Compiling Pattern Matching to In-Place Modifications

Paul Iannetta*, Laure Gonnord†, Gabriel Radanne‡

*ENS de Lyon, Inria & LIP
†UGA, Grenoble INP, LCIS & LIP
‡Inria & LIP

paul.iannetta@ens-lyon.fr
laure.gonnord@lcis.grenoble-inp.fr
gabriel.radanne@inria.fr

Our Starting Point

```c
struct tree {
  int a;
  struct tree *l, *r;
};

void mirror_left (struct tree *t) {
  t && t->right = t->left;
}
```

- \+ Performance
- \+ No copy
- \+ Flexibility
- – Manual memory handling

```ocaml
type tree =
| Empty
| Node of tree * int * tree

let mirror_left = function
| Empty -> Empty
| Node(l,o,r) -> Node(l,o,l)
```

- \+ Automatic memory handling
- – Hard to finely tune
- – No control on the layout of the type
- – Copies

Our Starting Point

```c
struct tree {
  int a;
  struct tree *l, *r;
};

vo

}
```

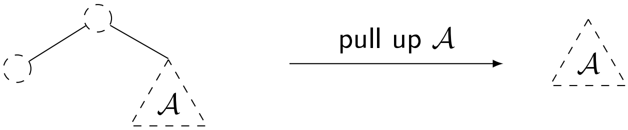
```ocaml
type tree =
| Empty
| Node of tree * int * tree
```

How to take the best of both worlds?

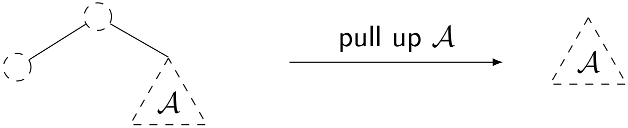
+ No copy
+ Flexibility
− Manual memory handling

− Hard to finely tune
− No control on the layout of the type
− Copies

A Language to Describe Structural Transformations



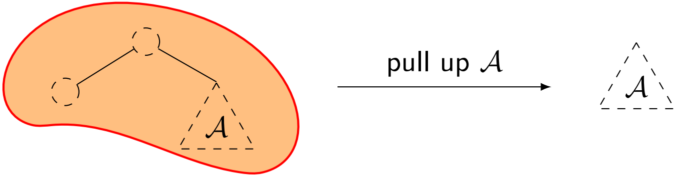
A Language to Describe Structural Transformations




```
type tree = Empty | Node (tree,int,tree)
```

The tree is stored in an array

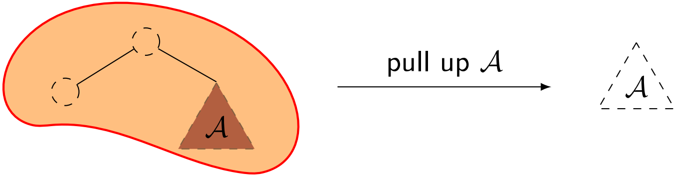
A Language to Describe Structural Transformations




```
type tree = Empty | Node (tree,int,tree)

pull_up (t : tree) : tree = rewrite t {
   | Node(a,i,Node(b,j,c)) -> Node(b,j,c)
   | Empty -> Empty
}
```

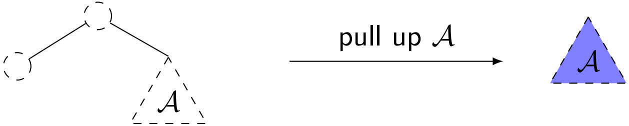
A Language to Describe Structural Transformations




```
type tree = Empty | Node (tree,int,tree)

pull_up (t : tree) : tree = rewrite t {
   | Node(a,i,Node(b,j,c)) -> Node(b,j,c)
   | Empty -> Empty
}
```

A Language to Describe Structural Transformations

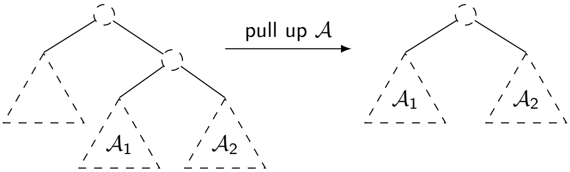



```
type tree = Empty | Node (tree,int,tree)

pull_up (t : tree) : tree = rewrite t {
  | Node(a,i,Node(b,j,c)) -> Node(b,j,c)
  | Empty -> Empty
}
```

Step1: Compute Subtree Movements on Pull Up

```
Node(a,i,Node(b,j,c)) -> Node(b,j,c)
```



$$\llparenthesis\, j : int \mid .2/tree.1/int \to .1/int \,\rrparenthesis$$

$$\llparenthesis\, i : int \mid .1/int \to \emptyset \,\rrparenthesis$$

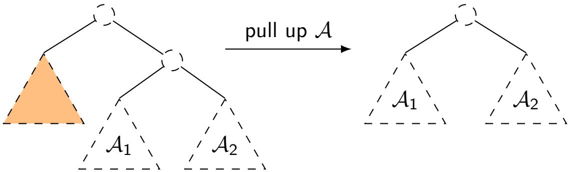
$$\llparenthesis\, b : tree \mid .2/tree.0/tree \to .0/tree \,\rrparenthesis$$

$$\llparenthesis\, a : tree \mid .0/tree \to \emptyset \,\rrparenthesis$$

$$\llparenthesis\, c : tree \mid .2/tree.2/tree \to .2/tree \,\rrparenthesis$$

Step1: Compute Subtree Movements on Pull Up

`Node(a, i ,Node(b, j, c)) -> Node(b, j, c)`



$\llparenthesis\, j : \mathit{int} \mid .2/\mathit{tree}.1/\mathit{int} \to .1/\mathit{int} \,\rrparenthesis$

$\llparenthesis\, i : \mathit{int} \mid .1/\mathit{int} \to \emptyset \,\rrparenthesis$

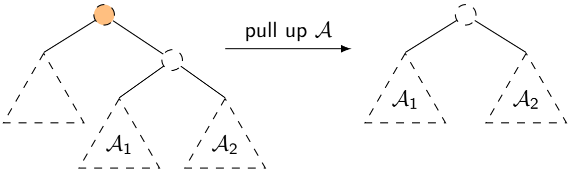
$\llparenthesis\, b : \mathit{tree} \mid .2/\mathit{tree}.0/\mathit{tree} \to .0/\mathit{tree} \,\rrparenthesis$

$\llparenthesis\, a : \mathit{tree} \mid .0/\mathit{tree} \to \emptyset \,\rrparenthesis$

$\llparenthesis\, c : \mathit{tree} \mid .2/\mathit{tree}.2/\mathit{tree} \to .2/\mathit{tree} \,\rrparenthesis$

Step1: Compute Subtree Movements on Pull Up

```
Node(a,i,Node(b,j,c)) -> Node(b,j,c)
```



$(\!| j : int \mid .2/tree.1/int \to .1/int |\!)$

$(\!| i : int \mid .1/int \to \emptyset |\!)$

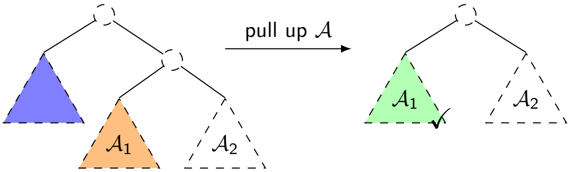
$(\!| b : tree \mid .2/tree.0/tree \to .0/tree |\!)$

$(\!| a : tree \mid .0/tree \to \emptyset |\!)$

$(\!| c : tree \mid .2/tree.2/tree \to .2/tree |\!)$

Step1: Compute Subtree Movements on Pull Up

```
Node( a , i ,Node( b , j , c )) -> Node( b , j , c )
```



$(\!| \, j : int \mid .2/tree.1/int \to .1/int \, |\!)$

$(\!| \, i : int \mid .1/int \to \emptyset \, |\!)$

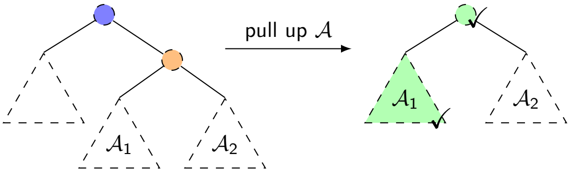
$(\!| \, b : tree \mid .2/tree.0/tree \to .0/tree \, |\!)$

$(\!| \, a : tree \mid .0/tree \to \emptyset \, |\!)$

$(\!| \, c : tree \mid .2/tree.2/tree \to .2/tree \, |\!)$

Step1: Compute Subtree Movements on Pull Up

```
Node( a , i ,Node( b , j , c )) -> Node( b , j , c )
```



$(\!| j : int \mid .2/tree.1/int \rightarrow .1/int |\!)$

$(\!| i : int \mid .1/int \rightarrow \emptyset |\!)$

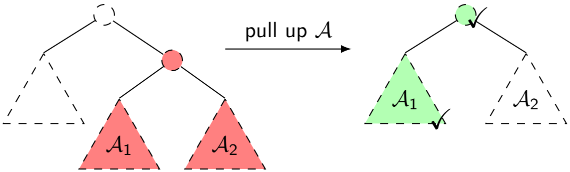
$(\!| b : tree \mid .2/tree.0/tree \rightarrow .0/tree |\!)$

$(\!| a : tree \mid .0/tree \rightarrow \emptyset |\!)$

$(\!| c : tree \mid .2/tree.2/tree \rightarrow .2/tree |\!)$

Step1: Compute Subtree Movements on Pull Up

```
Node( a , i ,Node( b , j ,c)) -> Node( b , j ,c)
```



$(\!|\, j : int \mid .2/tree.1/int \rightarrow .1/int \,|\!)$

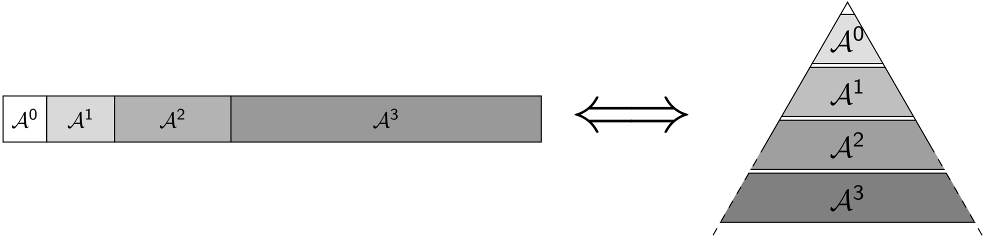
$(\!|\, i : int \mid .1/int \rightarrow \emptyset \,|\!)$

$(\!|\, b : tree \mid .2/tree.0/tree \rightarrow .0/tree \,|\!)$

$(\!|\, a : tree \mid .0/tree \rightarrow \emptyset \,|\!)$

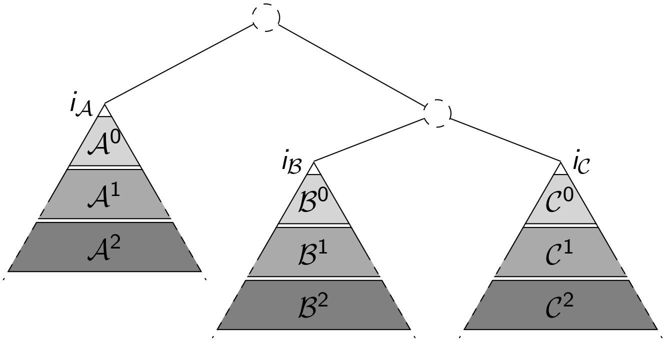
$(\!|\, c : tree \mid .2/tree.2/tree \rightarrow .2/tree \,|\!)$

Use Layers to Subdivide Memory Movements on Pull Up



Step 2: Layer-aware movements (1/2) on Pull Up

```
Node(a, i, Node(b, j, c)) -> Node(b, j, c)
```



$$(\!|\ .0/tree.\,\varphi^{k_0} \rightarrow \text{External}\ |\!) \quad \text{(a)}$$

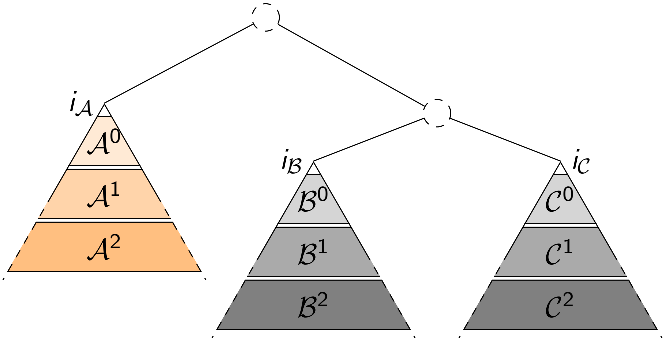
$$(\!|\ .1/int \rightarrow \text{External}\ |\!) \quad \text{(i)}$$

$$(\!|\ .2/tree.\,0/tree.\,\varphi^{k_1} \rightarrow .0/tree.\,\varphi^{k_1}\ |\!) \quad \text{(b)}$$

$$(\!|\ .2/tree.\,1/int \rightarrow .1/int\ |\!) \quad \text{(j)}$$

Step 2: Layer-aware movements (1/2) on Pull Up

```
Node(a, i ,Node(b , j ,c)) -> Node(b , j ,c)
```



$$(\!| \,.\, 0/tree.\, \varphi^{k_0} \rightarrow \text{External} \,|\!) \quad \text{(a)}$$

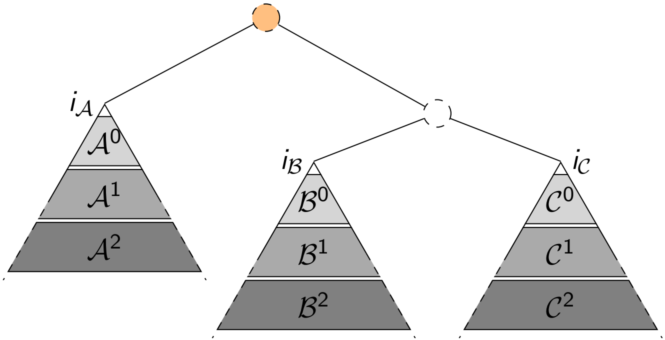
$$(\!| \,.\, 1/int \rightarrow \text{External} \,|\!) \quad \text{(i)}$$

$$(\!| \,.\, 2/tree.\, 0/tree.\, \varphi^{k_1} \rightarrow .\, 0/tree.\, \varphi^{k_1} \,|\!) \quad \text{(b)}$$

$$(\!| \,.\, 2/tree.\, 1/int \rightarrow .\, 1/int \,|\!) \quad \text{(j)}$$

Step 2: Layer-aware movements (1/2) on Pull Up

```
Node( a , i ,Node( b , j ,c)) -> Node( b , j ,c)
```



$$(\!|\, .\,0/\mathit{tree}.\,\varphi^{k_0} \rightarrow \text{External} \,|\!) \quad \text{(a)}$$

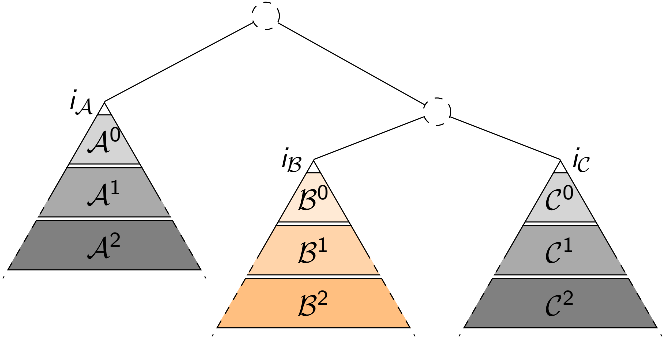
$$(\!|\, .\,1/\mathit{int} \rightarrow \text{External} \,|\!) \quad \text{(i)}$$

$$(\!|\, .\,2/\mathit{tree}.\,0/\mathit{tree}.\,\varphi^{k_1} \rightarrow .\,0/\mathit{tree}.\,\varphi^{k_1} \,|\!) \quad \text{(b)}$$

$$(\!|\, .\,2/\mathit{tree}.\,1/\mathit{int} \rightarrow .\,1/\mathit{int} \,|\!) \quad \text{(j)}$$

Step 2: Layer-aware movements (1/2) on Pull Up

```
Node( a , i ,Node( b , j ,c)) -> Node( b , j ,c)
```



$$(\!|\, .\, 0/\mathit{tree}.\, \varphi^{k_0} \rightarrow \text{External} \,|\!) \quad \text{(a)}$$

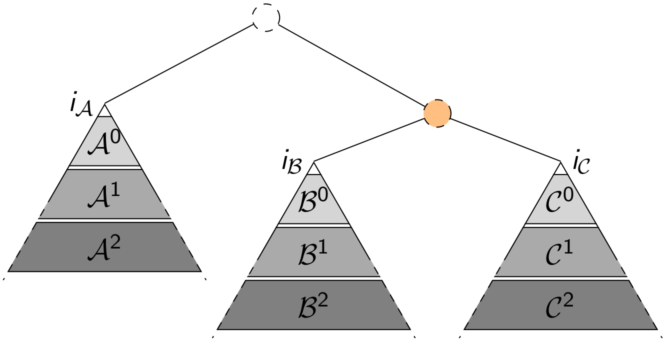
$$(\!|\, .\, 1/\mathit{int} \rightarrow \text{External} \,|\!) \quad \text{(i)}$$

$$(\!|\, .\, 2/\mathit{tree}.\, 0/\mathit{tree}.\, \varphi^{k_1} \rightarrow .\, 0/\mathit{tree}.\, \varphi^{k_1} \,|\!) \quad \text{(b)}$$

$$(\!|\, .\, 2/\mathit{tree}.\, 1/\mathit{int} \rightarrow .\, 1/\mathit{int} \,|\!) \quad \text{(j)}$$

Step 2: Layer-aware movements (1/2) on Pull Up

```
Node( a , i ,Node( b , j ,c)) -> Node( b , j ,c)
```



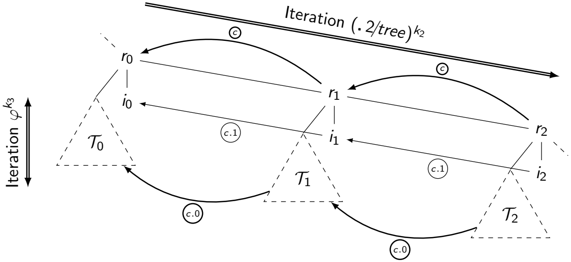
$$\llparenthesis\, .\,0/\mathit{tree}.\,\varphi^{k_0} \to \text{External} \,\rrparenthesis \qquad \text{(a)}$$

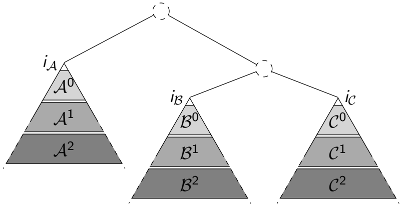
$$\llparenthesis\, .\,1/\mathit{int} \to \text{External} \,\rrparenthesis \qquad \text{(i)}$$

$$\llparenthesis\, .\,2/\mathit{tree}.\,0/\mathit{tree}.\,\varphi^{k_1} \to .\,0/\mathit{tree}.\,\varphi^{k_1} \,\rrparenthesis \qquad \text{(b)}$$

$$\llparenthesis\, .\,2/\mathit{tree}.\,1/\mathit{int} \to .\,1/\mathit{int} \,\rrparenthesis \qquad \text{(j)}$$

Step 2.5: Layer-aware movements (2/2) on Pull Up



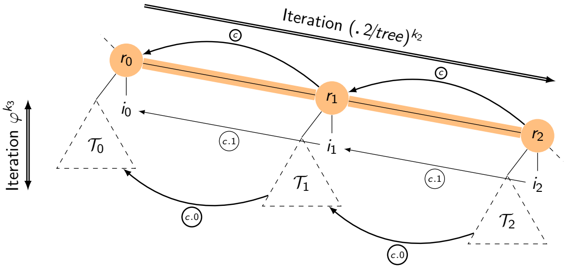


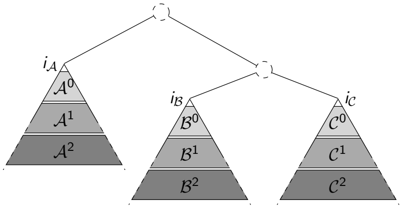
$$\llparenthesis\, (.\,2/tree)^{k_2+2} \rightarrow (.\,2/tree)^{k_2+1} \,\rrparenthesis \qquad (c)$$

$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,1/int \rightarrow (.\,2/tree)^{k_2+1}.\,1/int \,\rrparenthesis \qquad (c1)$$

$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,0/tree.\,\varphi^{k_3} \rightarrow (.\,2/tree)^{k_2+1}.\,0/tree.\,\varphi^{k_3} \,\rrparenthesis \qquad (c0)$$

Step 2.5: Layer-aware movements (2/2) on Pull Up



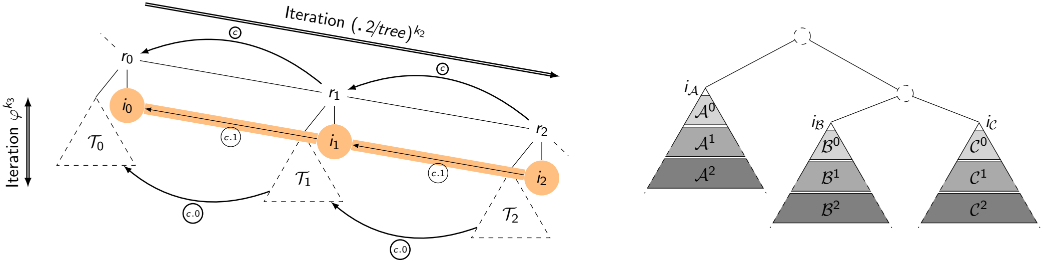


$$\llparenthesis\, (.\,2/tree)^{k_2+2} \to (.\,2/tree)^{k_2+1} \,\rrparenthesis \qquad (c)$$

$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,1/int \to (.\,2/tree)^{k_2+1}.\,1/int \,\rrparenthesis \qquad (c1)$$

$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,0/tree.\,\varphi^{k_3} \to (.\,2/tree)^{k_2+1}.\,0/tree.\,\varphi^{k_3} \,\rrparenthesis \qquad (c0)$$

Step 2.5: Layer-aware movements (2/2) on Pull Up

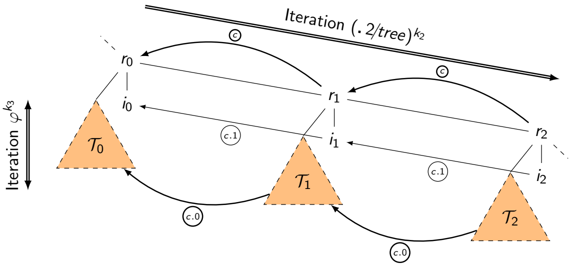


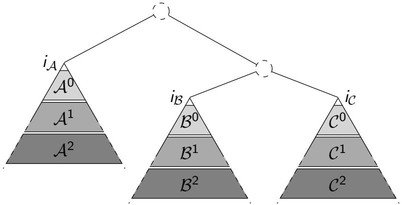
$$(\!| (.2/tree)^{k_2+2} \to (.2/tree)^{k_2+1} |\!) \qquad (c)$$

$$(\!| (.2/tree)^{k_2+2}.1/int \to (.2/tree)^{k_2+1}.1/int |\!) \qquad (c1)$$

$$(\!| (.2/tree)^{k_2+2}.0/tree.\varphi^{k_3} \to (.2/tree)^{k_2+1}.0/tree.\varphi^{k_3} |\!) \qquad (c0)$$

Step 2.5: Layer-aware movements (2/2) on Pull Up



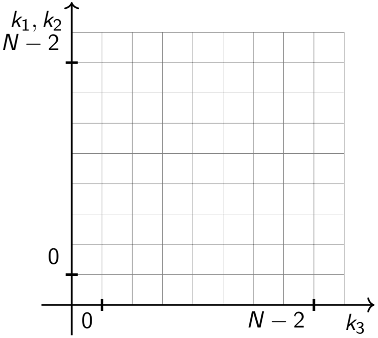


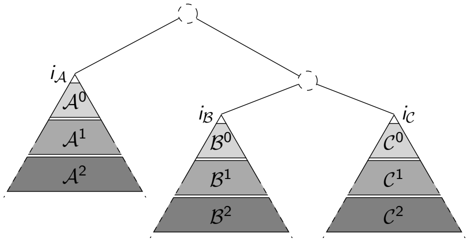
$$(\!|\ (.2/tree)^{k_2+2} \rightarrow (.2/tree)^{k_2+1}\ |\!) \qquad (c)$$

$$(\!|\ (.2/tree)^{k_2+2}.1/int \rightarrow (.2/tree)^{k_2+1}.1/int\ |\!) \qquad (c1)$$

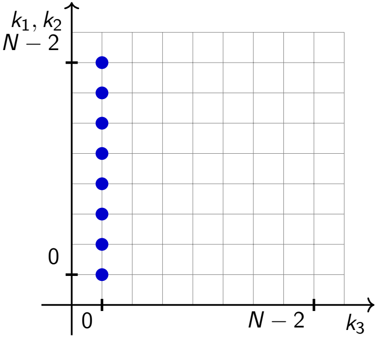
$$(\!|\ (.2/tree)^{k_2+2}.0/tree.\varphi^{k_3} \rightarrow (.2/tree)^{k_2+1}.0/tree.\varphi^{k_3}\ |\!) \qquad (c0)$$

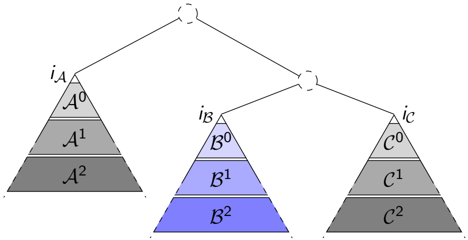
Step 3: Characterizing Memory Movements of Pull Up (Source)





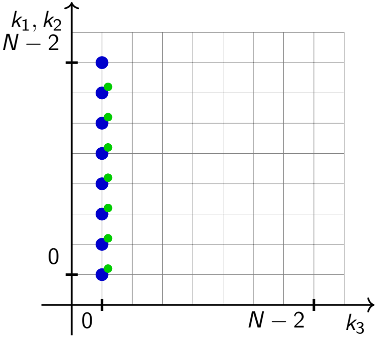
Step 3: Characterizing Memory Movements of Pull Up (Source)

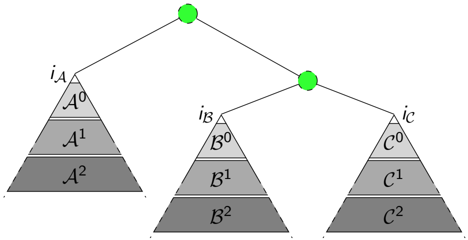




$$(\!|\, .2/tree.\, 0/tree.\, \varphi^{k_1} \rightarrow .\, 0/tree.\, \varphi^{k_1} \,|\!) \tag{b}$$

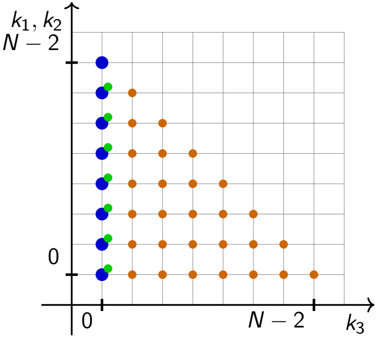
Step 3: Characterizing Memory Movements of Pull Up (Source)

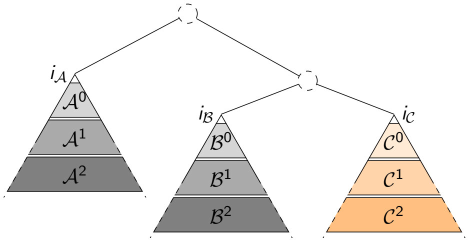




$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,1/int \rightarrow (.\,2/tree)^{k_2+1}.\,1/int \,\rrparenthesis \qquad (c1)$$

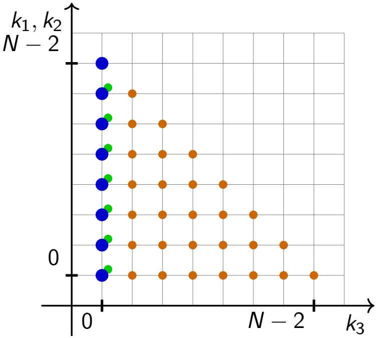
Step 3: Characterizing Memory Movements of Pull Up (Source)

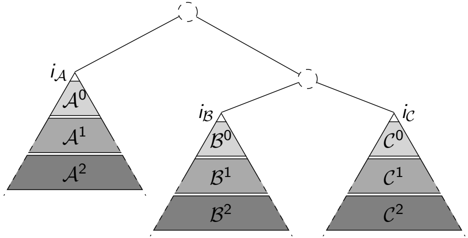




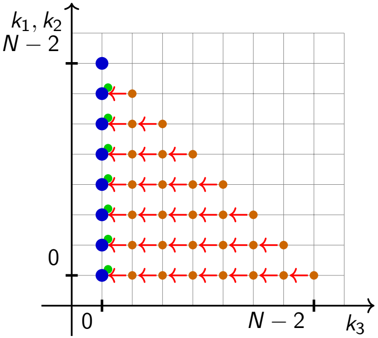
$$\llparenthesis\, (.\,2/tree)^{k_2+2}.\,0/tree.\,\varphi^{k_3} \rightarrow (.\,2/tree)^{k_2+1}.\,0/tree.\,\varphi^{k_3} \,\rrparenthesis \qquad (c0)$$

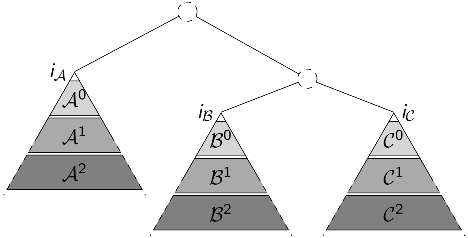
Characterizing Memory Movements of Pull Up (Dependencies)



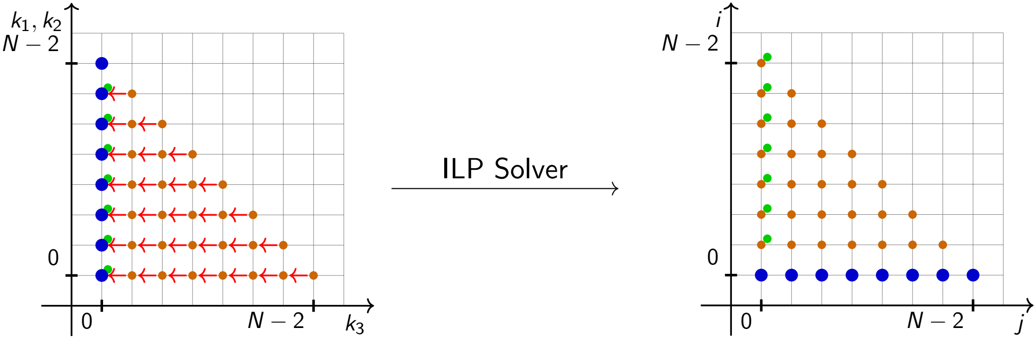


Characterizing Memory Movements of Pull Up (Dependencies)



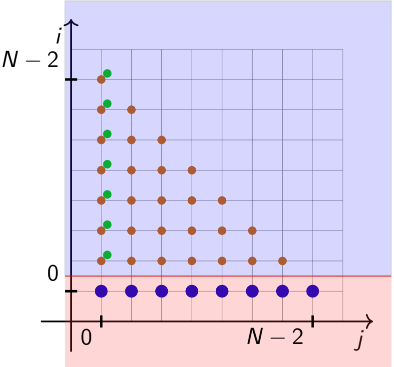


Step 4: Scheduling as an Integer Linear Program Problem



Step 5: From Schedule to Code

```
for (i = 0 ; i <= 0 ; i += 1) // P1
  for (j = 0 ; j <= N-2 ; j += 1)
    ⦇ . 2/tree . 0/tree.φ^j → . 0/tree.φ^j ⦈    // b
for (i = 1 ; i <= N-2 ; i+= 1)  // P2
  for (j = 0 ; j <= 0 ; j += 1)  // P'1,2
    ⦇ ( . 2/tree)^(i+1) . 1/int → ( . 2/tree)^i . 1/int ⦈   //c1
  for (j = 0 ; j <= N - i - 2 ; j += 1) // P'2,2
    ⦇ ( . 2/tree)^(i+1) . 0/tree.φ^j → ( . 2/tree)^i . 0/tree.φ^j ⦈   //c0
```



Application of Quilleré's Algorithm

Conclusion & Future Work

- A promising technique to compile pattern matching to in-place transformation:
  - Based on term-rewriting
  - Use linear equations on regexps!
  - Schedule the operations (pipelining, parallelization)
  - Generate code

Future Work

- Extend the input language
  - recursion
  - guards
- Improve the code generation
  - parallel code generation

Conclusion & Future Work

- A promising technique to compile pattern matching to in-place transformation:
  - Based on term-rewriting
  - Use linear equations on regexps!
  - Schedule the operations (pipelining, parallelization)
  - Generate code

Future Work

- Extend the input language
  - recursion
  - guards
- Improve the code generation
  - parallel code generation

Contents