Compiling Pattern Matching to In-Place Modifications

Paul Iannetta*, Laure Gonnord†, Gabriel Radanne‡

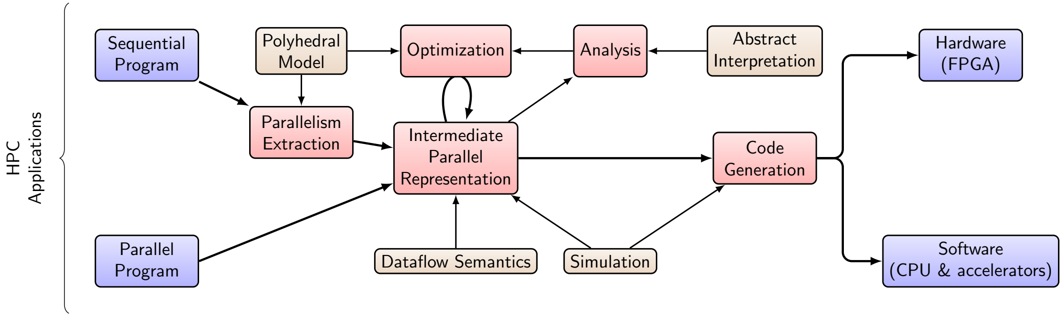
*ENS de Lyon, Inria & LIP
†UGA, Grenoble INP, LCIS & LIP
‡Inria & LIP

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

CASH: Topics

Optimized (software/hardware) compilation for HPC software with data-intensive computations.

Means: dataflow IR, static analyses, optimisations, simulation.



http://www.ens-lyon.fr/LIP/CASH/

Our Starting Point

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

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

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

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

- + Performance
- + Flexibility
- − Manual memory handling

- + Immutable
- − Immutable
- − Fix Memory Layout

Our Starting Point

```
st
};
vo
}
```

How to take the best of both worlds?

- Keep the DSL
- Keep a mutable semantics

+ Flexibility
– Manual memory handling

– Immutable
– Fix Memory Layout

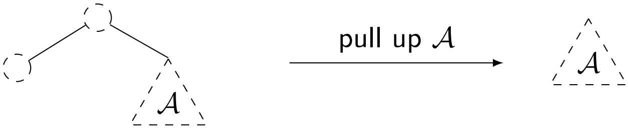
Contents

What is Rew?

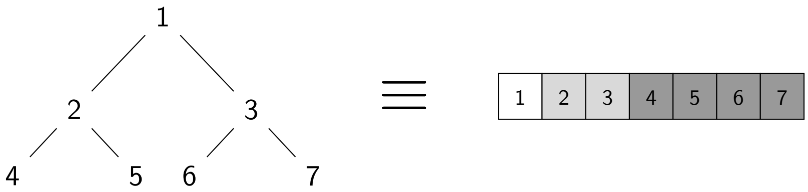
Rew is small DSL to:

- declare Algebraic Data Types with:
  - no sharing
  - no mutual recursion
  - all constructors slots must have a bounded size at compilation time
  - all constructors are of finite arity
- describe *s*tructural transformations on those through pattern matching.

Running Example: Pull Up

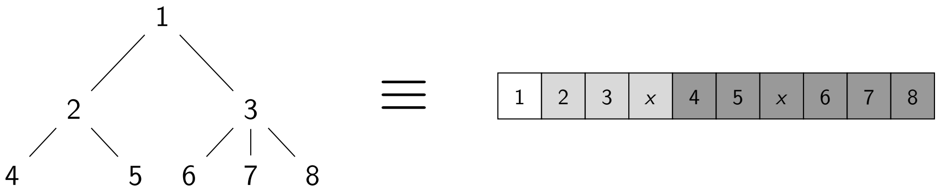


Memory Layout



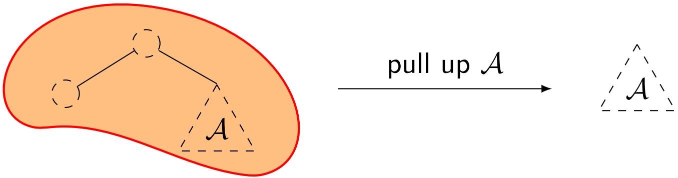

```
type tree = Empty
  | Node (tree,int,tree)
```

Memory Layout



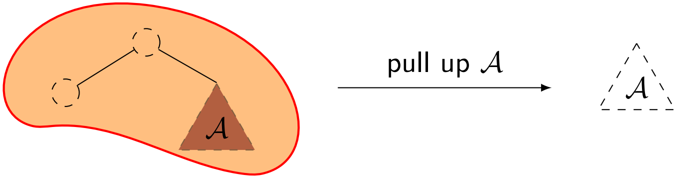

```
type tree = Empty
  | Node2 (int,tree,tree)
  | Node3 (int,tree,tree,tree)
```

A Language to Describe Structural Transformations



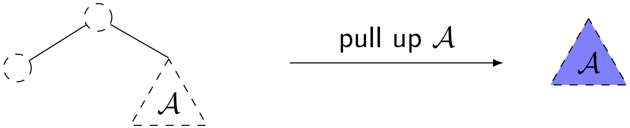

```
type tree = Empty | Node (tree,int,tree)

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

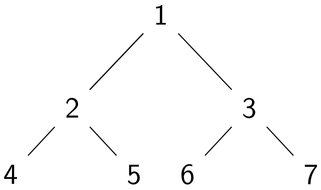
A Language to Describe Structural Transformations




```
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)
  | Node(a,i,Empty) -> Empty
  | Empty -> Empty
}
```

Our Notations for Locations

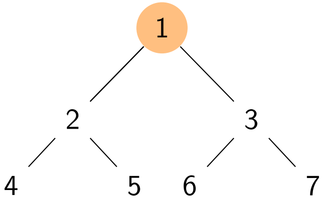
```
type tree = Empty | Node (tree,int,tree)
```



ϵ

Our Notations for Locations

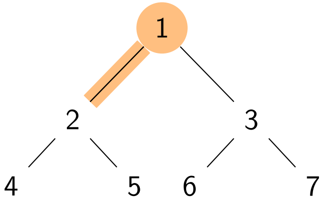
```
type tree = Empty | Node (tree,int,tree)
```



$.1/\mathit{int}$

Our Notations for Locations

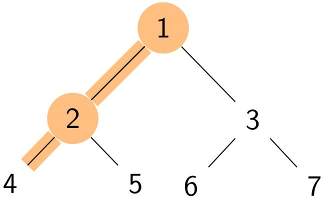
```
type tree = Empty | Node (tree,int,tree)
```



$.0/tree$

Our Notations for Locations

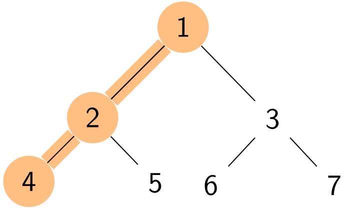
```
type tree = Empty | Node (tree,int,tree)
```



$$.0/tree.0/tree \equiv (.0/tree)^2$$

Our Notations for Locations

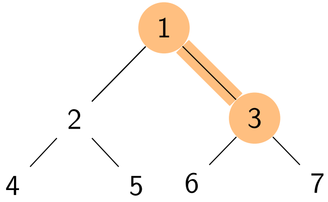
```
type tree = Empty | Node (tree,int,tree)
```



$$(.\,0/\mathit{tree})^2.\,1/\mathit{int}$$

Our Notations for Locations

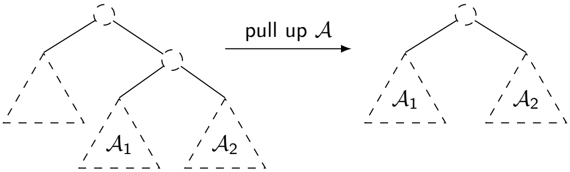
```
type tree = Empty | Node (tree,int,tree)
```



$.\,2/\mathit{tree}.\,1/\mathit{int}$

Step 1: Compute Subtree Movements

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



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

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

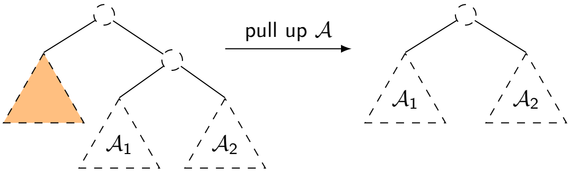
$$(\!|\, b : \mathit{tree} \mid \,.2/\mathrm{tree}.\,0/\mathrm{tree} \rightarrow \,.0/\mathrm{tree} \,|\!)$$

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

$$(\!|\, c : \mathit{tree} \mid \,.2/\mathrm{tree}.\,2/\mathrm{tree} \rightarrow \,.2/\mathrm{tree} \,|\!)$$

Step 1: Compute Subtree Movements

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



$$(\!|\ a : \mathit{tree} \mid .0/\mathsf{tree} \to \emptyset\ |\!)$$

$$(\!|\ i : \mathit{int} \mid .1/\mathsf{int} \to \emptyset\ |\!)$$

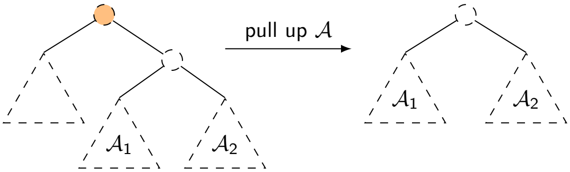
$$(\!|\ b : \mathit{tree} \mid .2/\mathsf{tree}.0/\mathsf{tree} \to .0/\mathsf{tree}\ |\!)$$

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

$$(\!|\ c : \mathit{tree} \mid .2/\mathsf{tree}.2/\mathsf{tree} \to .2/\mathsf{tree}\ |\!)$$

Step 1: Compute Subtree Movements

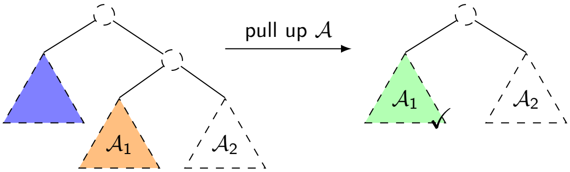
```
Node(a,i,Node(b,j,c)) -> Node(b,j,c)
```



$$(\!|\ a : tree \mid .0/tree \rightarrow \emptyset\ |\!)$$
$$(\!|\ i : int \mid .1/int \rightarrow \emptyset\ |\!)$$
$$(\!|\ b : tree \mid .2/tree.0/tree \rightarrow .0/tree\ |\!)$$
$$(\!|\ j : int \mid .2/tree.1/int \rightarrow .1/int\ |\!)$$
$$(\!|\ c : tree \mid .2/tree.2/tree \rightarrow .2/tree\ |\!)$$

Step 1: Compute Subtree Movements

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



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

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

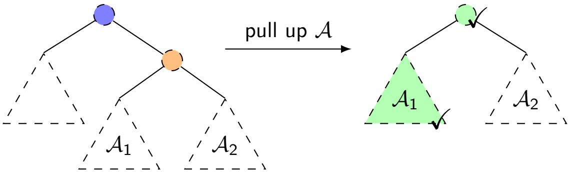
$$(\!|\, b : \mathit{tree} \mid .2/\mathrm{tree}.0/\mathrm{tree} \to .0/\mathrm{tree} \,|\!)$$

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

$$(\!|\, c : \mathit{tree} \mid .2/\mathrm{tree}.2/\mathrm{tree} \to .2/\mathrm{tree} \,|\!)$$

Step 1: Compute Subtree Movements

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



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

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

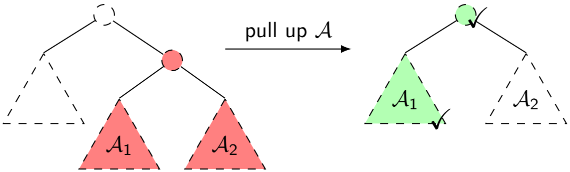
$$(\!|\, b : tree \mid .2/\text{tree}.0/\text{tree} \rightarrow .0/\text{tree} \,|\!)$$

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

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

Step 1: Compute Subtree Movements

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



$$(\!| \; a : tree \mid .0/\text{tree} \rightarrow \emptyset \; |\!)$$

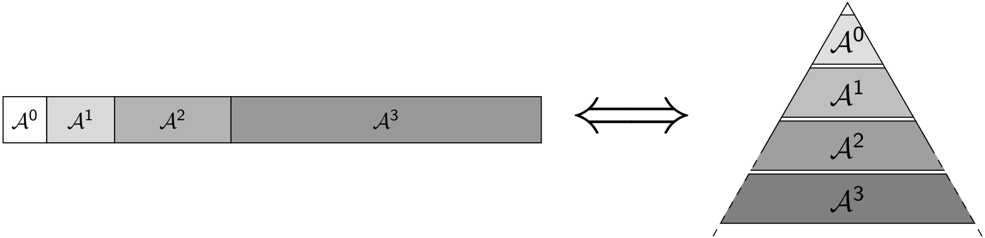
$$(\!| \; i : int \mid .1/\text{int} \rightarrow \emptyset \; |\!)$$

$$(\!| \; b : tree \mid .2/\text{tree}.0/\text{tree} \rightarrow .0/\text{tree} \; |\!)$$

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

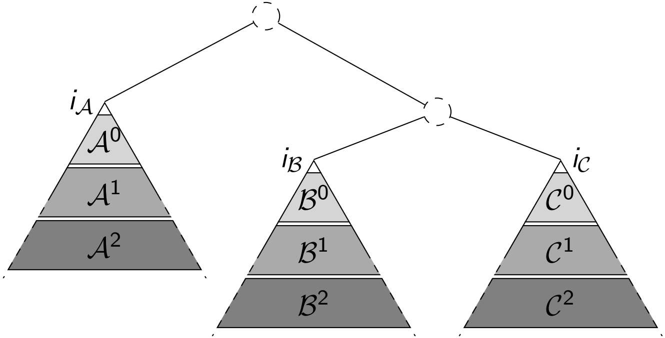
$$(\!| \; c : tree \mid .2/\text{tree}.2/\text{tree} \rightarrow .2/\text{tree} \; |\!)$$

Use Layers to Subdivide Memory Movements



Step 2: Layer-aware movements (1/2)

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



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

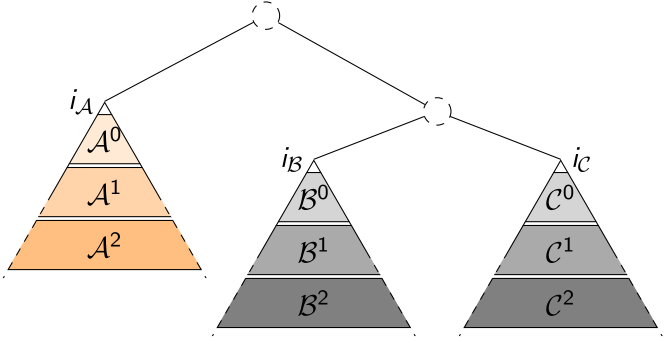
$$(\!|\ .\,1/\mathit{int} \to \emptyset\ |\!) \quad \text{(i)}$$

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

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

Step 2: Layer-aware movements (1/2)

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



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

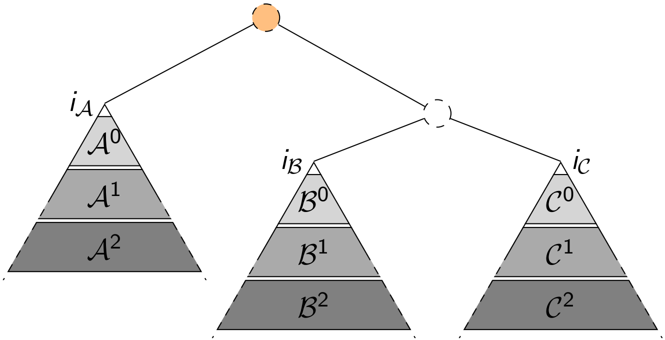
$$(\!|\, .\,1/int \rightarrow \emptyset \,|\!) \quad \text{(i)}$$

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

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

Step 2: Layer-aware movements (1/2)

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



$$(\!|\ .\,0/tree.\,\varphi^{k_0} \rightarrow \emptyset\ |\!) \tag{a}$$

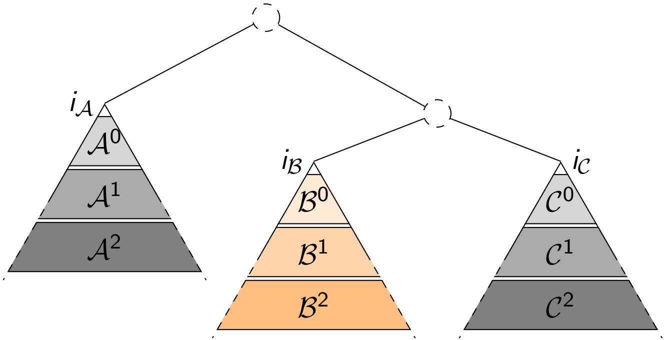
$$(\!|\ .\,1/int \rightarrow \emptyset\ |\!) \tag{i}$$

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

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

Step 2: Layer-aware movements (1/2)

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



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

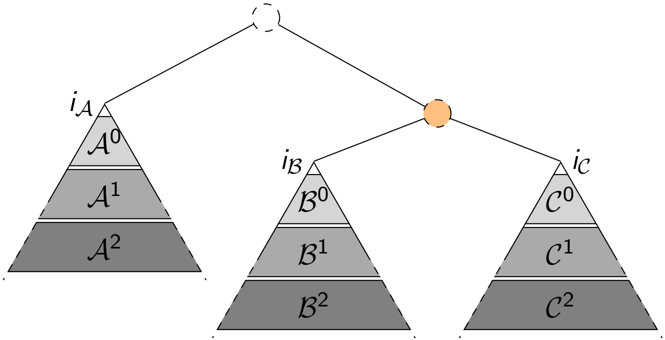
$$(\!|\, .\, 1/int \to \emptyset \,|\!) \quad \text{(i)}$$

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

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

Step 2: Layer-aware movements (1/2)

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



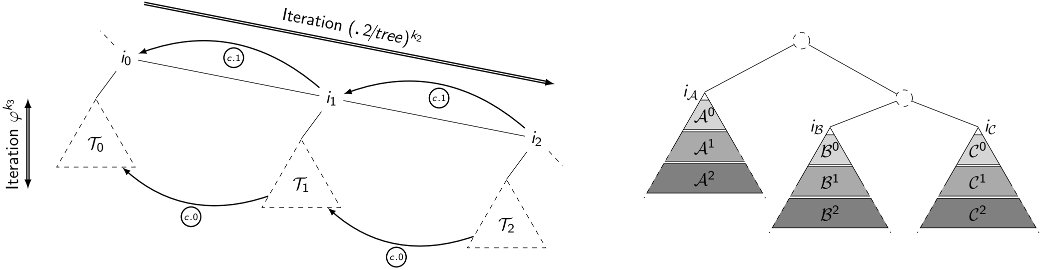
$$(\!|\ .\,0/tree.\,\varphi^{k_0} \to \emptyset\ |\!) \qquad (a)$$

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

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

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

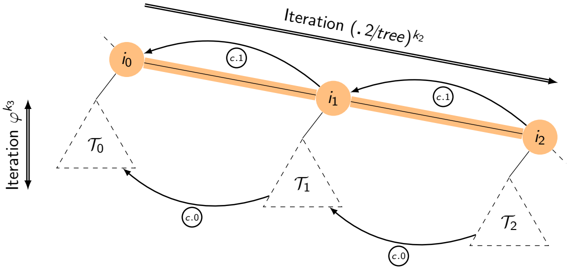
Step 2.5: Layer-aware movements (2/2)

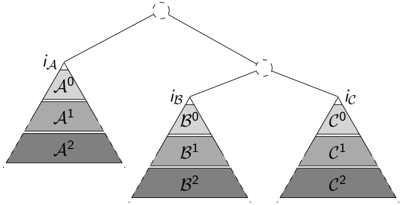


$$(\!| (.2/tree)^{k_2+2}.1/int \rightarrow (.2/tree)^{k_2+1}.1/int |\!) \quad (c1)$$

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

Step 2.5: Layer-aware movements (2/2)

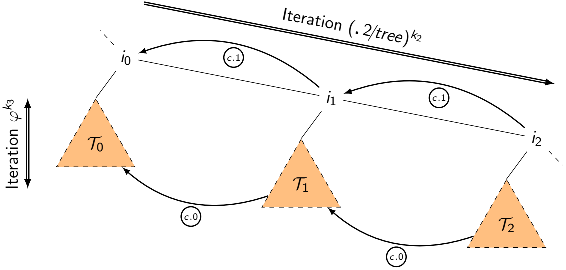


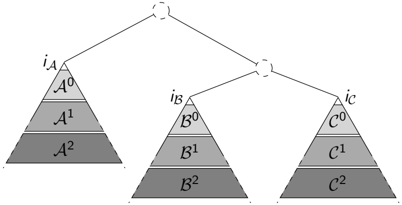


$$\llparenthesis\, (.2/tree)^{k_2+2}.1/int \rightarrow (.2/tree)^{k_2+1}.1/int \,\rrparenthesis \qquad (c1)$$

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

Step 2.5: Layer-aware movements (2/2)





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

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

Contents

Some definitions

Definition (Paths and memory movements)

$$
\begin{array}{lll}
k & \in \texttt{ItVars} & \text{(Iteration variables)} \\
\pi & ::= \ell \,.\, \pi \mid \ell^k \,.\, \pi \mid \varphi^k \mid \varepsilon & \text{(Path)} \\
m_\pi & ::= (\!|\, \pi \to \pi' \,|\!) & \text{(Move)}
\end{array}
$$

Definition (Admissible length of a path)

Given a path π, its *admissible length*, written $|\pi|$, is:

$$
\begin{array}{ll}
|\ell \,.\pi| = |\ell| + |\pi| & |\varphi^k| = k \\
|\ell^k \,.\pi| = |\ell| * k + |\pi| & |\varepsilon| = 0
\end{array}
$$

Domain of a move

Definition (Domain of a move)

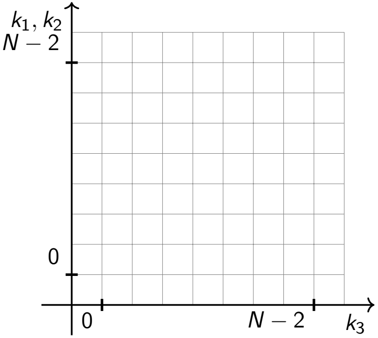
We consider a move $m = (\!| \pi \to \pi' |\!)$. The domain of m is written $\mathcal{D}_m$ and defined

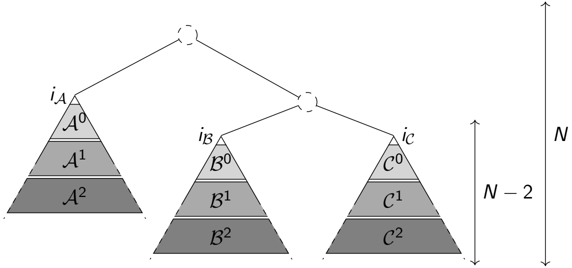
$$\mathcal{D}_m = \left\{ \vec{k} \;\middle|\; (0 \leq |\pi|(\vec{k}) \leq N) \wedge (0 \leq |\pi'|(\vec{k}) \leq N) \wedge (\vec{0} \leq \vec{k}) \right\}$$

We write D_m and $\vec{d_m}$ such that $\mathcal{D}_m = \left\{ \vec{k} \mid D_m \vec{k} + \vec{d_m} \geq \vec{0} \right\}$

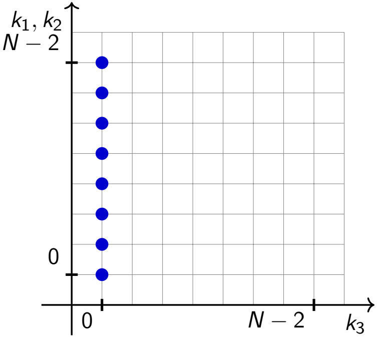
Since $|\pi|$ is a linear form on $\vec{k}$, the domain can also be defined as a polyhedron.

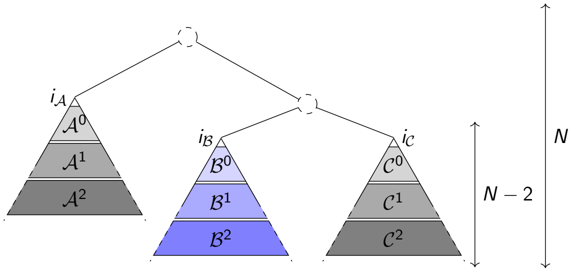
Step 3: Characterizing Memory Movements





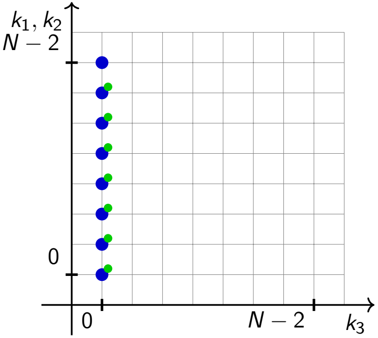
Step 3: Characterizing Memory Movements

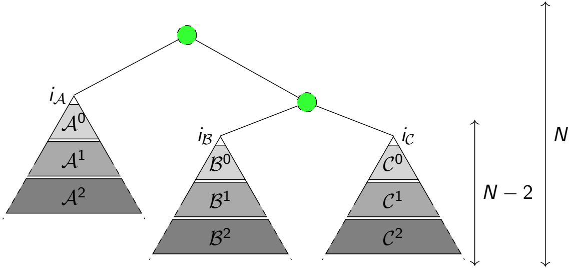




$$\llparenthesis\, .\, 2/tree.\, 0/tree.\, \varphi^{k_1} \rightarrow .\, 0/tree.\, \varphi^{k_1} \,\rrparenthesis \qquad \text{(b)}$$

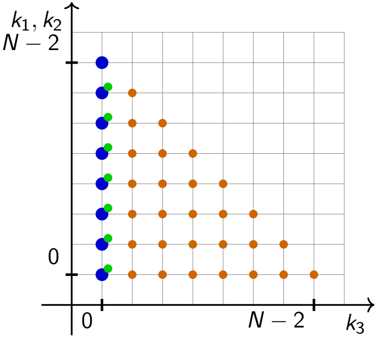
Step 3: Characterizing Memory Movements

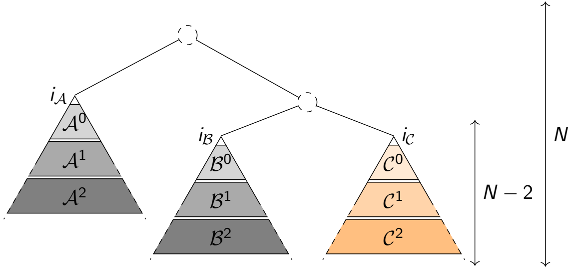




$$\llparenthesis\, (.\, 2/tree)^{k_2+2}.\, 1/int \rightarrow (.\, 2/tree)^{k_2+1}.\, 1/int\, \rrparenthesis \qquad (c1)$$

Step 3: Characterizing Memory Movements





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

Dependencies between moves

Definition ((R/W) Dependencies between moves)

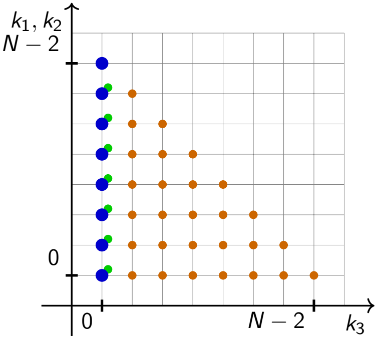
Let $m = (\!| \pi_p \rightarrow \pi_e |\!)$ and $m' = (\!| \pi'_p \rightarrow \pi'_e |\!)$ be two moves. Given $\mathcal{L}(\pi)$ the set of locations of π, we have:

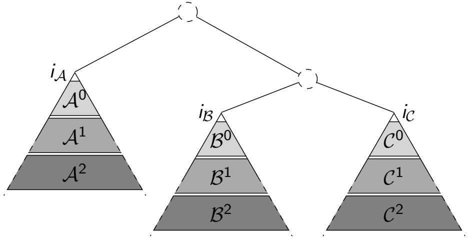
$$\mathcal{Q}_{(m,m')} = \left\{ \begin{pmatrix} \vec{k} \\ \vec{k'} \end{pmatrix} \middle| \exists \ell \in \mathcal{L}(\pi_p(\vec{k})) \cap \mathcal{L}(\pi'_e(\vec{k'})) \right\}$$

Lemma

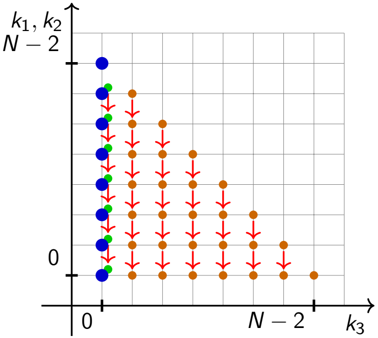
$\mathcal{Q}_{(m,m')}$ is a union of polyhedrons and computing its finite representation is decidable.

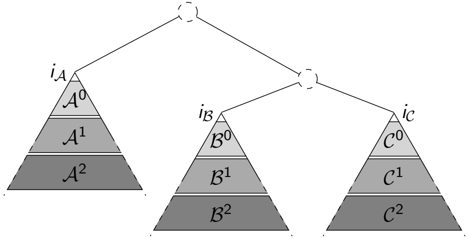
Characterizing Memory Movements (Dependencies)





Characterizing Memory Movements (Dependencies)





Schedule Definition

Definition (Schedule & Constraints)

A *schedule* for the graph $(\mathcal{M}, \mathcal{T})$ is a function $\rho : \mathcal{M} \times \mathbb{Z}^n \to \mathbb{N}^d$, from the graph vertices to $\mathbb{N}^d$, which is positive:

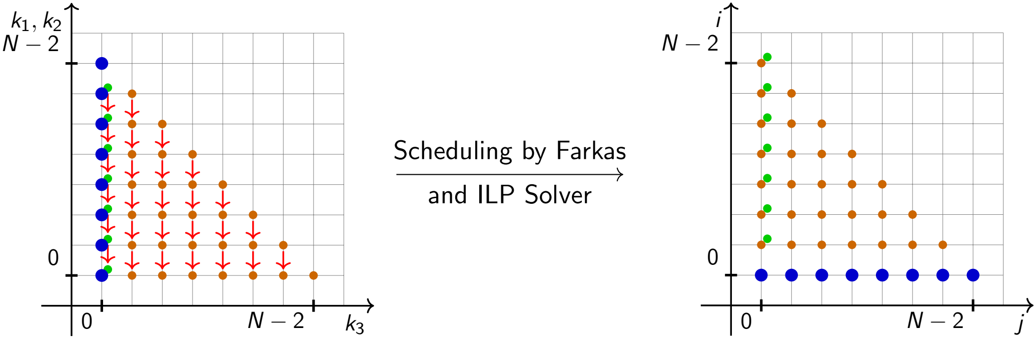
$$\vec{k} \in \mathcal{D}_m \Rightarrow \rho(m, \vec{k}) \geq \vec{0} \text{ (component-wise)} \qquad \text{(Positivity)}$$

and whose values *stricly* increase (according to $\preceq_d$, the standard lexicographic order on integer vectors) at each edge $t = (m, m') \in \mathcal{T}$:

$$\mathcal{Q}_{m,m'}(\vec{k}, \vec{k'}) \neq \emptyset \Rightarrow \rho(m, \vec{k}) \prec_d \rho(m', \vec{k'}) \qquad \text{(Increasing)}$$

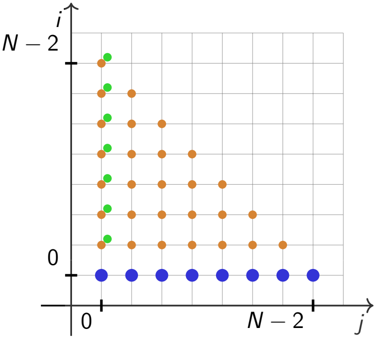
It is said *affine* if it is affine in the second parameter (the variables $\vec{k}$). If $d > 0$ the schedule is said to be multi-dimensional of dimension d.

Step 4: Scheduling as an Integer Linear Program Problem



Contents

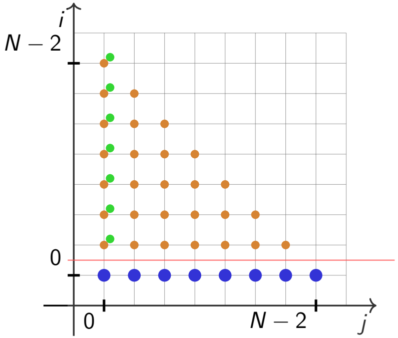
Step 5: From Schedule to Code



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

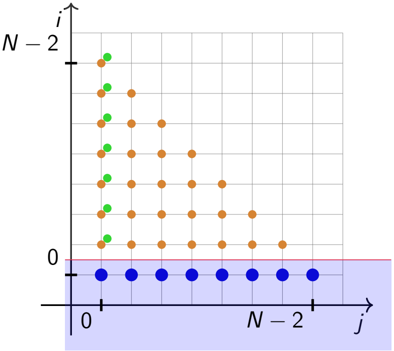
```
for (i = 0 ; i <= 0 ; i += 1)

for (i = 1 ; i <= N-2 ; i+= 1)
```



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

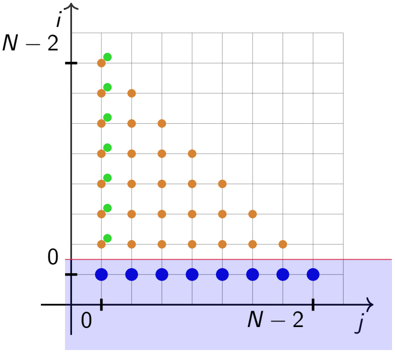
```
for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)

for (i = 1 ; i <= N-2 ; i+= 1)
```



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

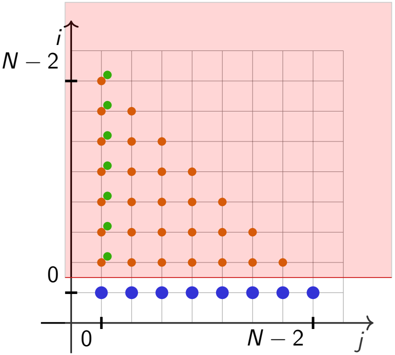
```
for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)
    (| . 2/tree . 0/tree.φ^j → . 0/tree.φ^j |)   // b
for (i = 1 ; i <= N-2 ; i+= 1)
```



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

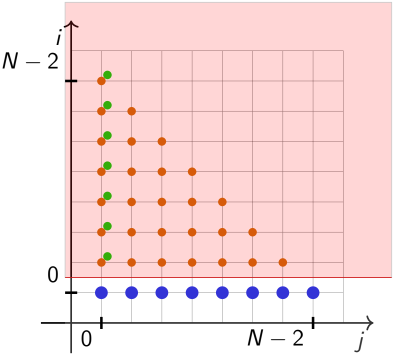
```
for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)
    ⦇ . 2/tree . 0/tree.φ^j → . 0/tree.φ^j ⦈   // b
for (i = 1 ; i <= N-2 ; i+= 1)
  for (j = 0 ; j <= 0 ; j += 1)

  for (j = 0 ; j <= N - i - 2 ; j += 1)
```



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

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



Application of Quilleré's Algorithm

Step 5: From Schedule to Code

`for (j = 0 ; j <= N-2 ; j += 1)`

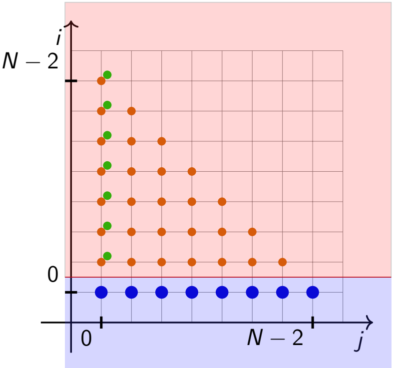
  $(\!|\ .\,2/tree\,.\,0/tree.\varphi^j \rightarrow .\,0/tree.\varphi^j\ |\!)$ `// b`

`for (i = 1 ; i <= N-2 ; i+= 1)`

  $(\!|\ (.\,2/tree)^{i+1}\,.\,1/int \rightarrow (.\,2/tree)^{i}\,.\,1/int\ |\!)$ `//c1`

  `for (j = 0 ; j <= N - i - 2 ; j += 1)`

    $(\!|\ (.\,2/tree)^{i+1}\,.\,0/tree.\varphi^j \rightarrow (.\,2/tree)^{i}\,.\,0/tree.\varphi^j\ |\!)$ `//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

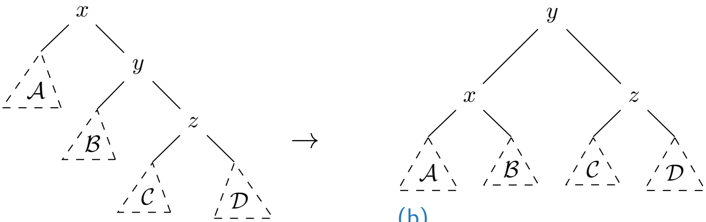
- Extend the input language
  - recursion
  - guards
- Improve the code generation
  - parallel code generation

Contents

AVL Trees

Definition

AVL Trees AVL Trees are self-rebalancing binary search trees relying on a rotation mechanism.



(a) A binary tree $\mathcal{T}$

(b) $\mathcal{T}' =$ L-rotation($\mathcal{T}$)

Figure: Left (L) rotation applied on to the unbalanced tree T

AVL Trees

Definition

AVL Trees AVL Trees are self-rebalancing binary search trees relying on a rotation mechanism.

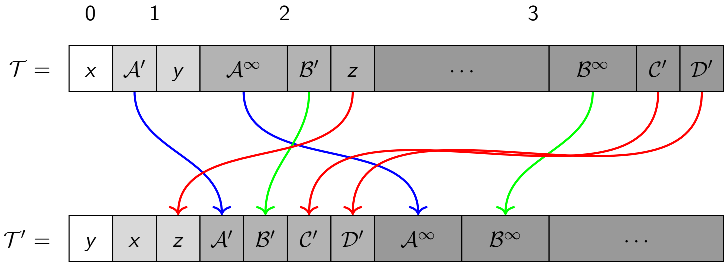


Figure: L-rotation on a tarbre

Low-level operations

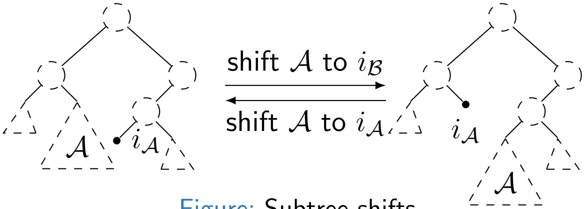


Figure: Subtree shifts

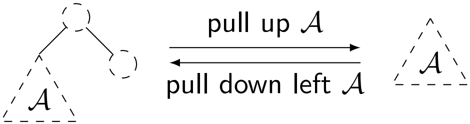
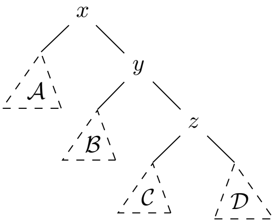


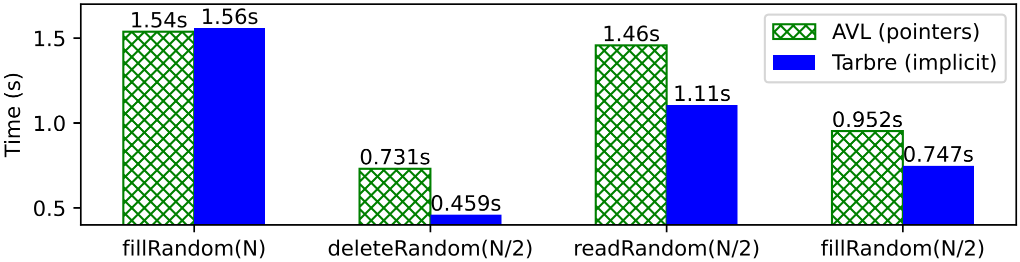
Figure: Subtree pull-ups and pull-downs

Rotations as low-level operations



Rotation		Right	Left
Steps	1.	pull-down-right($\mathcal{T}$, $i_{\mathcal{D}}$)	pull-down-left($\mathcal{T}$, $i_{\mathcal{A}}$)
	2.	shift($\mathcal{T}$, $i_{\mathcal{C}}$, $i_{\mathcal{C}} + 1$)	shift($\mathcal{T}$, $i_{\mathcal{B}}$, $i_{\mathcal{B}} - 1$)
	3.	pull-up($\mathcal{T}$, i_z)	pull-up($\mathcal{T}$, i_z)
	4.	move-values($\mathcal{T}$, i_x, i_y, i_z)	move-values($\mathcal{T}$, i_x, i_y, i_z)

Benchmark on a key-value store scenario ($N = 1M \approx 2^{20}$)

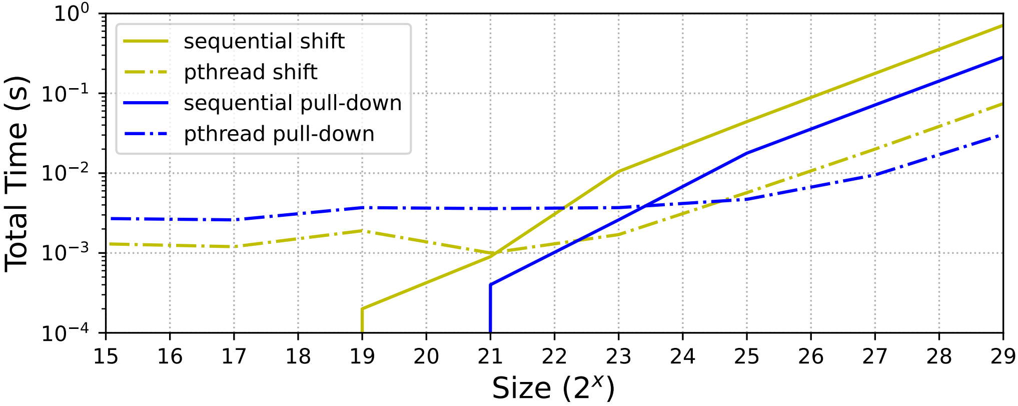


OCaml Type Checker

```
1223 <- addl(0,kind_478,layout_479) // Tree creation
1224 <- add(1223,arr_483) // Tree extension
find(1224,arr_483) // Lookup in tree 1224
find(1224,c_init_460)
...
free(1224) // Tree 1224 is freed
find(1223,create_430) // Lookup in tree 1223
```

	AVL (pointers)	tarbre (implicit)	std::set
Time (s)	5.64s	5.04s	5.78s
Memory	12Mo	13Mo	12Mo

Parallel Benchmark



Conclusion

- Summary:
  - A simple decomposition of rotations into simpler operations
  - Performance which are comparable on a sequential workload
  - Micro benchmarks show that there is room for improvement
- Current work:
  - Working on a complete parallel scenario

Contents