Evaluating Soundness of a Gradual Verifier with Property Based Testing

Jan-Paul Ramos-Dávila
Cornell University

Software and Societal Systems Department

What is Gradual Verification?

Static verification techniques do not provide good support for incrementality.

Dynamic verification approaches cannot provide static guarantees.

Gradual verification bridges this gap, supporting incrementality by allowing the user to specify a given program as much as they want, with a **formal guarantee of verifiability**.

The *gradual guarantee* states that verifiability and reducibility are **monotone with respect to precision**.

```
//Gradual C0 Example
void withdraw(Account* account)
//@requires acc(account→balance) &&
account→balance ⩾ 5
//@ensures acc(account→balance) &&
account→balance ⩾ 0
{
  account→balance -= 5;
}

// ? allows the verifier to assume anything
necessary to satisfy the withdraw precon
void withdraw(Account* account)
//@requires ? && acc(account→balance);
//@ensures ? && acc(account→balance) &&
account→balance ⩾ 0
{
  if(account→balance ⩽ 100)
    withdraw(account);
}
```

Why Property Based Testing?

Gradual C0's design has been proven sound and will catch all violations of a specification.

A number of bugs were **caught and fixed by hand**, which *Gradual C0*'s design was implemented incorrectly.

There are **no techniques** available to ensure the implementation of *Gradual C0* is correct!

Capturing the *truthiness* of a property's result **provides good coverage** for finding these implementations bugs.

By Example

```
-     if (x < v) {
+     if (v < x) {
        if (l ≠ NULL) {
          root→left = tree_add_helper(l, x, min, v-1);
        } else {
          root→left = create_tree_helper(x, min, v-1);
        }
      } else {
-     if (v < x) {
+     if (x < v) {
        if (r ≠ NULL) {
          root→right = tree_add_helper(r, x, v+1, max);
        } else {
          root→right = create_tree_helper(x, v+1, max);
        }
      }
    }
```

To prevent a trivial failure in a *Gradual C0* program, we must **avoid specifying preconditions and fold/unfolds** that won't be met while running.

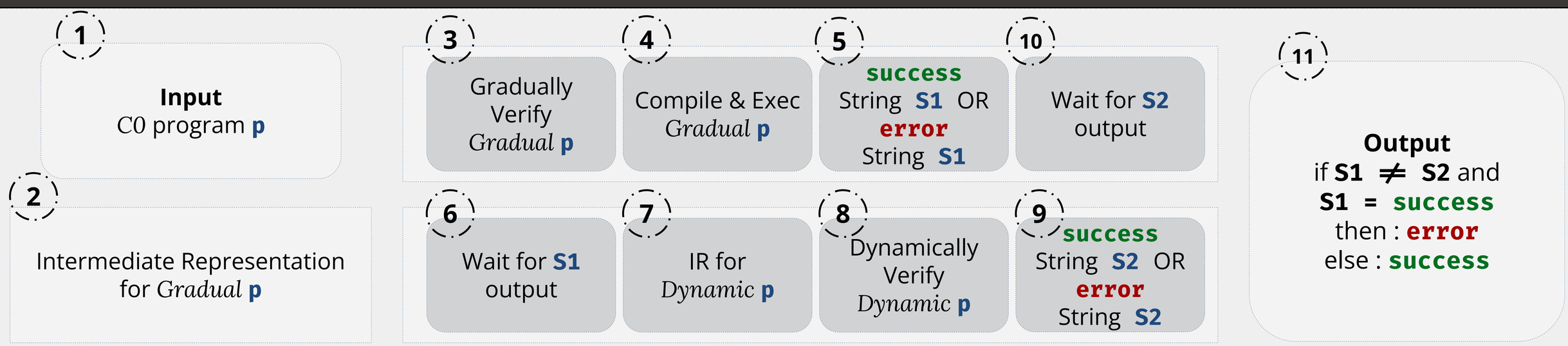
One of the main *Gradual C0* programs in our test suite is a *Binary Search Tree*.

First, we want to **break the tree order**, the left subtree has to be less than the right subtree. Therefore, we insert a node which is greater in the left hand side of the tree.

In *Gradual C0*, the *truthiness* for all programs consists of a pair of outputs: *dynamic* and *gradual verification* output message given by *Dynamic C0* and *Gradual C0* respectively.

Failed equivalence between the pair of outputs informs us of bugs in *Gradual C0*'s implementation that **do not break** the *gradual guarantee* and **would not have been caught otherwise**.

Soundness Evaluation *Checker* Architecture



Three-stage pipeline

1. **Reference model language** uses *Gradual C0*'s specifications

2. Input Generator is a test suite of examples that are not supposed to verify correctly. Ideally we randomly permute to test on.

3.1 Checker: Dynamic C0
Gradual C0 program that asserts runtime checks everywhere: **The ground truth**

3.2 Checker: Gradual C0
Gradual C0 is compared with *Dynamic C0*, expecting an error or a pass.

Test suite

```
/*@
predicate list(struct Node *l) =
  ? && (l ≠ NULL ? acc(l→value) &&
list(l→next) : true);
@*/
void append(Node *root, int value)
  //@requires ?;
  //@ensures ? && list(root);
  {
    Node *n = root;
    while (n→next ≠ NULL)
      //@loop_invariant ?;
      n = n→next;
    n→next = alloc(Node);
    n→next→value = value;
}
```

We caught **4 soundness bugs** at different implementation phases of *Gradual C0*.

Here's a soundness issue that was **identified** with our tool.