The functional essence of imperative binary search trees

Anton Lorenzen, Daan Leijen, Sam Lindley and *Wouter Swierstra*

Edinburgh, MSR and Utrecht

Pure functional programming

Compositional programs - each assembled from other functions that can be tested and verified independently.

There are a variety of *elementary* techniques for verifying that a program is correct:

- testing automatically;
- writing pen and paper proofs;
- developing a formal proofs using proof assistants.

Pure functional programming

Compositional programs - each assembled from other functions that can be tested and verified independently.

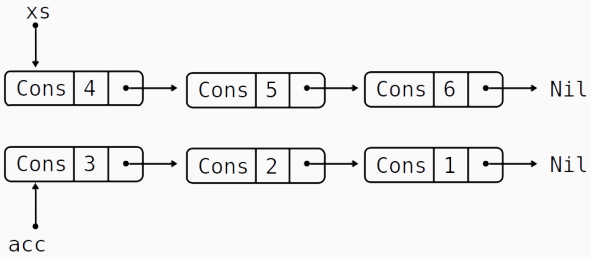
There are a variety of *elementary* techniques for verifying that a program is correct:

- testing automatically;
- writing pen and paper proofs;
- developing a formal proofs using proof assistants.

But these programs may allocate and deallocate more *memory* than their imperative counterparts.

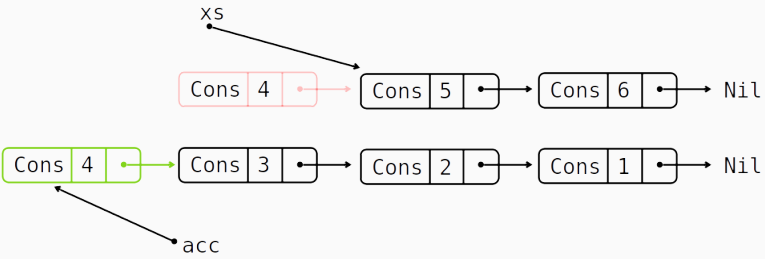
Functional reverse

```
reverseAcc :: List a → List a → List a
reverseAcc Nil         acc = acc
reverseAcc (Cons x xs) acc = reverseAcc xs (Cons x acc)
```



Towards in-place functional programming

```
reverseAcc :: List a → List a → List a
reverseAcc Nil         acc = acc
reverseAcc (Cons x xs) acc = reverseAcc xs (Cons x acc)
```



In-place functional programming

```
reverseAcc :: List a → List a → List a
reverseAcc Nil         acc = acc
reverseAcc (Cons x xs) acc = reverseAcc xs (Cons x acc)
```

The `reverseAcc` function has a few important properties:

- we can see that we are matching on one `Cons` cell on the left;
- and allocating one new `Cons` cell on the right.
- all other variables (like `x` or `acc`) are used linearly, i.e. they are not copied or discarded.

We will call such programs *fully in-place* – or fip for short.

Making this more precise..

$$\Gamma ::= \varnothing \mid \Gamma, x \mid \Gamma, \diamond_k \quad \text{(owned environment)}$$
$$\Delta ::= \varnothing \mid \Delta, y \quad \text{(borrowed environment)}$$

$$\frac{}{\Delta \mid x \vdash x}\ \textsc{var} \qquad \frac{}{\Delta \mid \varnothing \vdash C}\ \textsc{atom}$$

$$\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)}\ \textsc{tuple} \qquad \frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_k, \diamond_k \vdash C^k\ v_1 \ldots v_k}\ \textsc{reuse}$$

$$\frac{\overline{y} \in \Delta, \mathrm{dom}(\Sigma) \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash f(\overline{y}; e)}\ \textsc{call} \qquad \frac{\Delta, \Gamma_2 \mid \Gamma_1 \vdash e_1 \quad \Delta \mid \Gamma_2, \Gamma_3, \overline{x} \vdash e_2 \quad \overline{x} \notin \Delta, \Gamma_2, \Gamma_3}{\Delta \mid \Gamma_1, \Gamma_2, \Gamma_3 \vdash \text{let } \overline{x} = e_1 \text{ in } e_2}\ \textsc{let}$$

$$\frac{y \in \Delta \quad \Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma \vdash y\ e}\ \textsc{bapp} \qquad \frac{y \in \Delta \quad \Delta, \overline{x}_i \mid \Gamma \vdash e_i \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma \vdash \text{match } y \{ C_i\ \overline{x}_i \mapsto e_i \}}\ \textsc{bmatch}$$

$$\frac{\Delta \mid \Gamma \vdash e}{\Delta \mid \Gamma, \diamond_0 \vdash e}\ \textsc{empty} \qquad \frac{\Delta \mid \Gamma, \overline{x}_i, \diamond_k \vdash e_i \quad k = |\overline{x}_i| \quad \overline{x}_i \notin \Delta, \Gamma}{\Delta \mid \Gamma, x \vdash \text{match! } x \{ C_i\ \overline{x}_i \mapsto e_i \}}\ \textsc{dmatch!}$$

$$\frac{}{\Vdash \varnothing}\ \textsc{defbase} \qquad \frac{\Vdash \Sigma' \quad \overline{y} \mid \overline{x} \vdash e}{\Vdash \Sigma', f(\overline{y}; \overline{x}) = e}\ \textsc{deffun}$$

Fig. 4. Well-formed FIP expressions, where the multiplicity of each variable in Γ is 1.

In-place reverse in Koka

These rules (and corresponding *memory reuse*) have been implemented in the **Koka** compiler.

Re-implementing the `reverse` function in Koka becomes:

```
fip fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match xs
    Cons(x,xx) -> reverse-acc( xx, Cons(x,acc) )
    Nil        -> acc
```

The **`fip`** keyword indicates that a function can be executed *fully in place*.

The compiler checks that each function with the `fip` annotation can be executed without (de)allocating memory.

The compiled code simply reverses the pointers in a linked list.

Beyond list reversal

List reversal is not so interesting - what about algorithms for *trees*?

```
type tree
  Node( left : tree, key : int, right : tree )
  Leaf
```

Let's look at algorithms on *binary search trees*.

In particular, *restructuring* binary search trees, where accessing an element restructures the tree.

Beyond list reversal

List reversal is not so interesting - what about algorithms for *trees*?

```
type tree
  Node( left : tree, key : int, right : tree )
  Leaf
```

Let's look at algorithms on *binary search trees*.

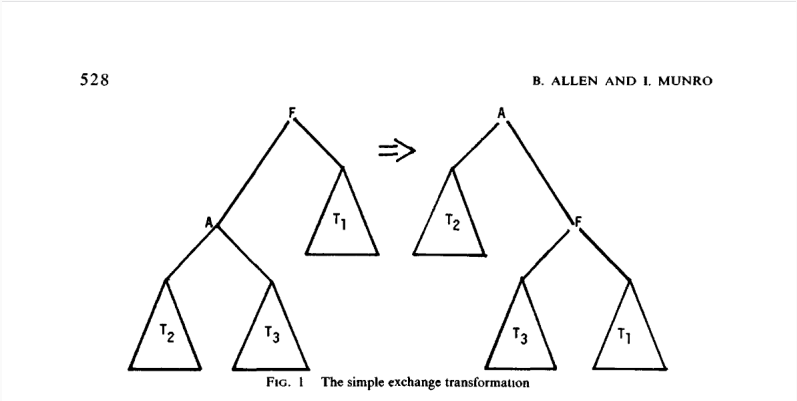
In particular, *restructuring* binary search trees, where accessing an element restructures the tree.

We aim to define an access function:

```
fun access-spec (t : tree, k : key) : tree
  Node (smaller(t,k), k, bigger(t,k))
```

which inserts the key, if it is not yet present, and moves it to the root if it is.

Rotation (Allen & Munro, 1976)




```
fip fun rotateRight (t : tree) : tree
  match t
    (Node (Node (t2, a, t3), f, t1) -> Node (t2, a, Node (t3, f, t1))
```

Move to root trees

Allen & Munroe suggest *repeated* rotations, ensuring the key being looked up is moved to the root of the new binary tree.

In this fashion, frequently accessed elements naturally ‘bubble up’ to the root of the tree.

Can we give a purely functional—yet in place—algorithm?

Warm up ; look up

```
fun lookup (k : int, t : tree)
  match t with
    Node (l, x, r) -> if k == x
                      then x
                      else if k < x
                        then lookup (k, l)
                        else lookup (k, r)
```

This is not in place: we only ever recurse on one subtree.

Warm up ; look up

```
fun lookup (k : int, t : tree)
  match t with
    Node (l, x, r) -> if k == x
                      then x
                      else if k < x
                        then lookup (k, l)
                        else lookup (k, r)
```

This is not in place: we only ever recurse on one subtree.

Key idea

Search through the tree recursively, accumulating the unvisited subtrees on a ‘stack’.

Once we find the element, unwind the ‘stack’ to rebuild the new tree, rotating as we move back up.

Stacks

A simple stack of subtrees will not work – to reconstruct a binary search tree we need to record if we went left or right!

```
type zipper
  Done
  // we went left; the tree stores bigger elements than the key being accessed
  Left(up : zipper, x : int, right : tree )
  // we went right; the tree stores smaller elements than the key being accessed
  Right(left : tree, x : int, up : zipper )
```

Stacks

A simple stack of subtrees will not work – to reconstruct a binary search tree we need to record if we went left or right!

```
type zipper
  Done
  // we went left; the tree stores bigger elements than the key being accessed
  Left(up : zipper, x : int, right : tree )
  // we went right; the tree stores smaller elements than the key being accessed
  Right(left : tree, x : int, up : zipper )
```

Using these zippers we can construct a pair of functions:

```
// search through the tree for the given key
fun access (key : int, t : tree, z : zipper) : (tree, zipper)
// rebuild the entire tree, rotating as necessary
fun rebuild (t : tree, z : zipper) : tree
```

Accessing a given key

```
fip(1) fun access(t : tree, k : int, z : zipper)
  match t
    Node(l,x,r) -> if x == k then rebuild(z, Node(l,k,r))
                   else if k < x then access(l, k, Left(z,x,r))
                   else access(r, k, Right(l,x,z))
    Leaf        -> rebuild(z, Node(Leaf,k,Leaf) )
```

The access function has the same structure as the (more familiar) `lookup` function.

If the key is already present, the tree is restructured; otherwise the new key is inserted.

The function is in-place, but may do at most one allocation.

What is missing?

We need to rebuild the complete tree - from our stack of subtrees:

```
fip fun rebuild(z : zipper, t : tree )
  match z
    Done            -> t
    Right(l,x,z) -> match t // we went right looking for k
      Node(s,k,b) -> rebuild(z, Node( Node(l,x,s), k, b))
    Left(z,x,r) -> match t // we went left looking for k
      Node(s,k,b) -> rebuild(z, Node( s, k, Node(b,x,r)))
```

Now we rebuild the entire tree, popping elements off the zipper.

In each case, we rotate the tree as required to ensure the result remains a binary search tree.

From this definition, it is easy to see that the key k stays at the root – just as we wanted!

What about the imperative implementations?

What about the imperative implementations?

```
Definition heap_mtr_insert_td : val :=
  fun: ( name, root ) {
    var: left_dummy := #0 in
    var: right_dummy := #0 in
    var: node := root in
    var: left_hook := &left_dummy in
    var: right_hook := &right_dummy in
    while: ( true ) {
      if: ( node != #0) {
        if: ( node->value == name ) {

          *left_hook = node->left;;
          *right_hook = node->right;;
          root = node;;
          break
        }
        else {
          if: ( node->value > name )
          {

            *right_hook = node;;
            right_hook = &(node->left);;
            node = node->left
          }

          else
          {
            *left_hook = node;;
            left_hook = &(node->right);;
            node = node->right
      } } }
      else {
        *left_hook = #0;;
        *right_hook = #0;;
        root = AllocN #3 #0;;
        root->value = name;;
        break
      }
    };;
    root->left = left_dummy;;
    root->right = right_dummy;;
    ret: root
  }.
```

```
node := root;
left_hook := addr(left(dummy));
right_hook := addr(right(dummy));

while node ≠ null do
    if value(node) = name then
        begin
            0(left_hook) := left(node);
            0(right_hook) := right(node);
            root := node;
            go to bottom
        end;
    if value(node) > name then
        begin
            0(right_hook) := node;
            right_hook := addr(left(node));
            node := left(node)
        end
    else
        begin
            0(left_hook) := node;
            left_hook := addr(right(node));
            node := right(node)
        end;

0(left_hook) := null;
0(right_hook) := null;
root := new_node ( );
value(root) := name;

bottom:
    left(root) := left(dummy);
    right(root) := right(dummy)
```

Fig. 1. The move-to-root top-down algorithm formalized in AddressC on the left, versus a screenshot of Stephenson's published algorithm on the right

Verifying imperative version

Using a proof assistant, we have given a formal proof of correctness of both top-down and bottom-up move to root trees.

That is, we can prove a Hoare triple of (roughly) the following form:

```
Lemma heap_mtr_insert_td_correct (k : key) (p : ptr) (t : tree) :
    { is_tree t p }
    heap_mtr_insert_td k p
    { is_tree (mtr_insert_td k t) p }.
```

In this way, we prove that the functional version (`mtr_insert_td`) coincides precisely with the (published) imperative algorithms (`heap_mtr_insert_td`).

Verifying the imperative version

- These proofs are non-trivial! If you've ever tried to write out a formal proof in Hoare logic for any program longer than 5 lines, you know there is a lot of bookkeeping involved.

Verifying the imperative version

- These proofs are non-trivial! If you've ever tried to write out a formal proof in Hoare logic for any program longer than 5 lines, you know there is a lot of bookkeeping involved.
- Theorem proving technology (Coq and Iris) are fairly impressive - the proof is about 50 loc, half of which is formulating the loop invariant.

Verifying the imperative version

- These proofs are non-trivial! If you've ever tried to write out a formal proof in Hoare logic for any program longer than 5 lines, you know there is a lot of bookkeeping involved.
- Theorem proving technology (Coq and Iris) are fairly impressive - the proof is about 50 loc, half of which is formulating the loop invariant.
- The functional implementation captures the key parts of the specification – it is essential for spelling out the loop invariant.

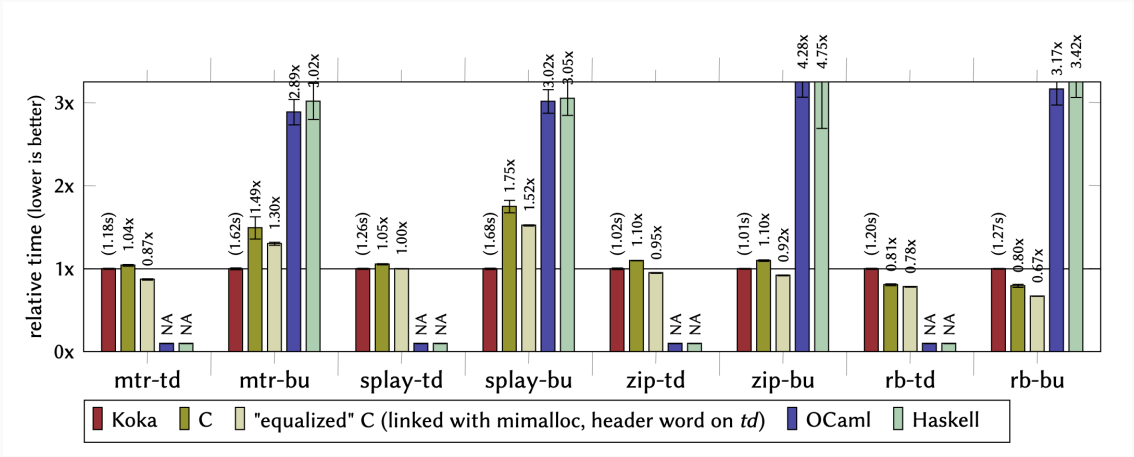
Verifying the imperative version

- These proofs are non-trivial! If you've ever tried to write out a formal proof in Hoare logic for any program longer than 5 lines, you know there is a lot of bookkeeping involved.
- Theorem proving technology (Coq and Iris) are fairly impressive - the proof is about 50 loc, half of which is formulating the loop invariant.
- The functional implementation captures the key parts of the specification – it is essential for spelling out the loop invariant.
- This is (to the best of our knowledge) the first formal proof of these algorithms.

What else?

- The zipper-based implementation can be derived by *program calculation* from the directly recursive definition.
- Besides this bottom-up version, we have a top-down variant version;
- These techniques also extend to more advanced data structures – including *splay trees* (Sleator and Tarjan 1985) and *zip trees* (Tarjan et al. 2021).

Benchmarks



10M pseudorandom insertions of a key between 0 and 100.000 on an initially empty tree.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Just as finding a *tail recursive* function yields a program that can be executed in constant *stack space*, finding a *fip* function yields a program that can be executed in constant *heap space*.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Just as finding a *tail recursive* function yields a program that can be executed in constant *stack space*, finding a *fip* function yields a program that can be executed in constant *heap space*.
- Elementary verification techniques only! Structural induction on trees suffices.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Just as finding a *tail recursive* function yields a program that can be executed in constant *stack space*, finding a *fip* function yields a program that can be executed in constant *heap space*.
- Elementary verification techniques only! Structural induction on trees suffices.
- Modern proof assistants can relate the functional and imperative versions - the functional algorithm captures the essence of the loop invariant.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Just as finding a *tail recursive* function yields a program that can be executed in constant *stack space*, finding a *fip* function yields a program that can be executed in constant *heap space*.
- Elementary verification techniques only! Structural induction on trees suffices.
- Modern proof assistants can relate the functional and imperative versions - the functional algorithm captures the essence of the loop invariant.
- But if you need to write the functional version to verify imperative code – why write imperative programs to begin with?

Questions?