Programmers' Eternal Dilemma: The Choice Between

Pure Reasoning

&

Raw Performance

Read our paper:

Anton Lorenzen   Daan Leijen   Wouter Swierstra   Sam Lindley

Functional Binary Search Trees

- Great for **reasoning**:
  - Typical exercise in verification courses
  - Can be automatically synthesized from a specification
  - Amortized time complexity can be proven automatically
- But **performance** is lacking..
- Let's consider *splay trees* as an example

```
fun splay(pivot, t)
  match t
    Node(a, x, b) -> if x > pivot then match a // search for pivot in left subtree
      Node(a1, y, a2) -> if y < pivot then      // search for pivot in right subtree
        val (small, big) = splay(pivot, a2)     // recursively partition the subtree
        (Node(a1, y, small), Node(big, x, b))   // add subtrees to the appropriate pile
    ...
```

- New keys are inserted at the root with two children: the existing keys smaller and bigger than the new key, as computed by *splay* (above).
- But this allocates $\mathcal{O}(\log n)$ new nodes and uses $\mathcal{O}(\log n)$ stack. ***Can we do better?***

In-Place Reuse

- One can safely update an immutable data structure in-place given unique ownership
- We use reference-counting to detect at runtime which blocks are unique

For unique data this function is *fully in-place*

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

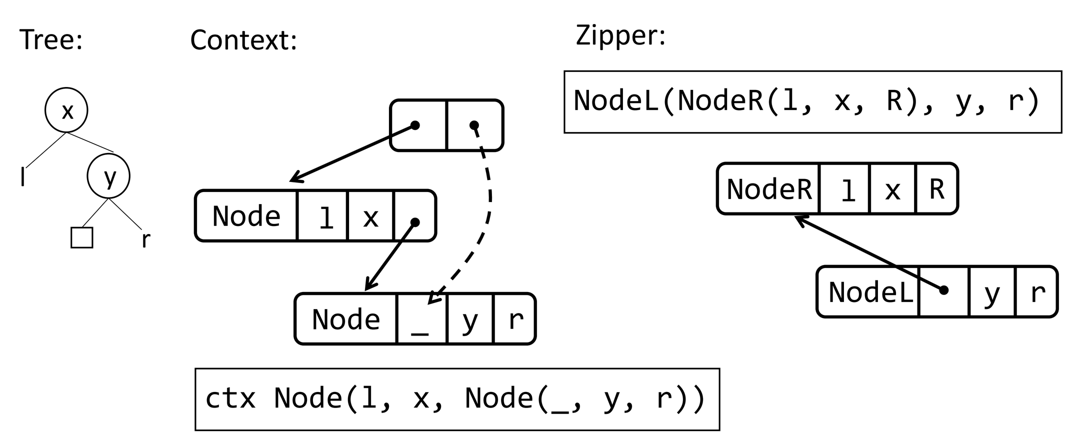
reuse opportunity

- Our compiler then detects reuse opportunities and removes allocations:

```
fip fun reverse( xs, acc )
  match xs
    Cons(x,xx) -> val loc = if is-unique(xs) // refcount==1?
        then &xs // then reuse the memory, else allocate:
        else { dup(x); dup(xx); decref(xs); alloc(2) }
      reverse( xx, Cons@loc(x, acc))
    Nil -> Nil
```

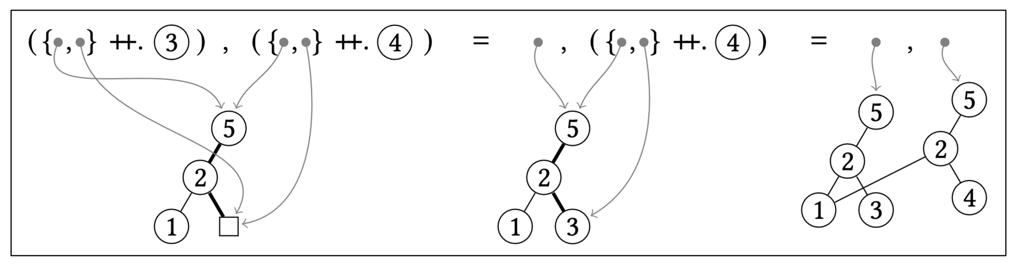
Constructor Contexts & Zippers

- Both describe data structures with constant time access to a single hole
- Constructor contexts store the path from the *root to the hole* and an extra pointer to the hole
- But zippers invert pointers: they store the path from the *hole to the root*



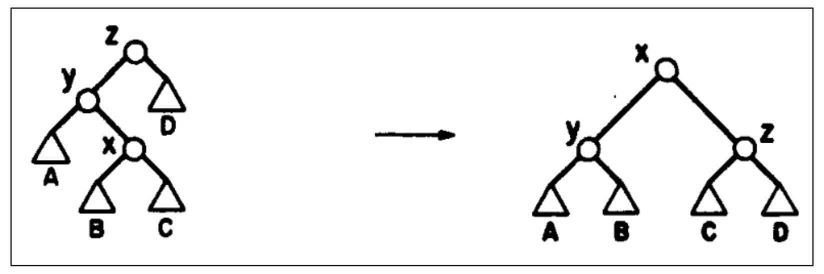

```
(++.) : cctx<a,b> -> b -> a
(++) : cctx<a,b> -> cctx<b,c> -> cctx<a,c>
```

- Both are semantically immutable: If contexts are shared, our runtime copies the path from root to hole and creates a new hole-pointer



Bottom-Up Algorithms

- A bottom-up insertion algorithm traverses down the tree searching for the new key and then *splays on the way back up*
- We represent the path upwards through the tree using a *zipper*



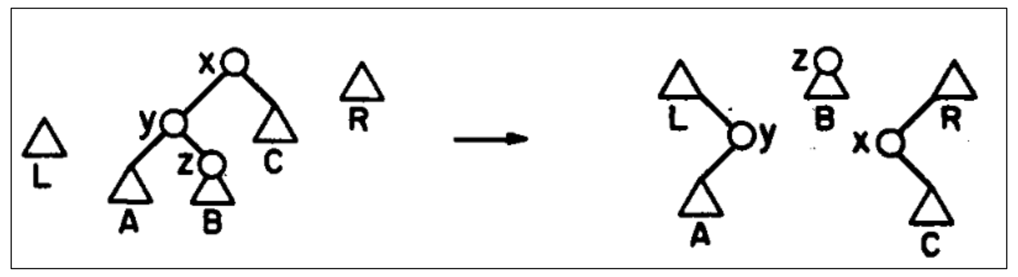
- Our functional algorithm uses $\mathcal{O}(1)$ new nodes and $\mathcal{O}(1)$ stack: It corresponds exactly to the description by Sleator & Tarjan (picture above)

```
fip fun splay( zipper, b, x, c )
  match zipper
    NodeR(a,y,NodeL(up,z,d)) ->
      up.splay( Node(a,y,b), x, Node(c,z,d) )
  ...
```

reuse

Top-Down Algorithms

- A top-down algorithm traverses down the tree searching for the new key and *splays while going down*
- We represent the already-traversed tree using two *constructor contexts* L and R

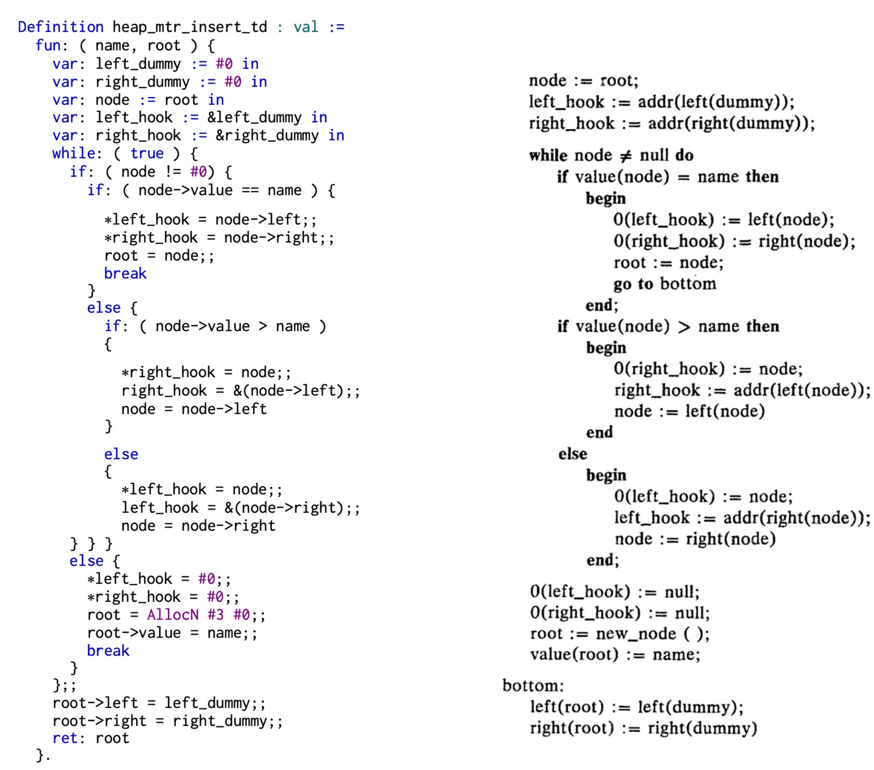


- Our functional algorithm uses $\mathcal{O}(1)$ new nodes and $\mathcal{O}(1)$ stack: It corresponds exactly to the top-down description by Sleator & Tarjan (Fig. 11 of their paper, reproduced above)

```
fip fun splay( t, k, l_ctx, r_ctx )
  match t
    Node(ayzb,x,c) -> if x > k then match ayzb
      Node(a,y,zb) -> if y < k then
        splay(zb,k, l_ctx ++ ctx Node(a,y,_), r_ctx ++ ctx Node(_,x,c))
  ...
```

reuse

Original, imperative algorithms



- We formalize precisely the imperative programs from the original papers in Iris HeapLang
- Using only loop invariants extracted from the functional programs, we can show functional correctness
- This shows that the functional programs capture the essence of the original algorithms

Benchmarks

- Our new algorithms in Koka perform on-par with the original C implementations of *move-to-root*, *splay*, *zip* and *red-black* trees