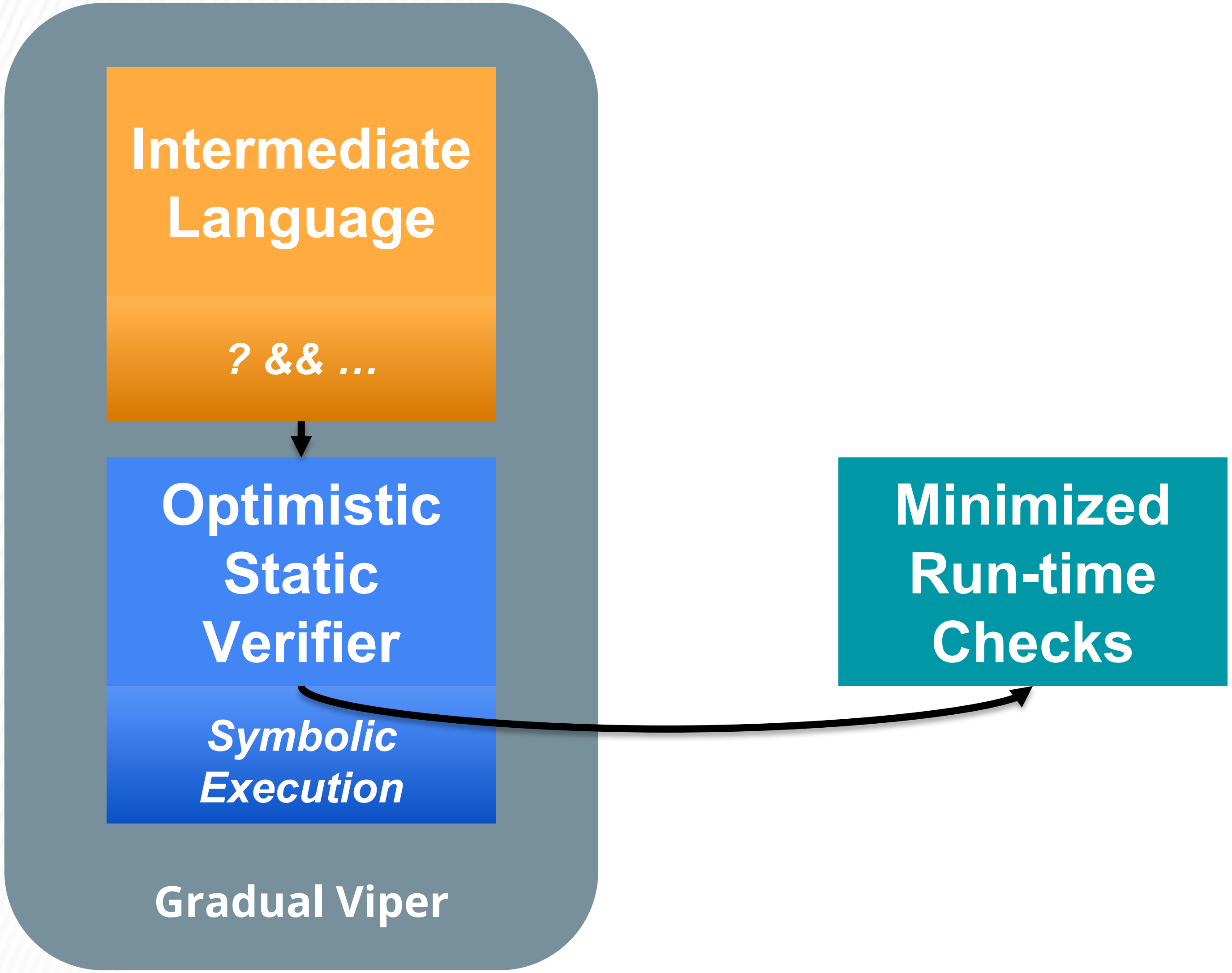
System Design of Gradual C0 [DiVincenzo et al. TOPLAS 2025]: *The First Gradual Verification Tool*



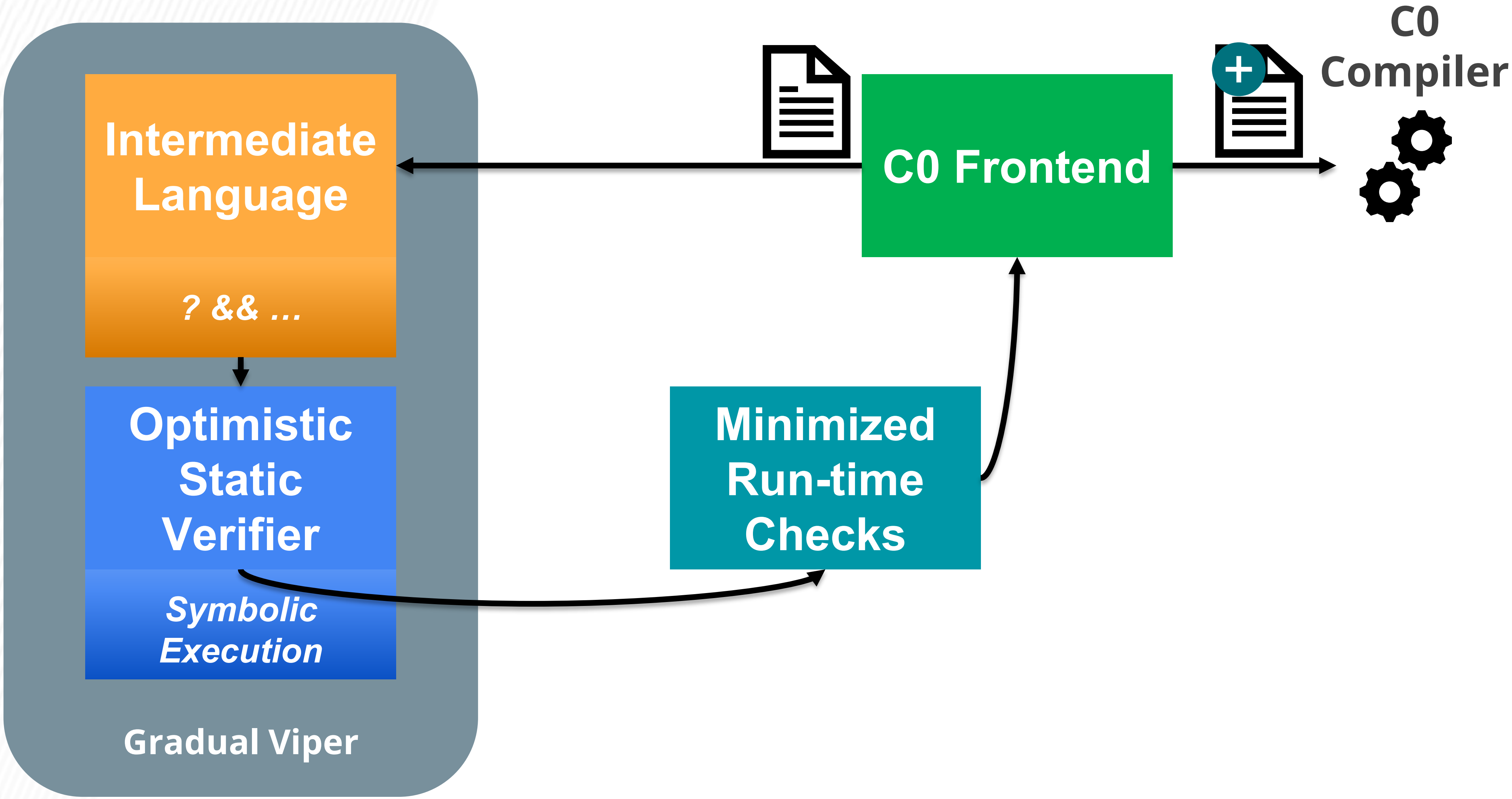
System Design of Gradual C0 [DiVincenzo et al. TOPLAS 2025]: *The First Gradual Verification Tool*



System Design of Gradual C0 [DiVincenzo et al. TOPLAS 2025]: *The First Gradual Verification Tool*



System Design of Gradual C0 [DiVincenzo et al. TOPLAS 2025]: *The First Gradual Verification Tool*



Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

Wise et al. 2020
Gradual Verification of Recursive Heap Data Structures

Theory of ? & Run-time Checking

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Zimmerman et al. 2024
Sound Gradual Verification with Symbolic Execution

Tool Implementation: Gradual C0

Run-time Performance Study

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Research Questions [DiVincenzo et al. TOPLAS 2025]:
A Quantitative Performance Evaluation of Gradual C0

[RQ1] As specifications are made more precise, can more verification conditions be eliminated statically?

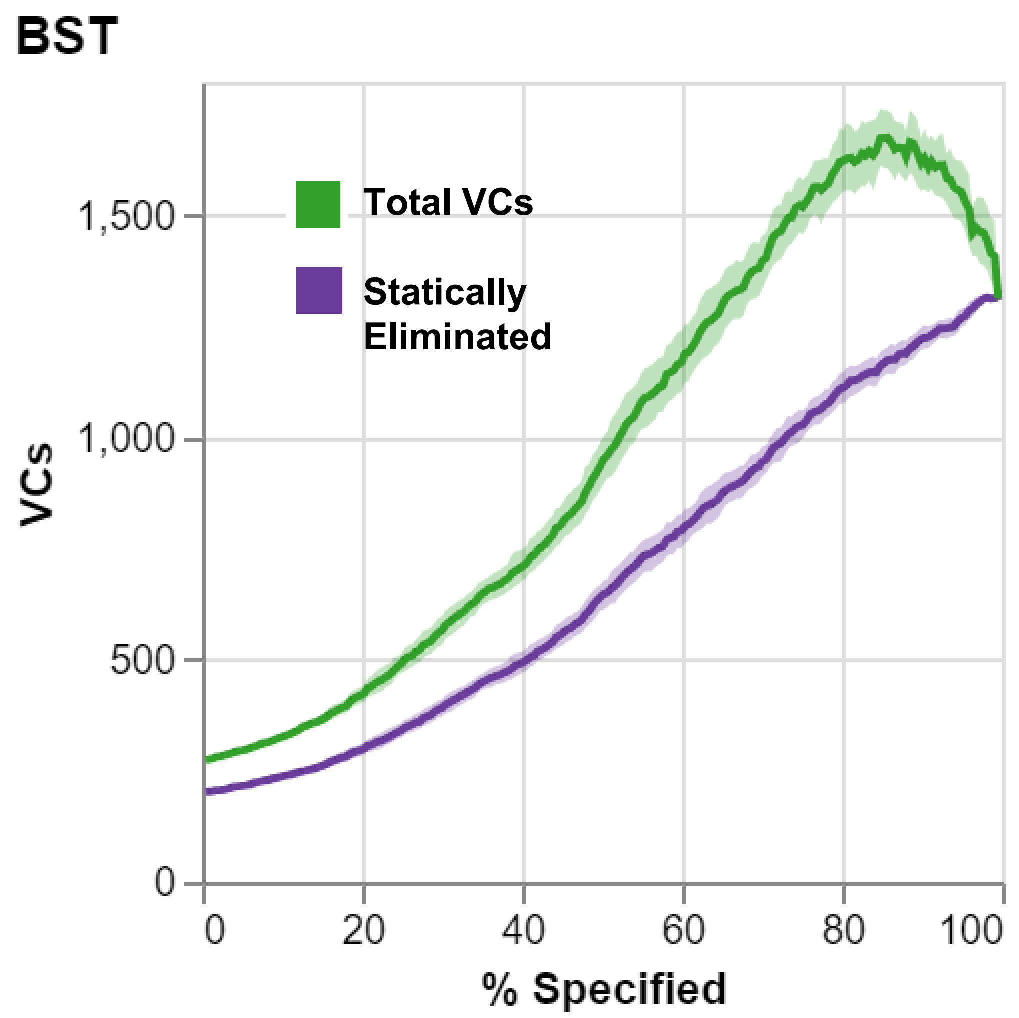
[RQ2] Does gradual verification result in less run-time overhead than a fully dynamic approach?

[RQ3] Are there types of specification constructs that significantly impact run-time performance?

Thousands of Partial Specifications Evaluated

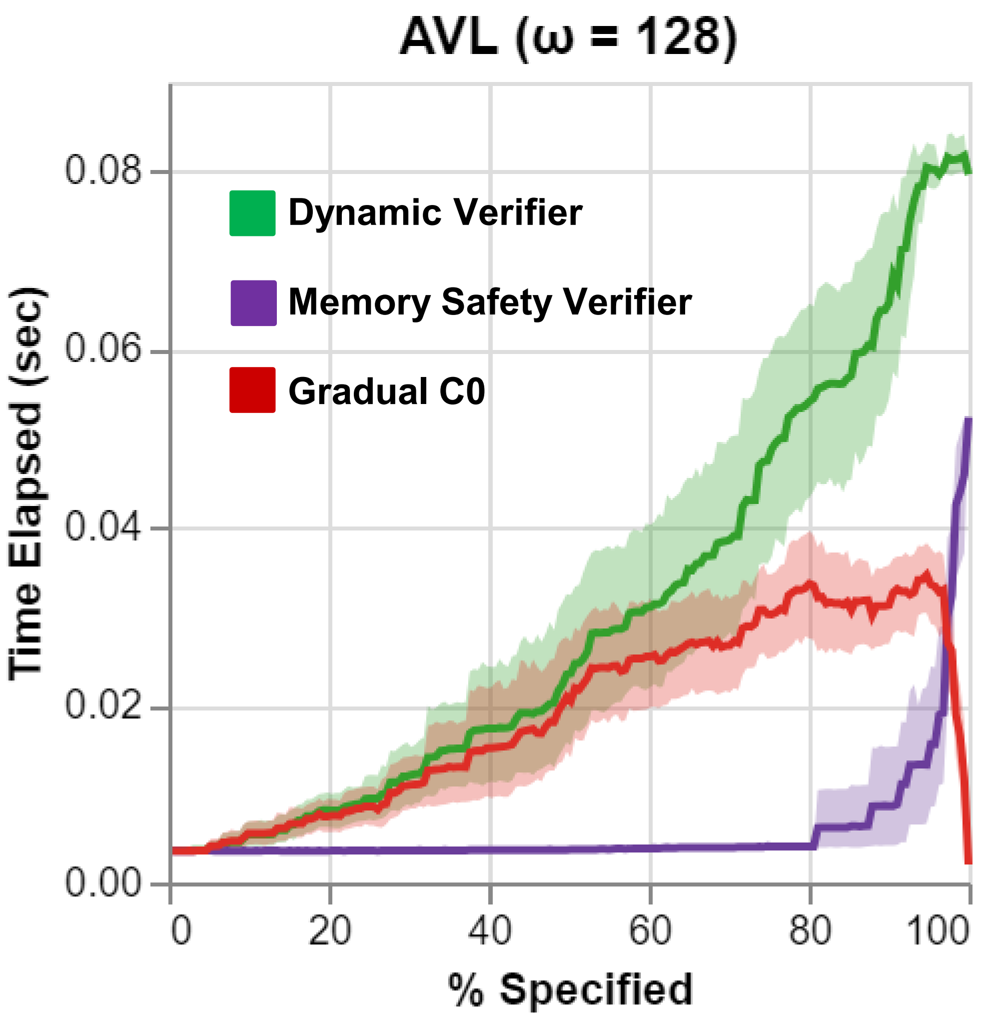
Benchmark	# of Sampled Partial Specifications
Sorted Linked List	1,745
Binary Search Tree	3,473
Composite Tree	2,577
AVL Tree	3,057

RQ1: As specifications are made more precise, can more verification conditions be eliminated statically? **YES**



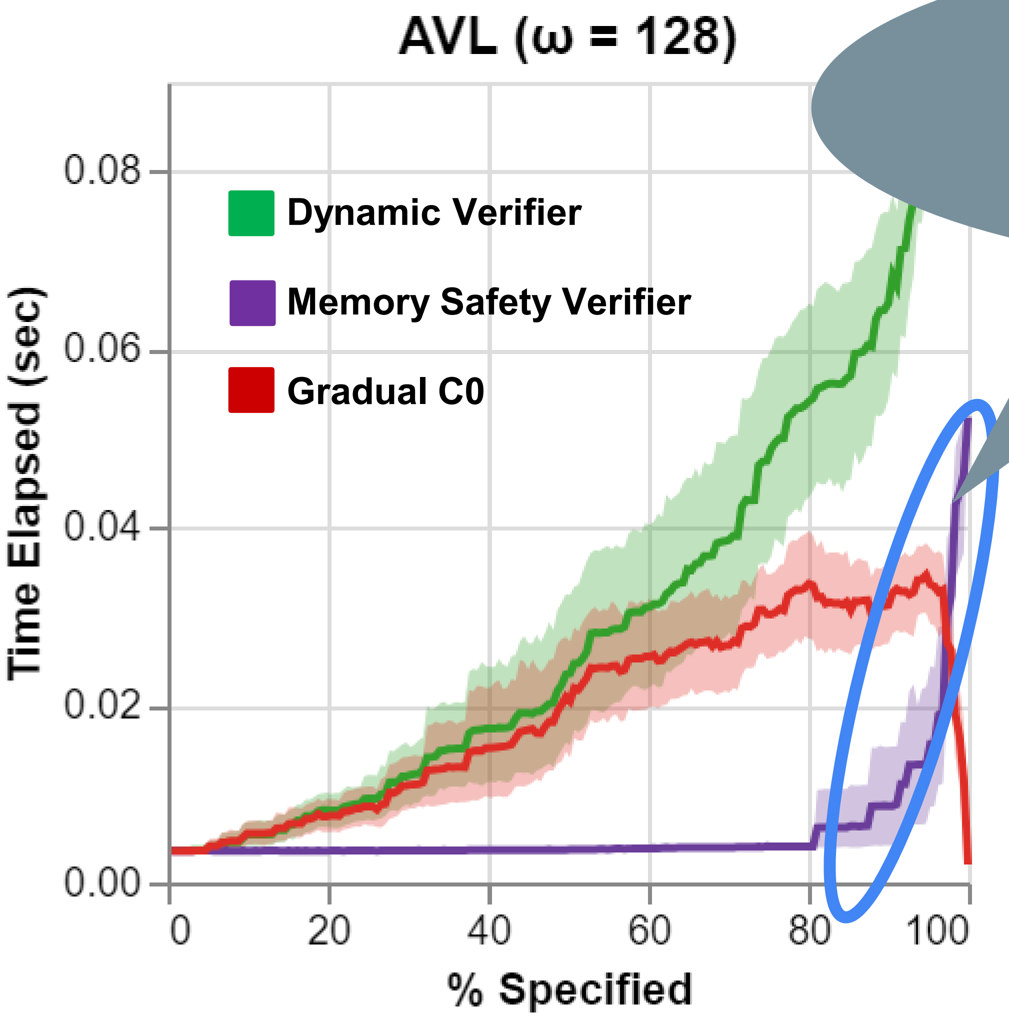
RQ2: Does gradual verification result in less run-time overhead than a fully dynamic approach?

YES



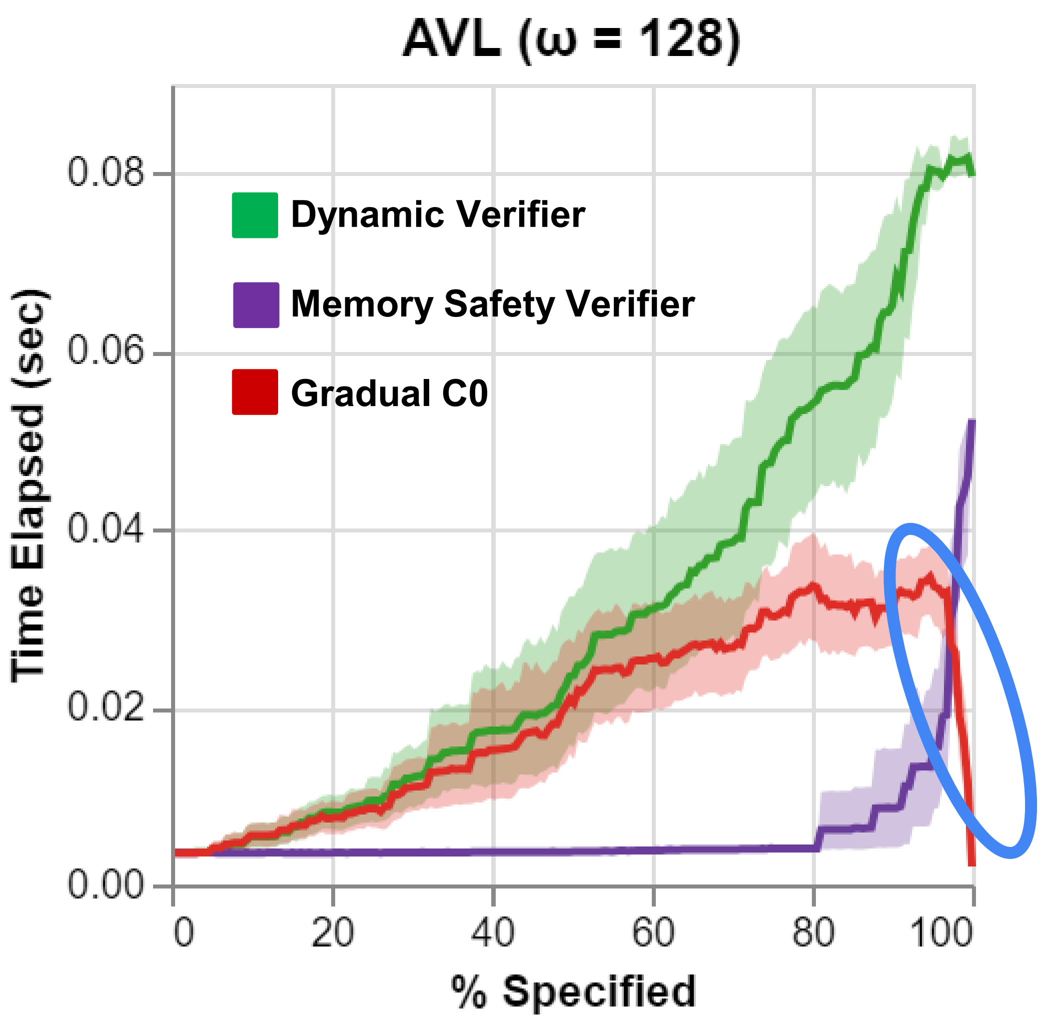
Gradual C0 reduces run-time overhead by 7-40%

RQ2: Does gradual verification result in less run-time overhead than a fully dynamic approach?

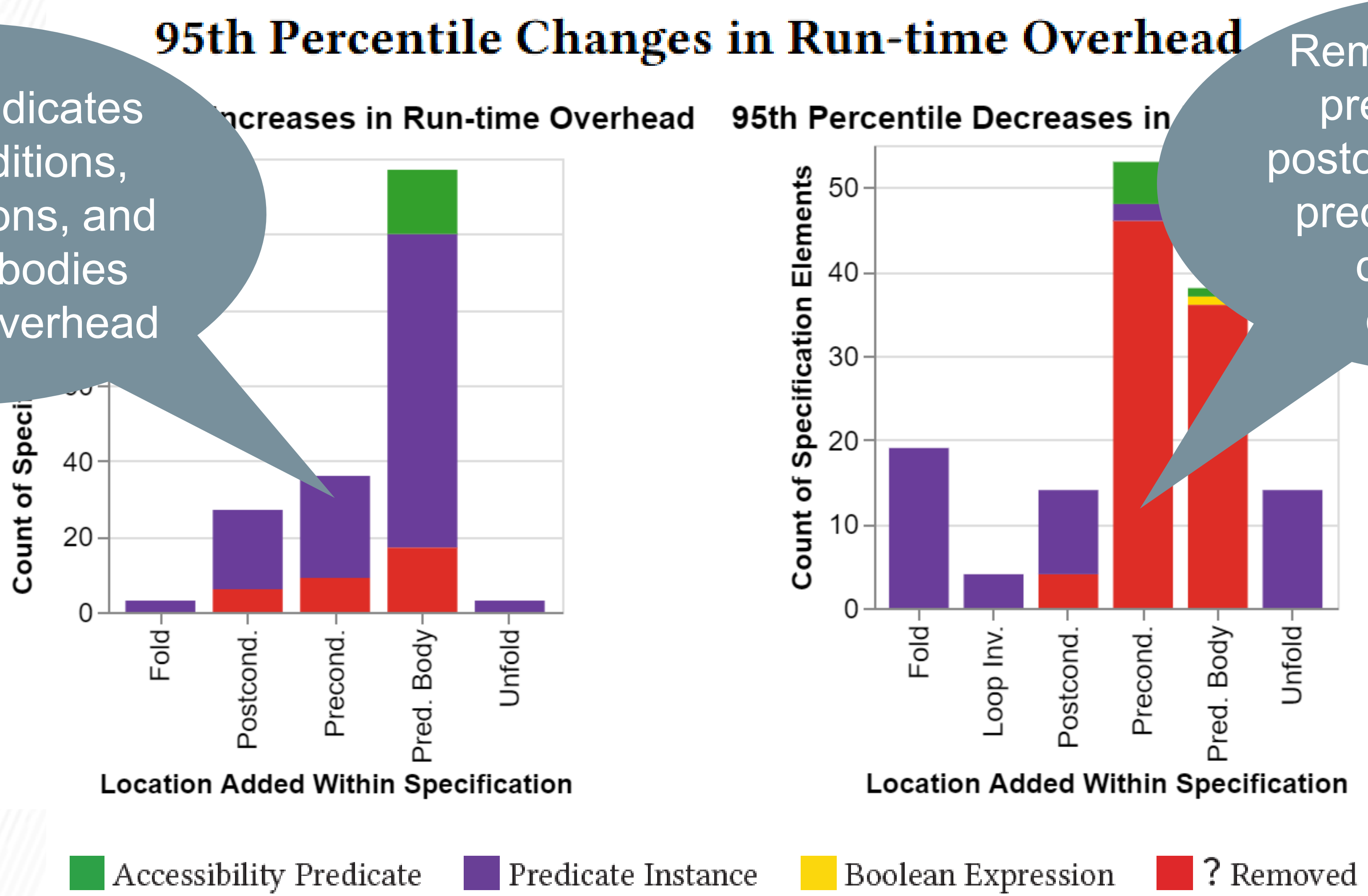


Tracking owned fields at method boundaries is expensive!

RQ2: Does gradual verification result in less run-time overhead than a fully dynamic approach?



RQ3: Are there types of specification constructs that significantly impact run-time performance? YES



Adding predicates to preconditions, postconditions, and predicate bodies increases overhead

Removing ? from preconditions, postconditions, and predicate bodies decreases overhead

Gradual Verification of Recursive Heap Data Structures

How Gradual Verification Works

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Theory of ? & Run-time Checking

Wise et al. 2020
Gradual Verification of Recursive Heap Data Structures

Tool Implementation: Gradual C0

Zimmerman et al. 2024
Sound Gradual Verification with Symbolic Execution

Run-time Performance Study

DiVincenzo et al. 2025
Gradual C0: Symbolic Execution for Gradual Verification

Gradual C0 Case Study: Gradually Verifying a C parser

Interesting Result:
Gradual C0 detected incorrect specifications for AVL tree earlier than static verification

Saved us a lot of time and effort!

A Case Study: Gradually Verifying a C Parser with Gradual C0 [Wise DiVincenzo CMU 2023]

[RQ1,RQ2] What trends, themes, or trade-offs occur during the verification process?

[RQ3] Is static or dynamic feedback helpful or detrimental?

[RQ4] How is gradual verification used to find bugs in real code?

[RQ5] What limitations does Gradual C0 have?

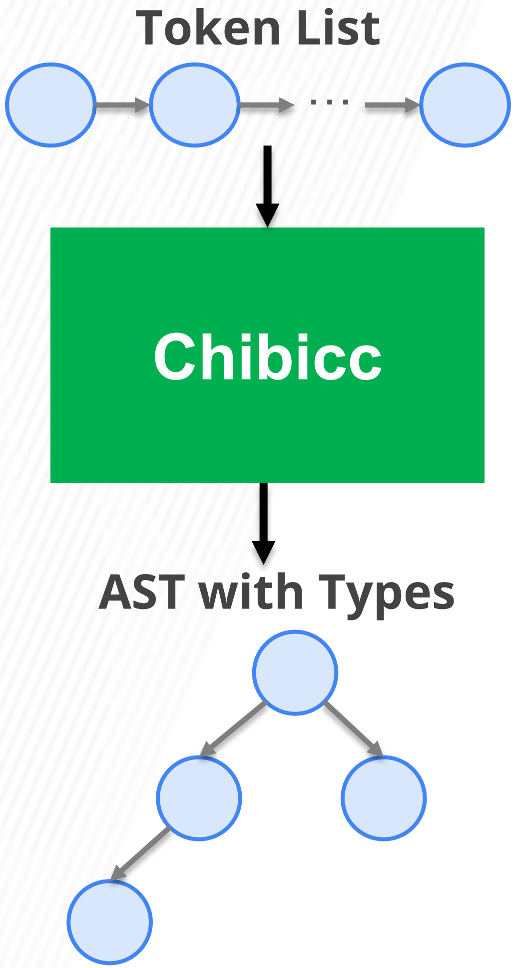
[RQ6] What challenges does the software being studied pose for gradual verification?

Objective

To explore how gradual verification is used to verify real application software

The Case for Study: A C Parser Called Chibicc

Chibicc *is a recursive descent parser for C programs found on Github with 10.4k stars.*



Code Features After Translation to C0 Code	
Lines of Program Code (LoPC)	3k
Functions	229
Loops	41
Modules	6

Design & Methodology

Gradually verified that the loops in Chibicc's main functions terminate using Gradual C0 over 1 week

- Recorded experience, thought process, and comments in a Google Form
- Saved data from Gradual C0, such as intermediate files, performance measures, and verification results
- Qualitatively coded text answers from Google Form
- Developed narrative from coded data and used Gradual C0 data to provide additional supporting details

Key Results [Wise DiVincenzo CMU 2023]

A Few Key Results

RQ1, RQ2 Avoided specifying ownership—an orthogonal property in orthogonal functions (over 200)—with Gradual C0 and still received useful verification feedback

RQ3, RQ4 Early and often reporting of relevant static and dynamic verification errors helped us uncover bugs in specification and code earlier than static verification alone

RQ5, RQ6 Dynamic verification is limited by execution coverage, but selective applications of static verification with Gradual C0 can cover for this weakness.

Thesis Work Shows:

It is possible to build gradual verification technology that

- *verifies programs manipulating recursive heap data structures*

Tool Implementation: Gradual C0
[DiVincenzo et al. 2025]

- *is sound and adheres to gradual properties*

Theory of ? & Run-time Checking
[Wise et al. 2020]
[Zimmerman et al. 2024]

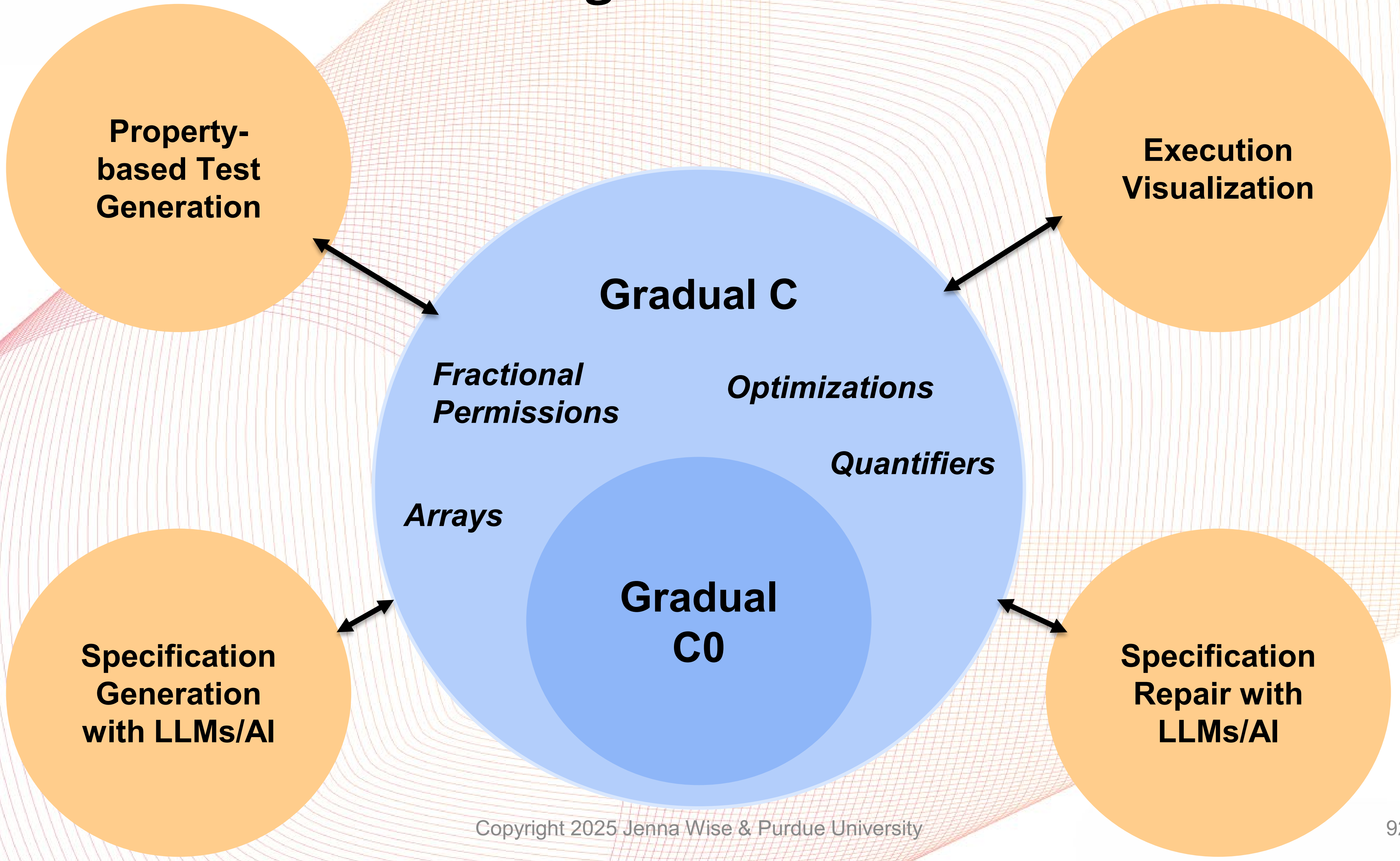
- *has minimized run-time overhead*

Run-time Performance Study
[DiVincenzo et al. 2025]

- *is useful in practice*

Gradual C0 Case Study: Gradually Verifying a C parser
[Wise DiVincenzo 2023]

New Extensions and Integrations with Gradual Verification

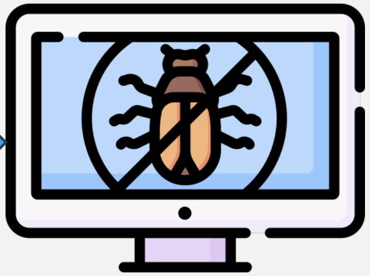


Gradual Verification: Assuring Programs Incrementally

Prof. Jenna DiVincenzo
https://jennalwise.github.io/

Gradual Verification – supports incremental verification by applying static (compile-time) verification where possible and dynamic (run-time) verification where necessary

Secure code as it is written.
Easier to write bug free code!



Current Projects:

Theory	Tool Building	User Studies

- Gradual Verifier for C
- Proof Synthesis for Gradual Verifiers
- Gradual Verification for Education