Optimization of a Gradual Verifier: Lazy evaluation of *Iso-recursive* Predicates as *Equi-recursive* at Runtime

Jan-Paul Ramos-Dávila
Cornell University

Gradual C0

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*.

Background

```
struct Node { int val ; struct Node * next ; };
typedef struct Node Node ;

//@ predicate acyclic(Node* root) = ?;

Node * insertLast ( Node * list , int val )
//@ requires ?;
//@ ensures acyclic(\result);
{
  //@ unfold acyclic(list);
  Node * y = list ;
  while (y -> next != NULL )
  //@ loop_invariant ? && y != NULL;
  { y = y -> next ; }
  y -> next = alloc ( struct Node );
  y -> next -> val = val ;
  y -> next -> next = NULL ;
  //@ fold acyclic(list);
  return list ;
}
```

Iso vs Equi

Iso-recursive predicates are isomorphic to their unfolding, and the isomorphism corresponds to folds/unfolds (highlight to the left.) We never have the problem of *not knowing how deep to unroll*!

Equi-recursive predicates are equal to their unfolding, therefore treating them as their complete unfolding.

Gradual C0 uses *iso-recursion* for static checking and *equi-recursion* for dynamic checking.

Optimizing Runtime Assertions

Before Opt.

```
assert(_1 - node->leftHeight < 2);
assert(node->leftHeight >= 0);
avlh(node->right, _1, _ownedFields);
avlh(_, node->leftHeight, _ownedFields);
```

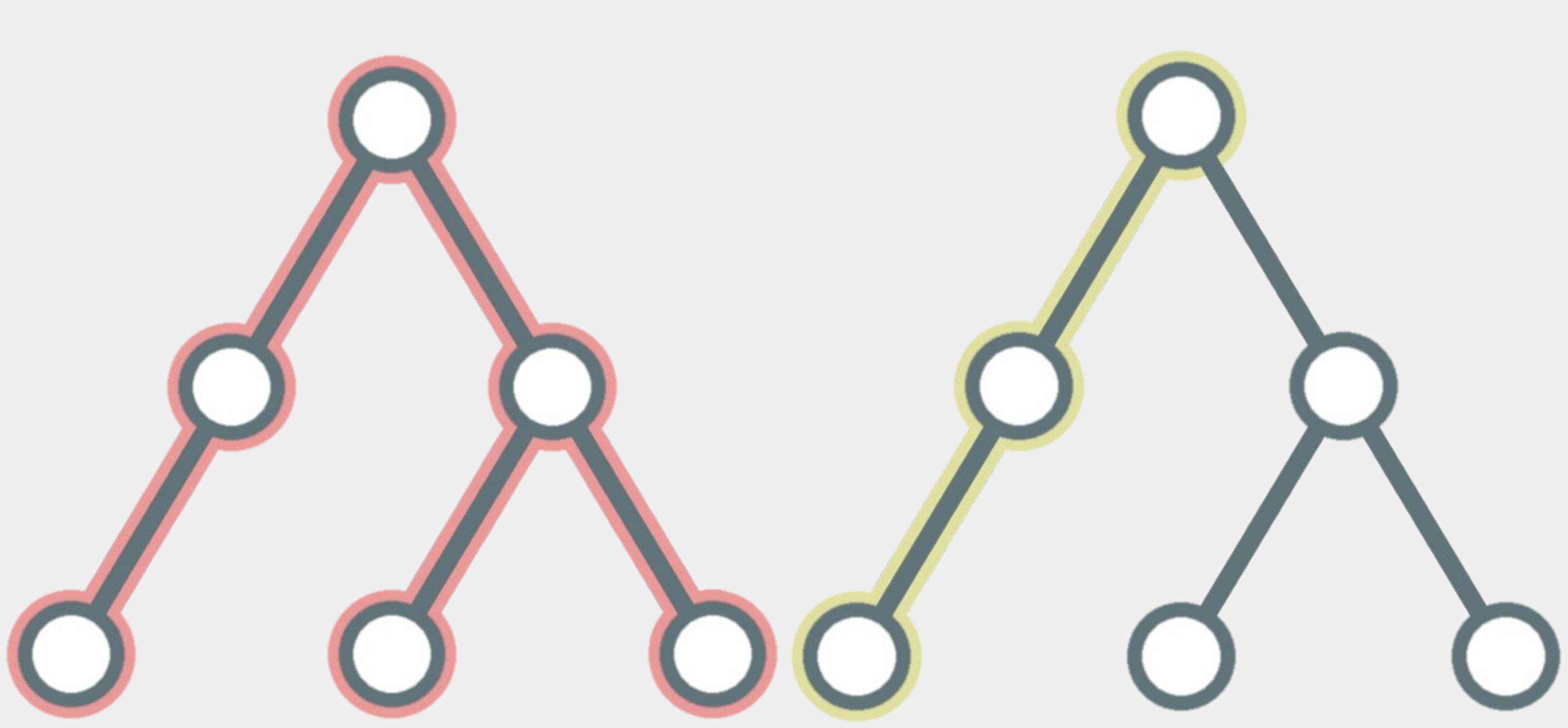
After Opt.

```
 assert(_1 - node->leftHeight < 2);
 assert(node->leftHeight >= 0);
 if (_ == node->right && _1 == node->leftHeight) {
   avlh(node->right->left, node->right->leftHeight)
   avlh(node->right->node->right,
          node->right->node->rightHeight)
   assert(node->leftHeight - node->rightHeight < 2)
   assert(node->rightHeight - node->leftHeight < 2)
   assert(node->leftHeight >= 0)
   assert(node->rightHeight >= 0)
   assert(root->leftHeight > root->rightHeight ?
          height1 == root->leftHeight+1 :
          height1 == root->rightHeight+1))
  } else {
    avlh(node->right, _1, _ownedFields);
    avlh(_, node->leftHeight, _ownedFields);
  }
```

At the *introduction of imprecise specifications* with static information we get *naive runtime checks* which re-assert the same logic from a predicate.

A *common pattern* for writing gradual specifications seems to be to *specify the post-condition but keep an imprecise pre-condition* (as in the code above).

While the verifier asserts *iso-recursive* predicates, there is a a side effect of *equivalent checks* for the predicate logic!



The **red tree** shows the program logic during the first iteration of the recursive call, whereas the **green tree** is the second iteration.

Runtime checks should *only verify the side of the tree which changes*, not the entire tree.

Pipeline

Slice Construction

Predicates are gathered and unfolded to **1-depth** if they exhibit recursive behavior.

Equivalence Identification

Keeps track of the path condition; identify which conditions overlap and discard. **Z3 SMT** solver identifies a more sophisticated identification.

Runtime Assertions

As detailed in the code to the left, **insert the unfolded predicates** into the verified code body.

Unbounded Recursion

Future work would implement an **equivalent loop transformation** algorithm for identifying the minimum required unfolds.