FBIP Crash Course

FBIP 崩溃课程

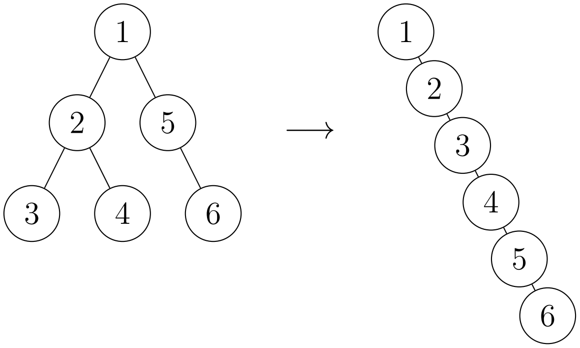
Jiaqi Wu

LUG, NJUPT

May 3, 2025

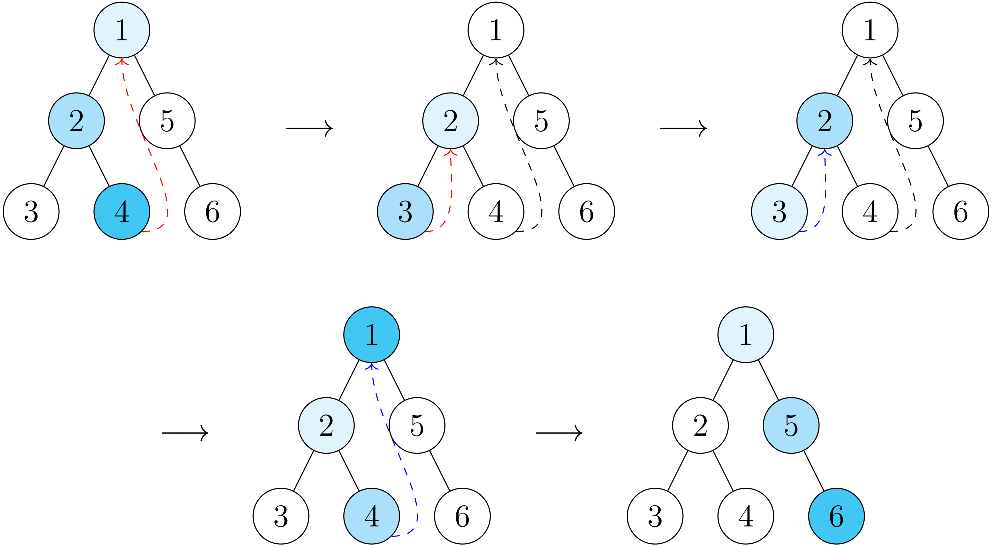
Intro: A simple binary tree traversal problem

Given a binary tree, flatten it into a linked list (à la unbalanced tree) in the same order as a pre-order traversal. (LeetCode 114.)



Flattening must be done in place. In particular, any allocation including stack allocation via recursion is NOT allowed.

Morris traversal using threaded binary tree

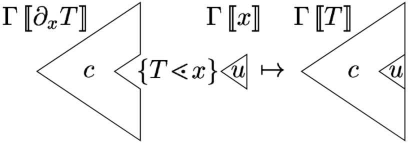


Solution

```c
void flatten(struct Tree *root) {
  struct Tree *cursor = root;
  while (cursor != NULL) {
    struct Tree *spine = cursor->left;
    if (spine == NULL) {
      cursor = cursor->right;
      continue; }
    while (spine->right != NULL && spine->right != cursor) {
      spine = spine->right; }
    if (spine->right == NULL) {
      spine->right = cursor;
      cursor = cursor->left;
    } else { // spine->right == cursor
      struct Tree *t = cursor->right;
      cursor->right = cursor->left;
      cursor->left = NULL;
      spine->right = t;
      cursor = cursor->right; } } }
```

Is it possible to write a purely functional program to achieve the same result?

Zipper



$$
\begin{aligned}
\partial_x x &\mapsto 1 \\
\partial_x (y \backslash x) &\mapsto 0 \\
\partial_x (S + T) &\mapsto \partial_x S + \partial_x T \\
\partial_x (S \times T) &\mapsto \partial_x S \times T + S \times \partial_x T \\
\partial_x (\mu y.F) &\mapsto \mu z.\partial_x F \mid y = \mu y.F + \partial_y F \mid y = \mu y.F \times z \\
\partial_x (F \mid y = S) &\mapsto \partial_x F \mid y = S + \partial_y F \mid y = S \times \partial_x S
\end{aligned}
$$

- Gérard Huet. “The Zipper”. In: JFP (1997)
- Conor McBride. “The derivative of a regular type is its type of onehole contexts”. 2001

Zipper (cont.)

$$\text{tree} \triangleq \mu x.1 + x \times \text{int} \times x$$

$$\partial_x \text{tree} \mapsto \mu x.1 + (\text{tree} \times \text{int} \times x) + (\text{tree} \times \text{int} \times x)$$

```
enum Tree {
    Node(Tree, Int, Tree)
    None
}
```

```
enum Zipper {
    Root
    Right(Tree, Int, Zipper)
    Left(Tree, Int, Zipper)
}
```

Solution (purely functional)

```
fn postorder(t : Tree, z : Zipper, d : Direction) -> Tree {
  match d {
    Down =>
      match t {
        Node(l, v, r) => postorder(l, Right(r, v, z), Down)
        None => postorder(None, z, Up)
      }
    Up =>
      match z {
        Root => t
        Right(r, v, z1) => postorder(r, Left(t, v, z1), Down)
        Left(l, v, z1) => postorder(f(Node(l, v, t)), z1, Up)
      }
  }
}

postorder(t, Root, Down)
```

But this is not in-place, is it?

A new paradigm: Functional But In-Place (FBIP)

- RC Instructions
- Reuse Analysis
- Drop & Reuse Specialization
- Persistence persistence

- Swift. “first non-research language to use an IR with explicit RC instructions”
- Sebastian Ullrich and Leonardo de Moura. “Counting immutable beans: reference counting optimized for purely functional programming”. In: IFL ’19
- Alex Reinking et al. “Perceus: garbage free reference counting with reuse”. In: PLDI 2021
- Anton Lorenzen, Daan Leijen, and Wouter Swierstra. “FP^2: Fully in-Place Functional Programming”. In: ICFP 2023

A new paradigm: Functional But In-Place (FBIP) (cont.)

```
fun map( xs : list⟨a⟩, f : a -> e b ) : e list⟨b⟩ {
  match(xs) {
    Cons(x,xx) -> Cons(f(x), map(xx,f))
    Nil        -> Nil
  }
}
```

(a) A polymorphic map function

$$\begin{tikzcd} \text{(a)} \arrow[d] \arrow[dd, bend right] & & \\ \text{(b)} \arrow[r] & \text{(c)} \arrow[r] & \text{(d)} \\ \text{(e)} \arrow[r] & \text{(f)} \arrow[r] & \text{(g)} \end{tikzcd}$$

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx); drop(xs)
      Cons( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(b) dup/drop insertion (2.2)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx)
      if (is-unique(xs))
        then drop(x); drop(xx); free(xs)
        else decref(xs)
      Cons( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(c) drop specialization (2.3)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      if (is-unique(xs))
        then free(xs)
        else dup(x); dup(xx); decref(xs)
      Cons( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(d) push down dup and fusion (2.3)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx);
      val ru = drop-reuse(xs)
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(e) reuse token insertion (2.4)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      dup(x); dup(xx);
      val ru = if (is-unique(xs))
                 then drop(x); drop(xx); &xs
                 else decref(xs); NULL
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(f) drop-reuse specialization (2.4)

```
fun map( xs, f ) {
  match(xs) {
    Cons(x,xx) {
      val ru = if (is-unique(xs))
                 then &xs
                 else dup(x); dup(xx);
                      decref(xs); NULL
      Cons@ru( dup(f)(x), map(xx, f))
    }
    Nil { drop(xs); drop(f); Nil }
  }
}
```

(g) push down dup and fusion (2.4)

Solution (purely functional)

```
fn postorder(t : Tree, z : Zipper, d : Direction) -> Tree {
  match d {
    Down =>
      match t {
        Node(l, v, r) => postorder(l, Right(r, v, z), Down)
        None => postorder(None, z, Up)
      }
    Up =>
      match z {
        Root => t
        Right(r, v, z1) => postorder(r, Left(t, v, z1), Down)
        Left(l, v, z1) => postorder(f(Node(l, v, t)), z1, Up)
      }
  }
}
```

Questions?

Thanks!