The Functional Essence of Imperative Binary Search Trees

INRIA, 2024-03-12

Anton Lorenzen

University of Edinburgh

Daan Leijen

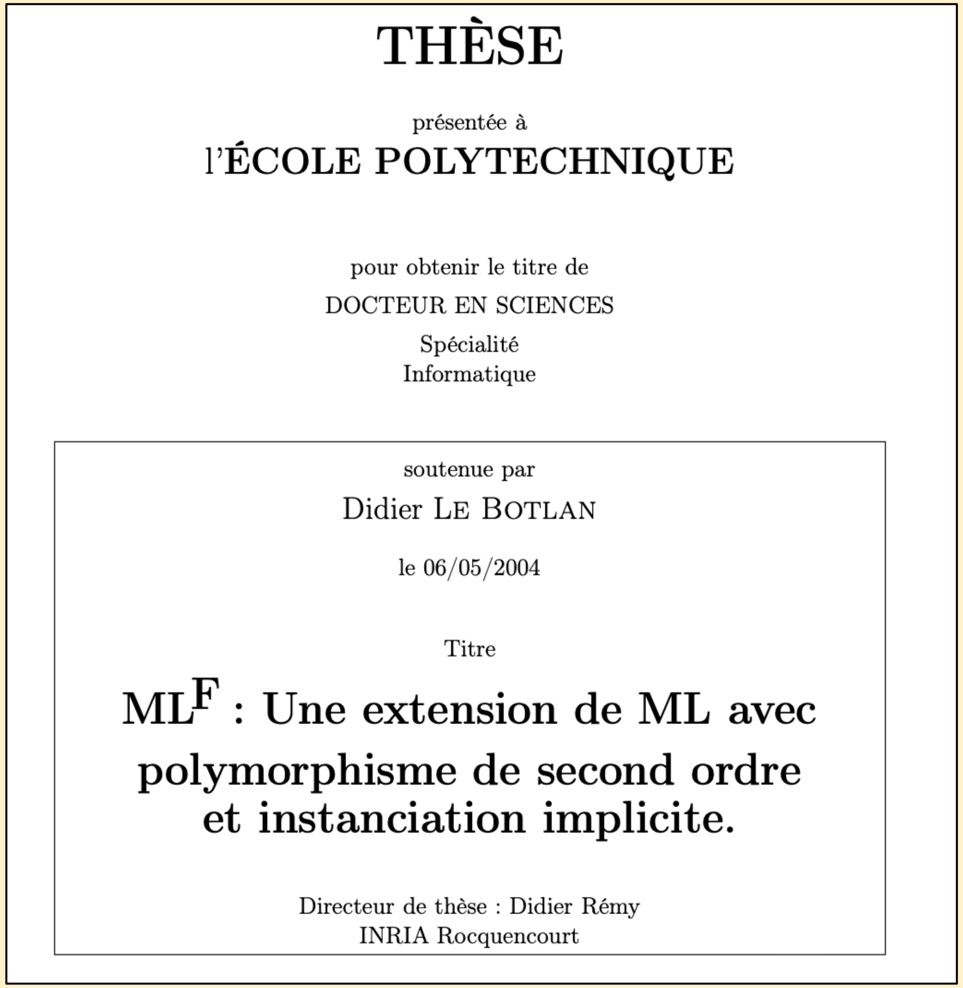
Microsoft Research

Wouter Swierstra

Universiteit Utrecht

Sam Lindley

University of Edinburgh



THÈSE

présentée à

l'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR EN SCIENCES

Spécialité
Informatique

soutenue par

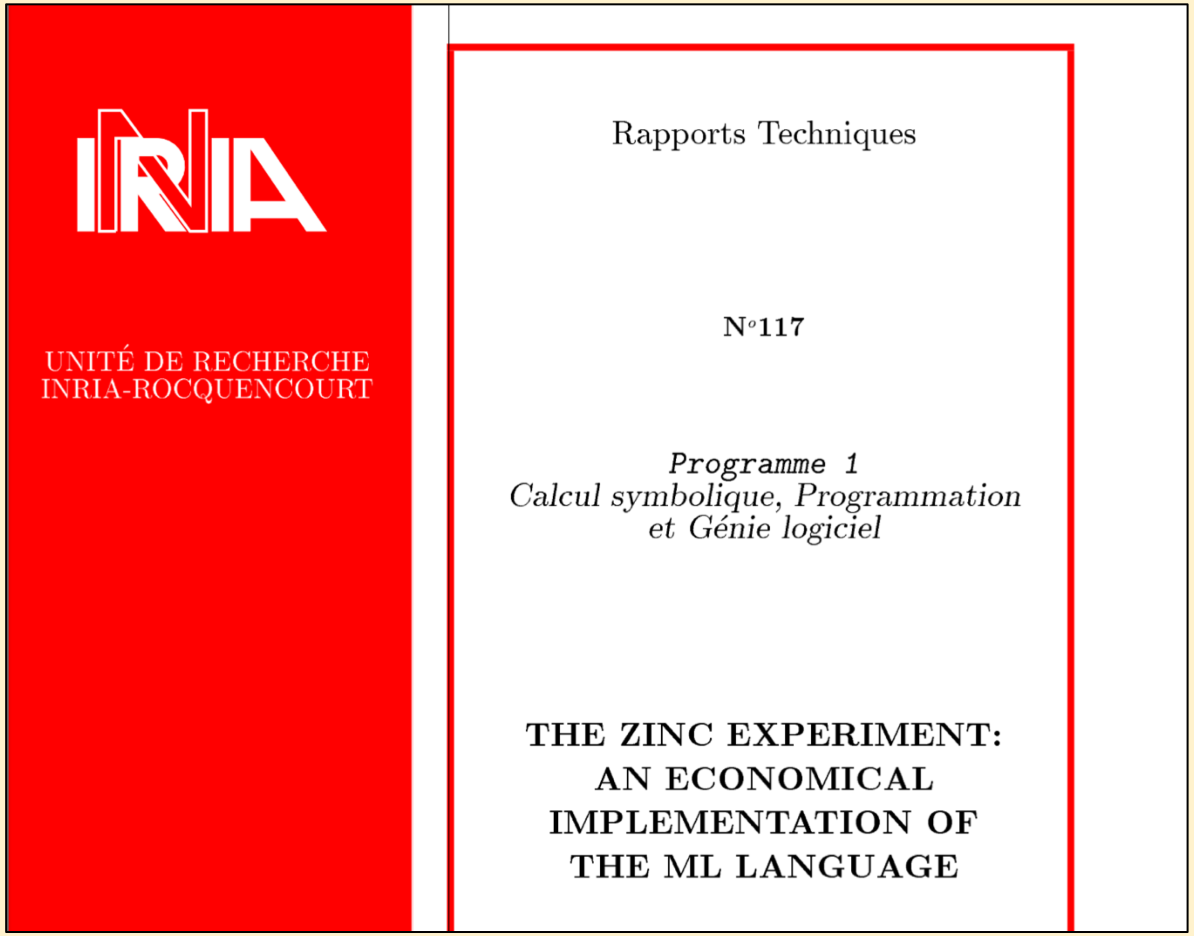
Didier Le Botlan

le 06/05/2004

Titre

ML^{F} : Une extension de ML avec polymorphisme de second ordre et instanciation implicite.

Directeur de thèse : Didier Rémy
INRIA Rocquencourt



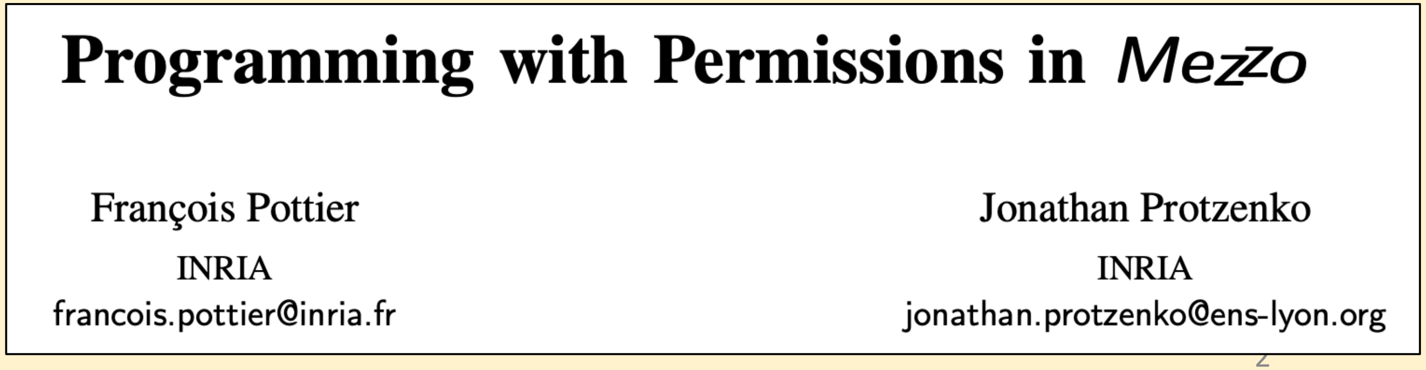
UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Rapports Techniques

N°117

Programme 1
Calcul symbolique, Programmation et Génie logiciel

THE ZINC EXPERIMENT: AN ECONOMICAL IMPLEMENTATION OF THE ML LANGUAGE



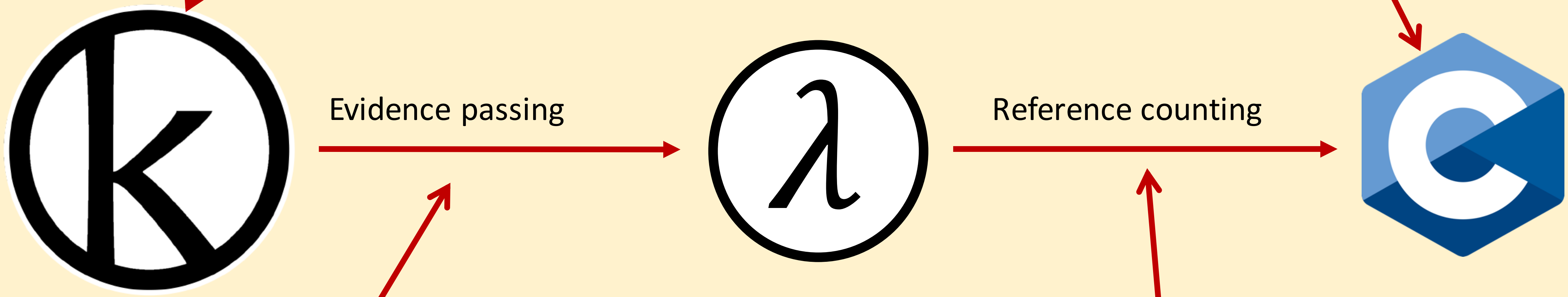
Programming with Permissions in *Mezzo*

François Pottier
INRIA
francois.pottier@inria.fr

Jonathan Protzenko
INRIA
jonathan.protzenko@ens-lyon.org

- Strict evaluation (like OCaml)
- Static types with Effect typing
- Algebraic Effect Handlers
- FIP (fully inplace programming)
- And much more (implicit parameters, etc)

Just using standard malloc/free.
No need for a GC, or root scanning,
or a special runtime system etc.

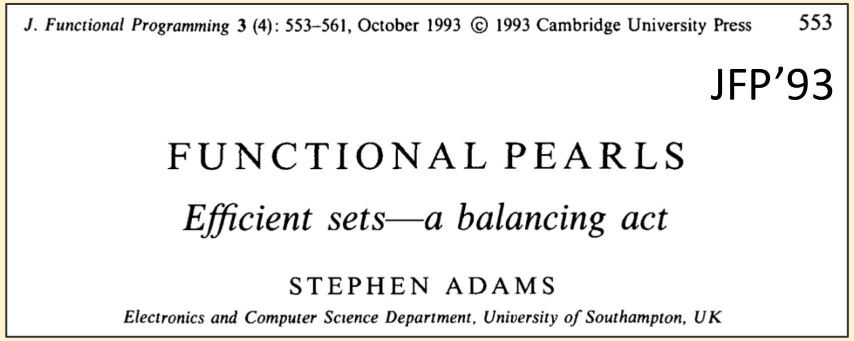


koka-lang.org

“Generalized Evidence Passing for Effect Handlers – Efficient Compilation of Effect Handlers to C”,
Ningning Xie and Daan Leijen, ICFP’21

“Perceus: Precise Reference Counting with Reuse and Specialization”, Reinking, Xie, de Moura, and Leijen, PLDI’21
“Reference counting with frame-limited reuse”,
Anton Lorenzen and Daan Leijen, ICFP’22

Data.Map in Haskell



J. Functional Programming **3** (4): 553–561, October 1993 © 1993 Cambridge University Press 553

JFP'93

FUNCTIONAL PEARLS

Efficient sets—a balancing act

STEPHEN ADAMS

Electronics and Computer Science Department, University of Southampton, UK

- Beautiful height-balanced trees:

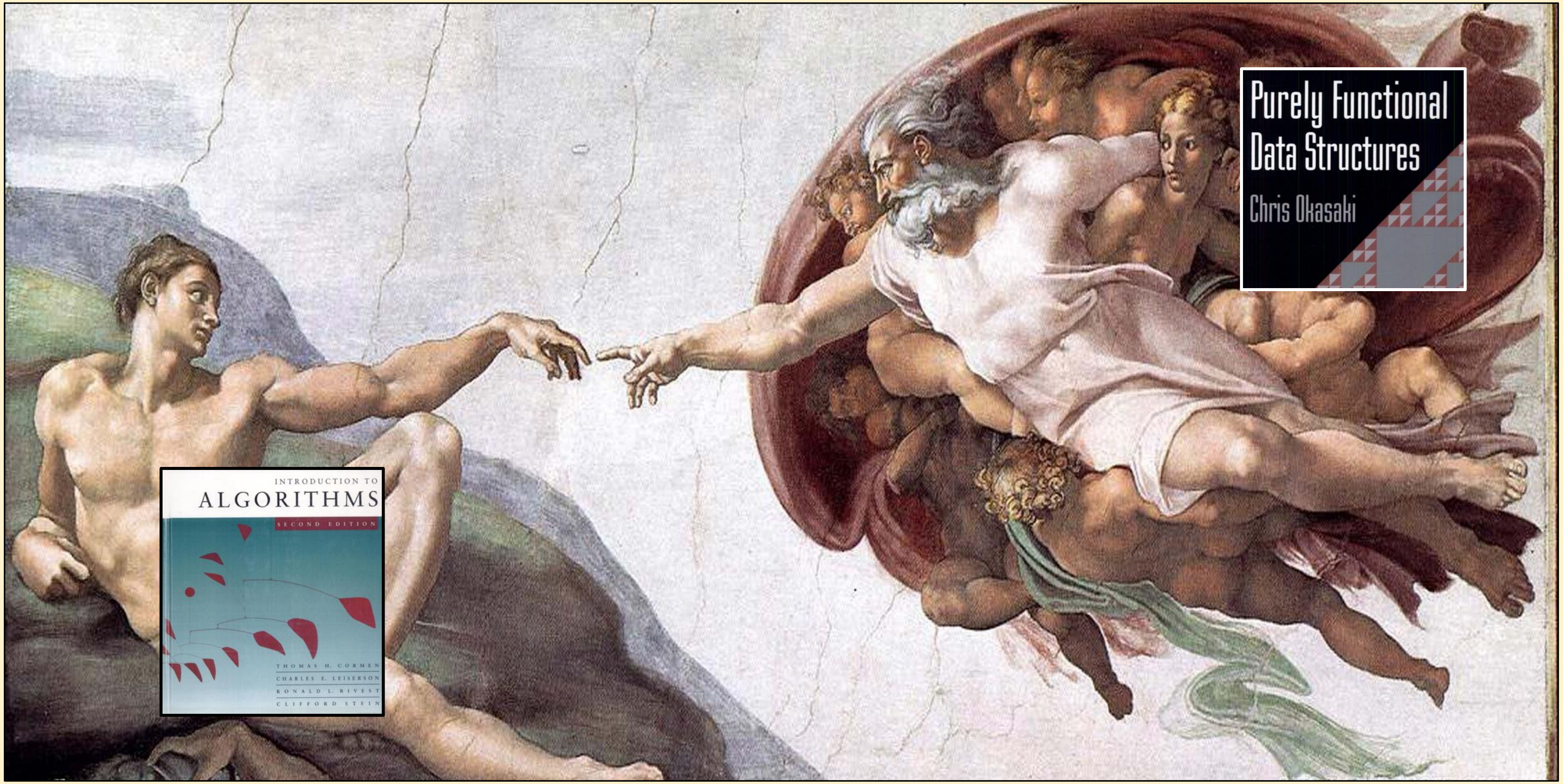
```
fun insert( t : tree, k : key ) : tree
  match t
    Node(l,x,r) -> if x < k then balance(Node(l,x,insert(r,k)))
                   elif x > k then balance(Node(insert(l,k),x,l))
                   else t
    Leaf -> Node(Leaf,k,Leaf) // insert if not found
```

But..

- Recursive – uses stack linear to the depth of the tree
- Allocates fresh Node's along the spine
- Is about **5x slower** that a corresponding iterative in-place C implementation

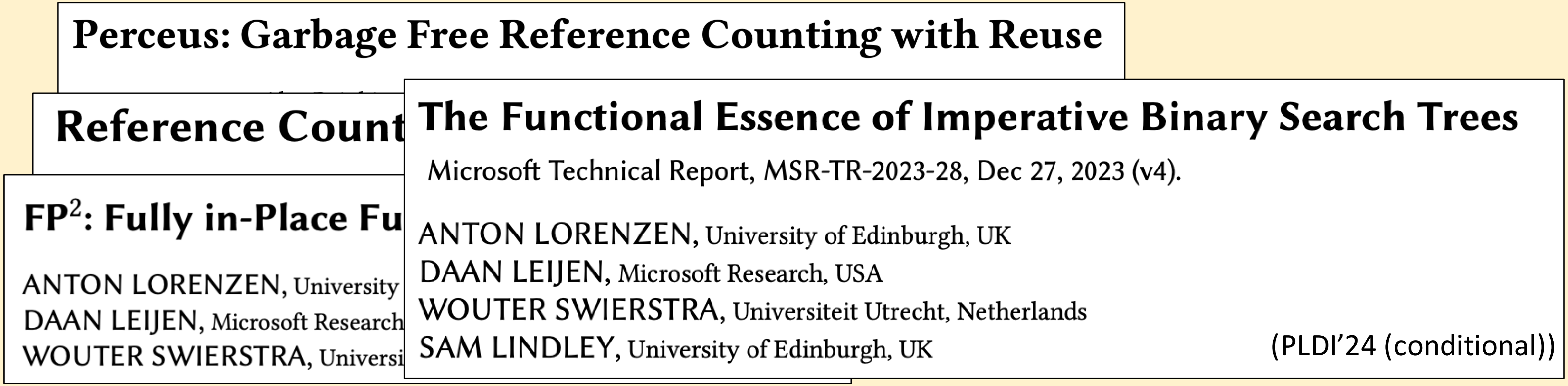
The FP Dream

- Can we write beautiful functional code that allows equational reasoning but still have performance close the corresponding imperative code?



Enabling Recent Developments

- **Perceus**: compiler guided precise reference counting
- **Reuse analysis**: avoid allocation by reusing unique objects at runtime
- **FIP**: fully in-place programming – the `**fip**` keyword guarantees constant stack usage and no allocation (if the parameters are unique)
- **Constructor Contexts**: first-class data structures with a *hole*



Perceus: Garbage Free Reference Counting with Reuse

Reference Count

FP^2: Fully in-Place Fu

ANTON LORENZEN, University
DAAN LEIJEN, Microsoft Research
WOUTER SWIERSTRA, Universi

The Functional Essence of Imperative Binary Search Trees

Microsoft Technical Report, MSR-TR-2023-28, Dec 27, 2023 (v4).

ANTON LORENZEN, University of Edinburgh, UK
DAAN LEIJEN, Microsoft Research, USA
WOUTER SWIERSTRA, Universiteit Utrecht, Netherlands
SAM LINDLEY, University of Edinburgh, UK

(PLDI'24 (conditional))

A Step towards the FP Dream

We are now able to write purely functional bottom-up and top-down tree algorithms with performance that rivals the best C implementations

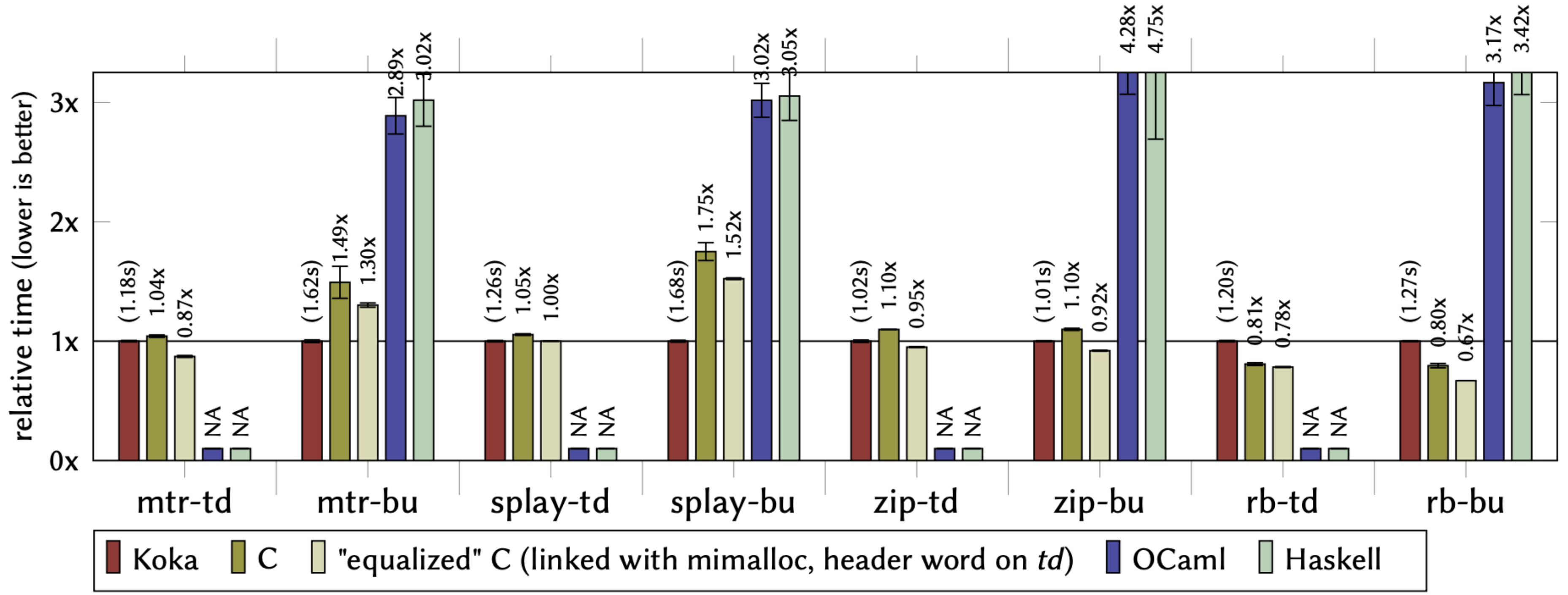


Fig. 4. Benchmarks on Ubuntu 22.04.2 (AMD 7950X 4.5Ghz) comparing the relative performance of C, ML, and Haskell against Koka for move-to-root (*mtr*), splay trees (*splay*), and zip trees (*zip*) for both top-down (*td*) and bottom-up (*bu*) variants. Each benchmark performs the same sequence of 10M pseudo-random insertions between 0 and 100 000 starting with an empty tree.

Detour:
Top-Down algorithms and
First-class Constructor Contexts

An imperative “top-down” map

A top-down algorithm traverses down a structure and eventually directly returns the final result

```
list_t* map( list_t* xs, int (*f)(int) )
{
  for(list_t* cur = xs; cur != NULL; cur = cur->tail) {
    cur->head = f(cur->head);
  }
  return xs;
}
```

A functional map in a recursive style

- Mapping a function over a list:

```
fun map( xs : list<a>, f : a -> b ) : list<b>
  match xs
    Cons(x,xx) -> Cons( f(x), map(xx,f) )
    Nil        -> Nil
```

- But it is not tail-recursive and uses stack linear in the length of the list.

Tail-Recursive map

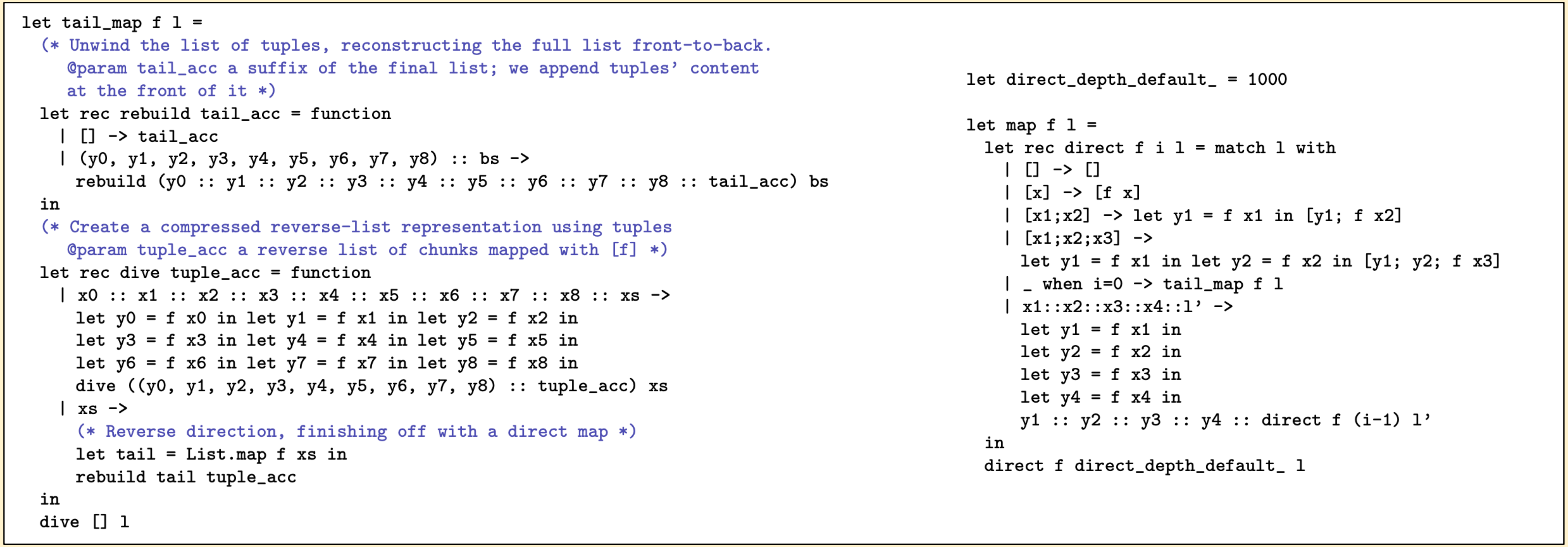
- We can use an accumulator to avoid using the stack:

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

fun map(xs,f)
  map-acc(xs,f,Nil)
```

- But now we need to *reverse* the list at the end ..
- This is essentially a *bottom-up* algorithm now

OCaml containers library..



```
let tail_map f l =
  (* Unwind the list of tuples, reconstructing the full list front-to-back.
     @param tail_acc a suffix of the final list; we append tuples’ content
     at the front of it *)
  let rec rebuild tail_acc = function
    | [] -> tail_acc
    | (y0, y1, y2, y3, y4, y5, y6, y7, y8) :: bs ->
      rebuild (y0 :: y1 :: y2 :: y3 :: y4 :: y5 :: y6 :: y7 :: y8 :: tail_acc) bs
  in
  (* Create a compressed reverse-list representation using tuples
     @param tuple_acc a reverse list of chunks mapped with [f] *)
  let rec dive tuple_acc = function
    | x0 :: x1 :: x2 :: x3 :: x4 :: x5 :: x6 :: x7 :: x8 :: xs ->
      let y0 = f x0 in let y1 = f x1 in let y2 = f x2 in
      let y3 = f x3 in let y4 = f x4 in let y5 = f x5 in
      let y6 = f x6 in let y7 = f x7 in let y8 = f x8 in
      dive ((y0, y1, y2, y3, y4, y5, y6, y7, y8) :: tuple_acc) xs
    | xs ->
      (* Reverse direction, finishing off with a direct map *)
      let tail = List.map f xs in
      rebuild tail tuple_acc
  in
  dive [] l
```

```
let direct_depth_default_ = 1000

let map f l =
  let rec direct f i l = match l with
    | [] -> []
    | [x] -> [f x]
    | [x1;x2] -> let y1 = f x1 in [y1; f x2]
    | [x1;x2;x3] ->
      let y1 = f x1 in let y2 = f x2 in [y1; y2; f x3]
    | _ when i=0 -> tail_map f l
    | x1::x2::x3::x4::l’ ->
      let y1 = f x1 in
      let y2 = f x2 in
      let y3 = f x3 in
      let y4 = f x4 in
      y1 :: y2 :: y3 :: y4 :: direct f (i-1) l’
  in
  direct f direct_depth_default_ l
```

- This is *not* the FP dream ☺

We need “First-Class Constructor Contexts”

- A composition of constructors with a single *hole*

```
ctx Cons(1,_)                   : ctx<list<int>>

ctx Node(Leaf,1,Node(Leaf,2,_)) : ctx<tree<int>>
```

- We can fill the hole using (++.) :

```
ctx Cons(1,Cons(2,_) ++. [3]  == [1,2,3]

ctx Node(Leaf,1,_) ++. Leaf   == Node(Leaf,1,Leaf)
```

Constructor Context Composition

- We can *compose* contexts using (++):

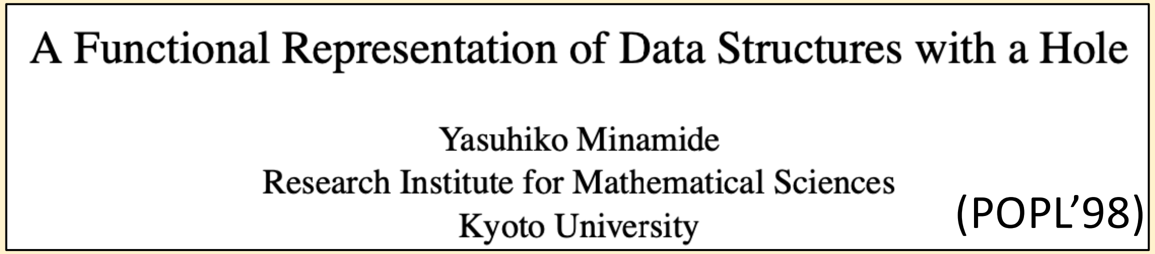
```
ctx Cons(1,_) ++ ctx Cons(2,_)           == ctx Cons(1,Cons(2,_))

ctx Node(Leaf,1,_) ++ ctx Node(_,2,Leaf) == ctx Node(Leaf,1,Node(_,2,Leaf))
```

- One way to implement contexts is as functions:

```
ctx K     = fn(x) K[x]
c1 ++ c2  = c1 o c2
c ++. e   = c e
```

Minamide Tuples



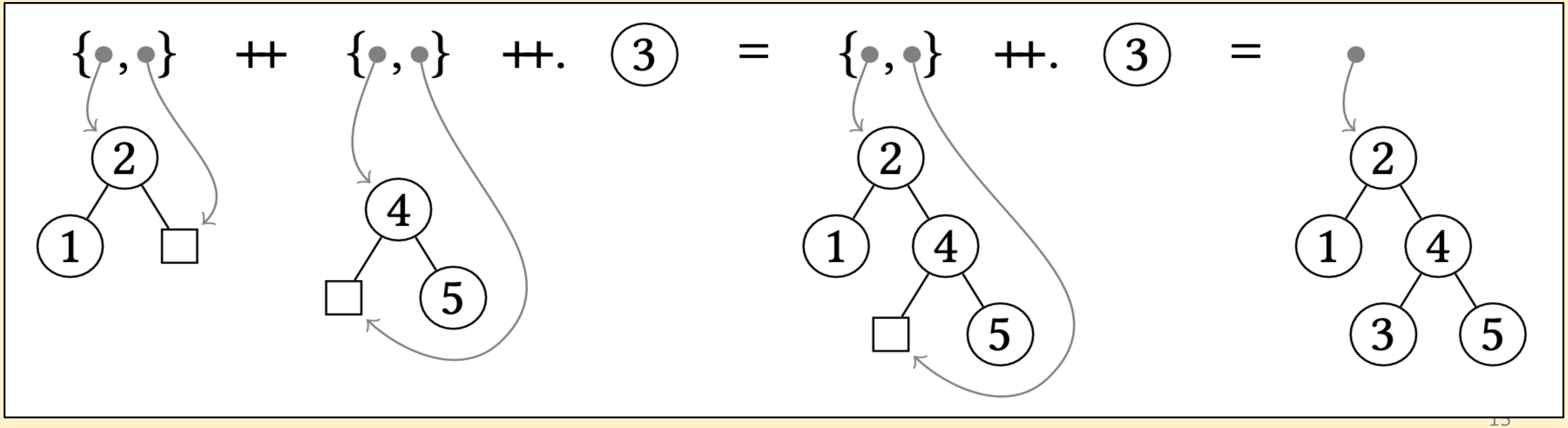
A Functional Representation of Data Structures with a Hole

Yasuhiko Minamide
Research Institute for Mathematical Sciences
Kyoto University

(POPL'98)

- We use a tuple {*x*,*h*} internally where *x* points to the top of the context and *h* points to the hole inside that context. We can update the hole directly:

```
ctx Node(single(1),2,_) ++ ctx Node(_,4,single(5)) ++. single(3)
```



.. But it is unsafe if contexts are shared

- Consider:

```
val c = ctx Node(1,_) in c ++ c ++. [2]   ==  [1,1,2]
```

- Minamide makes it safe by imposing a linear type system for contexts.

Safe and fast with static context paths

- We *statically compile* the indices of the *context path* so we can *dynamically copy* the context path at runtime (in linear time)

```
ctx Node(Node(Node(Leaf,1,Leaf),2,_),5,Leaf)
```

- Is compiled with context path indices [1,3] to:

$$\texttt{Node}_{1}(\texttt{Node}_{3}(\texttt{Node}(\texttt{Leaf},1,\texttt{Leaf}),2,\square),5,\texttt{Leaf})$$

- We use 8-bits in the header of constructors to support this

(and it is set at once together with the constructor tag)

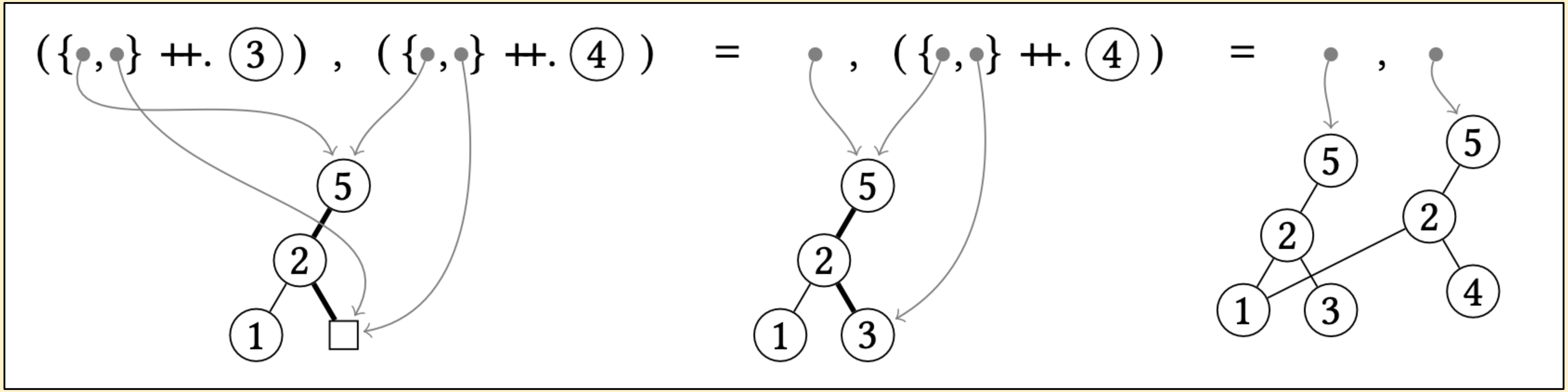
Tail Recursion Modulo Context: An Equational Approach

DAAN LEIJEN, Microsoft Research, USA
ANTON LORENZEN, University of Edinburgh, UK

(POPL'22)

Copy Dynamically on Demand

- With **Perceus**: if a context is not unique, copy the context path first
- With **GC:** use a distinguished value for the hole (□). If we find a hole (□) we are the first and update in-place; any subsequent access finds some other value and needs to copy the context path first.



(the thick edges are the context path with indices [1,3])

True Top-Down Map

- Instead of accumulating a *list*, we accumulate a *context* :

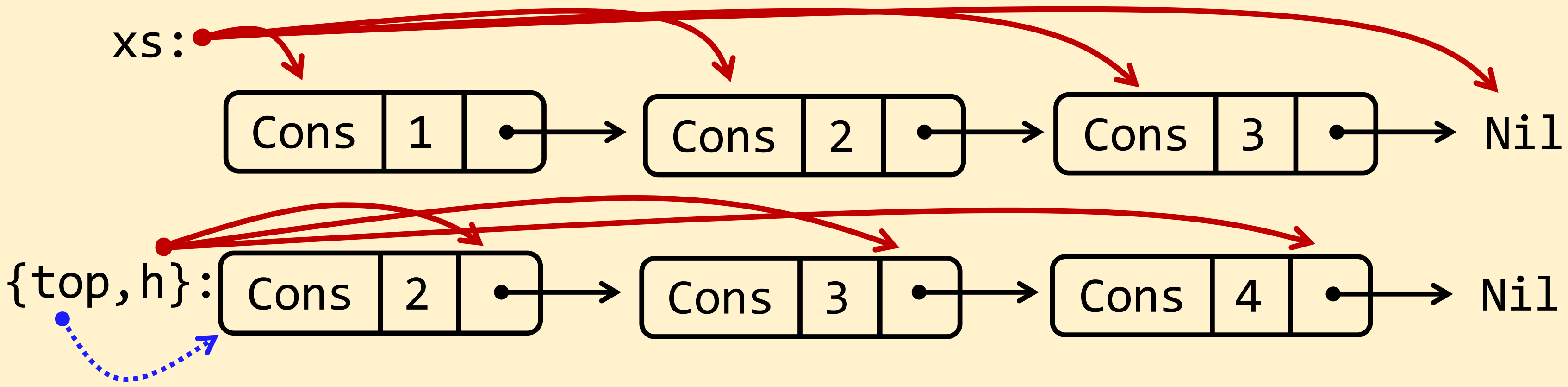
```
fun map-td( xs : list<a>, f : a -> b, acc : ctx<list<b>> ) : list<b>
  match xs
    Cons(x,xx) -> map-td( xx, f, acc ++ ctx Cons(f(x),_) )
    Nil        -> acc ++. Nil

fun map(xs,f)
  map-td(xs, f, ctx _)
```

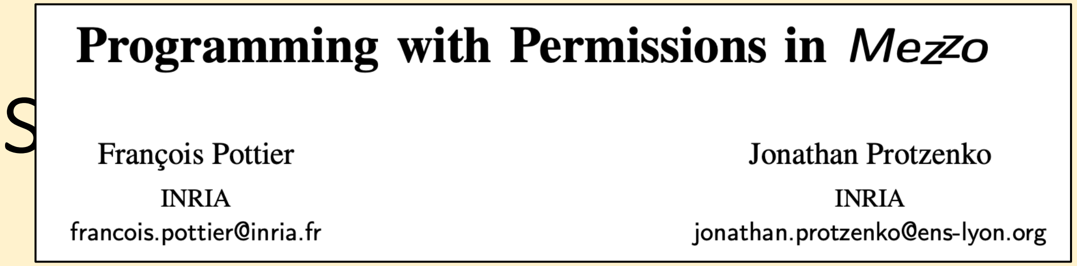
- No need to reverse at the end: we immediately return the root of the Minamide tuple
- The context is not shared: all compositions are in-place! (and constant time)

Top-Down with a Constructor Context

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



Another example: flatten a lis



Programming with Permissions in *Mezzo*

François Pottier
INRIA
francois.pottier@inria.fr

Jonathan Protzenko
INRIA
jonathan.protzenko@ens-lyon.org

```
fun flatten( xss : list<list<a>> ) : list<a>
  match xss
```

```
fip fun append-td( xs : list<a>, acc : ctx<list<a>> ) : ctx<list<a>>
  match xs
    Cons(x,xx) -> append-td( xx, acc ++ ctx Cons(x,_) )
    Nil -> acc

fbip fun flatten-td( xss : list<list<a>>, acc : ctx<list<a>> ) : ctx<list<a>>
  match xss
    Cons(xs,xxs) -> flatten-td( xxs, append-td( xs, acc ) )
    Nil -> acc

fbip fun flatten( xss : list<list<a>> ) : list<a>
  flatten-td( xss, ctx _) ++. Nil
```

Detour: FIP: Fully In-Place Functional Programming

Consider the imperative map again

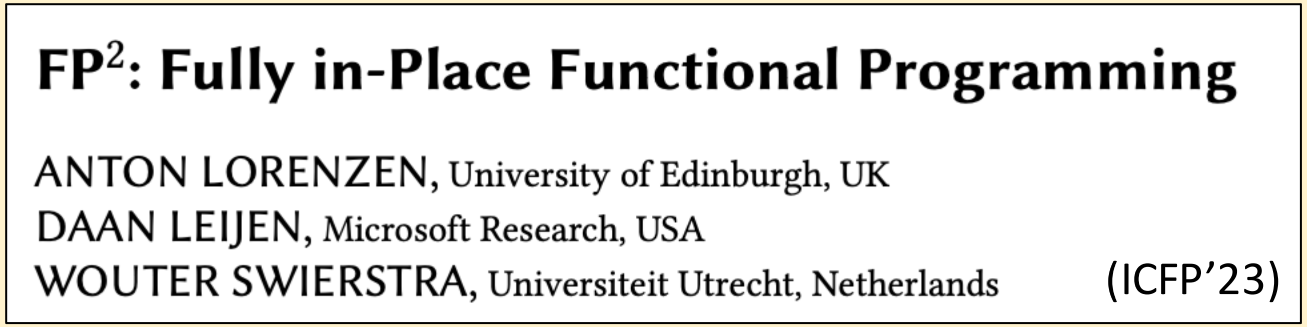
```
list_t* map( list_t* xs, int (*f)(int) )
{
  for(list_t* cur = xs; cur != NULL; cur = cur->tail) {
    cur->head = f(cur->head);
  }
  return xs;
}
```

But this assumes “ownership” of *xs* (so we can destructively update)

Fully In-Place Functional Programming

- What if we assume (for now) that we ”own” parameters?

- We defined a linear FIP calculus – we show that for any program in this FIP fragment we can always execute “in-place” -- using no heap allocation and constant stack space.



FP^2: Fully in-Place Functional Programming

ANTON LORENZEN, University of Edinburgh, UK
DAAN LEIJEN, Microsoft Research, USA
WOUTER SWIERSTRA, Universiteit Utrecht, Netherlands

(ICFP’23)

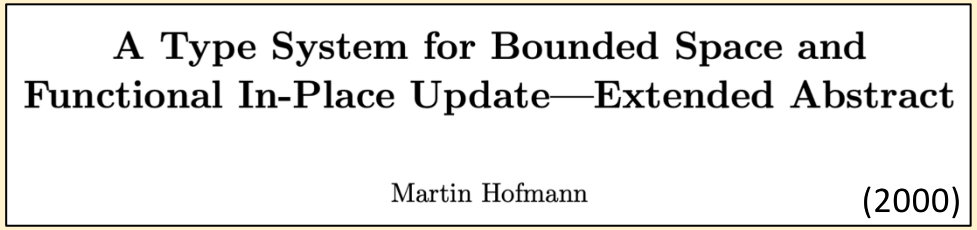
FIP map

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

fip fun map(xs,f)
  map-td(xs, f, ctx _)
```

- The `fip` keyword checks if the function is in the linear FIP fragment.
- It guarantees that if *xs* is unique, the function does not (de)allocate at all and uses constant stack space.
- The linear `match` consumes *xs* and we cannot use *xs* after this

Reuse credits $\Diamond_k$



A Type System for Bounded Space and Functional In-Place Update—Extended Abstract

Martin Hofmann

(2000)

- Since xs is matched linearly, we can *reuse* the Cons

```
fip fun map-td( xs : list<a>, ^f : a -> b, acc : ctx<list<b>> )
  match xs               ◇2
    Cons(x,xx) -> map-td( xx, f, acc ++ ctx Cons(f(x),_) )
    Nil        -> acc ++. Nil
```

- Formally, a linear match gives us the children (x,xx), but also a *reuse credit* of size 2, denoted as $\Diamond_2$
- The reuse credit is consumed by Cons – no allocation is needed.
- Inspired by the work of Aspinall and Hofmann (2002) which uses *space credits* – but unlike their work, we cannot split or combine reuse credits.

.. FIP in a Functional World

- Within FIP, no allocation takes place
- But who allocates the tree to be balanced, or the list to be mapped?

- When calling FIP functions from non-FIP functions how can we ensure the parameters are owned and can be updated destructively?

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

- One approach is to have a linear type system..
  (like the uniqueness types of Clean)

Our approach: use Perceus reuse

- In Koka we dynamically copy if the “owned” parameter is not unique at runtime
- This can be done efficiently, something like:

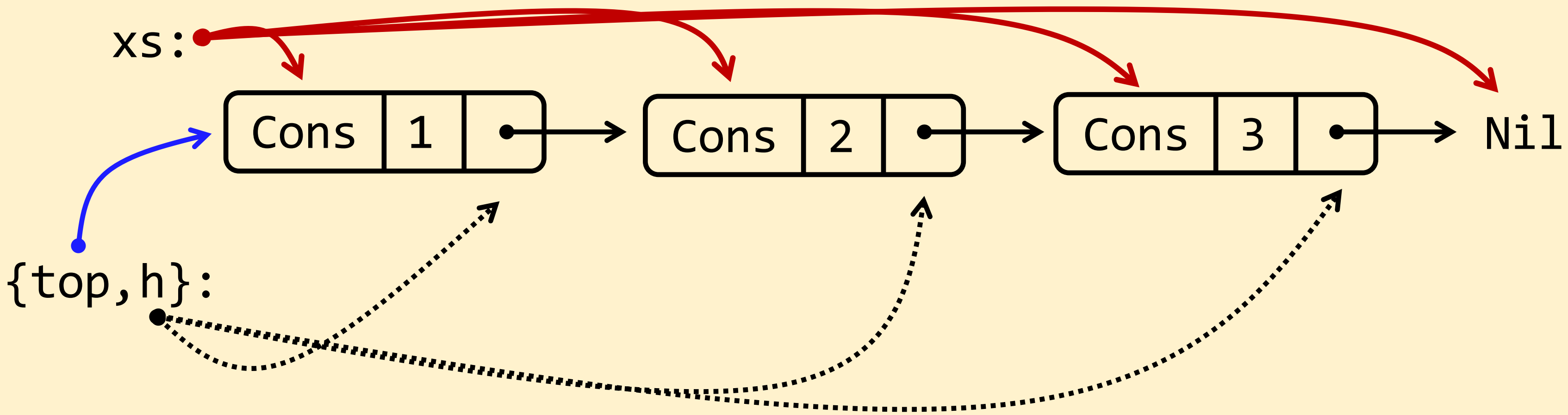
```
fip fun map-td( xs : list<a>, f : a -> b, acc : ctx<list<b>> ) : list<b>
  match xs
    Cons(x,xx) ->
      val loc = if is-unique(xs) then &xs
                                 else { dup(x); dup(xx); decref(xs); alloc(2) }
      map-td( xx, f, acc ++ ctx Cons@loc(f(x),_) )

    Nil -> acc ++. Nil
```

This is the runtime manifestation of the reuse credit in the FIP calculus, $\diamond_2$

Top-Down, In-Place

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



Fully In-Place Programming

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

- The best of both worlds: we have a pure functional program… but it is statically checked that the map can update the list “in-place” (if *xs* is unique at runtime).

- Much like a type checker, or a Rust-style borrow checker, this really changes the programmer experience – **fearless fip**!

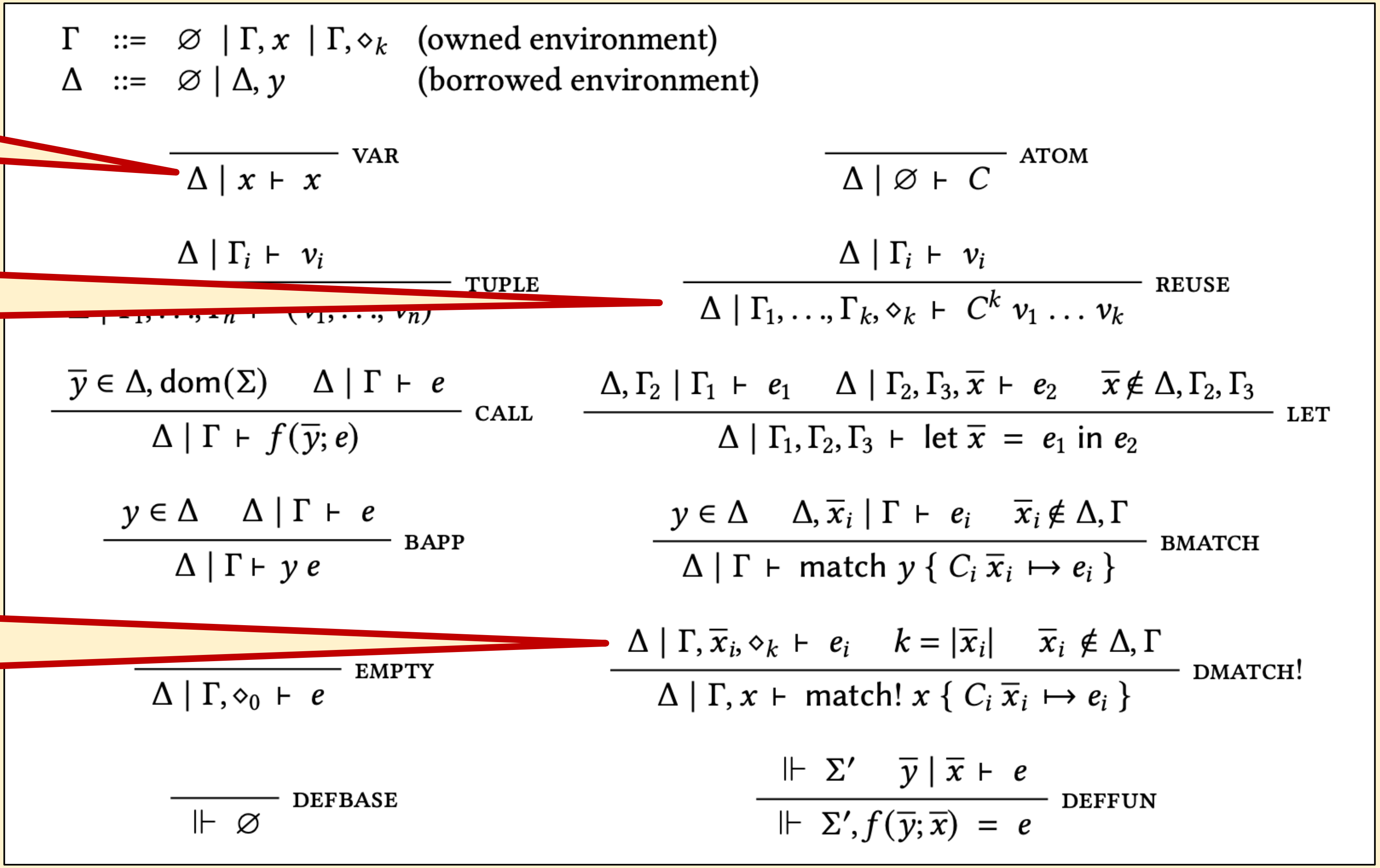
- Splay and Red-Black Trees, Merge- and Quick Sort, General map, etc.

A beautiful FIP calculus

A Linear calculus
(but no linear types!)

That we later can
use to allocate

A destructive match
gives us a $\diamond_k$ reuse
credit



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

$$\Delta \;::=\; \varnothing \mid \Delta, y \quad \text{(borrowed environment)}$$

$$\frac{}{\Delta \mid x \vdash x}\;\textsc{Var} \qquad \frac{}{\Delta \mid \varnothing \vdash C}\;\textsc{Atom}$$

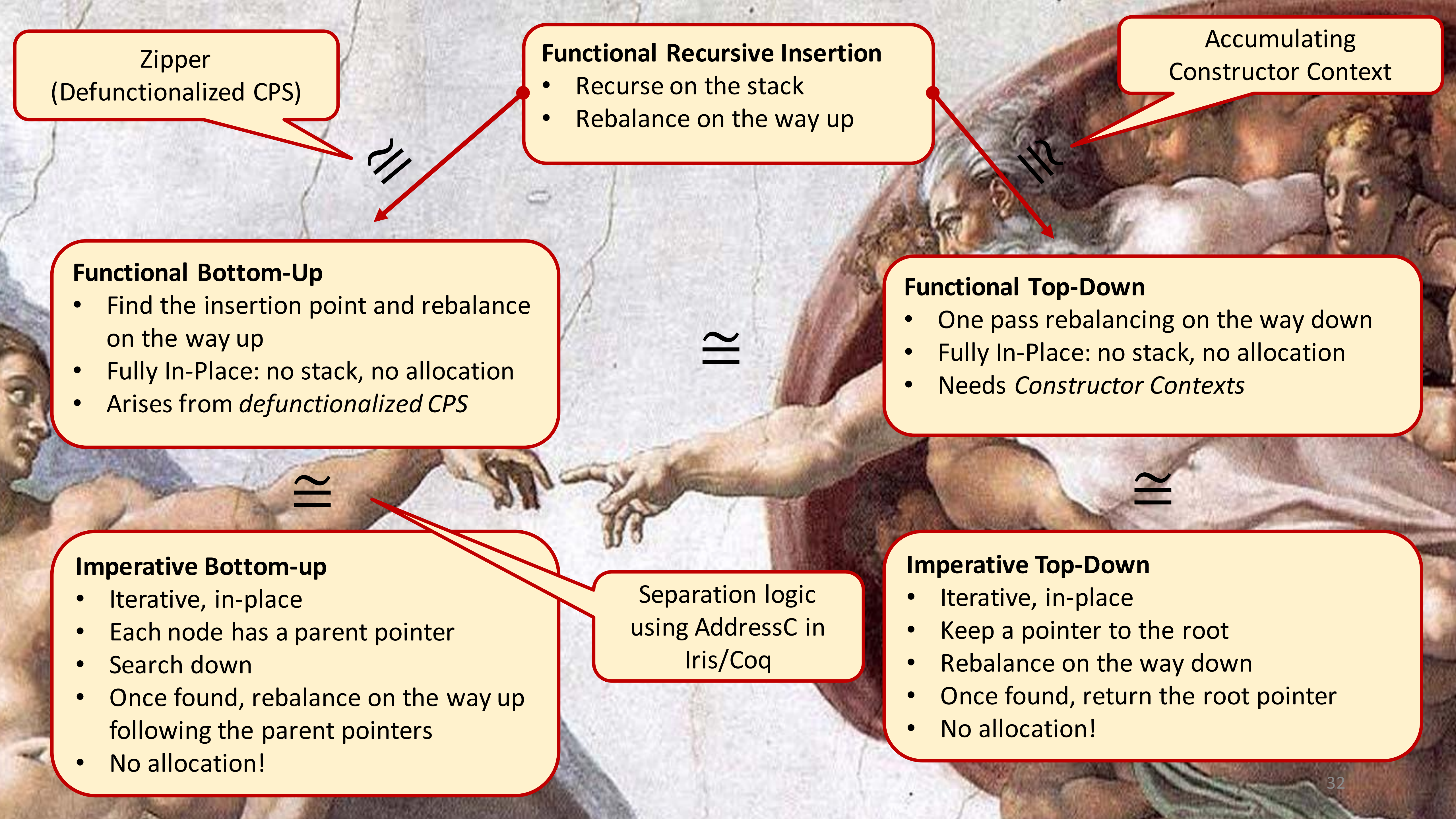
$$\frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_n \vdash (v_1, \ldots, v_n)}\;\textsc{Tuple} \qquad \frac{\Delta \mid \Gamma_i \vdash v_i}{\Delta \mid \Gamma_1, \ldots, \Gamma_k, \diamond_k \vdash C^k\, v_1 \ldots v_k}\;\textsc{Reuse}$$

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

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

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

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



Functional Recursive Insertion

- Recurse on the stack
- Rebalance on the way up

Zipper
(Defunctionalized CPS)

≅

Accumulating
Constructor Context

≅

Functional Bottom-Up

- Find the insertion point and rebalance on the way up
- Fully In-Place: no stack, no allocation
- Arises from *defunctionalized CPS*

≅

Functional Top-Down

- One pass rebalancing on the way down
- Fully In-Place: no stack, no allocation
- Needs *Constructor Contexts*

≅

Imperative Bottom-up

- Iterative, in-place
- Each node has a parent pointer
- Search down
- Once found, rebalance on the way up following the parent pointers
- No allocation!

Separation logic
using AddressC in
Iris/Coq

≅

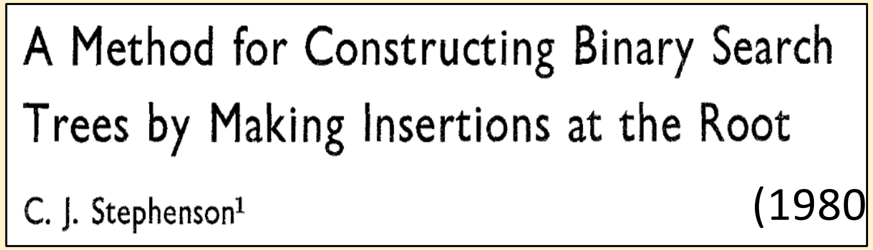
Imperative Top-Down

- Iterative, in-place
- Keep a pointer to the root
- Rebalance on the way down
- Once found, return the root pointer
- No allocation!

Move-To-Root Trees

- An inserted element always becomes the root node
  (so often accessed elements will be near the top)

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



A Method for Constructing Binary Search Trees by Making Insertions at the Root

C. J. Stephenson[1]

(1980

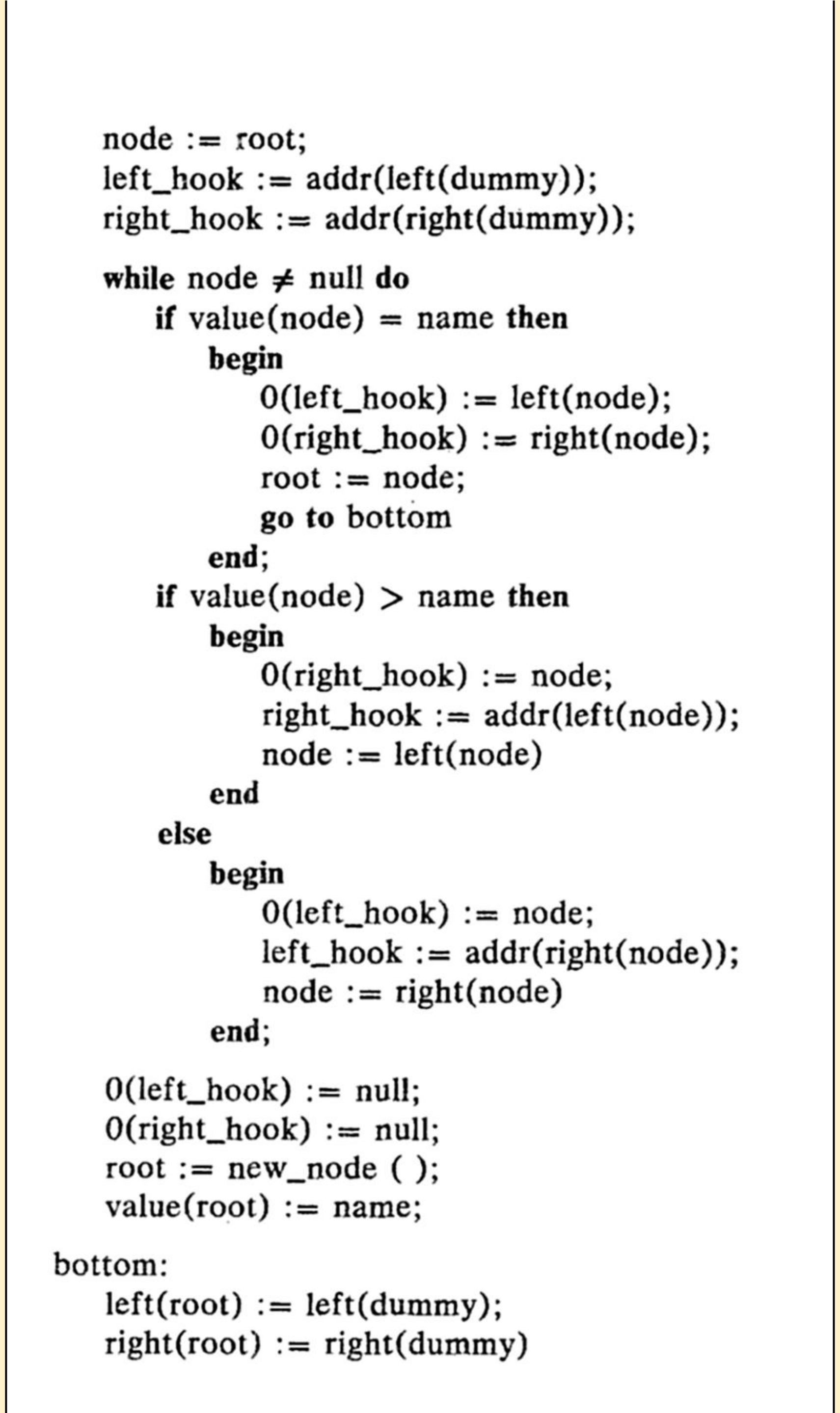
Move-To-Root: Top-Down

- We now use *two* accumulating contexts: one for the smaller tree, and one for the bigger one:

```
fip(1) fun insert-td( t : tree, k : key ) : tree
  down-td(t,k,ctx _, ctx _)

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

- This is fast and rivals the performance of the best iterative C implementation

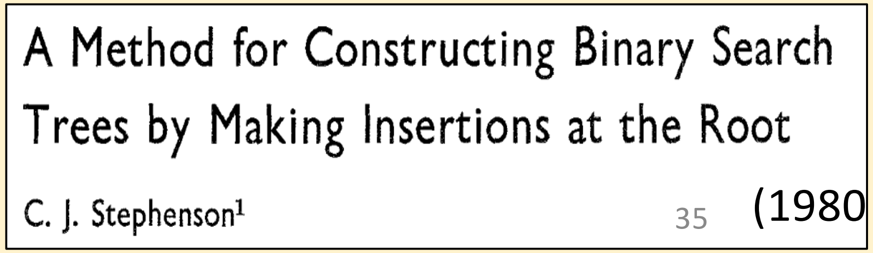


```
node := root;
left_hook := addr(left(dummy));
right_hook := addr(right(dummy));

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)
```



A Method for Constructing Binary Search Trees by Making Insertions at the Root

C. J. Stephenson[1]

(1980

Benchmarks

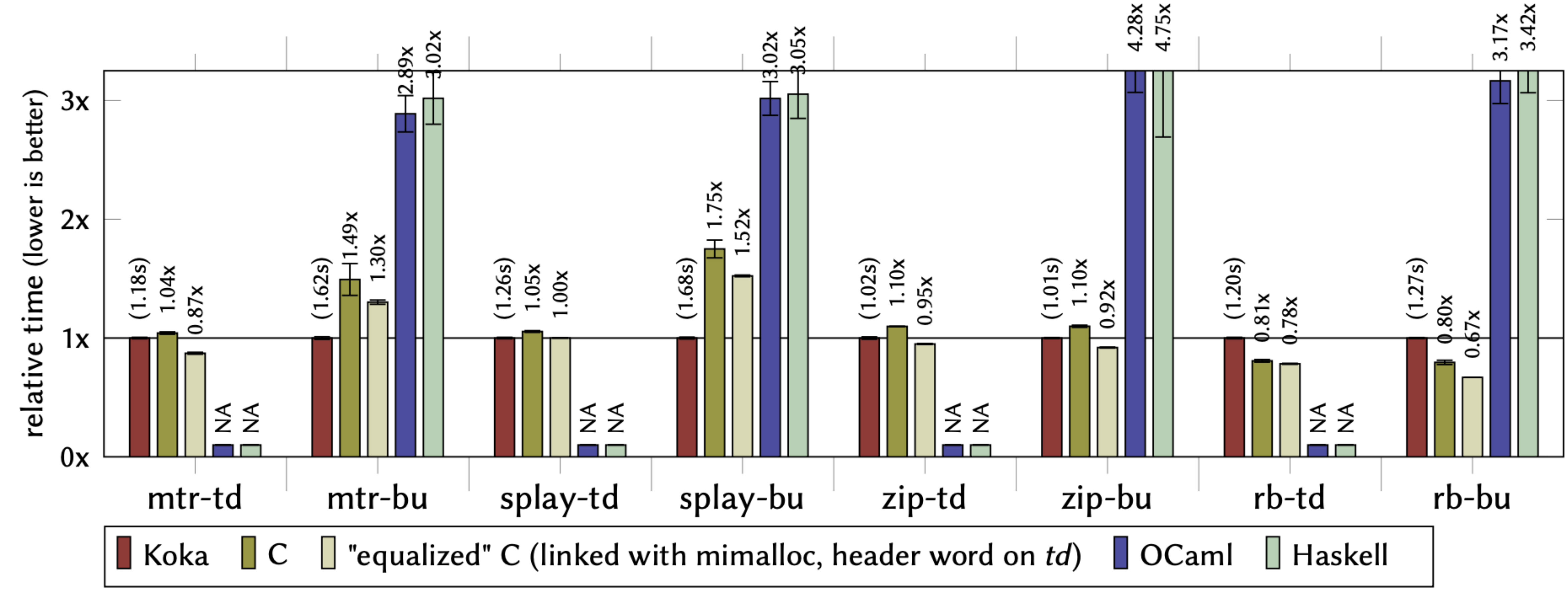
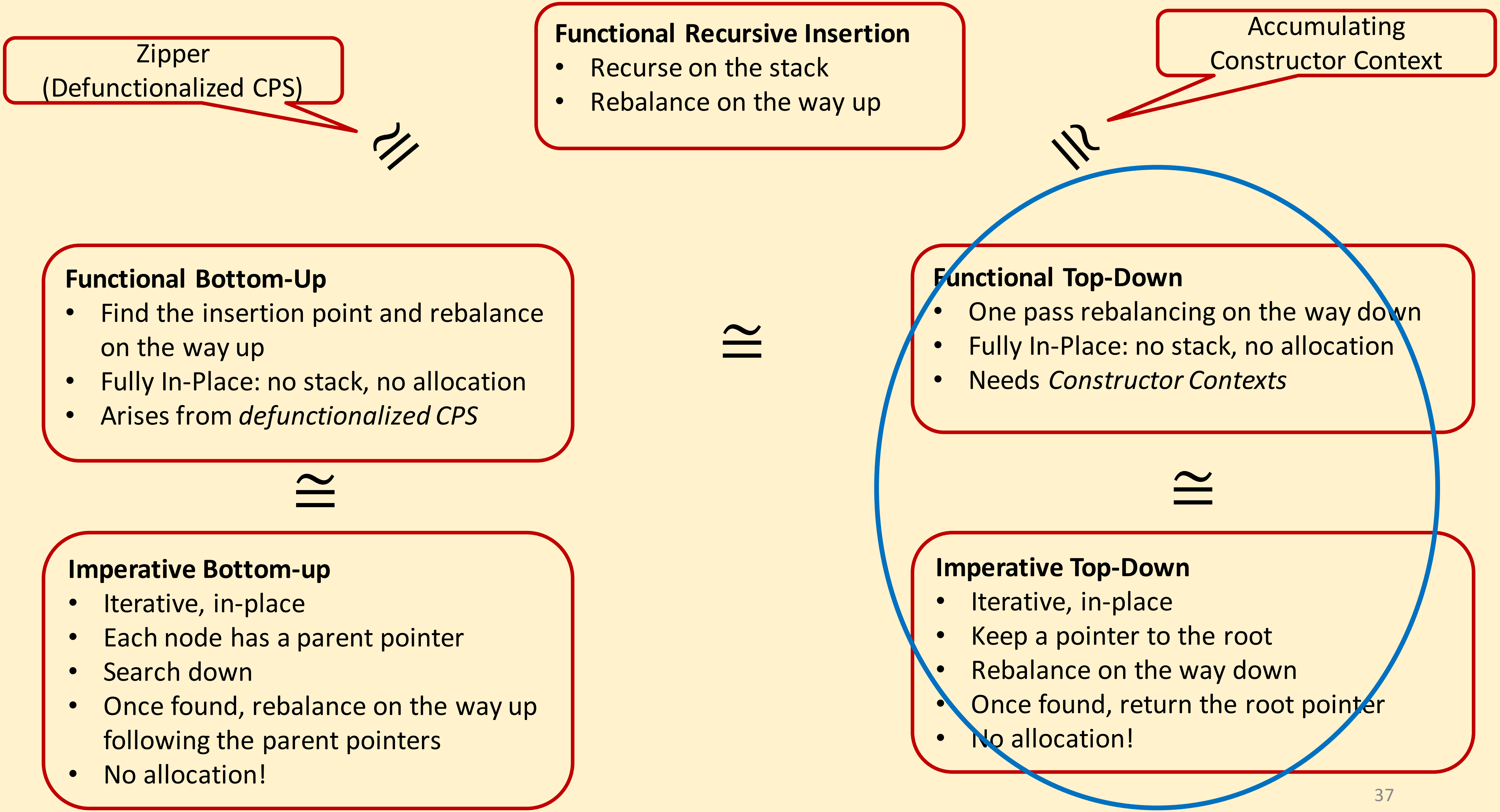


Fig. 4. Benchmarks on Ubuntu 22.04.2 (AMD 7950X 4.5Ghz) comparing the relative performance of C, ML, and Haskell against Koka for move-to-root (*mtr*), splay trees (*splay*), and zip trees (*zip*) for both top-down (*td*) and bottom-up (*bu*) variants. Each benchmark performs the same sequence of 10M pseudo-random insertions between 0 and 100 000 starting with an empty tree.



Zipper
(Defunctionalized CPS)

$\cong$

Functional Recursive Insertion

- Recurse on the stack
- Rebalance on the way up

Accumulating
Constructor Context

$\cong$

Functional Bottom-Up

- Find the insertion point and rebalance on the way up
- Fully In-Place: no stack, no allocation
- Arises from *defunctionalized CPS*

$\cong$

Functional Top-Down

- One pass rebalancing on the way down
- Fully In-Place: no stack, no allocation
- Needs *Constructor Contexts*

$\cong$

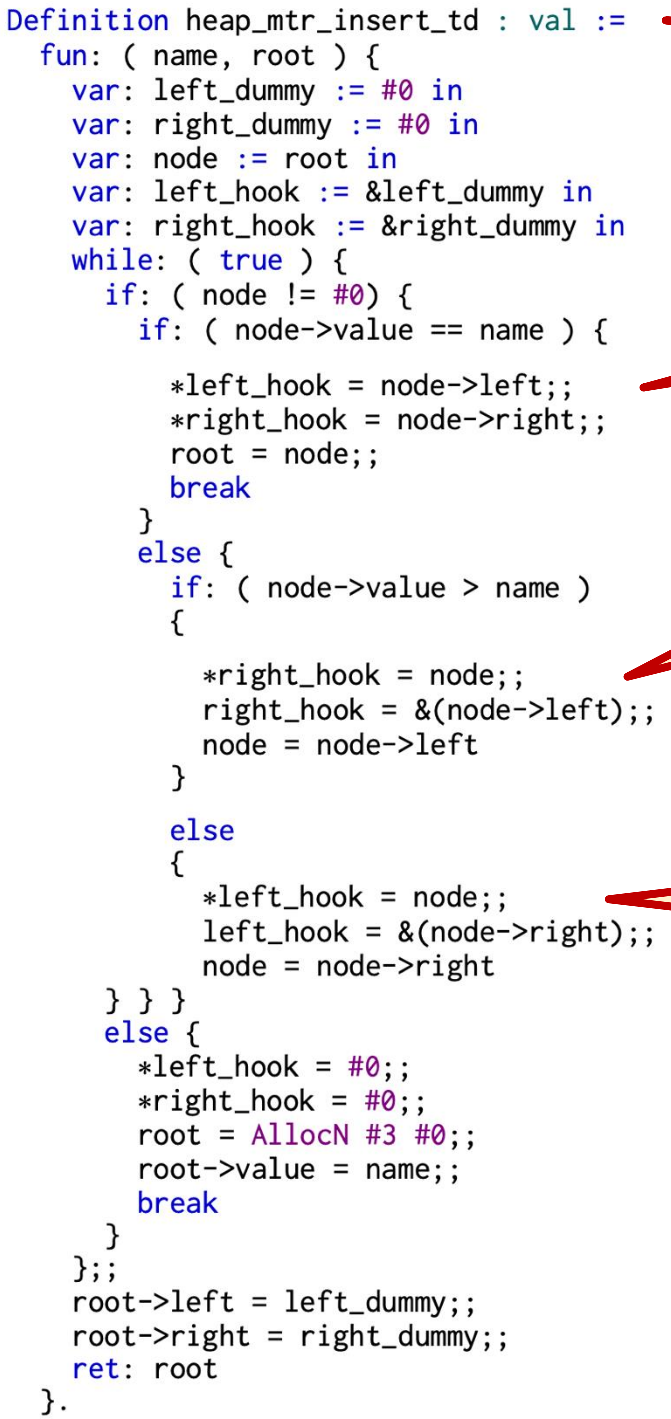
$\cong$

Imperative Bottom-up

- Iterative, in-place
- Each node has a parent pointer
- Search down
- Once found, rebalance on the way up following the parent pointers
- No allocation!

Imperative Top-Down

- Iterative, in-place
- Keep a pointer to the root
- Rebalance on the way down
- Once found, return the root pointer
- No allocation!



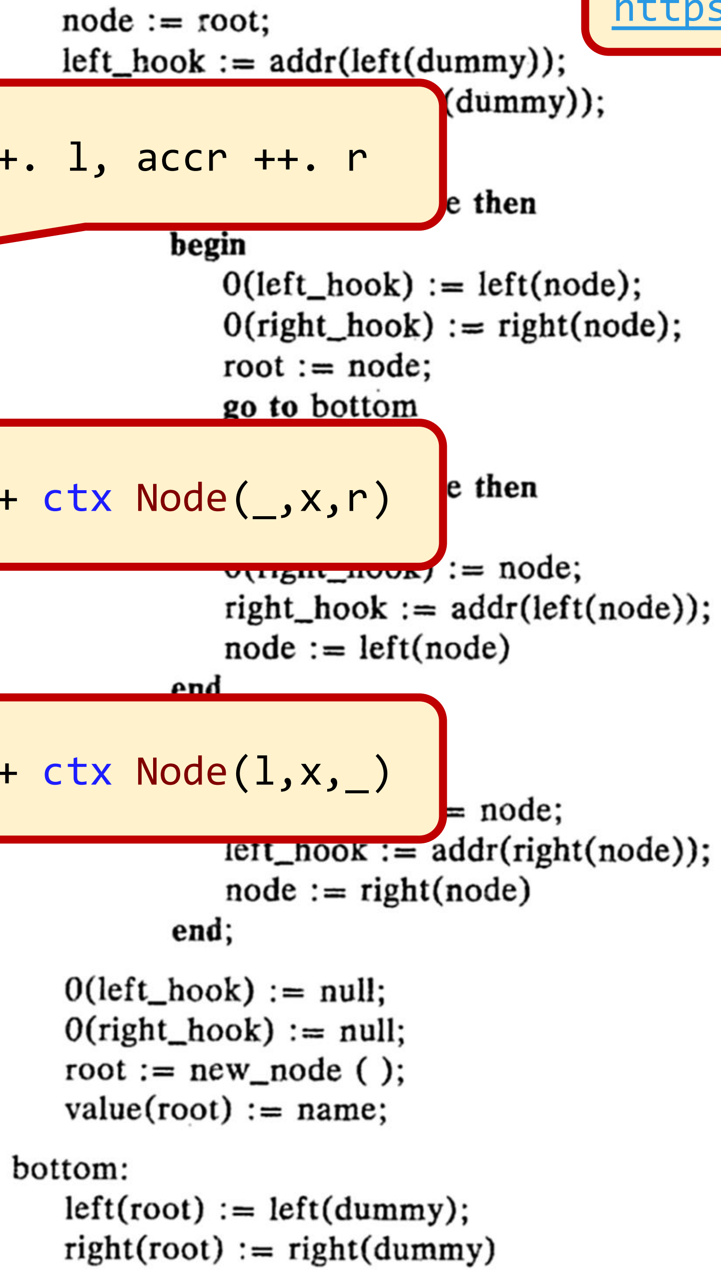
```
Definition heap_mtr_insert_td : val :=
  fun: ( name, root ) {
    var: left_dummy := #0 in
    var: right_dummy := #0 in
    var: node := root in
    var: left_hook := &left_dummy in
    var: right_hook := &right_dummy in
    while: ( true ) {
      if: ( node != #0) {
        if: ( node->value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

AddressC:
https://github.com/koka-lang/addressc



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

                          e then
    begin
        0(left_hook) := left(node);
        0(right_hook) := right(node);
        root := node;
        go to bottom

                          e then

                       := node;
        right_hook := addr(left(node));
        node := left(node)
    end

                          = 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)
```

accl ++. l, accr ++. r

accr ++ ctx Node(_,x,r)

accl ++ ctx Node(l,x,_)

Correctness

Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning

Ralf Jung
MPI-SWS &
Saarland University
jung@mpi-sws.org

David Swasey
MPI-SWS
swasey@mpi-sws.org

Filip Sieczkowski
Aarhus University
filips@cs.au.dk

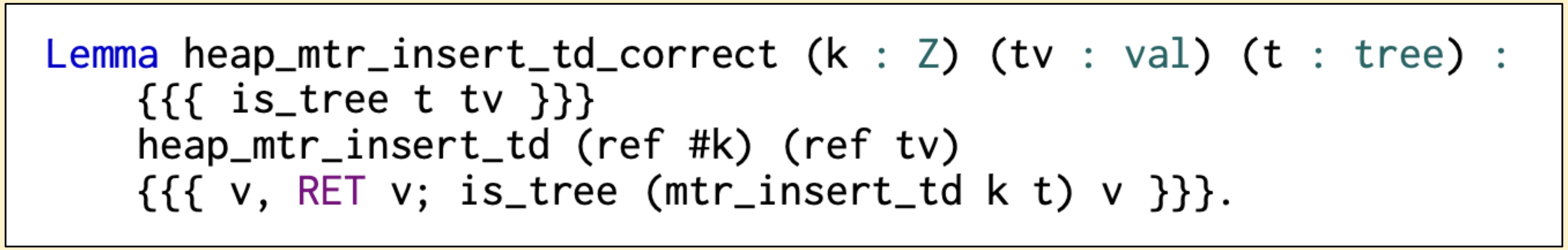
Kasper Svendsen
Aarhus University
ksvendsen@cs.au.dk

Aaron Turon
Mozilla Research
aturon@mozilla.com

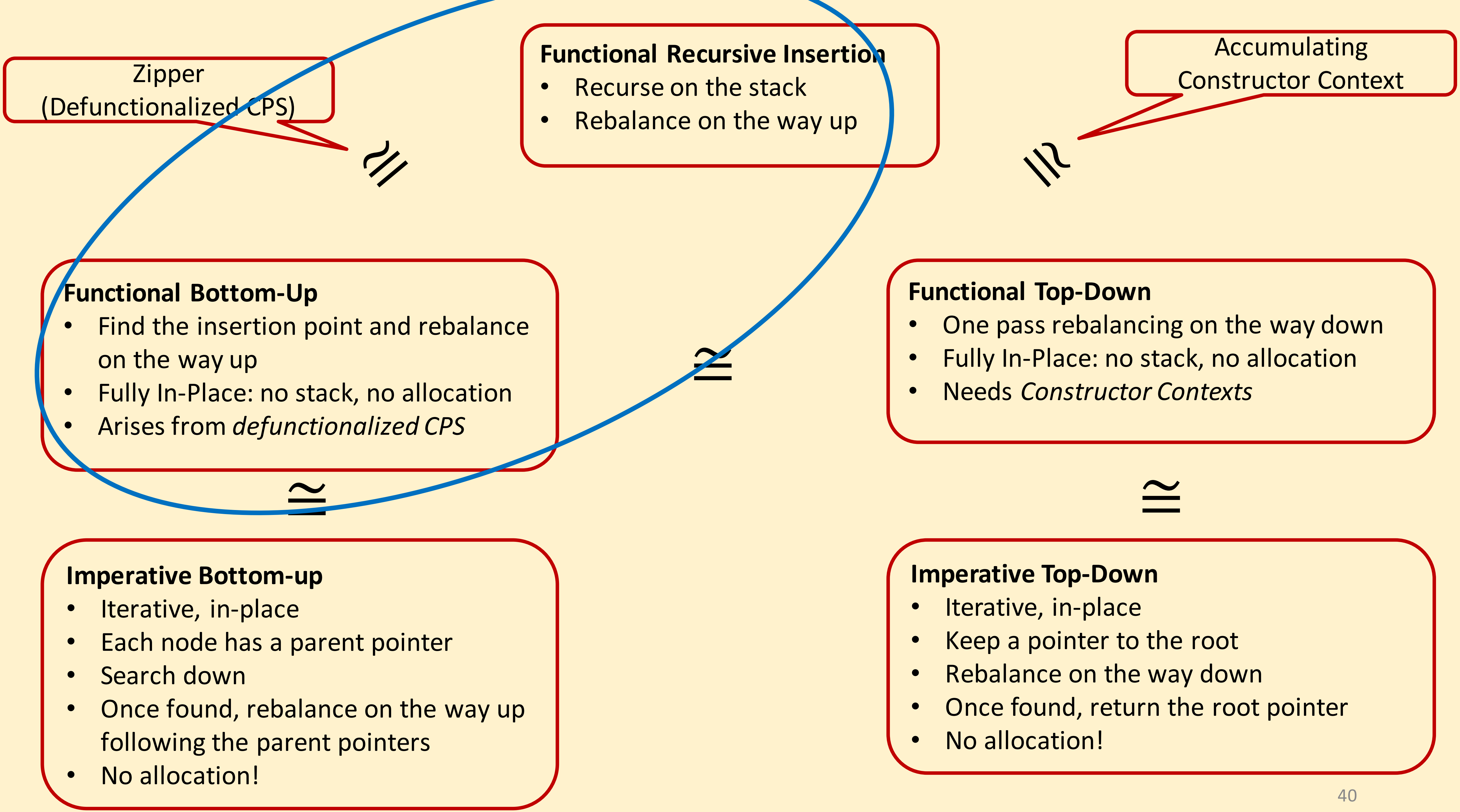
Lars Birkedal
Aarhus University
birkedal@cs.au.dk

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

- We can now use Iris/Coq to prove that the imperative algorithm corresponds to our functional top-down version:



```
Lemma heap_mtr_insert_td_correct (k : Z) (tv : val) (t : tree) :
    {{{ is_tree t tv }}}
    heap_mtr_insert_td (ref #k) (ref tv)
    {{{ v, RET v; is_tree (mtr_insert_td k t) v }}}.
```



Zipper
(Defunctionalized CPS)

Functional Recursive Insertion

- Recurse on the stack
- Rebalance on the way up

Accumulating
Constructor Context

≅ ≅

Functional Bottom-Up

- Find the insertion point and rebalance on the way up
- Fully In-Place: no stack, no allocation
- Arises from *defunctionalized CPS*

≅

Functional Top-Down

- One pass rebalancing on the way down
- Fully In-Place: no stack, no allocation
- Needs *Constructor Contexts*

≅ ≅

Imperative Bottom-up

- Iterative, in-place
- Each node has a parent pointer
- Search down
- Once found, rebalance on the way up following the parent pointers
- No allocation!

Imperative Top-Down

- Iterative, in-place
- Keep a pointer to the root
- Rebalance on the way down
- Once found, return the root pointer
- No allocation!

Move-To-Root: Recursive to Bottom-Up

- It turns out we can derive a bottom-up algorithm from a recursive one by using *defunctionalized CPS*

CPS conversion: add a continuation function

```
fun insert( t : tree, k : key )
  match t
    Node(l,x,r) -> if   x < k then match insert(r,k)
                                     Node(s,y,b) -> Node( Node(l,x,s), y, b)
```

Becomes:

```
fun insert( t : tree, k : key )
  down(t,k,id)

fun down ( t : tree, k : key, cont : tree -> tree ) : tree
  match t
   Node(l,x,r) -> if x < k then
      down( r, k, fn(t) match t
                          Node(s,y,b) -> cont(Node(Node(l,x,s),y,b)) )
   ...
```

Defunctionalize the CPS

```
fun down ( t : tree, k : key, cont : tree -> tree ) : tree
  match t
   Node(l,x,r) -> if x < k then
      down( r, k, fn(t) match t
                          Node(s,y,b) -> cont(Node(Node(l,x,s),y,b)) )
   ...
```

- Becomes:

```
type zipper
  Done
  NodeL(up : zipper, key : key, right : tree )
  NodeR(left : tree, key : key, up : zipper )

fip(1) fun down( t : tree, k : key, z : zipper )
  match t
    Node(l,x,r) -> if x < k then
       down( r, k, NodeR(l,x,z) )
    ...
```

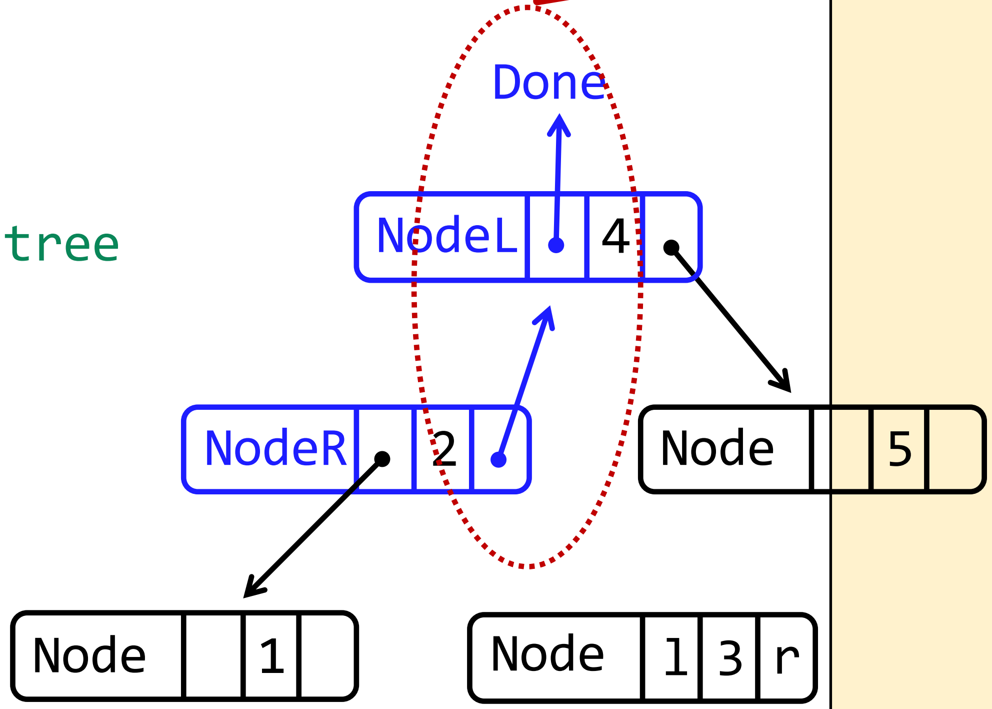
Defunctionalize the CPS:

A zipper represents the path *in reverse* (so we can unwind in linear time)

```
type zipper
  Done
  NodeL(up : zipper, key : key, right : tree )
  NodeR(left : tree, key : key, up : zipper )

fip(1) fun down( t : tree, k : key, z : zipper ) : tree
  match t
    Node(l,x,r) ->
      if   x < k then down( r, k, NodeR(l,x,z) )
      elif x > k then down ( l, k, NodeL(z,x,r) )
      else rebuild( z, Node(l,k,r) )
    Leaf -> rebuild( z, Node(Leaf,k,Leaf) )

fip fun rebuild( z : zipper, t : tree ) : tree
  match z
    Done -> t
    NodeR(l,x,up) -> match t // we came from the right
       Node(s,y,b) -> rebuild( up, Node( Node(l,x,s), y, b))
    NodeL(up,x,r) -> match t // we came from the left
       Node(s,y,b) -> rebuild( up, Node( s, y, Node(b,x,r)))
```

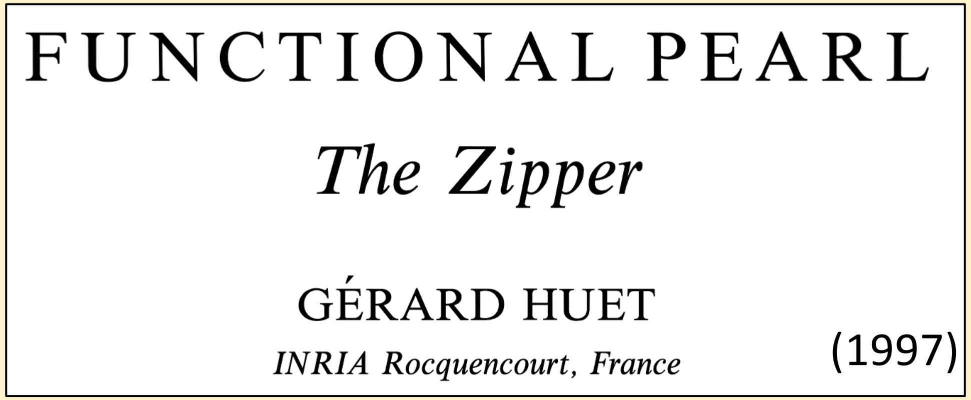


Zippers are FIP ?

- Huet already observed that the zipper operations could be implemented in-place, where he concludes his paper with the following paragraph:

“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.”

- Our FIP calculus makes Huet’s insight precise!



FUNCTIONAL PEARL

The Zipper

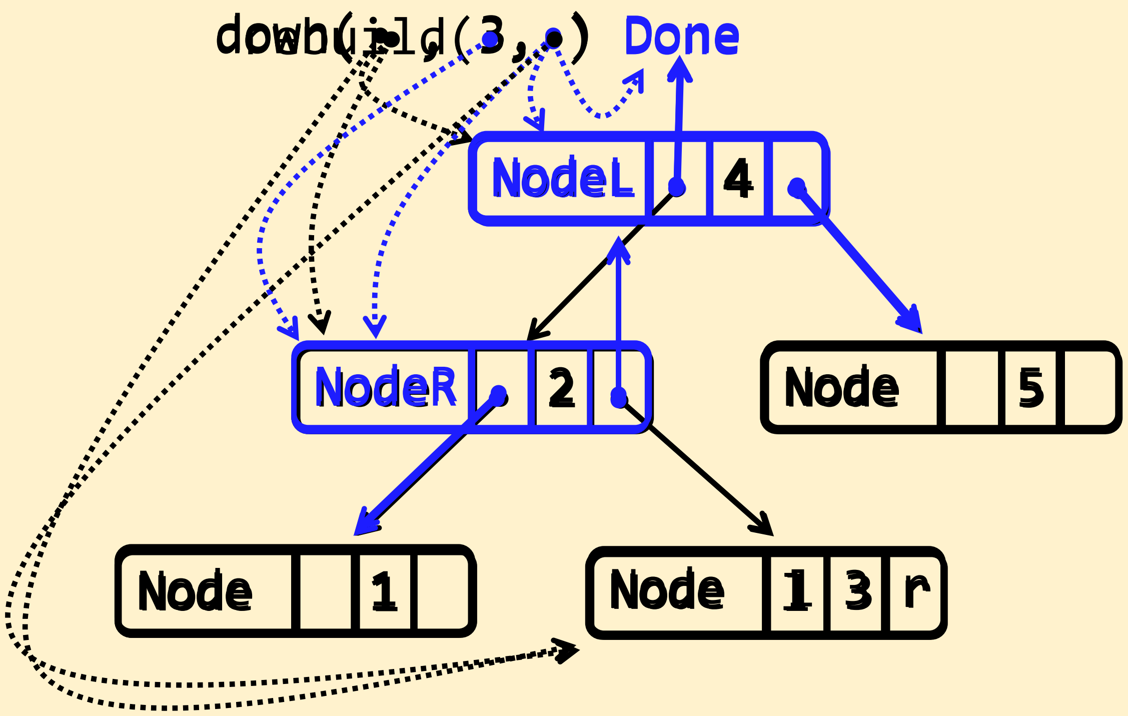
GÉRARD HUET

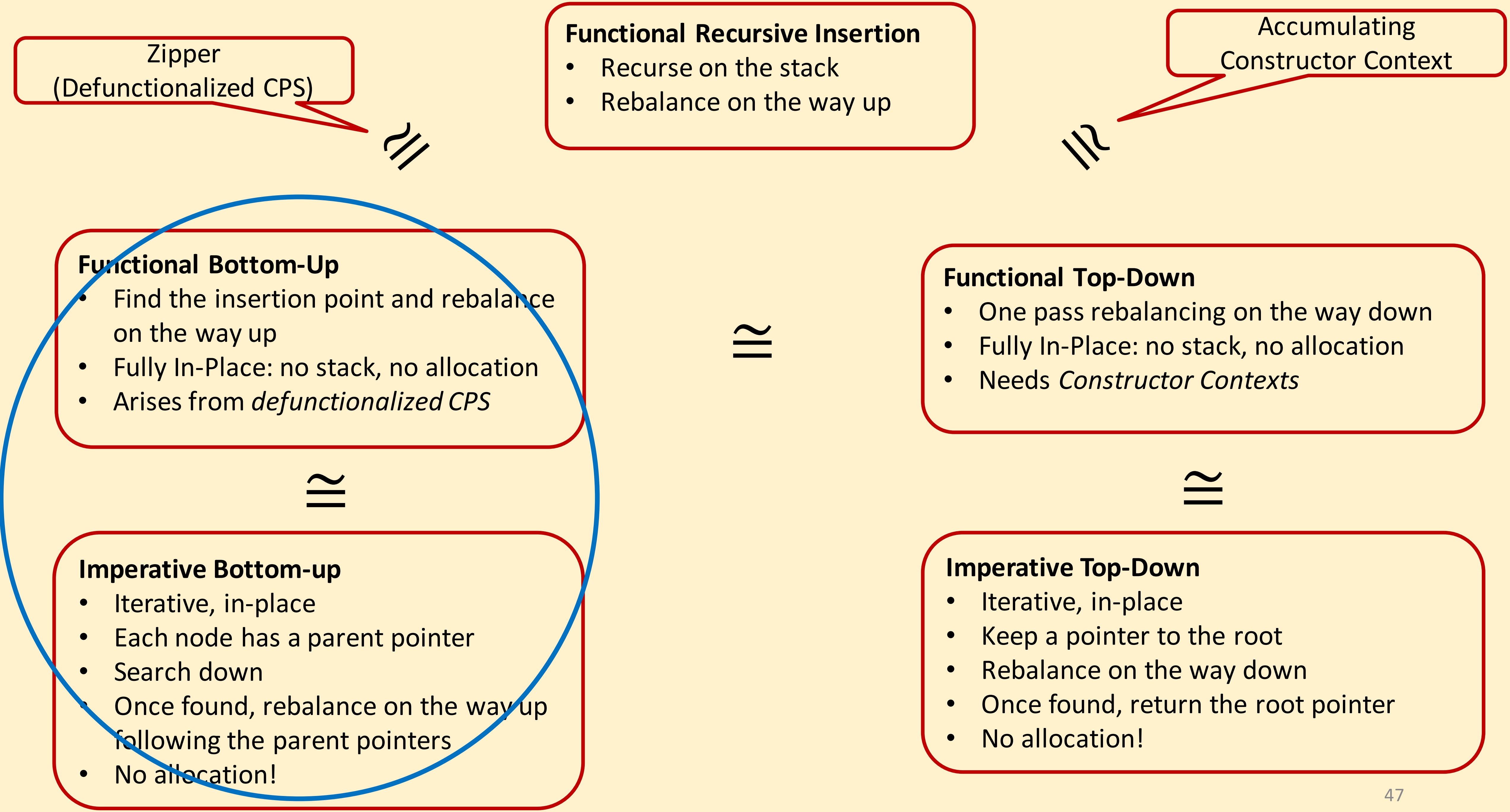
INRIA Rocquencourt, France

(1997)

Inserting an element – fully in-place

```
fip(1) fun down( t : tree, k : key, z : zipper )
  match t ●
    Node(l,x,r) ->
      if  x < k then down(r,k,NodeR(l,x,z)) ●
      elif x > k then down ( l, k, NodeL(z,x,r) ) ●
      else rebuild( z, Node(l,k,r) ) ●
    Leaf -> rebuild( z, Node(Leaf,k,Leaf) )
```





Functional Recursive Insertion

- Recurse on the stack
- Rebalance on the way up

Zipper
(Defunctionalized CPS)

≅

Accumulating
Constructor Context

≅

Functional Bottom-Up

- Find the insertion point and rebalance on the way up
- Fully In-Place: no stack, no allocation
- Arises from *defunctionalized CPS*

≅

Functional Top-Down

- One pass rebalancing on the way down
- Fully In-Place: no stack, no allocation
- Needs *Constructor Contexts*

≅

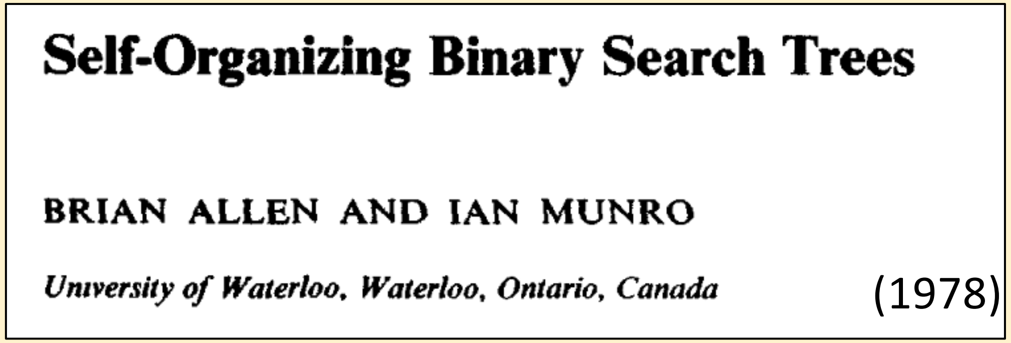
Imperative Bottom-up

- Iterative, in-place
- Each node has a parent pointer
- Search down
- Once found, rebalance on the way up following the parent pointers
- No allocation!

≅

Imperative Top-Down

- Iterative, in-place
- Keep a pointer to the root
- Rebalance on the way down
- Once found, return the root pointer
- No allocation!



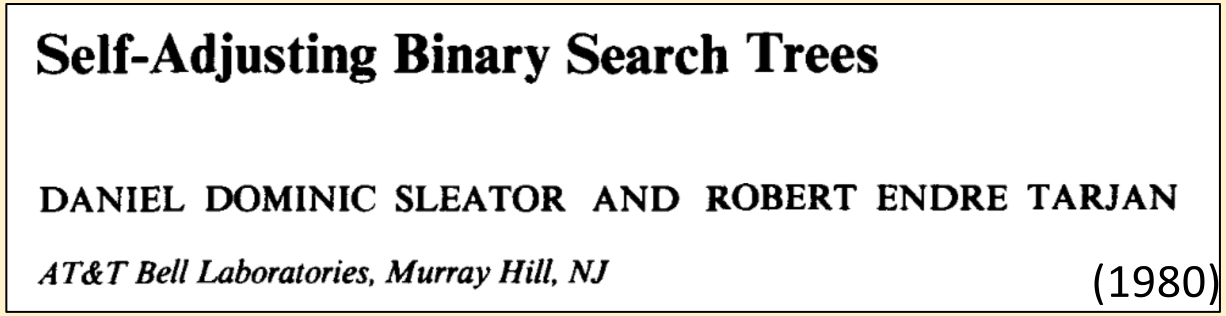
Self-Organizing Binary Search Trees

BRIAN ALLEN AND IAN MUNRO

University of Waterloo, Waterloo, Ontario, Canada

(1978)

- This corresponds to Allen and Munro’s (1978) bottom-up imperative algorithm
- And we can do the same for **Splay trees**, bottom-up and top-down (and it turns out the published bottom-up and top-down splay algorithms are *not* equivalent where only bottom-up has the *transformation* property)

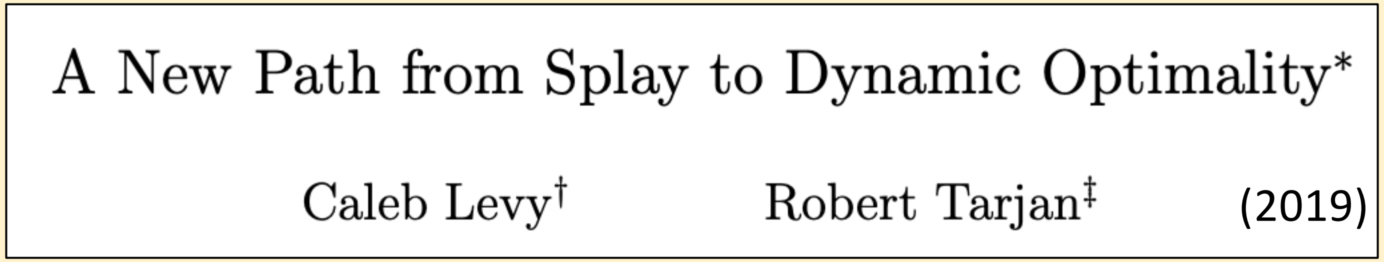


Self-Adjusting Binary Search Trees

DANIEL DOMINIC SLEATOR AND ROBERT ENDRE TARJAN

AT&T Bell Laboratories, Murray Hill, NJ

(1980)



A New Path from Splay to Dynamic Optimality*

Caleb Levy† Robert Tarjan‡

(2019)

Benchmarks

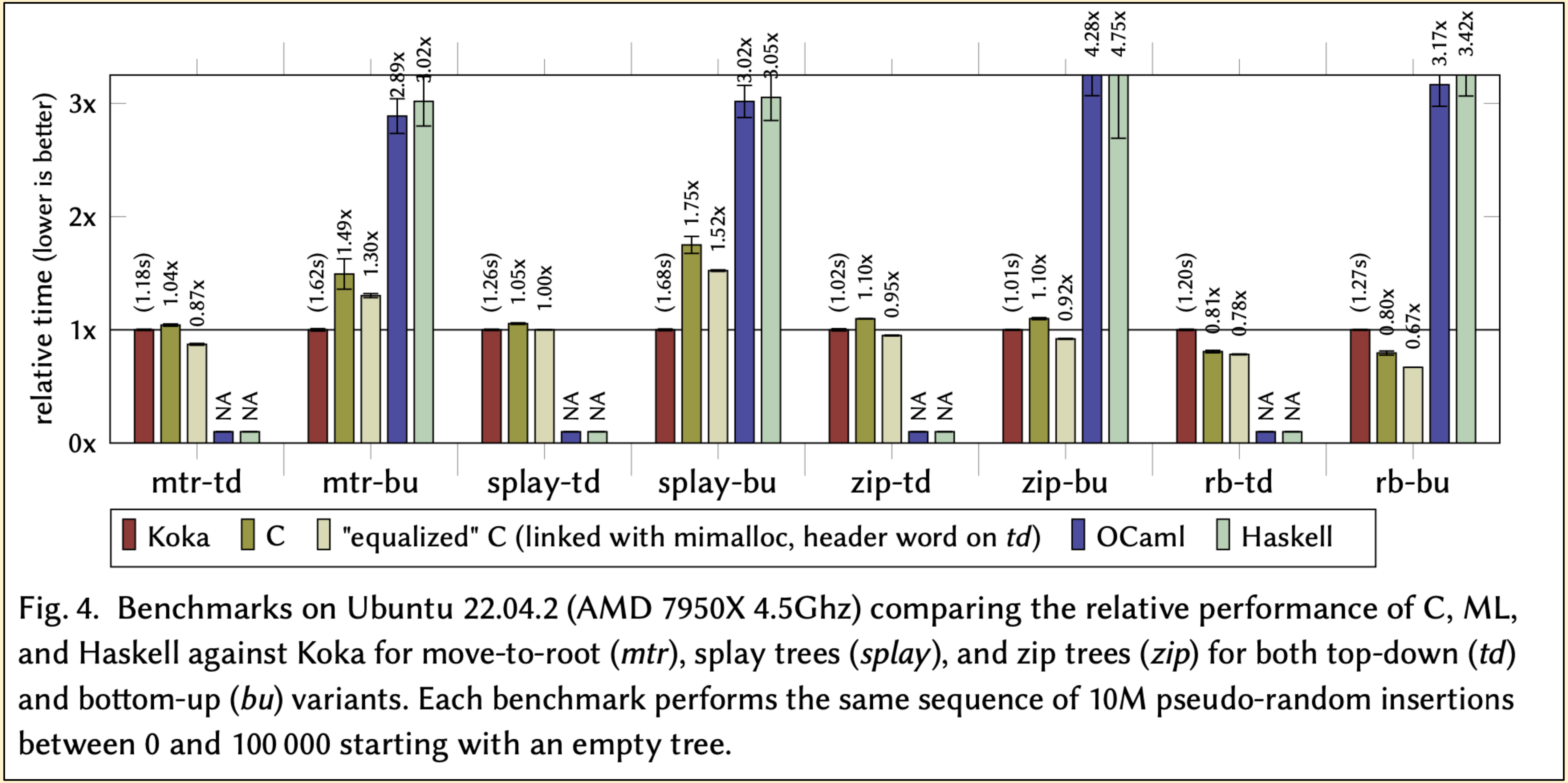
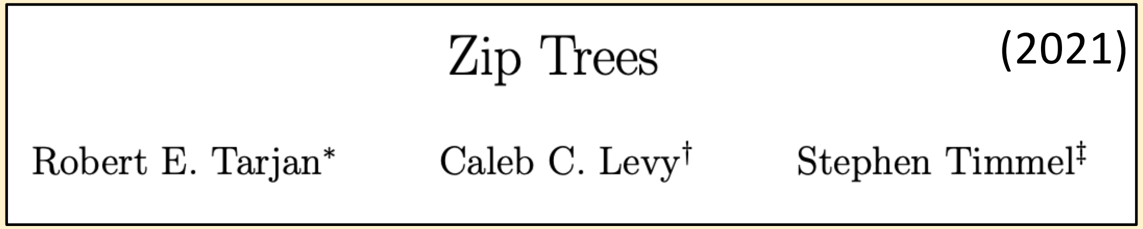


Fig. 4. Benchmarks on Ubuntu 22.04.2 (AMD 7950X 4.5Ghz) comparing the relative performance of C, ML, and Haskell against Koka for move-to-root (*mtr*), splay trees (*splay*), and zip trees (*zip*) for both top-down (*td*) and bottom-up (*bu*) variants. Each benchmark performs the same sequence of 10M pseudo-random insertions between 0 and 100 000 starting with an empty tree.

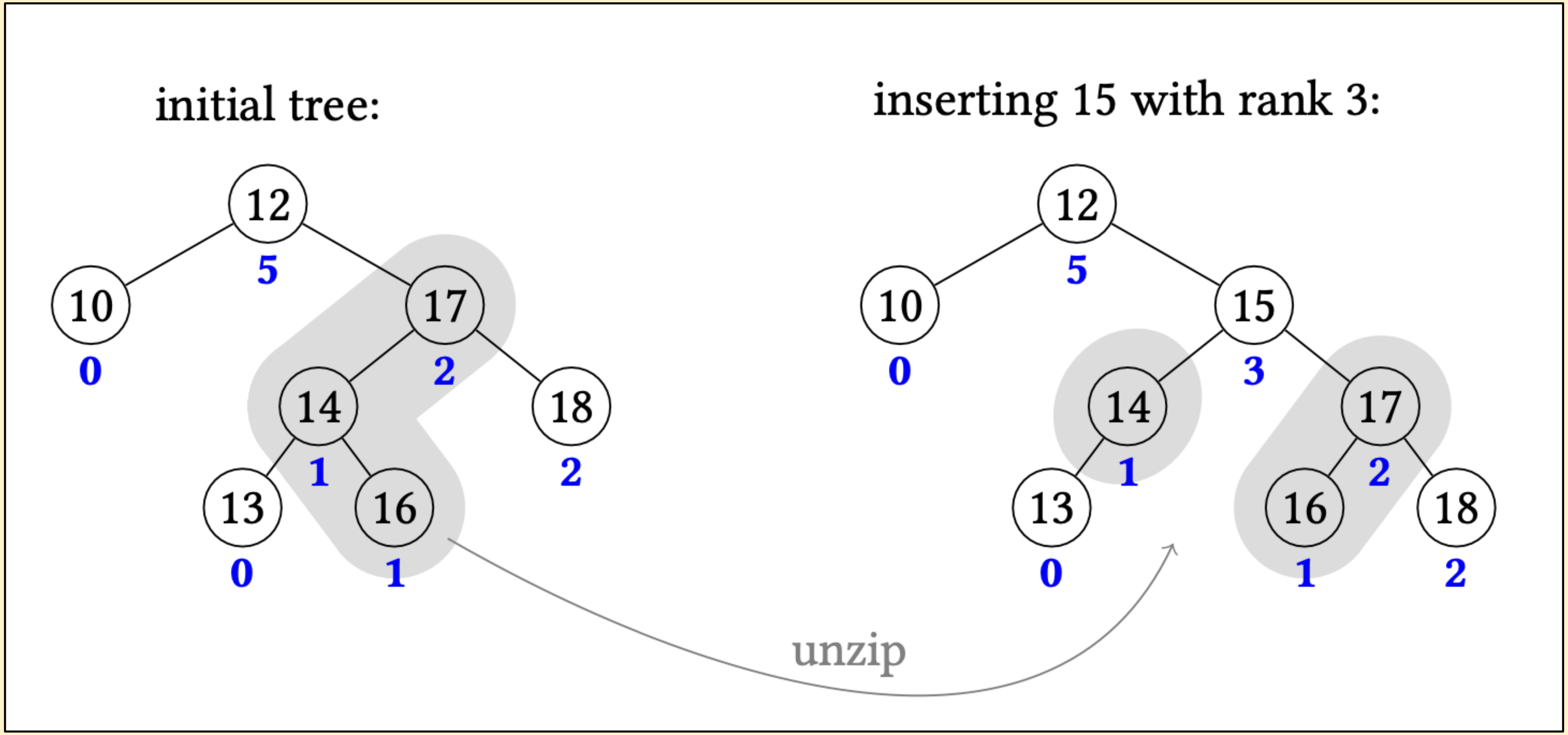
Zip Trees



Zip Trees (2021)

Robert E. Tarjan[*] Caleb C. Levy[†] Stephen Timmel[‡]

- The functional equivalent of skip lists:



Zip Tree Insertion

```
fun insert( t : ztree, rank : rank, k : key ) : ztree
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk, x), (rank, k))
      -> if (x < k) then Node(rnk, l, x, insert(r,rank,k))
                    else Node(rnk, insert(l,rank,k), x, r)
  _ -> Node(rank, smaller(t,k), x, bigger(t,k))            // unzip
```

- unzip *is* move-to-root insertion!

```
fun mtr-insert( t : tree, k : key ) : tree
  Node(smaller(t,k), k, bigger(t,k))
```

Zip Trees: Top-Down

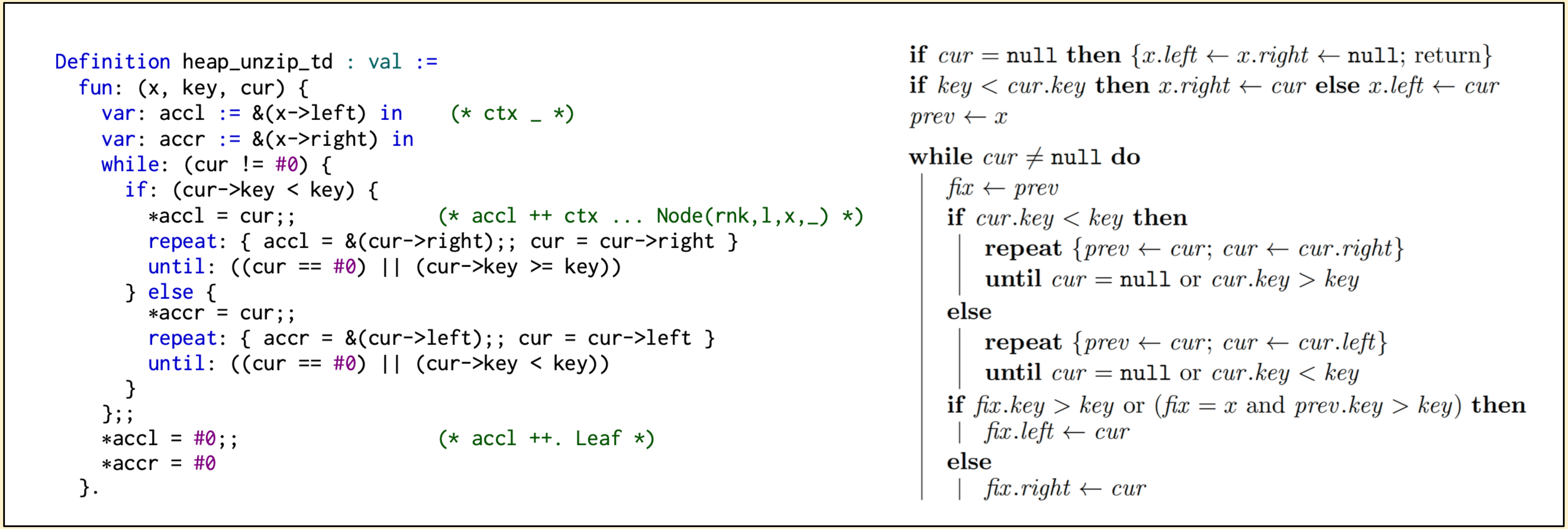
- Again, we accumulate the tree above our insertion point:

```
fip(1) fun insert-td( t : ztree, rank : rank, k : key ) : ztree
  down-td( t, rank, k, ctx _)


fip(1) fun down-td( t : ztree, rank : rank, k : key, acc : ctx<ztree> )
  match t
    Node(rnk,l,x,r) | is-higher-rank( (rnk,x), (rank,k) )
      -> if (x < k) then down-td( r, rank, k, acc ++ ctx Node(rnk,l,x,_) )
                    else down-td( l, rank, k, acc ++ ctx Node(rnk,_,x,r) )
    _ -> acc ++. mtr-insert-td(t,k)   // unzip
```

A new imperative top-down algorithm

- We derive our imperative unzip from the fast path of the functional one:



```
Definition heap_unzip_td : val :=
  fun: (x, key, cur) {
    var: accl := &(x->left) in     (* ctx _ *)
    var: accr := &(x->right) in
    while: (cur != #0) {
      if: (cur->key < key) {
        *accl = cur;;             (* accl ++ ctx ... Node(rnk,l,x,_) *)
        repeat: { accl = &(cur->right);; cur = cur->right }
        until: ((cur == #0) || (cur->key >= key))
      } else {
        *accr = cur;;
        repeat: { accr = &(cur->left);; cur = cur->left }
        until: ((cur == #0) || (cur->key < key))
      }
    };;
    *accl = #0;;                  (* accl ++. Leaf *)
    *accr = #0
  }.
```

```
if $cur = \texttt{null}$ then $\{x.left \leftarrow x.right \leftarrow \texttt{null}$; return$\}$
if $key < cur.key$ then $x.right \leftarrow cur$ else $x.left \leftarrow cur$
$prev \leftarrow x$

while $cur \neq \texttt{null}$ do
    $fix \leftarrow prev$
    if $cur.key < key$ then
        repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.right\}$
        until $cur = \texttt{null}$ or $cur.key > key$
    else
        repeat $\{prev \leftarrow cur;\ cur \leftarrow cur.left\}$
        until $cur = \texttt{null}$ or $cur.key < key$
    if $fix.key > key$ or $(fix = x$ and $prev.key > key)$ then
        $fix.left \leftarrow cur$
    else
        $fix.right \leftarrow cur$
```

- And we can prove it correct again.

Benchmarks

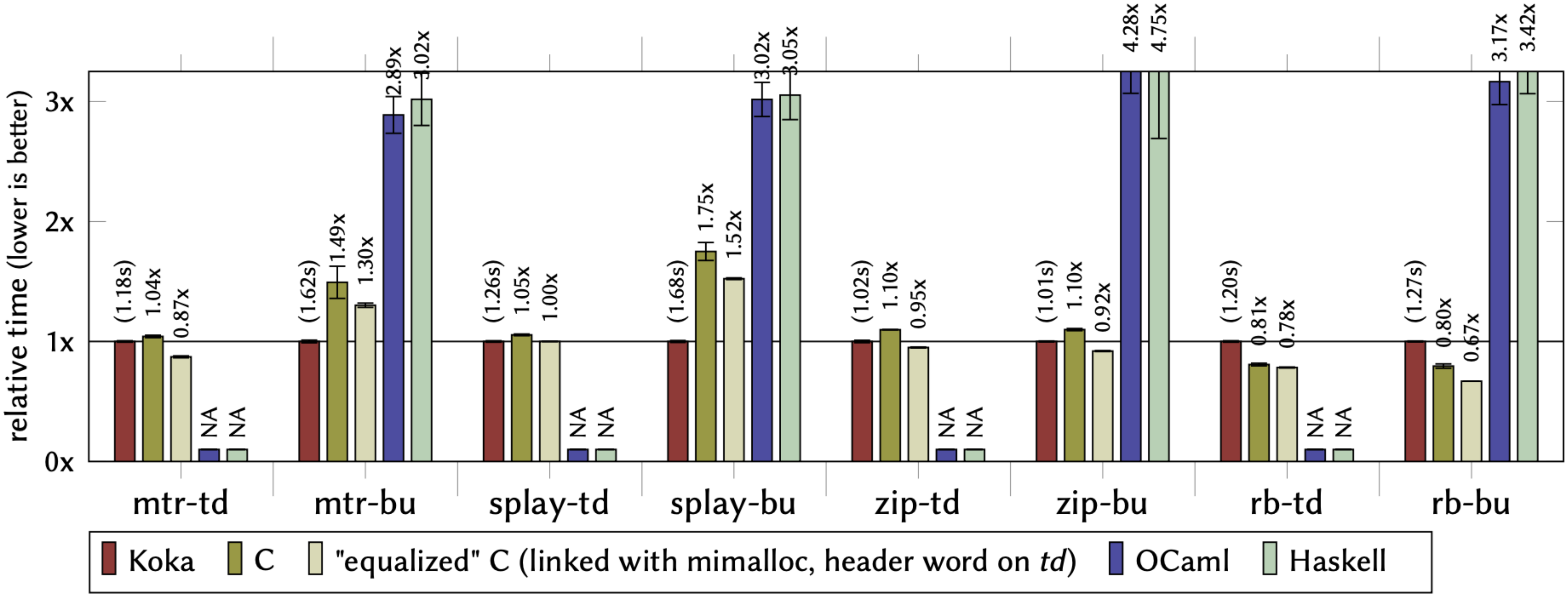


Fig. 4. Benchmarks on Ubuntu 22.04.2 (AMD 7950X 4.5Ghz) comparing the relative performance of C, ML, and Haskell against Koka for move-to-root (*mtr*), splay trees (*splay*), and zip trees (*zip*) for both top-down (*td*) and bottom-up (*bu*) variants. Each benchmark performs the same sequence of 10M pseudo-random insertions between 0 and 100 000 starting with an empty tree.

Thank you!

- Koka: https://koka-lang.github.io

- "FP^2: Fully in-Place Functional Programming",
  Anton Lorenzen, Daan Leijen, and Wouter Swierstra, ICFP'23
  https://www.microsoft.com/en-us/research/publication/fp2-fully-in-place-functional-programming/

- "The Functional Essence of Imperative Binary Search Trees",
  Anton Lorenzen, Daan Leijen, Wouter Swierstra, Sam Lindley, PLDI'24 (conditional)
  https://www.microsoft.com/en-us/research/people/daan/