Utrecht University

The functional essence of imperative binary search trees

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

Edinburgh, MSR and Utrecht

Functional programming 101 : list reversal

```haskell
data List a = Nil | Cons a (List a)

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

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.

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

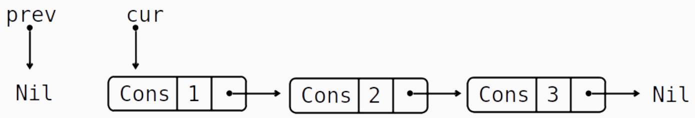
Linked list reversal in C

```
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; }
```

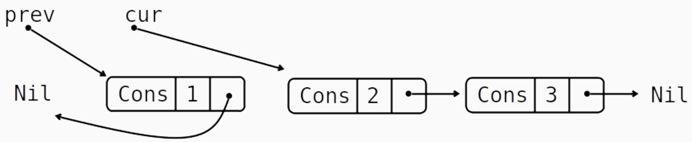
Each cell in the list stores a value and a pointer to the remaining lists.

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

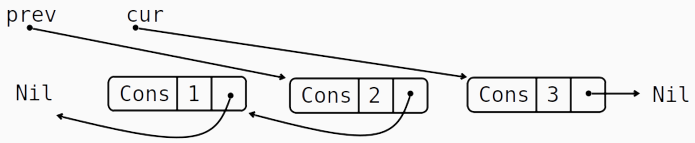
In-place execution



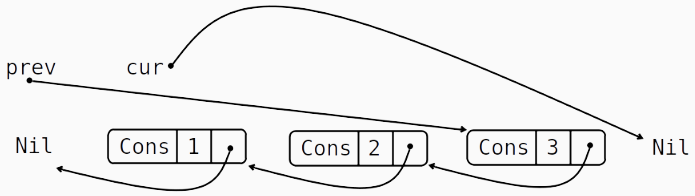
In-place execution



In-place execution



In-place execution



Correctness?

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

- Hoare logic is not suitable for reasoning about this kind of imperative code: what if the list structure in memory has cycles? Or if the two pointers map to lists sharing the same memory locations?
- Extensions of Hoare logic, notably separation logic, have had a lot of success - but these methods are certainly not elementary.

Correctness?

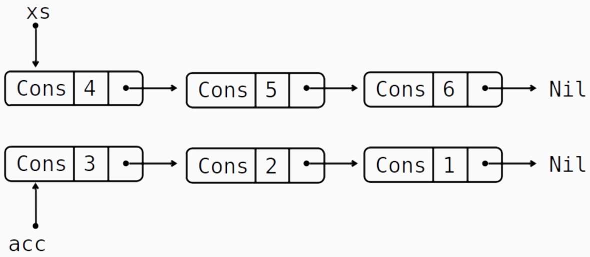
Giving a *formal proof* that such a program is correct is **very hard**.

- Hoare logic is not suitable for reasoning about this kind of imperative code: what if the list structure in memory has cycles? Or if the two pointers map to lists sharing the same memory locations?
- Extensions of Hoare logic, notably 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.

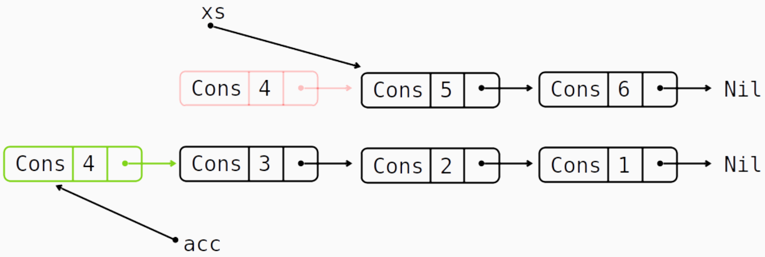
Functional reverse

```
reverseAcc :: List 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..

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

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

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

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

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

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

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

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.

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)

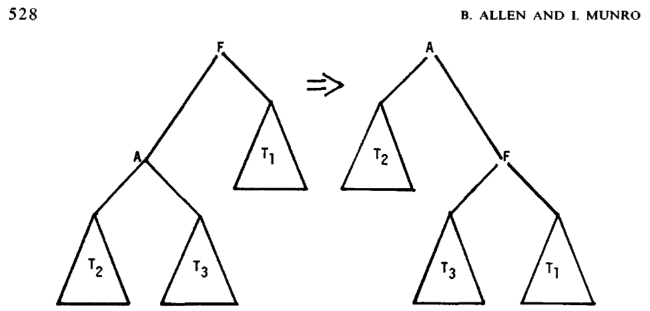


FIG. 1 The simple exchange transformation

```
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.

In-place reuse

```
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 reuse

```
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!

Zippers - defunctionalised continuations

How do you come up with code like this?

Zippers - defunctionalised continuations

How do you come up with code like this?

It is ‘just’ the tail recursive version arising from the defunctionalised CPS-transformed direct implementation that is in-place, but not tail recursive:

```
fun access( t : tree, k : key )
  match t
    Node(l,x,r) -> if x < k then match access(r,k) // lookup and rotate
                                 Node(s,y,b) -> Node( Node(l,x,s), y, b)
                   elif x > k then match access(l,k) // lookup and rotate
                                 Node(s,y,b) -> Node( s, y, Node(b,x,r))
                   else Node(l,k,r)
    Leaf -> Node(Leaf,k,Leaf)
```

And the implementation above follows calculationally from the original `access-spec`.

Towards *constructor contexts*

The zippers are used as a ‘stack of trees’ – we have immediate access to the tree we last added to the zipper.

But that’s not always what we want: many *top down* algorithms work by *accumulating* (unfinished) trees, extending the tree with new nodes at the fringe.

Towards *constructor contexts*

The zippers are used as a ‘stack of trees’ – we have immediate access to the tree we last added to the zipper.

But that’s not always what we want: many *top down* algorithms work by *accumulating* (unfinished) trees, extending the tree with new nodes at the fringe.

Consider the following fragment of code from the previous slide:

```
Node(l,x,r) ->
  if x < k then match access(r,k) // lookup and rotate
                  Node(s,y,b) -> Node( Node(l,x,s), y, b)
...
```

We *could* implement these unfinished trees as zippers, just as we *could* implement queues using lists...

Koka’s constructor contexts

Koka let’s you write *constructor contexts*, or elements of an algebraic datatypes with a single hole:

```
Node(Leaf, 3 , □)
```

Koka’s constructor contexts

Koka let’s you write *constructor contexts*, or elements of an algebraic datatypes with a single hole:

```
Node(Leaf, 3 , □)
```

There are two operations on these contexts:

```
fun (++) : ctx -> ctx -> ctx     // append contexts
fun (++.) : ctx -> tree -> tree  // fill in the hole
```

Tail recursive map

Using such contexts, we can write a tail recursive `map` function in a single pass:

```
fip fun map-td(xs : list<a>, f : a -> b, acc : list-ctx<b> ) : list<b>
  match xs
    Cons(x,xs) -> map-td( xx, f, acc ++ Cons(f(x), □) )
    Nil        -> acc ++. Nil
```

And start our map with an empty accumulator:

```
fip fun map(xs,f)
  map-td(xs, f, □)
```

This pattern – accumulating (partial) results in a constructor context – pops up again and again.

Implementing constructor contexts

We could implement constructor contexts using functions – similar to ‘difference lists’ – or zippers.

Implementing constructor contexts

We could implement constructor contexts using functions – similar to ‘difference lists’ – or zippers.

But instead, we represent them as ‘Minamide tuples’, allocating an additional pointer to the context’s hole.

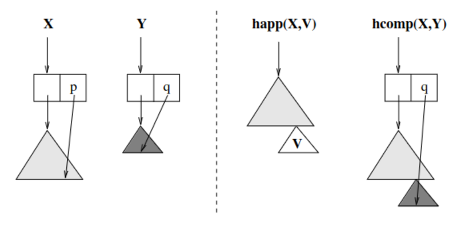


Figure 1: Application and Composition of Hole Abstractions

Then *all* operations on constructor contexts require constant time.

Towards top down move to root trees

Let’s take a closer look at the direct implementation:

```
Node(l,x,r) ->
  if x < k then match access(r,k) // lookup and rotate
                  Node(s,y,b) -> Node( Node(l,x,s), y, b)
...
```

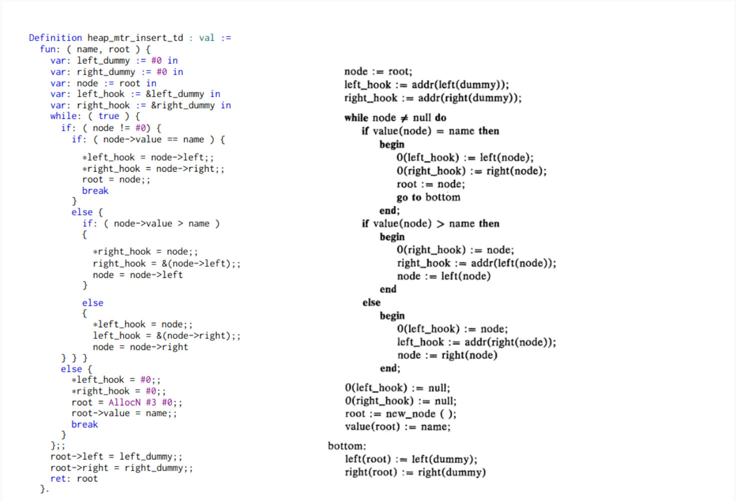
We can also phrase this in terms of two accumulating constructor contexts – the left and right subtrees – where we collect the parts of the trees we do not visit.

Top down move to root trees

```
fip(1) fun access-td(t : tree, k : key, accl : ctx, accr : ctx) : tree
  match t
    Node(l,x,r) ->
      if x < k then access-td( r, k, accl ++ Node(l,x,□), accr )
      elif x > k then access-td( l, k, accl, accr ++ Node(□,x,r) )
      else Node( accl ++. l, x, accr ++. r )
    Leaf -> Node( accl ++. Leaf, k, accr ++. Leaf)
```

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.

From move-to-root trees to splay trees to zip 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 to zip 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.

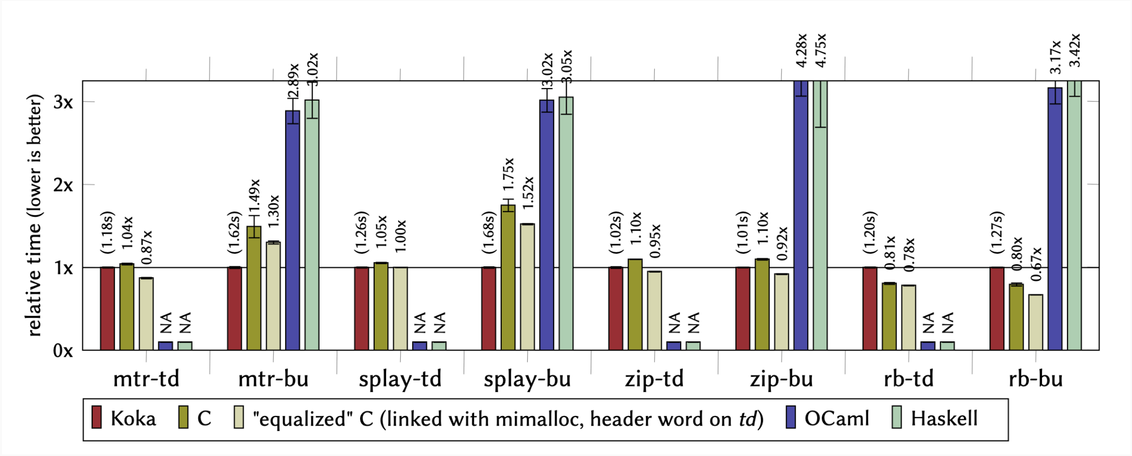
Similarly, the more recent *zip trees* (Tarjan et al. 2021) can be shown to have a fully in-place functional counterpart.

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.

Why?

From the algorithm description or imperative code, it can be quite hard to understand what splay trees or zip trees do – or why they work at all!

Why?

From the algorithm description or imperative code, it can be quite hard to understand what splay trees or zip trees do – or why they work at all!

We start with a simple functional specification, `access-spec`.

Why?

From the algorithm description or imperative code, it can be quite hard to understand what splay trees or zip trees do – or why they work at all!

We start with a simple functional specification, `access-spec`.

We write a ‘direct style’ recursive implementation.

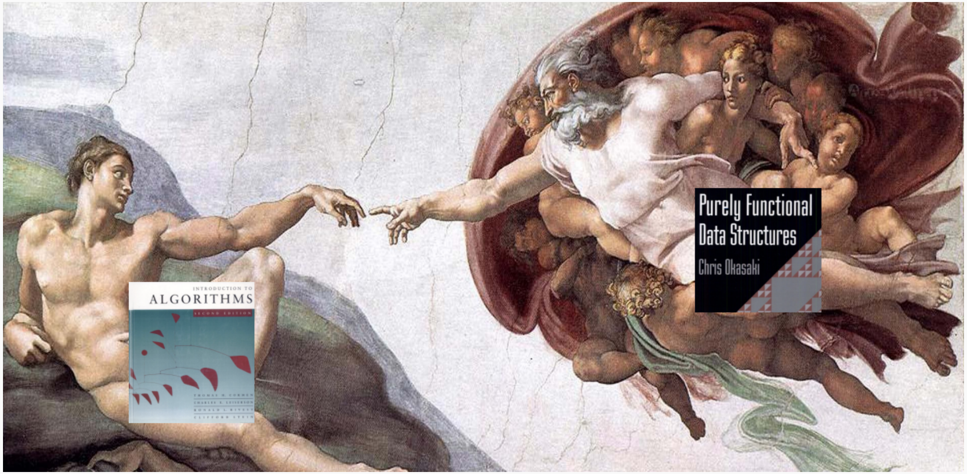
Why?

From the algorithm description or imperative code, it can be quite hard to understand what splay trees or zip trees do – or why they work at all!

We start with a simple functional specification, `access-spec`.

We write a 'direct style' recursive implementation.

Transforming this to fip functions is *guaranteed* to produce fast code.



Is there a way to combine the indulgences of impurity with the benefits of purity? — Phil Wadler [1990]

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.
- 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.
- Okasaki's famous book on *Purely functional datastructures* is chock-full of fip algorithms...
- 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.
- Okasaki's famous book on *Purely functional datastructures* is chock-full of fip algorithms…
- 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.
- Okasaki's famous book on *Purely functional datastructures* is chock-full of fip algorithms...
- 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.
- Okasaki's famous book on *Purely functional datastructures* is chock-full of fip algorithms...
- 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?

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) )...
```

Zippers

The original zipper paper describes a functional approach to navigating through a tree (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))
```

Zippers

The original zipper paper describes a functional approach to navigating through a tree (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))
```

Huet (1997) writes:

> *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!

Zipper-based traversal turn out to be very useful.

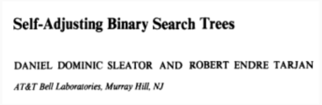
Schorr-Waite-Deutsch style `map` on trees

```
fip fun load (t : tree, f : int -> int, z : zipper) : tree
  match t
    Node (l, x, r) -> load(l,f,Left(z, f(x), r))
    Leaf           -> unload(z,f,Leaf)

fip fun unload (z : zipper, f : int -> int, t : tree) : tree
  match z
    Left (z,x,r)   -> load(r,f,Right(t,x,z))
    Right (l,x,z)  -> unload(z,f,Node(l,x,t))
    Done           -> t
```

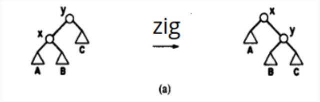
This traversal does not allocate any heap or stack space – yet works for any* data type!

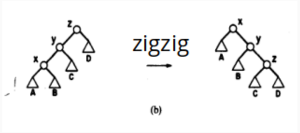
Rebuilding splay trees

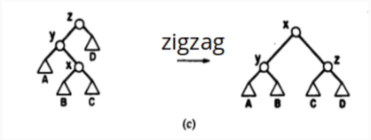

Self-Adjusting Binary Search Trees

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))))
```

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.