Lecture 26: Scaling Up Verification: Heap-Manipulating Programs and Gradual Verification

17-355/17-655/17-819: Program Analysis

Rohan Padhye and Jonathan Aldrich

May 6, 2021

* Course materials developed with Claire Le Goues

Hoare Logic-based Verification So Far

- Focus on imperative programs without functions or memory allocation
  - E.g. the WHILE language

```
S   ::= x := a | skip | S1 ; S2
      |   if P then S1 else S2 | while P do S
a   ::= x | n | a1 opa a2
...
```

- What about other constructs?
  - Function calls
  - Heap data structures – e.g. pointers and records

A Language with Functions and Records

- We define a program as a list of declarations D followed by a list of statements
  - Functions have one parameter, and include pre- and post-condition specifications
  - Records are a named structure with a set of field declarations

$$
\begin{array}{lcl}
D & ::= & \text{fun } g(x) \text{ requires } P \text{ ensures } Q \; \{\; S \;\} \\
 & | & \text{record } R \; \{\; \mathit{flds} \;\} \\
flds & ::= & \text{field } f; \mid \mathit{flds}\; \mathit{flds}
\end{array}
$$

Extended expression and statement syntax

- We also extend statements and expressions
  - Function call and field assignment statements
    - Function calls are statements because they have side effects – awkward in an expression
  - New record and field read expressions

$$
\begin{array}{lll}
S & ::= & x := a \\
 & | & \text{skip} \\
 & | & S_1;\ S_2 \\
 & | & \text{if } b \text{ then } S_1 \text{ else } S_2 \\
 & | & \text{while } b \text{ do } S \\
 & | & \boxed{x := g(a)} \\
 & | & \boxed{x.f := a}
\end{array}
\qquad
\begin{array}{lll}
b & ::= & \text{true} \\
 & | & \text{false} \\
 & | & \text{not } b \\
 & | & b_1\ op_b\ b_2 \\
 & | & a_1\ op_r\ a_2
\end{array}
\qquad
\begin{array}{lll}
a & ::= & x \\
 & | & n \\
 & | & a_1\ op_a\ a_2 \\
 & | & \boxed{\text{new} R} \\
 & | & \boxed{a.f}
\end{array}
\qquad
\begin{array}{lll}
op_b & ::= & \text{and} \mid \text{or} \\
op_r & ::= & < \mid \leqslant \mid = \\
 & | & > \mid \geqslant \\
op_a & ::= & + \mid - \mid * \mid /
\end{array}
$$

Verifying Functions

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \}}{\text{fun } g(x) \text{ requires } P \text{ ensures } Q\ \{\ S\ \}\ OK}\ \textit{fn-defn}$$

- Example (we extend to multiple arguments)

```
fun exp(x,n)
   requires n ≥ 0
   ensures result = x^n {
result := 1
count := 0
while count < n do
   result := result * x
   count := count + 1
}
```

Verifying Function Calls – An Example

$$\frac{decls(g) = \text{fun } g(y) \text{ requires } P \text{ ensures } Q \ldots}{\{\ [a/y]P\ \}\ x := g(a)\ \{\ [a/y, x/result]Q\ \}}\ \textit{fn-call}$$

fun exp(x,n)

requires n $\geq$ 0

ensures result = x^n { … }

{ **true** }

j := 2

z = exp(y, j)

{ z = y^2 }

Verifying Field Assignments

$$\frac{x.f \notin a}{\{\ [a/x.f]P\ \}\ x.f := a\ \{\ P\ \}}\ \textit{field-assign (simplified)}$$

{ true }

x.f = 2

z = y * x.f

{ z = y*2 }

The Challenge of Aliasing

{ true }

x = new R

x.f = 1

y = x

y.f = 2

{ x.f = 1 }

$$\frac{x.f \notin a}{\{\ [a/x.f]P\ \}\ x.f := a\ \{\ P\ \}}$$

- This program verifies!
  But it's not correct
- Issue: P contains elements that might be affected by the assignment
- How can we fix this problem?

Addressing Aliasing

- If we know x = y, then we can update y.f when we update x.f
- If we know $x \neq y$, then we can preserve knowledge of y.f when we update x.f
- If we don't know whether x = y or not, we "forget" knowledge of y.f
  - One possibility: replace all occurrences of y.f with an existentially quantified variable
- Challenge 1: tracking aliasing doesn't scale
  - If you have n variables, there are n * (n-1) / 2 aliasing conditions!
    - For w, x, y, z: $w \neq x \land w \neq y \land w \neq z \land x \neq y \land x \neq z \land y \neq z$
    - Too much specification to be realistic
- Challenge 2: tracking aliasing is unmodular

Tracking Aliasing Conditions is Unmodular

```
fun doubleXF(x)
    requires x ≠ y ∧ x.f = n ∧ y.f = m
    ensures x ≠ y ∧ x.f = n*2 ∧ y.f = m {
    x.f = x.f * 2
}
```

```
x = new R
y = new R
x.f = 1
y.f = 3
doubleXF(x)
assert x.f = 2 ∧ y.f = 3
```

doubleXF doesn't use y. It's unmodular for its spec to mention y.

The Frame Rule supports modular specification

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \quad vars(R) \cap assigned(S) = \varnothing}{\{\ P \land R\ \}\ S\ \{\ Q \land R\ \}}\ \textit{frame (simplified)}$$

- The frame rule allows us to reason about direct effects of S (transforming P to Q), and “carry over” other things we know (in R)
  - One caution: we must be sure that R does not mention any variables assigned by S
- With the Frame Rule, we can call a function that does not mention y in its spec and still preserve our knowledge about y

How the Frame Rule Helps

```
fun double(x)
    requires x=n
    ensures result = n*2 {
    result = x * 2
}
```

```
x = 1
y = 3
x = double(x)
assert x = 2 ∧ y = 3
```

> We must apply the frame rule here to carry over our knowledge that y=3

$$\dfrac{\dfrac{decls(double) = \text{fun } double(x) \text{ requires } x = n \text{ ensures } result = n * 2 \ldots}{\{\ x = 1\ \}\ x := double(x)\ \{\ x = 2\ \}}\ \textit{fn-call}}{\{\ x = 1 \land y = 3\ \}\ x := double(x)\ \{\ x = 2 \land y = 3\ \}}\ \textit{frame}$$

But we need a frame rule that addresses aliasing!

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \qquad vars(R) \cap assigned(S) = \emptyset}{\{\ P \wedge R\ \}\ S\ \{\ Q \wedge R\ \}}\ \textit{frame (simplified)}$$

Idea: Let's make sure that P describes all of the object-field combinations that S could access.

What if R mentions a field of an object that is assigned in S?

Resource Logics talk about state that is *owned*

```
fun doubleXF(x)
    requires acc(x.f) * x.f = n
    ensures acc(x.f) * x.f = n*2 {
    x.f = x.f * 2
}
```

We're only allowed to mention x.f in the formula because we have asserted **acc**(x.f)

acc(x.f) means we own x.f and can use it in this function and its specification

* is a special kind of conjunction (see next slide)

This is a research logic called Implicit Dynamic Frames (IDF)

The full Frame Rule, considering aliasing

$$\frac{\{\ P\ \}\ S\ \{\ Q\ \} \quad vars(R) \cap assigned(S) = \varnothing \quad P, R, S\ \textit{self-framed}}{\{\ P\ *\ R\ \}\ S\ \{\ Q\ *\ R\ \}}\ \textit{frame (full)}$$

The *separating conjuction* * is like $\wedge$, but any given object field can be owned by only one side.

Thus **acc**(x.f) * **acc**(y.f) implies $x \neq y$

A *self-framed* formula only mentions object fields that it owns.

x.f = 3 is not self-framed.
acc(x.f) * x.f=3 is self-framed

The allocation rule in Implicit Dynamic Frames

$$\frac{}{\{true\}\ x := \text{new}\ R\ \{\ \forall_* f \in fields(R)\ .\ \text{acc}(x.f)\ \}}\ alloc$$

- Provides the permission to all fields in the newly allocated object

Quiz: check the full example by filling in the { }'s

```
fun doubleXF(x)
    requires acc(x.f) * x.f = n
    ensures acc(x.f) * x.f = n*2 {
    x.f = x.f * 2
}
```

```
{ true }
x = new R
y = new R
{                                        }
x.f = 1
y.f = 3
{                                        }
doubleXF(x)
{ acc(x.f) * x.f = 2 * acc(y.f) * y.f = 3 }
```

What about recursive data structures?

```
record Node { int val; Node next; }
predicate list(Node n, int sum) =
   if (n ≠ null)
      then ∃s1 . acc(n.val) * acc(n.next)
         * list(n.next, s1) * sum=n.val+s1


fun cons(Node n, int v)
   requires list(n, s)
   ensures list(result, s+v)
result = new Node
result.val = v
result.next = n
fold list(result, s+v)
```

Can define recursive predicates that describe properties of a data structure—in this case that a list sums to a particular value

Functions can use **fold** and **unfold** to move between a predicate and its unfolded definition

Gradual Verification of Recursive Heap Data Structures

Jenna Wise (Carnegie Mellon University, Johannes Bader (Jane Street), Cameron Wong (Jane Street),

Jonathan Aldrich (Carnegie Mellon University), Éric Tanter (University of Chile), Joshua Sunshine (Carnegie Mellon University)

Dynamic verification increases runtime overhead for weaker assurances

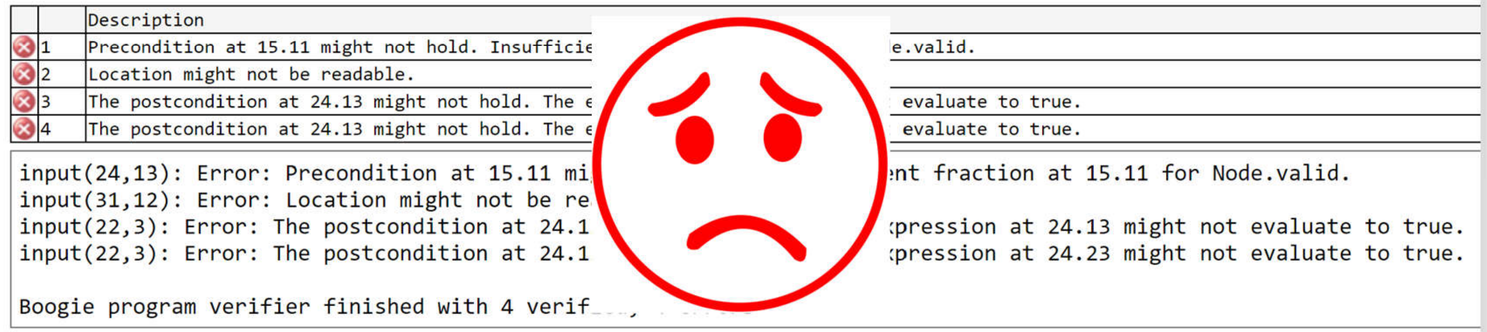
Static verification has a large upfront specification cost

Gradual verification allows developers to deal with this cost incrementally

- without unnecessary effort
- with immediate feedback

Naïve Verification Attempt

```
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: Missing Preconditions

```
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;
}
```

Naïve Verification Attempt: Missing Loop Invariants

```
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;
}
```

Naïve Verification Attempt: Missing Folds and Unfolds

```
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
  }
    FOLDS/UNFOLDS
  return m;
}
```

Naïve Verification Attempt: Missing Lemmas

```
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;
}
```

Naïve Verification Attempt: Missing Specifications

```
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;
}
```

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 INVARIANT  {
    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 INVARIANT {
    if(curr.val > m) { m := curr.val; }
    curr := curr.next;
  }
    FOLDS/UNFOLDS
  return m;
}
```

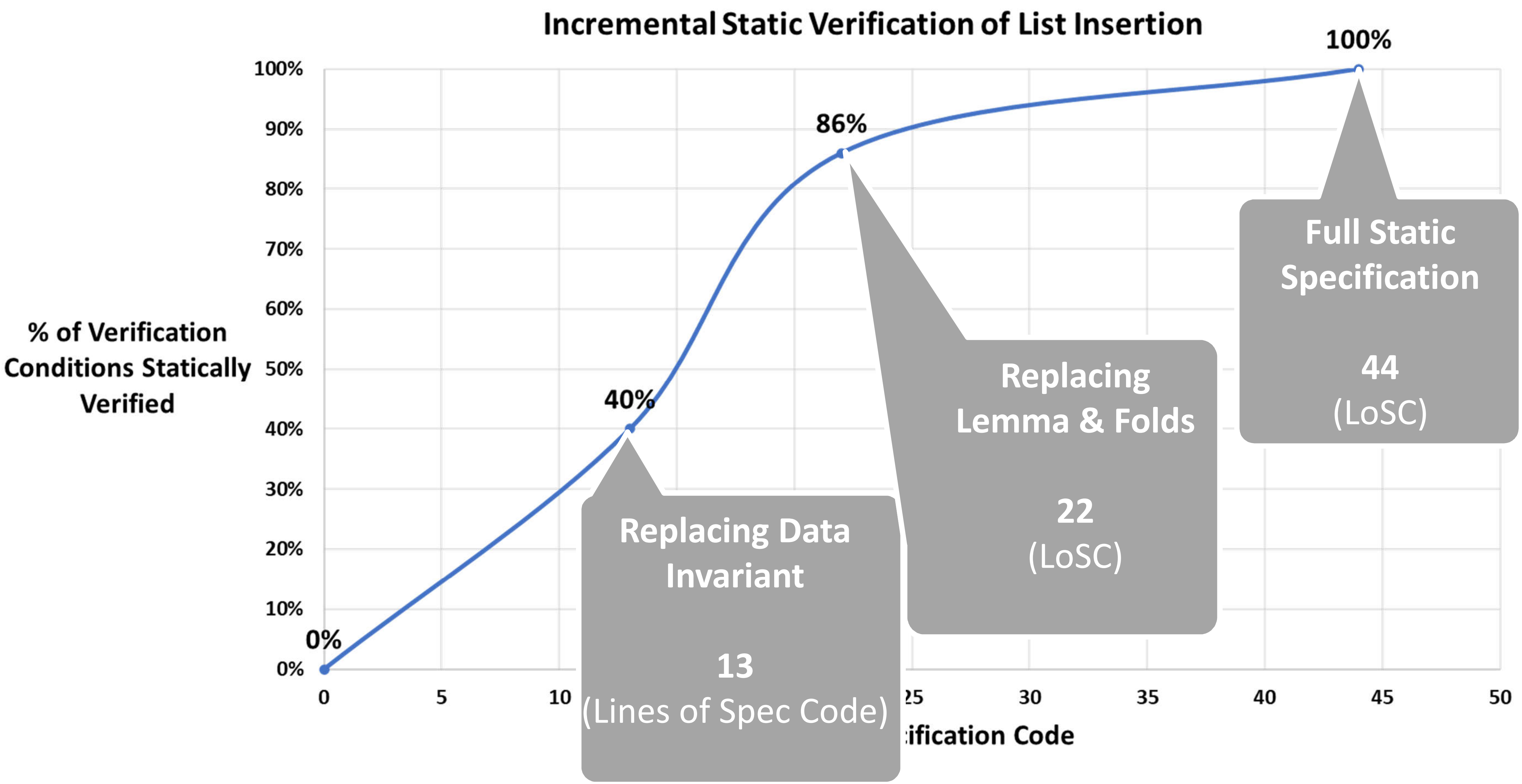
Gradual Verification to the Rescue

```
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 INVARIANT
    if(curr.val > m) { m := curr.val; }
    curr := curr.next;
      FOLDS/UNFOLDS
  }
    FOLDS/UNFOLDS
  return m;
}
```

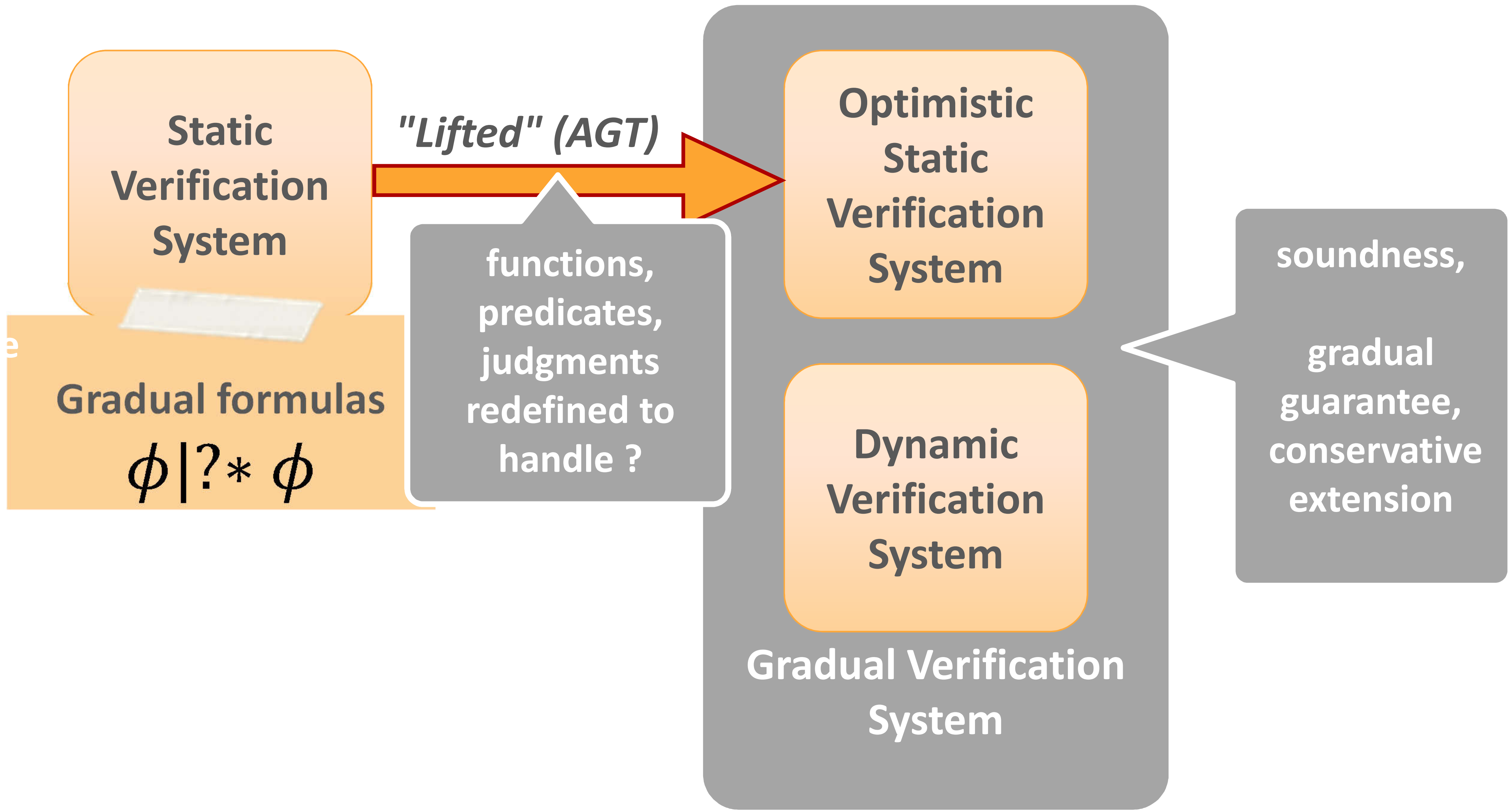
Naïve Verification Attempt: Missing Specifications

```
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 INVARIANT
    if(curr.val > m) { m := curr.val; }
    curr := curr.next;
      FOLDS/UNFOLDS
          LEMMAS
  }
    FOLDS/UNFOLDS
  return m;
}
```

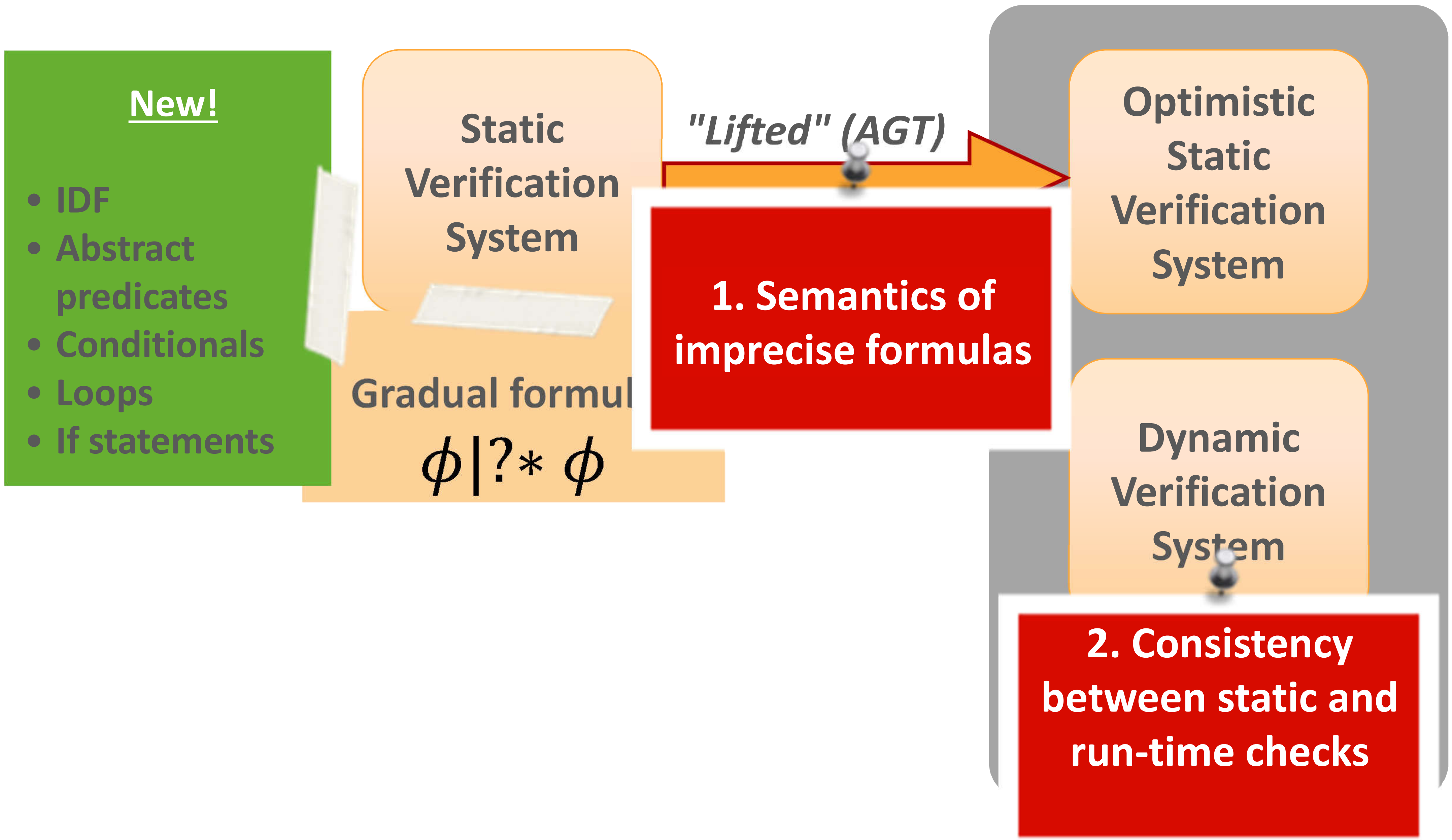
Incremental Static Verification of List Insertion



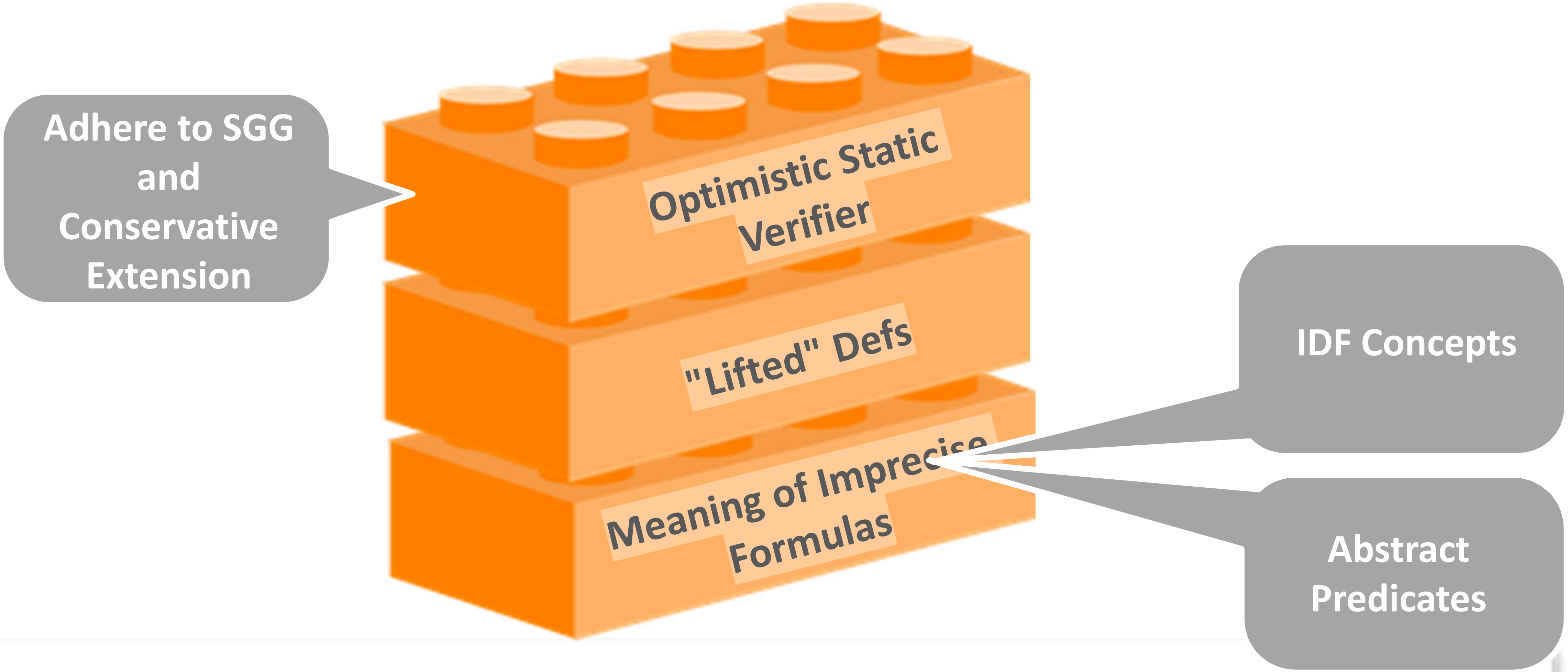
Gradual Verification Framework



Extending the Prior Gradual Verification Approach



1. Giving the Right Meaning to Imprecise Formulas



Static gradual guarantee - any specification increment with correct specifications will not fail to statically verify

Meaning of Imprecise Formulas: By Example

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

Separating conjunction - preds must refer to different heap locations

Accessibility predicate - denotes permission to access a heap location

```
predicate acyclic(Node root) =
  if root == null then true else acc(root.val)
    * acc(root.next) * acyclic(root.next)
```

Meaning of Imprecise Formulas: By Example

```
predicate acyclic(Node root) =
  if root == null then true else acc(root.val)
    * acc(root.next) * acyclic(root.next)
```

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

assert acyclic(l);
```

Meaning of Imprecise Formulas: By Example

```
predicate acyclic(Node root) =
  if root == null then true else acc(root.val)
    * acc(root.next) * acyclic(root.next)
```

```
{ 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);
```

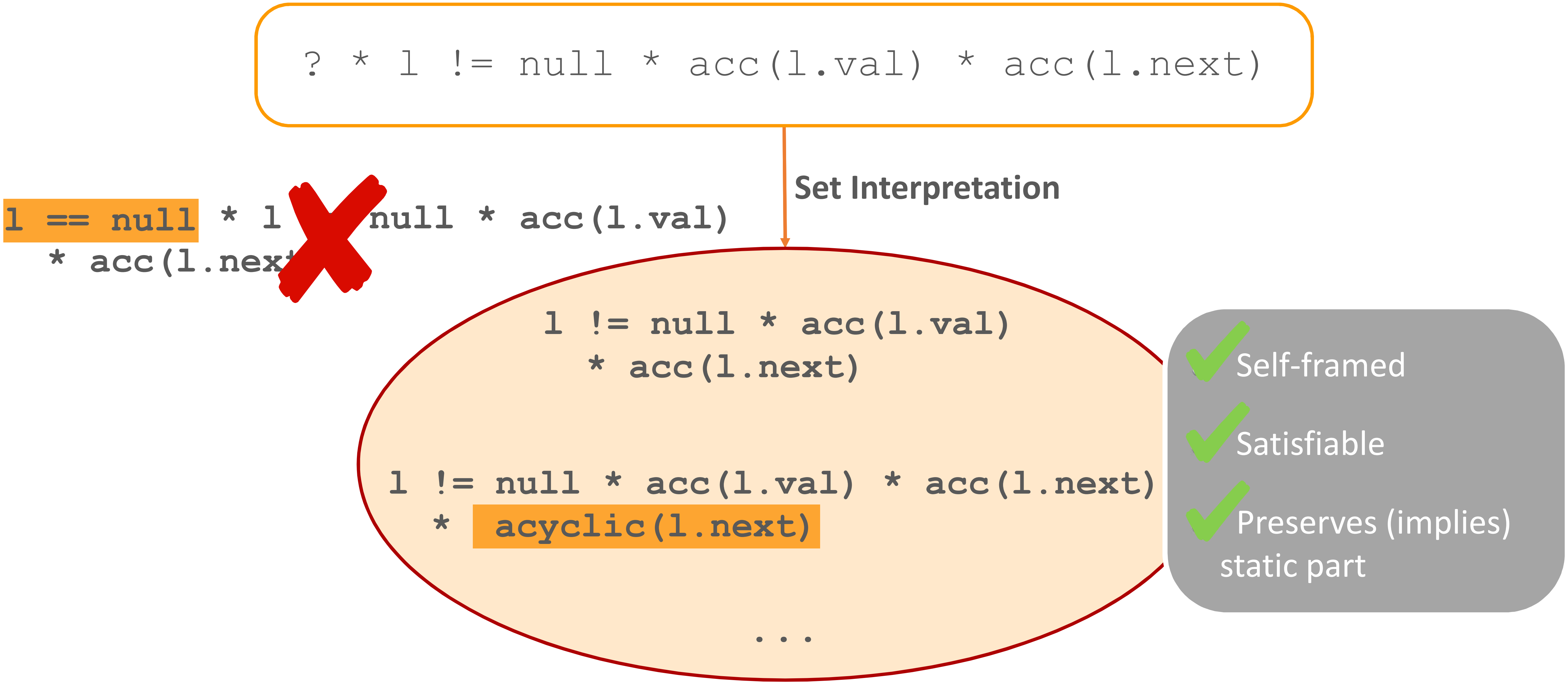
Meaning of Imprecise Formulas: By Example

```
predicate acyclic(Node root) =
  if root == null then true else acc(root.val)
    * acc(root.next) * acyclic(root.next)
```

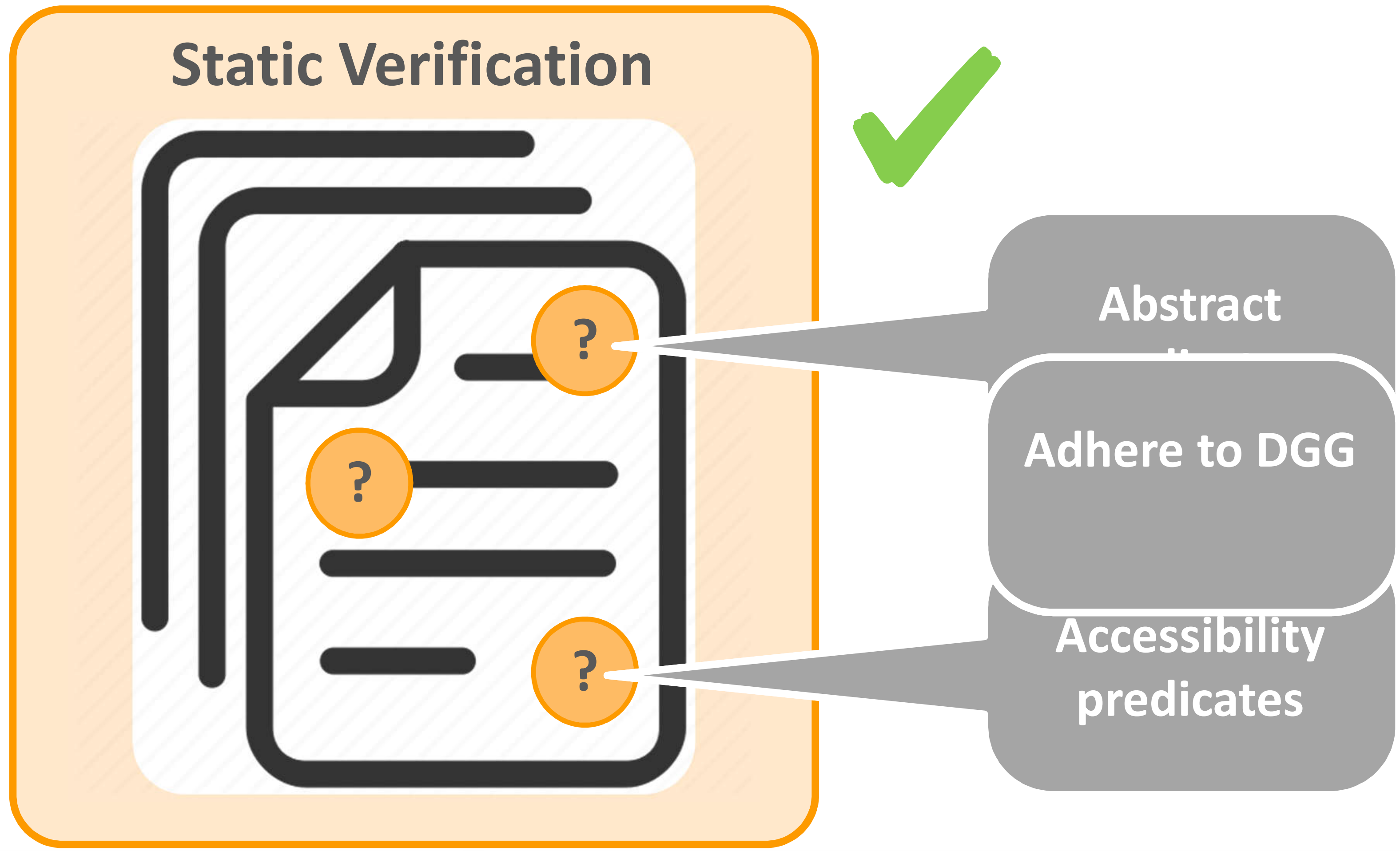
? gives
acyclic(l.next)

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

Meaning of Imprecise Formulas: By Example



2. Run-time checking



Dynamic gradual guarantee – reducing the precision of specifications does not change the runtime system's observable behavior for a verified program

Dynamically Verifying Predicates

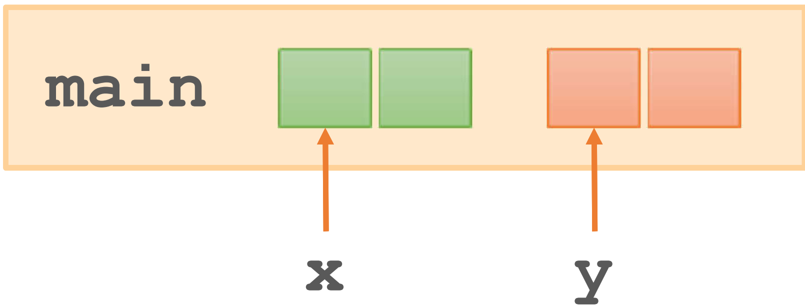
```
predicate acyclic(Node root) =
  if root == null then true else acc(root.val)
    * acc(root.next) * acyclic(root.next)
```

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

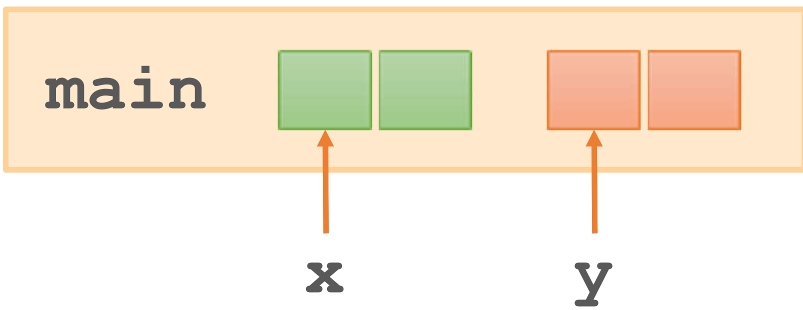
? gives
`acyclic(l.next)`

Equi-recursive

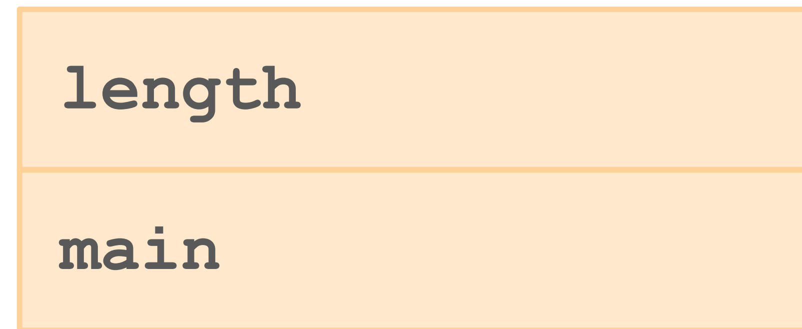
Dynamically Verifying Accessibility Predicates



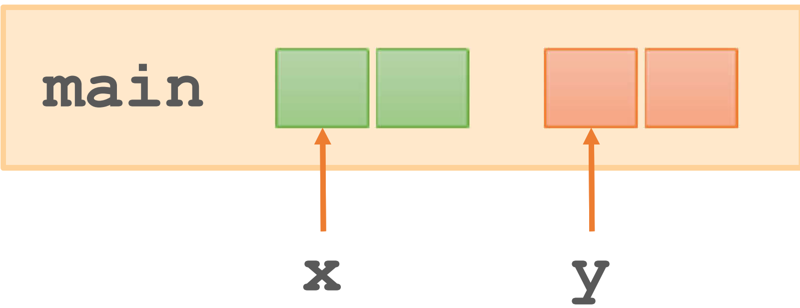
Dynamically Verifying Accessibility Predicates



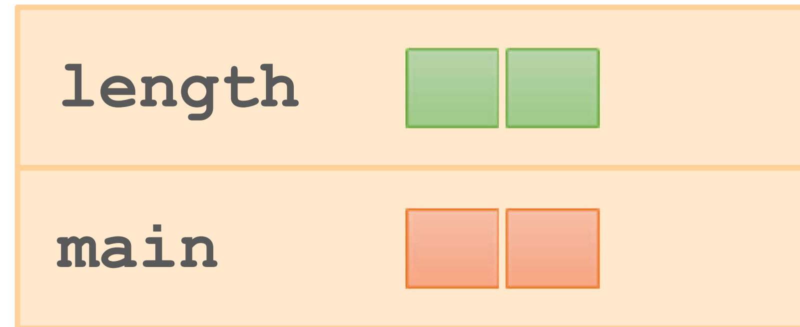

```
length(Node x)
  requires
    acyclic(x)
```



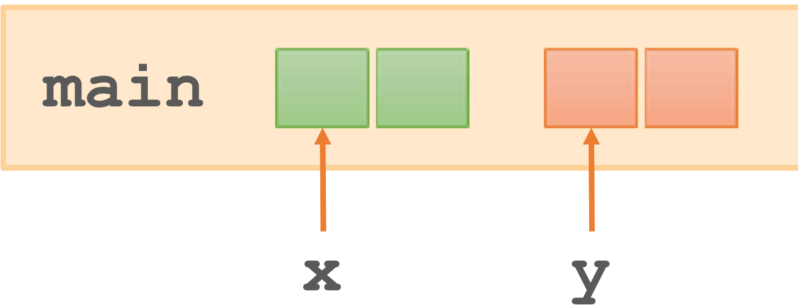
Dynamically Verifying Accessibility Predicates



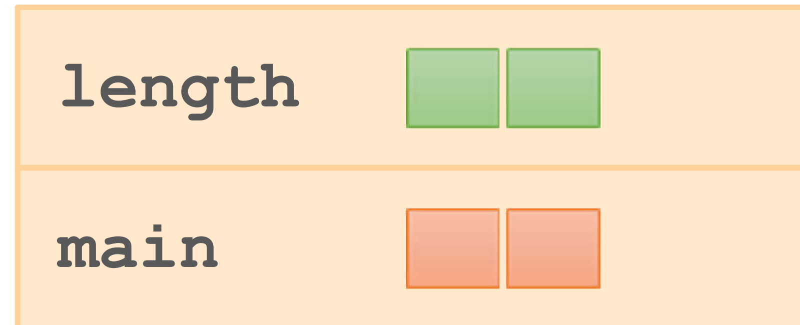

```
length(Node x)
  requires
    acyclic(x)
```



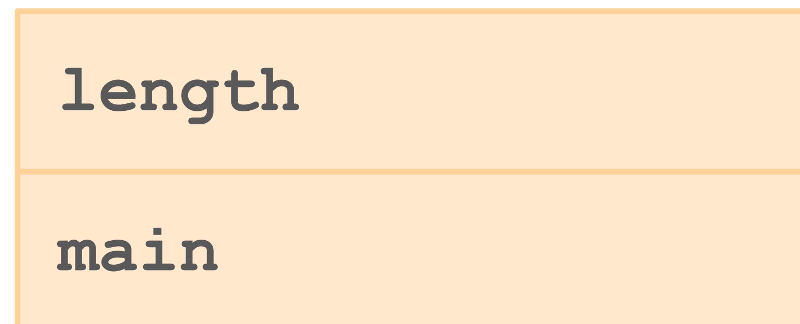
Dynamically Verifying Accessibility Predicates



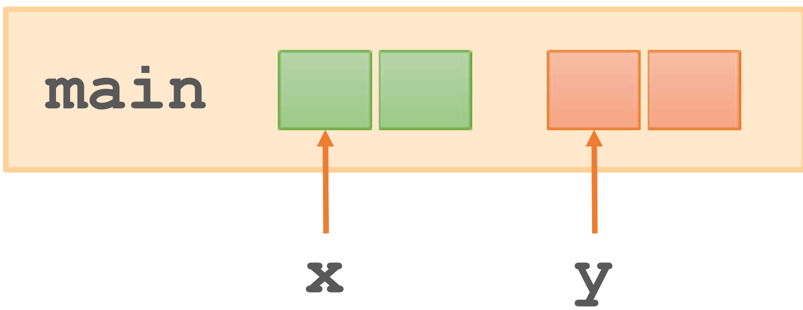

```
length(Node x)
  requires
    acyclic(x)
```



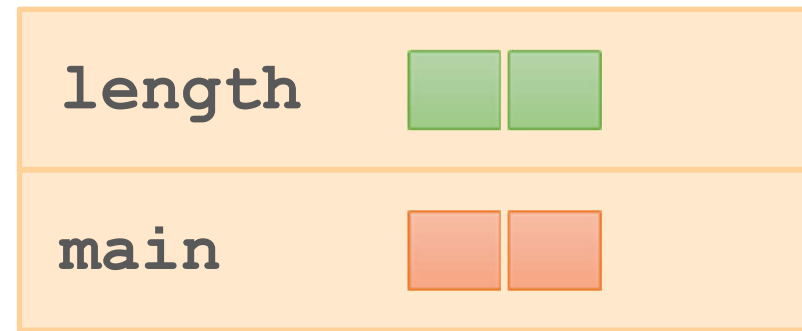

```
length(Node x)
  requires ?
```



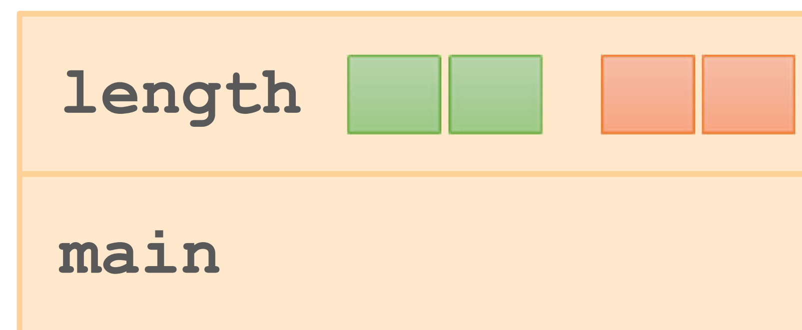
Dynamically Verifying Accessibility Predicates




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




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



Incremental static verification is made possible with Gradual Verification

Challenges

1. Semantics of imprecise formulas

Solution: Any precise formula that is
- Self-framed
- Satisfiable
- Implies static part

2. Consistency between static & run-time checks

Solution:
- Acc preds: ownership set
- Abstract preds: equi-recursively

Current & Future Work

- Prototype implementation
- Formative user studies
- Performance studies
- Summative user studies