Stackless traversals with Zippers (or: Borrowing for Perceus Reference Counting)

Anton Lorenzen

University of Bonn

May 14, 2022

Contents

- Link inversion
- Functional programming
- Zippers
- CPS transformation
- Benchmarks

- Reference counting
- Perceus & reuse analysis
- Borrowing

Traversing a tree

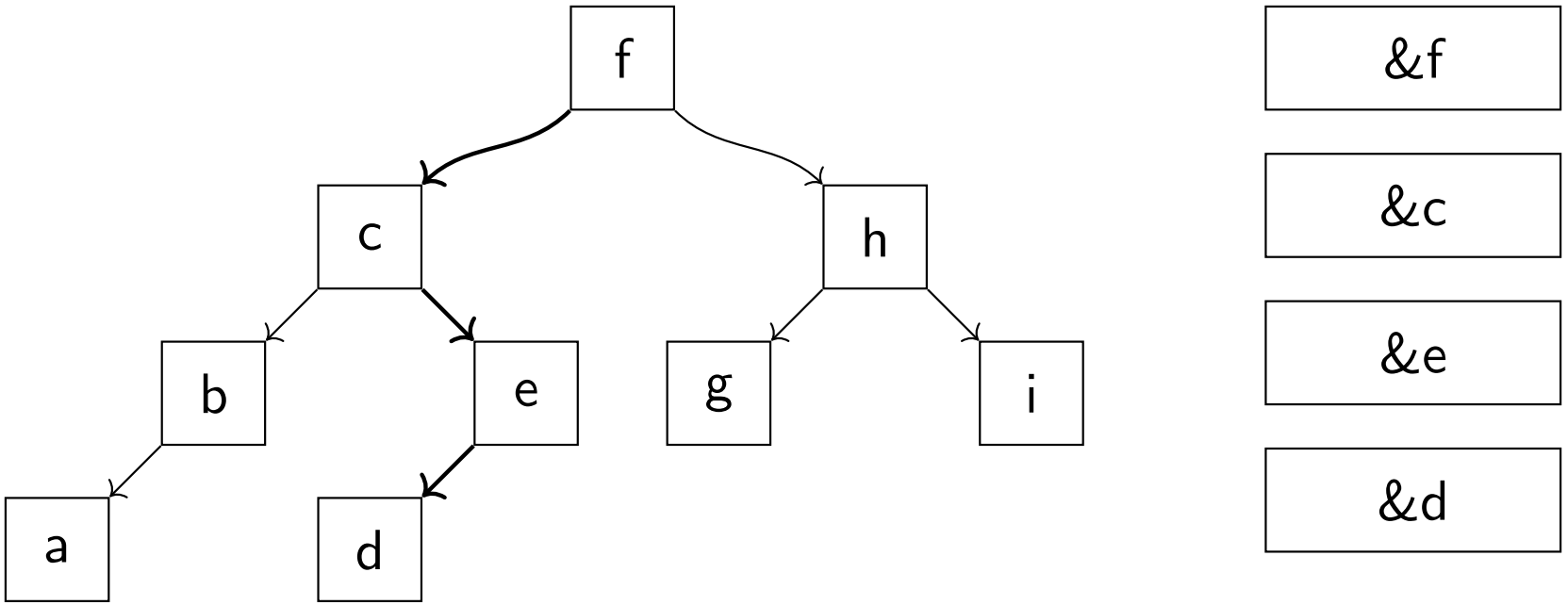


Figure: A binary tree, with a stack for visiting the node `d`

Traversing a tree . . . with link inversion

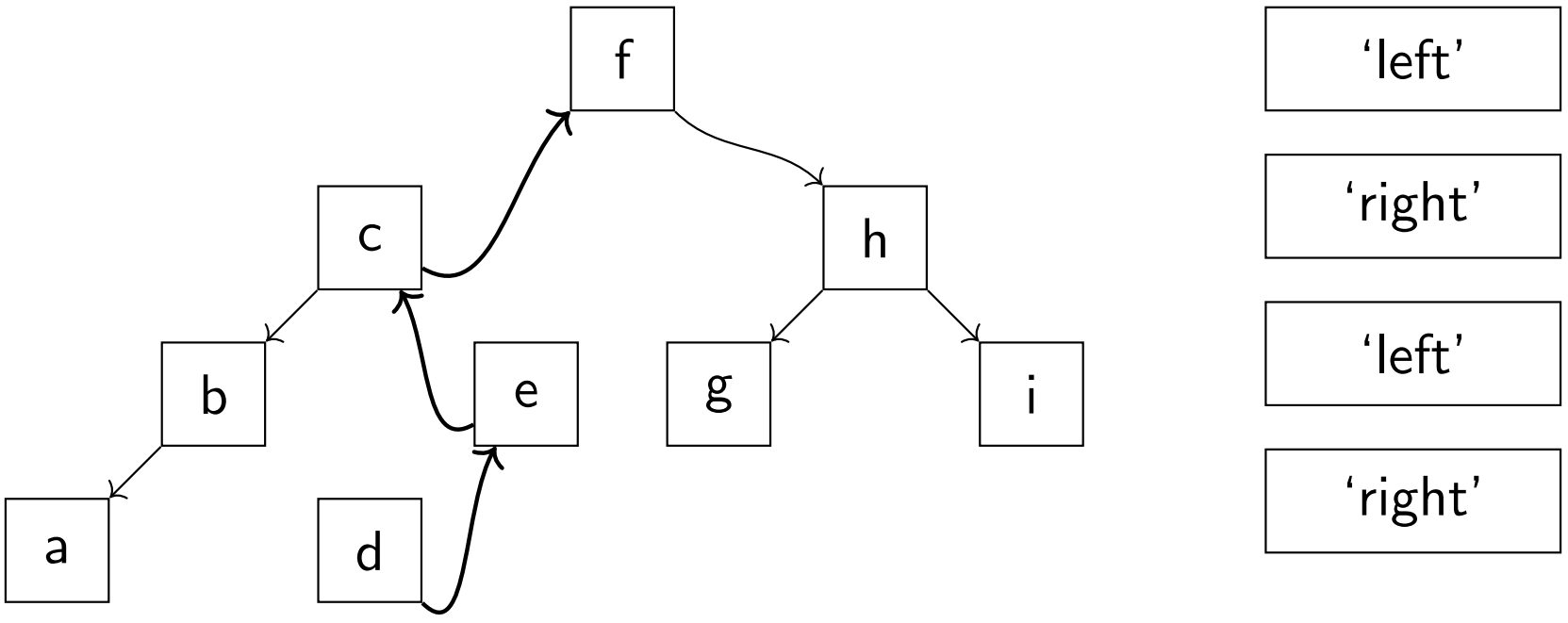


Figure: A binary tree with link-inversion on the path to `d`

Traversing a tree . . . with link inversion

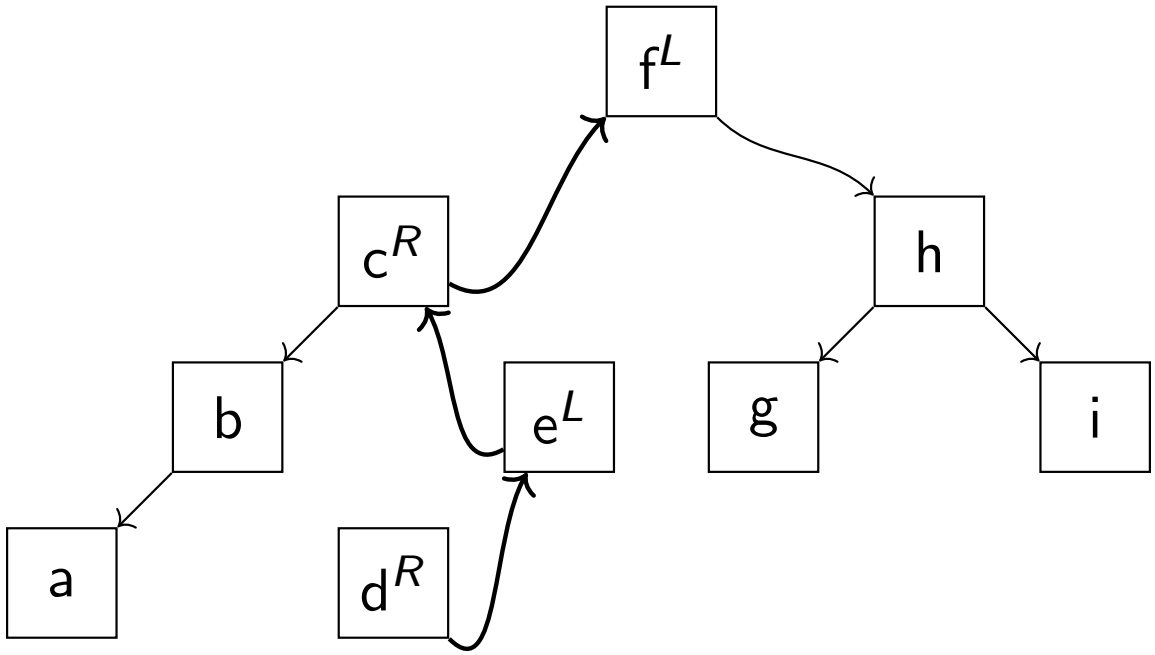


Figure: A binary tree with link-inversion on the path to `d`

Algebraic data types

```
type bool
  False
  True

type complex
  Complex( real : double, imaginary : double )

type result
  Success( result : int )
  Error( msg : string )
```

Standard tree traversal

```
type tree<a>
  Bin( left : tree<a>
     , elem : a
     , right : tree<a> )
  Tip

fun map(t : tree<a>, f : a -> b) : tree<b>
  match(t)
    Bin(l, x, r)
      -> Bin(map(l, f), f(x), map(r, f))
    Tip -> Tip
```

Traversing a tree ... with link inversion (again)

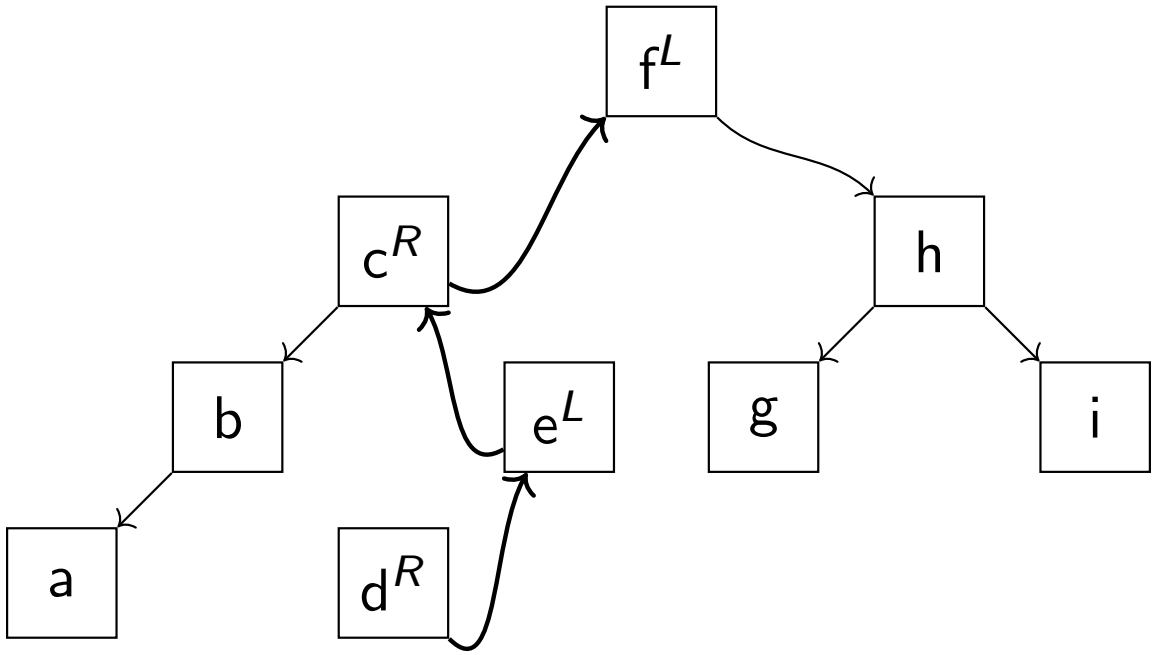


Figure: A binary tree with link-inversion on the path to `d`

Zippers

```
type zipper<a,b>
  BinL( left : zipper<a,b>
      , elem : a
      , right : tree<a> )
  BinR( left : tree<b>
      , elem : b
      , right : zipper<a,b> )
  Top
```

CPS transformation

```
fun map(t : tree<a>, f : a -> b) : tree<b>
  match(t)
    Bin(l, x, r)
      -> Bin(map(l, f), f(x), map(r, f))
    Tip -> Tip
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> k(Bin(map(l, f), f(x), map(r, f)))
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> k(Bin( map-cps(l, f, id), f(x)
              , map-cps(r, f, id)))
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> (fn(lDone) {
            k(Bin( lDone, f(x)
                 , map-cps(r, f, id)))
          })( map-cps(l, f, id) )
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> map-cps(l, f, fn(lDone) {
           k(Bin( lDone, f(x)
                , map-cps(r, f, id)))
         })
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> map-cps(l, f, fn(lDone) {
           let y = f(x)
           map-cps(r, f, fn(rDone) {
             k(Bin(lDone, y, rDone))
           })
         })
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> map-cps(l, f, fn(lDone) {
           // free variables: r, k, x, f
           let y = f(x)
           map-cps(r, f, fn(rDone) {
             // free variables: lDone, k, y
             k(Bin(lDone, y, rDone))
           })
         })
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, id)
```

CPS transformation

```
fun map-cps(t, f, k)
  match(t)
    Bin(l, x, r)
      -> map-cps(l, f, fn(lDone, f) {
           // free variables: r, k, x
           let y = f(x)
           map-cps(r, f, fn(rDone, f) {
             // free variables: lDone, k, y
             k(Bin(lDone, y, rDone), f)
           })
         })
    Tip -> k(Tip)

fun map(t : tree<a>, f : a -> b) : tree<b>
  map-cps(t, f, fn(done, f) { done })
```

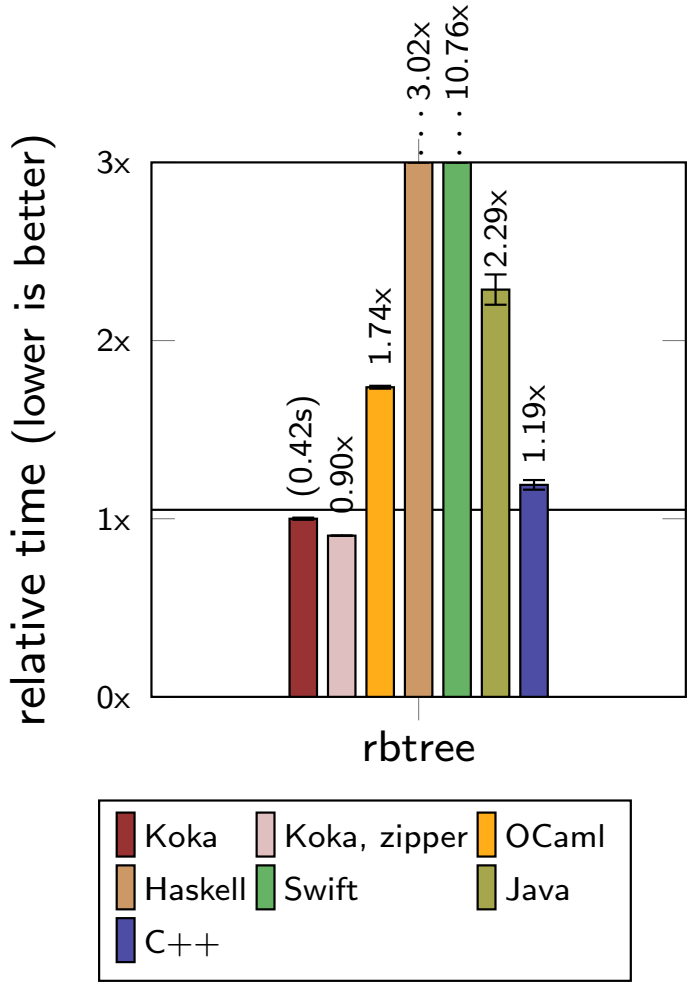
Zippers

```
type zipper<a,b>
  // fn(lDone, f) { ... }
  // with free variables: r, k, x
  BinL( left : zipper<a,b>
      , elem : a
      , right : tree<a> )

  // fn(rDone, f) { ... }
  // with free variables: lDone, k, y
  BinR( left : tree<b>
      , elem : b
      , right : zipper<a,b> )

  // fn(done, f) { done }
  // with no free variables
  Top
```

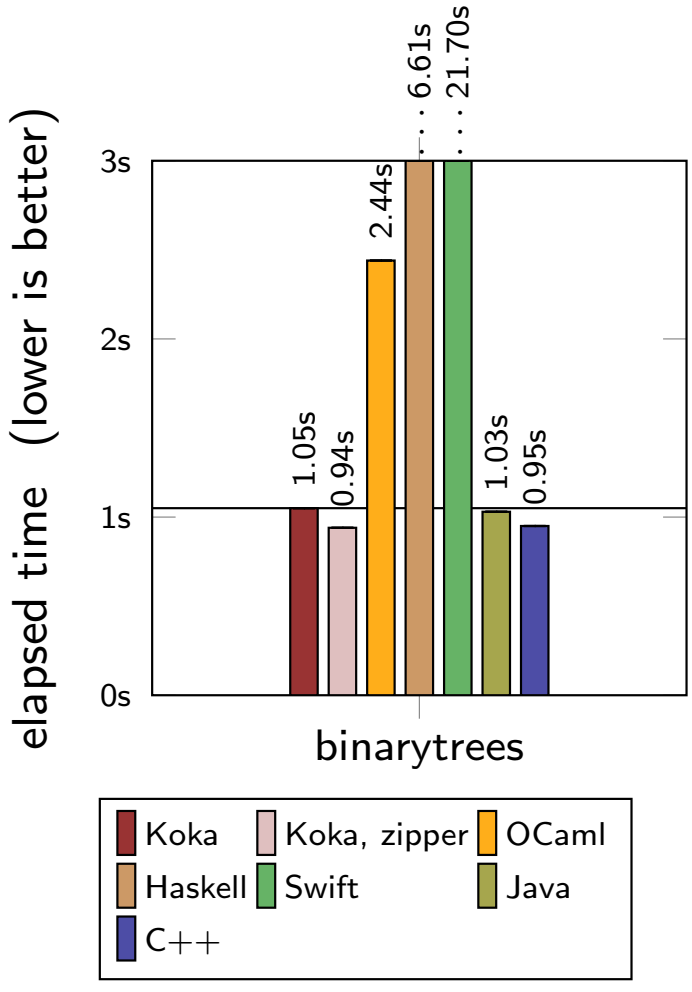
Link-inverted tree traversal

```
fun map(t : tree<a>, f : a -> b) : tree<b>
  down(t, f, Top)

fun down(t, f, z)
  match(t)
    Bin(l, x, r) -> down(l, f, BinL(z, x, r))
    Tip -> up(Tip, f, z)

fun up(t, f, z)
  match(z)
    BinL(z', x, r)
      -> down(r, f, BinR(t, f(x), z'))
    BinR(l, x, z') -> up(Bin(l, x, t), f, z')
    Top -> t
```

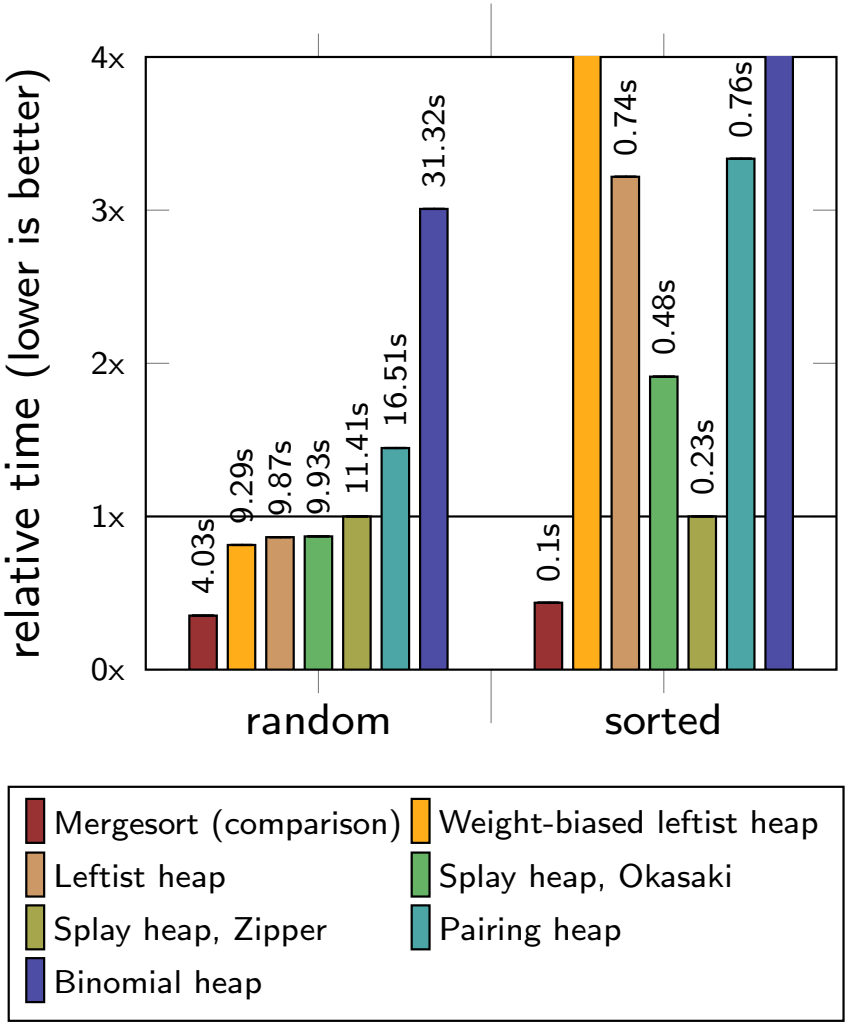
Benchmarks


relative time (lower is better)
3x
2x
1x
0x
(0.42s)
0.90x
1.74x
3.02x
10.76x
2.29x
1.19x
rbtree
Koka
Koka, zipper
OCaml
Haskell
Swift
Java
C++

Benchmarks


elapsed time (lower is better)
3s
2s
1s
0s
1.05s
0.94s
2.44s
6.61s
21.70s
1.03s
0.95s
binarytrees
Koka
Koka, zipper
OCaml
Haskell
Swift
Java
C++

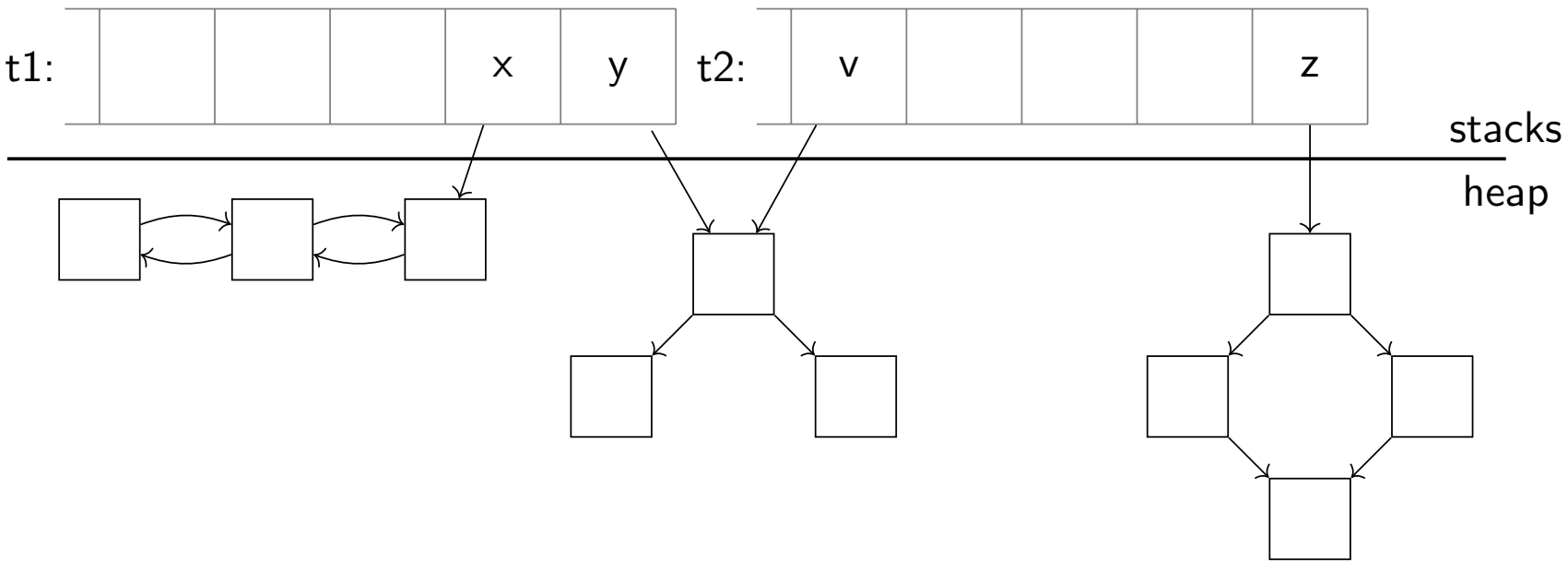
Benchmarks


relative time (lower is better)
4x
3x
2x
1x
0x
4.03s
9.29s
9.87s
9.93s
11.41s
16.51s
31.32s
0.1s
0.74s
0.48s
0.23s
0.76s
random
sorted
Mergesort (comparison)
Weight-biased leftist heap
Leftist heap
Splay heap, Okasaki
Splay heap, Zipper
Pairing heap
Binomial heap

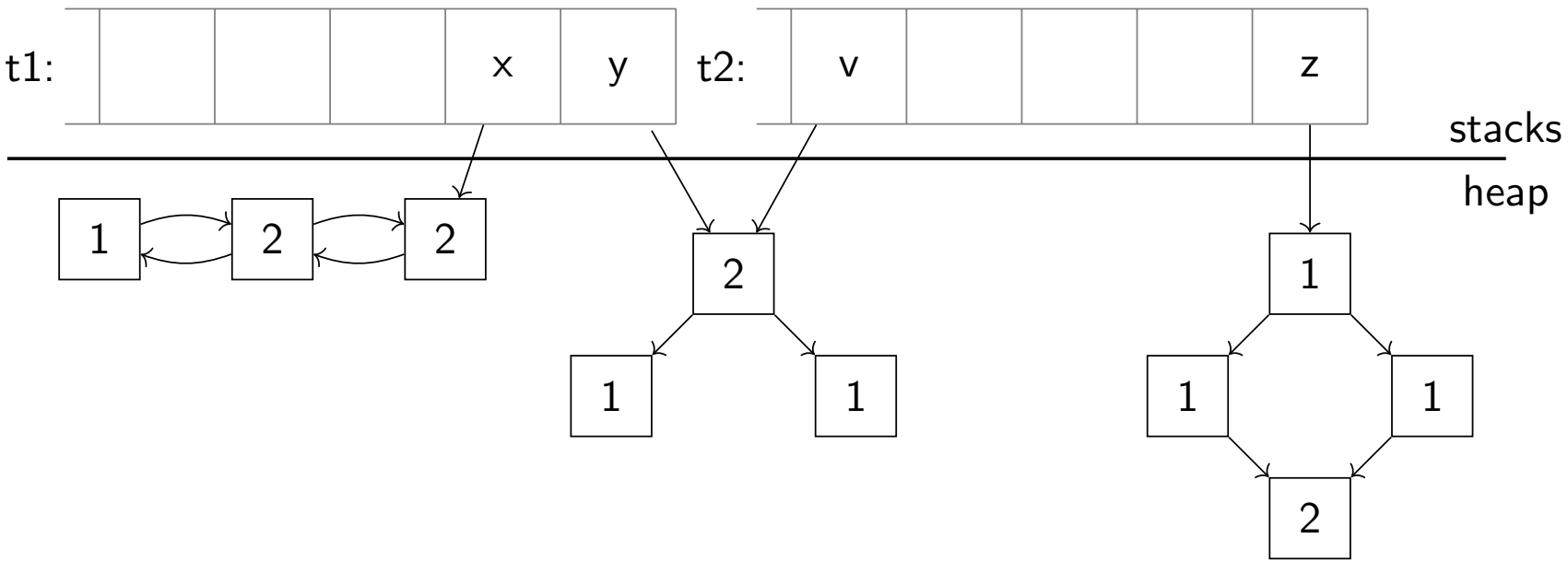
Link-inverted tree traversal

```
fun map(t : tree<a>, f : a -> b) : tree<b>
  down(t, f, Top)

fun down(t, f, z)
  match(t)
    Bin(l, x, r) -> down(l, f, BinL(z, x, r))
    Tip -> up(Tip, f, z)

fun up(t, f, z)
  match(z)
    BinL(z', x, r)
      -> down(r, f, BinR(t, f(x), z'))
    BinR(l, x, z') -> up(Bin(l, x, t), f, z')
    Top -> t
```

Reference Counting



Reference Counting



Scope-based Reference Counting

```
int foo(std::shared_ptr<std::vector<int>> v) {
    do_something(v);
    allocate_a_lot();
    // v.~shared_ptr<std::vector<int>>();
}
```

Perceus Reference Counting

```
fun foo(v : vector) : int
  do-something(v); // passing ownership
  allocate-a-lot();
```

Reuse analysis

```
fun down(t, f, z)
  match(t)
    Bin(l, x, r) ->
      let ru = if(is-unique(t)) { &t }
               else decref(t); NULL
      down(l, f, BinL@ru(z, x, r))
    Tip -> up(Tip, f, z)
```

Conclusion

Stackless algorithms can be ~10% faster

They can often be derived by the CPS transform

With reuse analysis they are easy to implement