Purely Functional, Fully in-Place Programming

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

Edinburgh, MSR and Utrecht

Pure functional programming

- Programming without assignment statements

Pure functional programming

- Programming without assignment statements
- Recursive functions rather than loops

Pure functional programming

- Programming without assignment statements
- Recursive functions rather than loops
- Algebraic data types rather than classes and objects

Pure functional programming

- Programming without assignment statements
- Recursive functions rather than loops
- Algebraic data types rather than classes and objects

Each function *only* depends on its inputs - there is no state or memory being mutated.

Pure functional programming

- Programming without assignment statements
- Recursive functions rather than loops
- Algebraic data types rather than classes and objects

Each function *only* depends on its inputs - there is no state or memory being mutated.

In the famous paper, *Why functional programming matters* Hughes (1984) writes:

> *The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.*

Analogy: goto's considered harmful

In the 1960's-1970's, the idea of *structured programming* emerged.

By avoiding 'spaghetti code' using jumps and goto's, the control flow of a progam becomes more predictable.

Reasoning about structured programs becomes easier: a large program can be broken into smaller pieces that are assembled in a predictable fashion.

Analogy: goto's considered harmful

In the 1960's-1970's, the idea of *structured programming* emerged.

By avoiding 'spaghetti code' using jumps and goto's, the control flow of a progam becomes more predictable.

Reasoning about structured programs becomes easier: a large program can be broken into smaller pieces that are assembled in a predictable fashion.

In purely functional programs, these pieces are *independent* - there is no (implicit) memory or state connecting them.

Functional programming 101 : list reversal

```haskell
-- 'linked lists' as an algebraic data type
data List = Nil | Cons Int List

reverse :: List -> List
reverse xs = reverseAcc xs Nil
  where
  reverseAcc :: List -> List -> List
  reverseAcc Nil         acc = acc
  reverseAcc (Cons x xs) acc = reverseAcc xs (Cons x acc)
```

Testing functional programs

```
reverseProperty :: List -> Bool
reverseProperty xs = reverse (reverse xs) == xs
```

Property-based testing libraries like *QuickCheck* generate inputs for our reverse function, trying to falsify the property we have formulated:

```
> quickCheck reverseProperty
```

Testing functional programs

```
reverseProperty :: List -> Bool
reverseProperty xs = reverse (reverse xs) == xs
```

Property-based testing libraries like *QuickCheck* generate inputs for our reverse function, trying to falsify the property we have formulated:

```
> quickCheck reverseProperty

OK, passed 100 tests.
```

Reasoning about functional programs

Theorem ∀ xs , reverse (reverse xs) = xs

Proof *Base case* (when xs is Nil)

reverse (reverse Nil)

{ *by definition of* reverse }

= reverseAcc (reverseAcc Nil Nil) Nil

{ *by definition of* reverseAcc }

= reverseAcc Nil Nil

{ *by definition of* reverseAcc }

= Nil

Inductive case (when xs is of the form Cons y ys)...

Proof assistants

We can even formalise such proofs in a *proof assistant* (such as Coq, Agda, Lean, or others) that checks our reasoning:

```
Lemma rev_involutive : forall xs, reverse (reverse xs) = xs.
Proof.
  induction xs; [ reflexivity |  simpl in *; auto].
  rewrite (rev_unit (rev xs) a).
  now rewrite IHxs.
Qed.
```

Pure functional programming

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

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

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

How does this compare to list reversal in an imperative language?

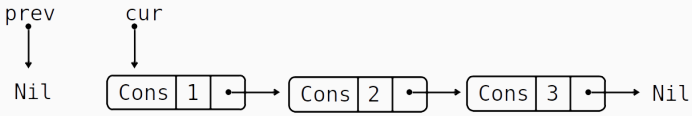
Linked list reversal in C

```
list_t* reverse( list_t* curr ) {
  list_t* prev = NULL;
  while(curr ≠ NULL) {
    list_t* next = curr->tail;
    curr->tail = prev;
    prev = curr;
    curr = next; }
  return prev; }
```

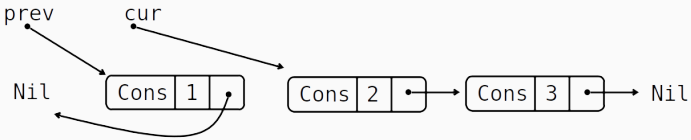
Each cell in the list stores a value and a pointer to the remaining lists.

To reverse the list, we update the pointer in each cell to point to the *previous* element, until there are no cells left.

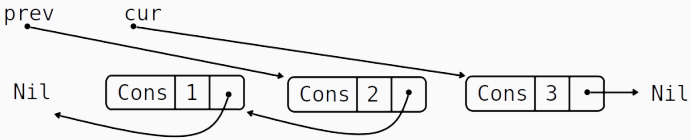
In-place execution



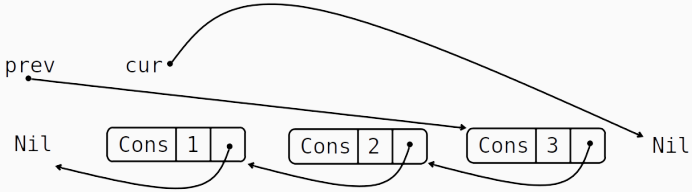
In-place execution



In-place execution



In-place execution



Correctness?

Giving a *formal proof* that such a program is correct is **very hard**.

- The Hoare logic typically used to reason about imperative code is unsuitable: what if the list structure in memory has cycles? Or if the two pointers map to lists sharing the same memory locations? How do we find the *loop invariant* guaranteeing the correctness of reversal? What *variant* guarantees termination?
- Extensions of Hoare logic, notably flavours of separation logic, have had a lot of success - but these methods are certainly not elementary.

Correctness?

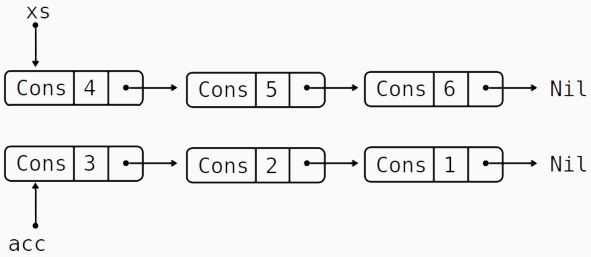
Giving a *formal proof* that such a program is correct is **very hard**.

- The Hoare logic typically used to reason about imperative code is unsuitable: what if the list structure in memory has cycles? Or if the two pointers map to lists sharing the same memory locations? How do we find the *loop invariant* guaranteeing the correctness of reversal? What *variant* guarantees termination?
- Extensions of Hoare logic, notably flavours of separation logic, have had a lot of success - but these methods are certainly not elementary.

But the C algorithm has an important property: it is executed *in-place* - it does not need to allocate new memory or deallocate unused memory.

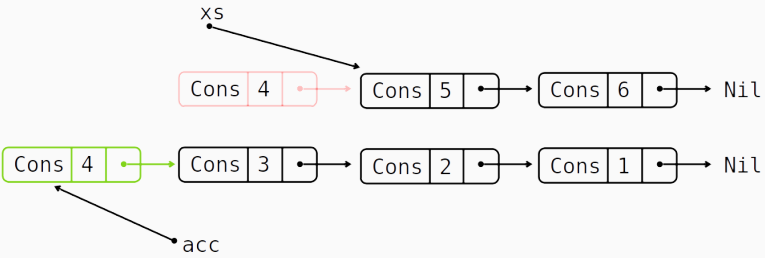
Functional reverse

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



Functional reverse

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



The trade offs...

Despite the apparent virtues of functional programming, *memory management matters*.

The trade offs...

Despite the apparent virtues of functional programming, *memory management matters*.

Now reversing a linked list is not that exciting...

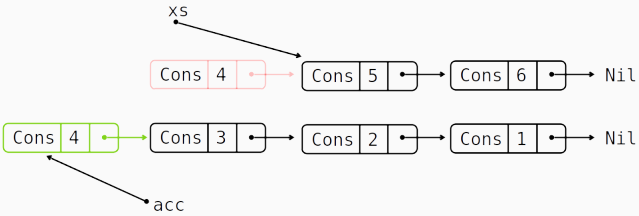
But many datastructures – such as *splay trees* or *red black trees* – use careful pointer management to restructure/rebalance the tree.

Overly liberal allocation & garbage creation has a real performance impact.

Towards in-place functional programming

Let's revisit our reversal function.

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



Here we see that we are *both* allocating a new Cons cell *and* creating a new 'garbage' Cons cell *in each recursive step*.

In-place functional programming

```
reverseAcc :: List → List → List
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 shared.

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

Making this more precise..

$$\begin{array}{lcll} \Gamma & ::= & \varnothing \mid \Gamma, x \mid \Gamma, \diamond_k & \text{(owned environment)} \\ \Delta & ::= & \varnothing \mid \Delta, y & \text{(borrowed environment)} \end{array}$$

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

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

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

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

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

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

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 FIP function can be executed in constant stack space, without allocating or deallocating memory.

This gives great *performance*, while still writing purely functional programs.

Beyond list reversal

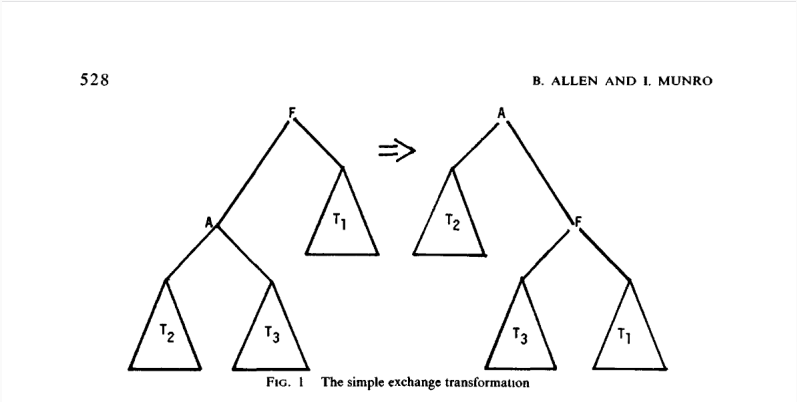
List reversal is not so interesting - what about algorithms for *trees*?

```
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.

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: lookup

The `lookup` function for a binary search tree should be familiar – even if we write it in a functional language:

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

Warm-up: lookup

The `lookup` function for a binary search tree should be familiar – even if we write it in a functional language:

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

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 (at the root).

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

In-place?

```
fip fun access (...)
    Node(l,x,r) -> if ... then access(l, k, Left(z,x,r))
```

Why is this in-place? We match on a node, but extend our zipper?

In-place?

```
fip fun access (...)
    Node(l,x,r) -> if ... then access(l, k, Left(z,x,r))
```

Why is this in-place? We match on a node, but extend our zipper?

Memory re-use is *not* restricted to the same constructors or even the same types!

Instead, we only check that the variables are not duplicated or discarded;

And that the *sizes* of deallocations and allocations line up.

Reconstructing the tree

```
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!

Formal verification

With a bit more work, we can (and have!) written machine checked proofs that the `access` function:

- does not discard elements from the input tree;
- returns a binary search tree, when passed a binary search tree;

Formal verification

With a bit more work, we can (and have!) written machine checked proofs that the `access` function:

- does not discard elements from the input tree;
- returns a binary search tree, when passed a binary search tree;

Furthermore, the `fip` annotations ensure that it execute in-place.

Zippers?

The name ‘zipper’ is taken from existing literature in functional programming (Huet 1997).

A pair of a zipper and tree, allows you to ‘move focus’ to a child, without loss of information:

```
fip fun left (t : tree, z : zipper) : (tree, zipper)
  match t
    Node (l, x, r) -> (l, Left(z,x,r))
```

These are typically used for ‘local’ operations on binary trees; or visiting each node, as in the Deutsch-Schorr-Waite algorithm.

Zippers?

The name ‘zipper’ is taken from existing literature in functional programming (Huet 1997).

A pair of a zipper and tree, allows you to ‘move focus’ to a child, without loss of information:

```
fip fun left (t : tree, z : zipper) : (tree, zipper)
  match t
    Node (l, x, r) -> (l, Left(z,x,r))
```

These are typically used for ‘local’ operations on binary trees; or visiting each node, as in the Deutsch-Schorr-Waite algorithm.

> *Efficient destructive algorithms on binary trees may be programmed with these completely applicative primitives, which all use constant time, since they all reduce to local pointer manipulation.*

Using our fip calculus we can make this precise – and check this property statically!

(These zippers generalise very smoothly to *any* other algebraic datatypes.)

Bottom-up vs top-down algorithms

The approach described by Allen & Munro yields a *bottom-up* algorithm:

1. Find the key in the tree (the `access` function)
2. Work our way back up, rotating as necessary (the `rebuild` function)

Bottom-up vs top-down algorithms

The approach described by Allen & Munro yields a *bottom-up* algorithm:

1. Find the key in the tree (the `access` function)
2. Work our way back up, rotating as necessary (the `rebuild` function)

Alternatively, Stephenson (1980) defines a *top-down* algorithm for move-to-root trees.

This algorithm accumulates two (unfinished) trees, corresponding to the left and right children of the final root.

As we search for the desired key, we ‘enqueue’ new subtrees on these unfinished trees.

Bottom-up vs top-down algorithms

The approach described by Allen & Munro yields a *bottom-up* algorithm:

1. Find the key in the tree (the `access` function)
2. Work our way back up, rotating as necessary (the `rebuild` function)

Alternatively, Stephenson (1980) defines a *top-down* algorithm for move-to-root trees.

This algorithm accumulates two (unfinished) trees, corresponding to the left and right children of the final root.

As we search for the desired key, we 'enqueue' new subtrees on these unfinished trees.

A purely functional account for such algorithms requires some work – just as implementing a purely functional stack is easier than a queue.

What about the imperative pseudocode?

What about the imperative pseudocode?

```
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.
- The logics and theorem proving technology 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.
- The logics and theorem proving technology 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 – and forms the heart of 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.
- The logics and theorem proving technology 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 – and forms the heart of the loop invariant.
- This is (to the best of our knowledge) the first formal proof of these algorithms.

From move-to-root trees to splay trees

Move-to-root trees guarantee that a freshly accessed key always moves to the root.

But it does not do any *rebalancing* along the way...

From move-to-root trees to splay trees

Move-to-root trees guarantee that a freshly accessed key always moves to the root.

But it does not do any *rebalancing* along the way...

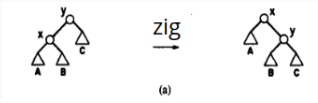
The more established *splay trees* (Sleator and Tarjan 1985) address precisely this issue.

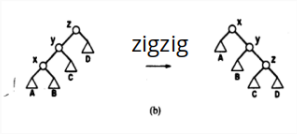
The key difference is in the `rebuild` function that tries to rebalance the resulting tree.

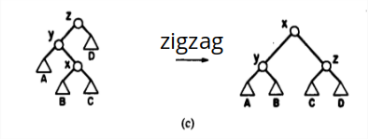
Rebuilding splay trees

Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN
AT&T Bell Laboratories, Murray Hill, NJ







Rebuilding splay trees

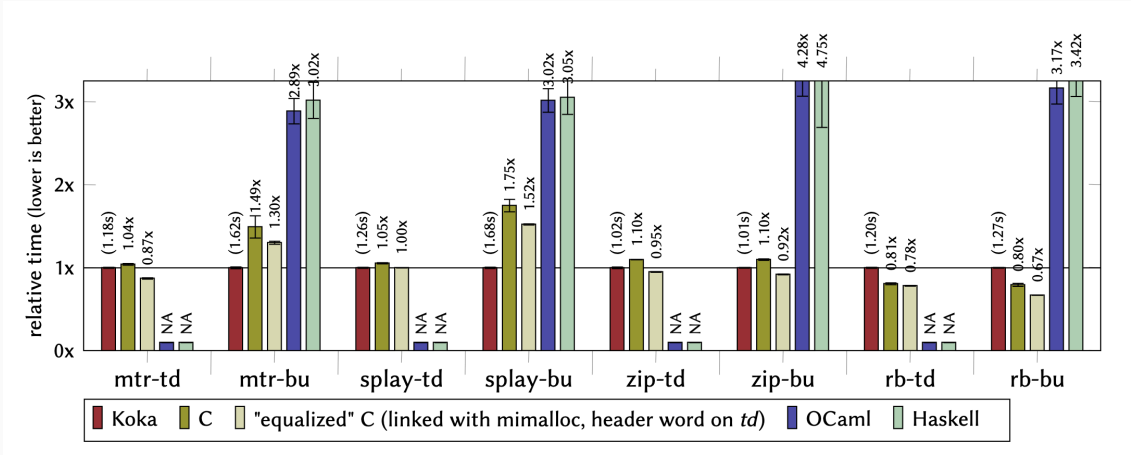
```
fip fun rebuild(z : zipper, t : tree ) : tree
  match tree
    Node(tl,tx,tr) -> match z
      Done -> Node(tl,tx,tr)
      Right(rl,rx,Done) -> Node(Node(rl,rx,tl),tx,tr)                                  // zig
      Left(Done,lx,lr) -> Node(tl,tx,Node(tr,lx,lr))
      Right(rl,rx,Right(l,x,up)) -> rebuild(up, Node(Node(Node(l,x,rl),rx,tl),tx,tr)) // zigzig
      Left(Right(l,x,up),lx,lr) -> rebuild(up, Node(Node(l,x,tl),tx,Node(tr,lx,lr)))
      Right(rl,rx,Left(up,x,r)) -> rebuild(up, Node(Node(rl,rx,tl),tx,Node(tr,x,r)))  // zigzag
      Left(Left(up,x,r),lx,lr) -> rebuild(up, Node(tl,tx,Node(tr,lx,Node(lr,x,r))))
```

Benchmarks - is it fast?

Comparing the same algorithm across different languages is always going to be unfair.

- Koka, Haskell, and OCaml have automatic memory management - as opposed to C;
- Haskell and OCaml use mark-and-sweep garbage collectors; Koka uses reference counting.
- Koka uses arbitrary precision integers for keys and all comparisons and arithmetic operations include branches for the case where big integer arithmetic is required;
- Haskell is lazy, most other functional languages are not.

Benchmarks



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

Beyond splay trees

- We have similar results for red-black trees and the more recent *ziptrees* (Tarjan, Levy & Timmel 2021) - in some cases, this leads to (slightly) improved algorithms over the best known implementations.
- Using similar ideas, we can write *in place* sorting algorithms in a functional style – including mergesort and quicksort.
- Okasaki's famous book on *Purely functional datastructures* is chock-full of fip algorithms...

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.
- Elementary verification techniques only! Structural induction on trees suffices.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Elementary verification techniques only! Structural induction on trees suffices.
- Modern proof assistants can relate the functional and imperative versions.

Conclusions

- Best-of-both worlds approach: purely functional programs with low memory usage.
- Elementary verification techniques only! Structural induction on trees suffices.
- Modern proof assistants can relate the functional and imperative versions.
- Performance is on par with best known implementations.

Questions?

When to execute in place?

Even if a function is fip, it is not *always* safe to execute it in place.

Consider the following example:

```
fun makePalindrome( xs : list<a>) : list<a>
  return (append(xs, reverse(xs))
```

Clearly this call to `reverse` should not run in place!

When to execute in place?

Even if a function is fip, it is not *always* safe to execute it in place.

Consider the following example:

```
fun makePalindrome( xs : list<a>) : list<a>
  return (append(xs, reverse(xs))
```

Clearly this call to `reverse` should not run in place!

When can we tell it is safe to execute fip functions in place?

Static vs dynamic checks

Koka uses a *reference counted garbage collection*.

As a result, we know at execution time how many references exist to any given value.

We can check if a reference is unique at run-time and re-use existing memory locations when possible:

```
fip fun reverse-acc( xs : list<a>, acc : list<a> ) : list<a>
  match xs
    Cons(x,xx) ->
      val addr = if is-unique(xs) then &xs else { dup(x); dup(xx); decref(xs); alloc(2) }
      reverse-acc( xx, Cons@addr(x,acc) )...
```

So what's our the papers

- In place versions of:
  - Splay trees;
  - Schorr Waite traversal of trees;
  - Generic map over any algebraic datatype;
  - Red black tree insertion;
  - Mergesort;
  - Finger tree insertions;
  - …
- Top-down & bottom-up implementations of:
  - move to root trees;
  - splay trees;
  - zip trees;
  - proofs that they 'are all equal';
  - proofs that the functional versions coincide with published imperative ones.
- Lots of metatheory, showing that it is *safe* to execute fip programs in place.