Compiling Pattern Matching to In-Place Modifications

Paul Iannetta*, Laure Gonnord†, Gabriel Radanne‡
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

How to take the best of both worlds?
Keep the DSL
Keep a mutable semantics
Our Starting Point

How to take the best of both worlds?

- Keep the DSL
- Keep a mutable semantics

+ Flexibility
- Manual memory handling

- Immutable
- Fix Memory Layout
Contents

1. Presentation of Rew
   • Structural Transformations
   • Coarse Memory Movements
   • Fine-Grain Memory Movements

2. Scheduling Memory Movements
   • Memory Movements Domains
   • Inter-Movement Dependencies

3. Code Generation
   • Schedule Computation

4. Some Manual Experiments
   • AVL Trees
   • Decomposition of Rotations
   • Benchmarks
What is \textsc{Rew}?

\textsc{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 structural transformations on those through pattern matching.
Running Example: Pull Up

\[ \text{pull up } A \rightarrow A \]
Memory Layout

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

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

1 2 3 4 5 6 7 8
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

\[
\text{type tree } = \text{ Empty } | \text{ Node (tree, int, tree)}
\]

\[
pull\_up (t : \text{ tree}) : \text{ tree } = \text{ rewrite } t \{
| \text{ Node(a,i,Node(b,j,c)) } -> \text{ Node(b,j,c)}
| \text{ Node(a,i,Empty) } -> \text{ Empty}
| \text{ Empty } -> \text{ Empty}
\}
\]
A Language to Describe Structural Transformations

\[
\text{type tree} = \text{Empty} \mid \text{Node} (\text{tree}, \text{int}, \text{tree})
\]

\[
pull\_up (t : \text{tree}) : \text{tree} = \text{rewrite} t \{
| \text{Node}(a,i,\text{Node}(b,j,c)) \rightarrow \text{Node}(b,j,c)
| \text{Node}(a,i,\text{Empty}) \rightarrow \text{Empty}
| \text{Empty} \rightarrow \text{Empty}
\}
\]
Our Notations for Locations

type tree = Empty | Node (tree, int, tree)
Our Notations for Locations

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

```
.1/int
```
Our Notations for Locations

\[
\text{type } \text{tree} = \text{Empty} \mid \text{Node (tree, int, tree)}
\]

```
0
/  \
1- \\
2 3
4 5 6 7
```

.0/tree
Our Notations for Locations

\[ \text{type } \text{tree} = \text{Empty} \mid \text{Node} \left( \text{tree}, \text{int}, \text{tree} \right) \]

\[ .0/\text{tree} .0/\text{tree} \equiv ( .0/\text{tree} )^2 \]
Our Notations for Locations

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

```
(. 0/tree)^2 . 1/int
```
Our Notations for Locations

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

2/\textit{tree}. 1/\textit{int}
Step 1: Compute Subtree Movements

Node(a, i, Node(b, j, c)) → Node(b, j, c)

\[
\begin{align*}
\text{Pull up } A &:
\begin{aligned}
\text{A}_1 &: \text{tree} | .0/\text{tree} \rightarrow \emptyset \\
\text{A}_2 &: \text{int} | .1/\text{int} \rightarrow \emptyset \\
\text{B} &: \text{tree} | .2/\text{tree}.0/\text{tree} \rightarrow .0/\text{tree} \\
\text{C} &: \text{int} | .2/\text{tree}.1/\text{int} \rightarrow .1/\text{int} \\
\text{D} &: \text{tree} | .2/\text{tree}.2/\text{tree} \rightarrow .2/\text{tree}
\end{aligned}
\end{align*}
\]
Step 1: Compute Subtree Movements

\[
\text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c)
\]

\[
\begin{align*}
| a : \text{tree} | & .0/\text{tree} \rightarrow \emptyset \\
| i : \text{int} | & .1/\text{int} \rightarrow \emptyset \\
| b : \text{tree} | & .2/\text{tree}.0/\text{tree} \rightarrow .0/\text{tree} \\
| j : \text{int} | & .2/\text{tree}.1/\text{int} \rightarrow .1/\text{int} \\
| c : \text{tree} | & .2/\text{tree}.2/\text{tree} \rightarrow .2/\text{tree}
\end{align*}
\]
Step 1: Compute Subtree Movements

\[ \text{Node}(a,i,\text{Node}(b,j,c)) \rightarrow \text{Node}(b,j,c) \]

pull up \( A \)

\[
\begin{align*}
|a: \text{tree} & \rightarrow \emptyset | \\
|i: \text{int} & \rightarrow \emptyset | \\
|b: \text{tree} & \rightarrow .0/\text{tree} | \\
|j: \text{int} & \rightarrow .1/\text{int} | \\
|c: \text{tree} & \rightarrow .2/\text{tree} |
\end{align*}
\]
Step 1: Compute Subtree Movements

\[ \text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c) \]
Step 1: Compute Subtree Movements

\[ \text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c) \]
Step 1: Compute Subtree Movements

Node(a, i, Node(b, j, c)) → Node(b, j, c)

(a: tree | .0/tree → ∅)
(i: int | .1/int → ∅)
(b: tree | .2/tree.0/tree → .0/tree)
(j: int | .2/tree.1/int → .1/int)
(c: tree | .2/tree.2/tree → .2/tree)
Fine-Grain Memory Movements

Use Layers to Subdivide Memory Movements

$A^0 \leftrightarrow A^1 \leftrightarrow A^2 \leftrightarrow A^3$
Step 2: Layer-aware movements (1/2)

Node(a, i, Node(b, j, c)) → Node(b, j, c)
Step 2: Layer-aware movements (1/2)

Node($a$, $i$, Node($b$, $j$, $c$)) $\rightarrow$ Node($b$, $j$, $c$)

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

(i) $\{.1/int \rightarrow \emptyset\}$

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

(j) $\{.2/tree.1/int \rightarrow .1/int\}$
Step 2: Layer-aware movements (1/2)

\[ \text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c) \]
Step 2: Layer-aware movements (1/2)

\[ \text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c) \]

- (a) \( \langle .0/\text{tree.}\varphi^{k_0} \rightarrow \emptyset \rangle \)
- (i) \( \langle .1/\text{int} \rightarrow \emptyset \rangle \)
- (b) \( \langle .2/\text{tree.}0/\text{tree.}\varphi^{k_1} \rightarrow .0/\text{tree.}\varphi^{k_1} \rangle \)
- (j) \( \langle .2/\text{tree.}1/\text{int} \rightarrow .1/\text{int} \rangle \)
Step 2: Layer-aware movements (1/2)

\[ \text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c) \]

(a) \( \langle .0/\text{tree.} \varphi^{k_0} \rightarrow \emptyset \rangle \)

(i) \( \langle .1/int \rightarrow \emptyset \rangle \)

(b) \( \langle .2/\text{tree.} 0/\text{tree.} \varphi^{k_1} \rightarrow .0/\text{tree.} \varphi^{k_1} \rangle \)

(j) \( \langle .2/\text{tree.} 1/int \rightarrow .1/int \rangle \)
Step 2.5: Layer-aware movements (2/2)

\[ \| (\cdot 2/\text{tree})^{k_2+2} \cdot 1/\text{int} \rightarrow (\cdot 2/\text{tree})^{k_2+1} \cdot 1/\text{int} \| \] \hspace{1cm} (c1)

\[ \| (\cdot 2/\text{tree})^{k_2+2} \cdot 0/\text{tree}, \phi^{k_3} \rightarrow (\cdot 2/\text{tree})^{k_2+1} \cdot 0/\text{tree}, \phi^{k_3} \| \] \hspace{1cm} (c0)
Step 2.5: Layer-aware movements (2/2)

\[ ((.2/\text{tree})^{k_2+2}.1/\text{int} \rightarrow (.2/\text{tree})^{k_2+1}.1/\text{int}) \]

\[ ((.2/\text{tree})^{k_2+2}.0/\text{tree}.\varphi^{k_3} \rightarrow (.2/\text{tree})^{k_2+1}.0/\text{tree}.\varphi^{k_3}) \]
Step 2.5: Layer-aware movements (2/2)

\[ \left( \frac{.2}{\text{tree}} \right)^{k_2 + 2} \cdot \frac{1}{\text{int}} \rightarrow \left( \frac{.2}{\text{tree}} \right)^{k_2 + 1} \cdot \frac{1}{\text{int}} \right) \quad (c1) \]

\[ \left( \frac{.2}{\text{tree}} \right)^{k_2 + 2} \cdot \frac{0}{\text{tree}} \cdot \varphi^{k_3} \rightarrow \left( \frac{.2}{\text{tree}} \right)^{k_2 + 1} \cdot \frac{0}{\text{tree}} \cdot \varphi^{k_3} \right) \quad (c0) \]
Contents

1 Presentation of REW
   - Structural Transformations
   - Coarse Memory Movements
   - Fine-Grain Memory Movements

2 Scheduling Memory Movements
   - Memory Movements Domains
   - Inter-Movement Dependencies

3 Schedule Computation

4 Some Manual Experiments
   - AVL Trees
   - Decomposition of Rotations
   - Benchmarks
Some definitions

Definition (Paths and memory movements)

\[ k \in \text{ItVars} \]  
\[ \pi ::= \ell . \pi | \ell^k . \pi | \varphi^k | \varepsilon \]  
\[ m_\pi ::= (\pi \rightarrow \pi') \]

Definition (Admissible length of a path)

Given a path \( \pi \), its *admissible length*, written \( |\pi| \), is:

\[ |\ell . \pi| = |\ell| + |\pi| \]
\[ |\ell^k . \pi| = |\ell| \ast k + |\pi| \]
\[ |\varphi^k| = k \]
\[ |\varepsilon| = 0 \]
Domain of a move

**Definition (Domain of a move)**

We consider a move \( m = (\pi \rightarrow \pi') \). The domain of \( m \) is written \( D_m \) and defined

\[
D_m = \left\{ k \mid (0 \leq |\pi| (k) \leq N) \land (0 \leq |\pi'| (k) \leq N) \land (\vec{0} \leq \vec{k}) \right\}
\]

We write \( D_m \) and \( \vec{d}_m \) such that

\[
D_m = \left\{ \vec{k} \mid D_m\vec{k} + \vec{d}_m \geq \vec{0} \right\}
\]

Since \(|\pi|\) is a linear form on \( \vec{k} \), the domain can also be defined as a polyhedron.
Step 3: Characterizing Memory Movements

\[ k_1, k_2, N - 2 \]

\[ A^0, A^1, A^2, B^0, B^1, B^2, C^0, C^1, C^2 \]
Step 3: Characterizing Memory Movements

\[ (\ell. \text{2/tree.0/tree.} \varphi^{k_1} \rightarrow \text{.0/tree.} \varphi^{k_1}) \]
Step 3: Characterizing Memory Movements

\[
\langle (\cdot 2/\text{tree})^{k_2+2} \cdot 1/\text{int} \rightarrow (\cdot 2/\text{tree})^{k_2+1} \cdot 1/\text{int} \rangle
\]

(c1)
Step 3: Characterizing Memory Movements

\[ \langle \frac{2}{\text{tree}} \rangle^{k_2+2} \cdot 0/\text{tree} \cdot \varphi^{k_3} \rightarrow \langle \frac{2}{\text{tree}} \rangle^{k_2+1} \cdot 0/\text{tree} \cdot \varphi^{k_3} \rangle \] (c0)
Inter-Movement Dependencies

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 $\pi$, we have:

$$Q_{(m,m')} = \left\{ \left( \begin{array}{c} \vec{k} \\ \vec{k}' \end{array} \right) \left| \exists \ell \in \mathcal{L}(\pi_p(\vec{k})) \cap \mathcal{L}(\pi'_e(\vec{k}')) \right. \right\}$$

Lemma

$Q_{(m,m')}$ is a union of polyhedrons and computing its finite representation is decidable.
Characterizing Memory Movements (Dependencies)

$k_1, k_2$

$N - 2$

$0$

$k_3$

$A^0$

$A^1$

$A^2$

$B^0$

$B^1$

$B^2$

$C^0$

$C^1$

$C^2$
Characterizing Memory Movements (Dependencies)
Schedule Definition

**Definition (Schedule & Constraints)**

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

$$\tilde{k} \in D_m \Rightarrow \rho(m, \tilde{k}) \geq \tilde{0} \text{ (component-wise)}$$

(Positivity)

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

$$Q_{m, m'}(\tilde{k}, \tilde{k}') \neq \emptyset \Rightarrow \rho(m, \tilde{k}) <_d \rho(m', \tilde{k}')$$

(Increasing)

It is said *affine* if it is affine in the second parameter (the variables $\tilde{k}$). If $d > 0$ the schedule is said to be multi-dimensional of dimension $d$. 
Step 4: Scheduling as an Integer Linear Program Problem

Scheduling by Farkas and ILP Solver
Contents

1 Presentation of Rew
   - Structural Transformations
   - Coarse Memory Movements
   - Fine-Grain Memory Movements

2 Scheduling Memory Movements
   - Memory Movements Domains
   - Inter-Movement Dependencies

3 Code Generation
   - Application of the Algorithm to the Running Example

4 Some Manual Experiments
   - AVL Trees
   - Decomposition of Rotations
   - Benchmarks
Step 5: From Schedule to Code

Application of Quilleré’s Algorithm
Step 5: From Schedule to Code

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

Application of Quilleré’s Algorithm
Application of the Algorithm to the Running Example

Step 5: From Schedule to Code

```c
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)
    \( L_{i}\) \( 2/\text{tree.}_i \) \( \rightarrow \) \( 0/\text{tree.}_i \) \( \phi_j \) \( \rightarrow \) \( 0/\text{tree.}_i \) \( M_{b} \)

for (i = 1 ; i <= N-2 ; i+= 1)
Step 5: From Schedule to Code

\begin{align*}
\text{for (i = 0 ; i <= 0 ; i += 1)} \\
\quad \text{for (j = 0 ; j <= N-2 ; j += 1)} \\
\quad \quad (L_{.2/\text{tree}.} 0/\text{tree}.\phi_j \rightarrow 0/\text{tree}.\phi_j) \quad // \text{ b} \\
\text{for (i = 1 ; i <= N-2 ; i+= 1)} \\
\quad \text{for (j = 0 ; j <= 0 ; j += 1)} \\
\quad \quad \text{for (j = 0 ; j <= N - i - 2 ; j += 1)}
\end{align*}
Step 5: From Schedule to Code

for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)
    L\textunderscore 2/\text{tree}\cdot 0/\text{tree}\cdot \varphi^j \rightarrow 0/\text{tree}\cdot \varphi^j \quad // \quad b
for (i = 1 ; i <= N-2 ; i+= 1)
  for (j = 0 ; j <= 0 ; j += 1)
    L\textunderscore (2/\text{tree})_{i^+1}\cdot 1/\text{int} \rightarrow (2/\text{tree})_i\cdot 1/\text{int} \quad //c1
for (j = 0 ; j <= N - i - 2 ; j += 1)
  L\textunderscore (2/\text{tree})_{i^+1}\cdot 0/\text{tree}\cdot \varphi^j \rightarrow (2/\text{tree})_i\cdot 0/\text{tree}\cdot \varphi^j \quad //c0

Application of Quilleré’s Algorithm
Step 5: From Schedule to Code

```c
for (j = 0 ; j <= N-2 ; j += 1)
    (.2/\text{tree}.0/\text{tree}.\varphi^j \rightarrow .0/\text{tree}.\varphi^j) // b

for (i = 1 ; i <= N-2 ; i+= 1)
    ((.2/\text{tree})^{i+1}.1/int \rightarrow (.2/\text{tree})^i.1/int) // c1

for (j = 0 ; j <= N - i - 2 ; j += 1)
    ((.2/\text{tree})^{i+1}.0/\text{tree}.\varphi^j \rightarrow (.2/\text{tree})^i.0/\text{tree}.\varphi^j) // c0
```

Application of Quilleré’s Algorithm
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

1. **Presentation of REW**
   - Structural Transformations
   - Coarse Memory Movements
   - Fine-Grain Memory Movements

2. **Scheduling Memory Movements**
   - Memory Movements Domains
   - Inter-Movement Dependencies

3. **Code Generation**
   - Schedule Computation

4. **Some Manual Experiments**
   - AVL Trees
   - Decomposition of Rotations
   - Benchmarks
Definition

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

(a) A binary tree $T$

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

Figure: Left (L) rotation applied on to the unbalanced tree $T$
**AVL Trees**

**Definition**

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

\[
\mathcal{T} = \begin{array}{ccccccc}
0 & 1 & 2 & 3 \\
x & A' & y & A^\infty & B' & z & \cdots & B^\infty & C' & D'
\end{array}
\]

\[
\mathcal{T}' = \begin{array}{ccccccc}
0 & 1 & 2 & 3 \\
y & x & z & A' & B' & C' & D' & A^\infty & B^\infty & \cdots
\end{array}
\]

*Figure: L-rotation on a tarbre*
Low-level operations

Figure: Subtree shifts

Figure: Subtree pull-ups and pull-downs
Decomposition of Rotations

Rotations as low-level operations

<table>
<thead>
<tr>
<th>Rotation</th>
<th>Right</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. pull-down-right($T, i_D$)</td>
<td>pull-down-left($T, i_A$)</td>
<td></td>
</tr>
<tr>
<td>2. shift($T, i_C, i_C + 1$)</td>
<td>shift($T, i_B, i_B - 1$)</td>
<td></td>
</tr>
<tr>
<td>3. pull-up($T, i_z$)</td>
<td>pull-up($T, i_z$)</td>
<td></td>
</tr>
<tr>
<td>4. move-values($T, i_x, i_y, i_z$)</td>
<td>move-values($T, i_x, i_y, i_z$)</td>
<td></td>
</tr>
</tbody>
</table>
Benchmark on a key-value store scenario ($N = 1M \approx 2^{20}$)
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

<table>
<thead>
<tr>
<th>AVL (pointers)</th>
<th>tarbre (implicit)</th>
<th>std::set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>5.64s</td>
<td>5.04s</td>
</tr>
<tr>
<td>Memory</td>
<td>12Mo</td>
<td>13Mo</td>
</tr>
</tbody>
</table>
Parallel Benchmark

- **Sequential Shift**
- **pthread Shift**
- **Sequential Pull-down**
- **pthread Pull-down**

<table>
<thead>
<tr>
<th>Size (2x)</th>
<th>Total Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10^-4</td>
</tr>
<tr>
<td>4</td>
<td>10^-3</td>
</tr>
<tr>
<td>10</td>
<td>10^-2</td>
</tr>
<tr>
<td>3</td>
<td>10^-1</td>
</tr>
<tr>
<td>10</td>
<td>10^0</td>
</tr>
</tbody>
</table>

**Graphs**
- Logarithmic scale for Total Time (s)
- Size (2^x) on the x-axis
- Total Time (s) on the y-axis
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
1 Presentation of **REW**
   - Structural Transformations
   - Coarse Memory Movements
   - Fine-Grain Memory Movements

2 Scheduling Memory Movements
   - Memory Movements Domains
   - Inter-Movement Dependencies
   - Schedule Computation

3 Code Generation
   - Application of the Algorithm to the Running Example

4 Some Manual Experiments
   - AVL Trees
   - Decomposition of Rotations
   - Benchmarks