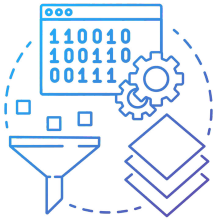
Compiling Trees
Combining Data Layouts and the Polyhedral Model

Paul Iannetta (ENS de Lyon, Inria & LIP)
paul.iannetta@ens-lyon.fr

Jury:

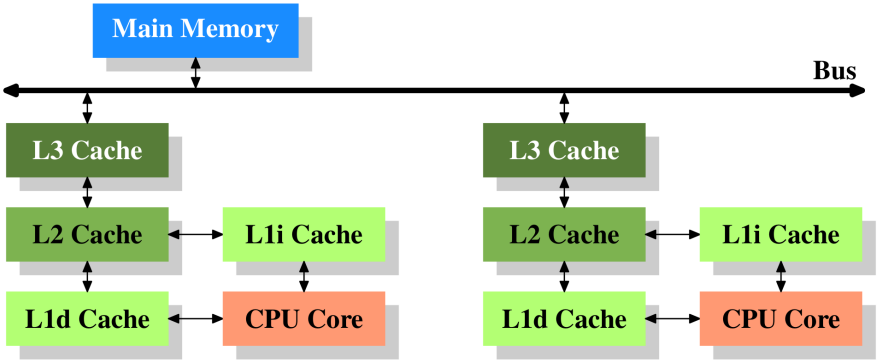
Charles	Henri-Pierre,	Rapporteur.	Gonnord	Laure,	Directrice de thèse.
Clauss	Philippe,	Rapporteur.	Radanne	Gabriel,	Co-encadrant.
Collange	Caroline,	Examinatrice.	Morel	Lionel,	Co-encadrant.
Keller	Gabriele,	Examinatrice.	Meister	Benoît,	Guest

What is Compilation?



- Programmers want to write easy to maintain software
- Offload complexity to the compiler (Hardware diversity)
- We don't want that people write hand tuned code.
- ... the compiler has to be smart enough to use the hardware cleverly enough
- let's talk a bit about the hardware...

The Basic Blocks Around a CPU



- Memory contains everything (slow – 50 $\sim$ 100ns)
- Cache contains only the bare minimum (fast – 0.5 $\sim$ 7ns)

Programs Which Play Well With The Hardware

- Programs whose behavior can be accurately predicted (control flow)
  - The CPU always knows what to do

Programs Which Play Well With The Hardware

- Programs whose behavior can be accurately predicted (control flow)
  - The CPU always knows what to do
- Programs with predictable read/write patterns (io patterns)
  - Prefetch and fill the cache from memory
  - Avoid waiting for data

Programs Which Play Well With The Hardware

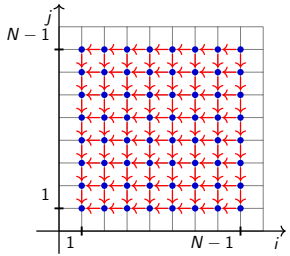
- Programs whose behavior can be accurately predicted (control flow)
  - The CPU always knows what to do
- Programs with predictable read/write patterns (io patterns)
  - Prefetch and fill the cache from memory
  - Avoid waiting for data
- Programs with independant parts (parallelism opportunities)
  - Share the work between CPUs
  - Better use of system ressources

A Very Nice Loop

```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```

A Very Nice Loop

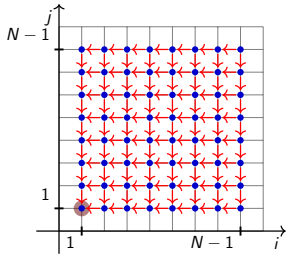
```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```



- Statement
- ⟵ Dependence

A Very Nice Loop

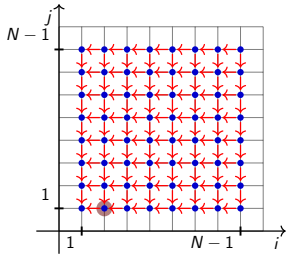
```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```



- Statement
- ⟵ Dependence

A Very Nice Loop

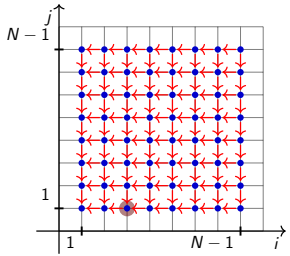
```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```



- Statement
- ⟵ Dependence

A Very Nice Loop

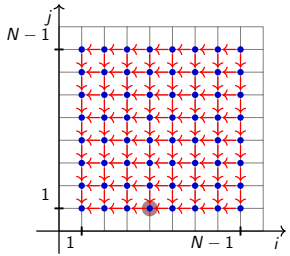
```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```



- Statement
- ⟵ Dependence

A Very Nice Loop

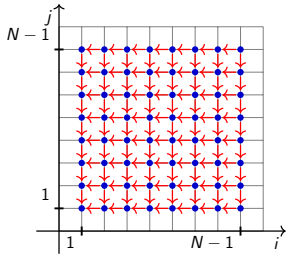
```
          for (int i = 1 ; i < N ; ++i)
            for (int j = 1 ; j < N ; ++j)
/* s: */        A[i][j] = A[i][j-1] + A[i-1][j];
```



- Statement
- ⟵ Dependence

A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```



Change the loop traversal

Improve inter-instruction parallelism

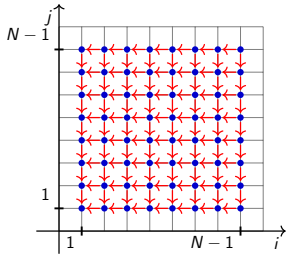
- Statement

⟵ Dependence

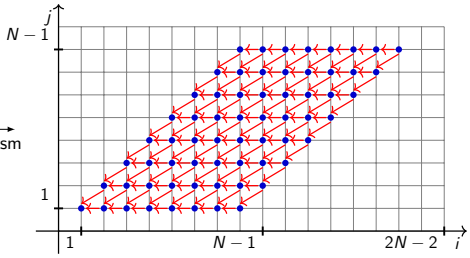
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



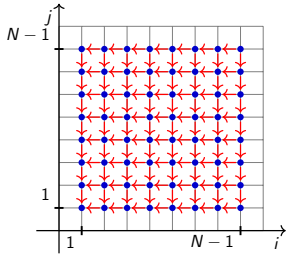
Change the loop traversal
Improve inter-instruction parallelism



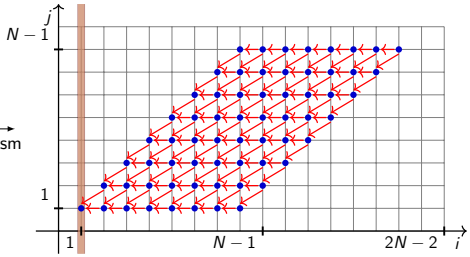
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



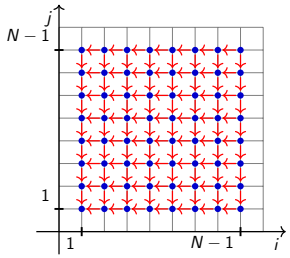
Change the loop traversal
Improve inter-instruction parallelism



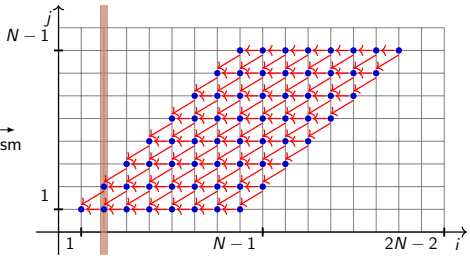
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



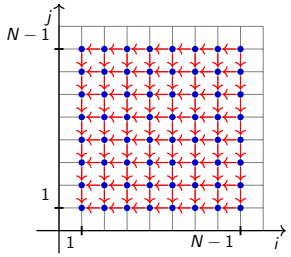
Change the loop traversal
Improve inter-instruction parallelism



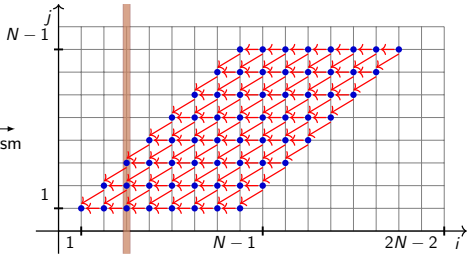
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



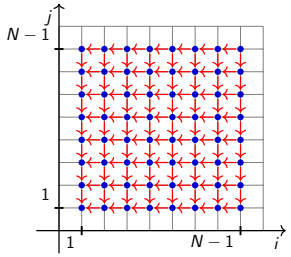
Change the loop traversal
Improve inter-instruction parallelism



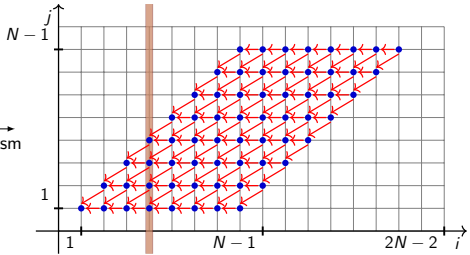
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



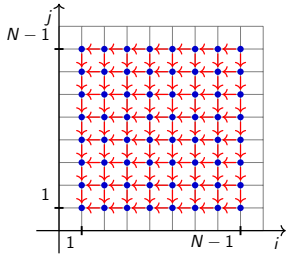
Change the loop traversal
Improve inter-instruction parallelism



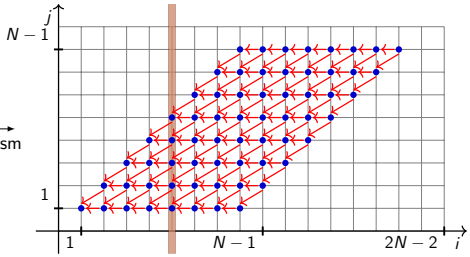
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



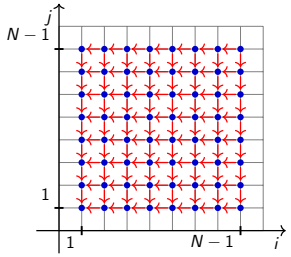
Change the loop traversal
Improve inter-instruction parallelism



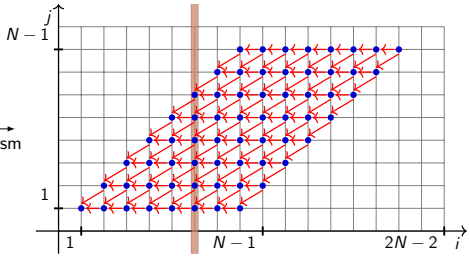
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



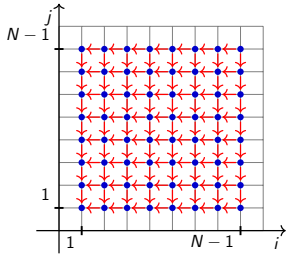
Change the loop traversal
Improve inter-instruction parallelism



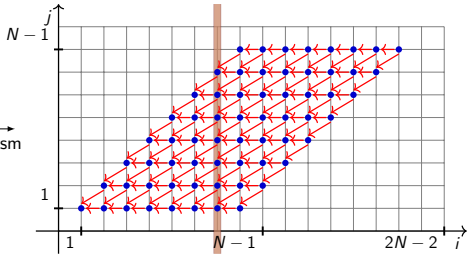
A Very Nice Loop

```
for (int i = 1 ; i < N ; ++i)
  for (int j = 1 ; j < N ; ++j)
/* s: */    A[i][j] = A[i][j-1] + A[i-1][j];
```

• Statement
⟵ Dependence



Change the loop traversal
Improve inter-instruction parallelism



What About the Real World?

- Linear Algebra kernels
- Simulations
- Machine learning
- Image Processing
- ...


```
for (int i = 0 ; i < N ; ++i)
  for (int j = 0 ; j < N ; ++i)
    B[i][j] = (A[i-1][j-1] + A[i-1][j] + A[i-1][j+1]
            +  A[i][j-1]   + A[i][j]   + A[i][j+1]
            +  A[i+1][j-1] + A[i+1][j] + A[i+1][j+1])
            / 9.0
```

Uniform Blur

The Polyhedral Model

The Polyhedral Model (French Success Story since 1991!)

Compiles programs with affine loops ranging over arrays accessed with affine functions into efficient C code by: 1) Analysing dependencies, 2) Reschedule the instructions, 3) Generate C code.

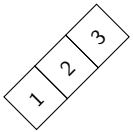
Pros:

- The best tool when it comes to polyhedral programs
- 30-year of active research and improvements
- Available in productions compilers (gcc, llvm)

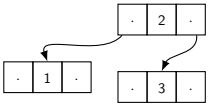
Cons:

- Still limited to arrays

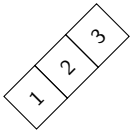
Still Limited to Arrays, the Whys?



Properties	Array	Pointers
Contiguous in Memory	✔	✖
Random Access	✔	✖

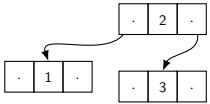


Still Limited to Arrays, the Whys?



Properties	Array	Pointers
Contiguous in Memory	✔	✖
Random Access	✔	✖

Other nice data structures?



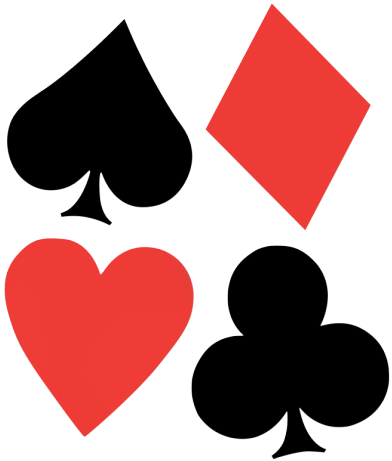
Research Questions

- Are Algebraic Data Types Regular Enough?
- How to Compile ADTs reshaping efficiently?

Are Algebraic Data Types Regular Enough?

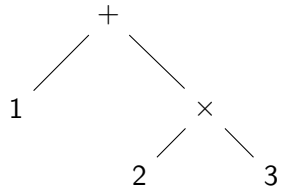
```
type suit =
  | Heart
  | Diamond
  | Clubs
  | Spade
type value =
  | Ace
  | King
  | Queen
  | Jack
  | Number of int
type card = (suit, value)
```

A simple ADT



```
type expr =
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr

let rec eval = function
  | Cst (i)      -> i
  | Add (e1, e2) -> eval e1 + eval e2
  | Mul (e1, e2) -> eval e1 * eval e2

(* 1 + 2 * 3 = 7 *)
eval (Add(Cst(1), Mul(Cst(2), Cst(3)))
```

The expr ADT and its eval function



This is “regular”, isn’t it?

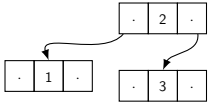
How Are Encoded ADTs Into Memory?

- The constant part (enumeration with arguments) has a fixed size

How Are Encoded ADTs Into Memory?

- The constant part (enumeration with arguments) has a fixed size
- The recursion is captured by using pointers

```
struct tree {
  int val;
  struct tree *left, *right;
};
```



How Are Encoded ADTs Into Memory?

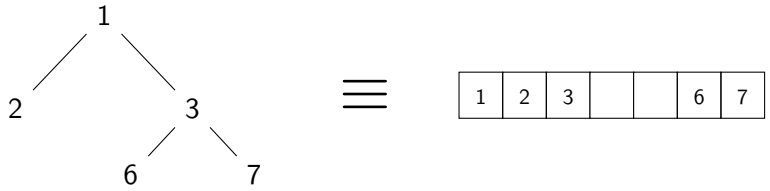
- The constant part (enumeration with arguments) has a fixed size
- The recursion is captured by using pointers
- But pointers = no predictability

How Are Encoded ADTs Into Memory?

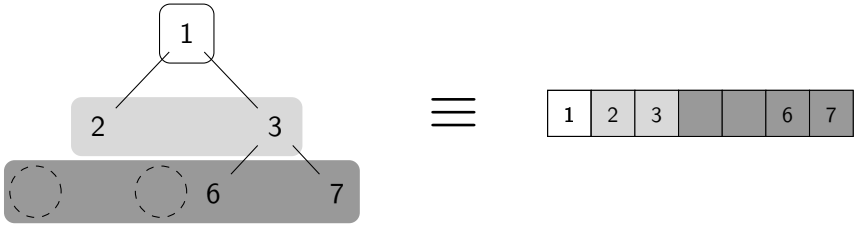
- The constant part (enumeration with arguments) has a fixed size
- The recursion is captured by using pointers
- But pointers = no predictability

What about exploiting the regular properties of ADTs...
...and store them into arrays!

Breadth-first Layout and Its Properties

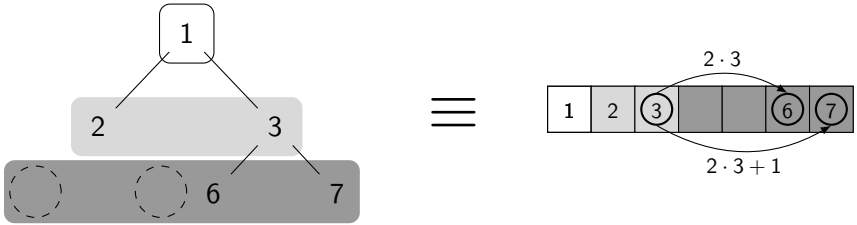


Breadth-first Layout and Its Properties



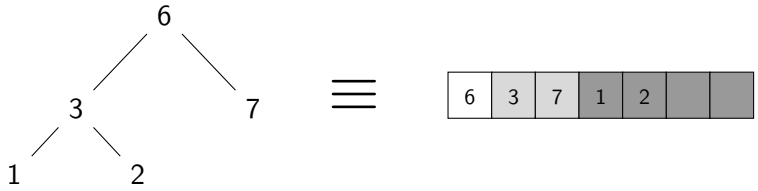
- A notion of layers

Breadth-first Layout and Its Properties



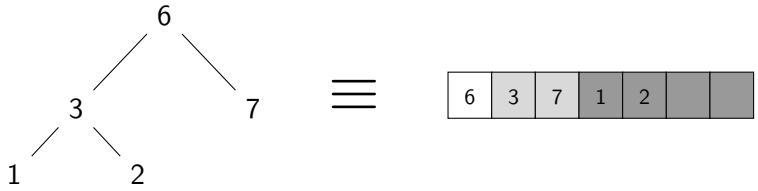
- A notion of layers
- A simple formula for finding parent and children (it's a shift away)

Breadth-first Layout and Its Properties



- A notion of layers
- A simple formula for finding parent and children (it's a shift away)
- The tree can be compressed, if only values matter.

Breadth-first Layout and Its Properties



- A notion of layers
- A simple formula for finding parent and children (it's a shift away)
- The tree can be compressed, if only values matter.
- Insertion happens at the end, or in a hole

The Layer Tower

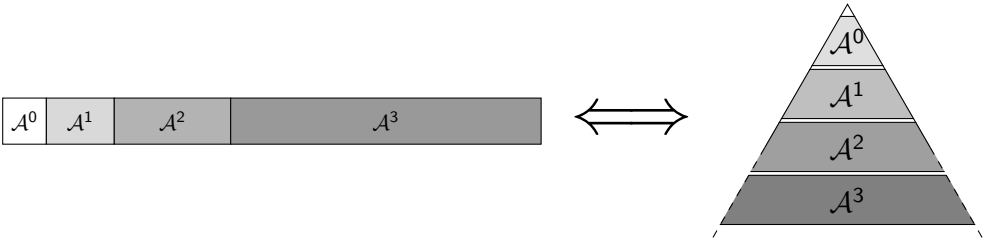


Figure: A graphical notation

Other Layouts

- Variation without holes
- Obviously, depth-first layout
- Cache oblivious community has designed many byzantine layouts
- Most famous: Van Em Boas Layout (see manuscript)

Other Layouts

- Variation without holes
- Obviously, depth-first layout
- Cache oblivious community has designed many byzantine layouts
- Most famous: Van Em Boas Layout (see manuscript)

The rest of the presentation uses the breadth-first layout

A Data-Structure to Study

We want a data-structure:

- which is actually used
- which would benefit from optimizations
- which is neither to complex nor to simple

A Data-Structure to Study

We want a data-structure:

- which is actually used
- which would benefit from optimizations
- which is neither to complex nor to simple

We chose AVL Trees

```
type avl =
    | Empty
    | Node of avl * int * avl
```

Operations:

- Insert/Delete/Find
- Rotations (Structural Modifications)

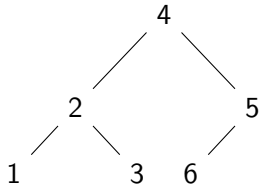
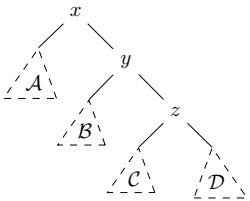
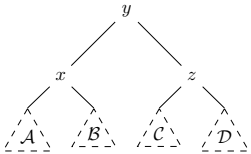


Figure: An AVL



(a) A binary tree T



(b) T after L-rotation

AVL Implementation

AVL with pointers	Implicit layout (Tarbres)
Rotations are easy and cheap $\mathcal{O}(1)$	Rotations in place?
Bad cache behavior (find/map)	Nice cache behavior (find/map)

AVL Rebalancing: Low Level Operations 1/2

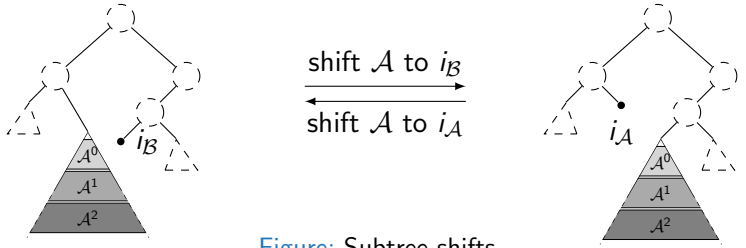


Figure: Subtree shifts

AVL Rebalancing: Low Level Operations 1/2

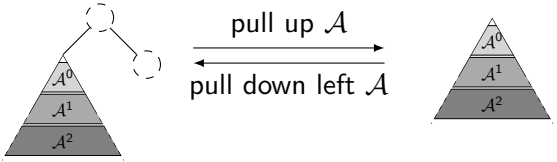
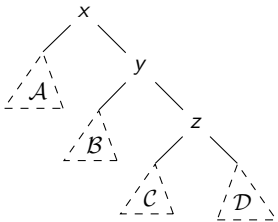


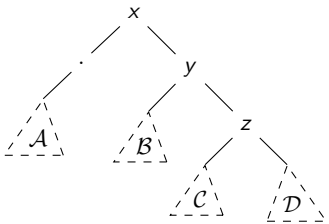
Figure: Subtree pull-ups and pull-downs

AVL Rebalancing: Hands on!



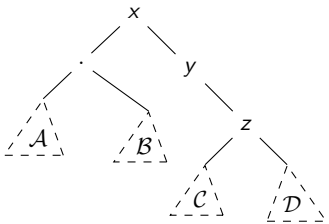
- Original Unbalanced State

AVL Rebalancing: Hands on!



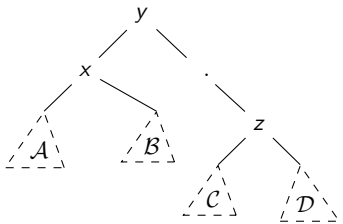
- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$

AVL Rebalancing: Hands on!



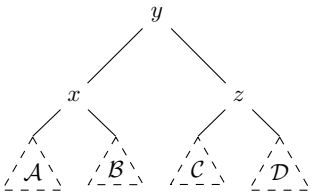
- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$
- Shift Left $\mathcal{B}$

AVL Rebalancing: Hands on!



- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$
- Shift Left $\mathcal{B}$
- Rename

AVL Rebalancing: Hands on!



- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$
- Shift Left $\mathcal{B}$
- Rename
- Move up (z-$\mathcal{B}$-$\mathcal{C}$)

Implementation

- Support for insert/delete/find/map
- Rotation as a combination of low-level operations
- Support for scalar-type and pointer-type values
- The low-level operations can be:
  - Naive: move one layer at a time
  - Pipelined: try to start the next move as soon as possible
  - Parallelized
- Available at: `https://gitlab.inria.fr/paiannet/calv/`

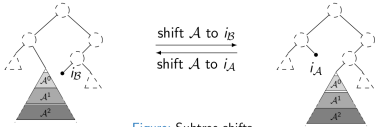
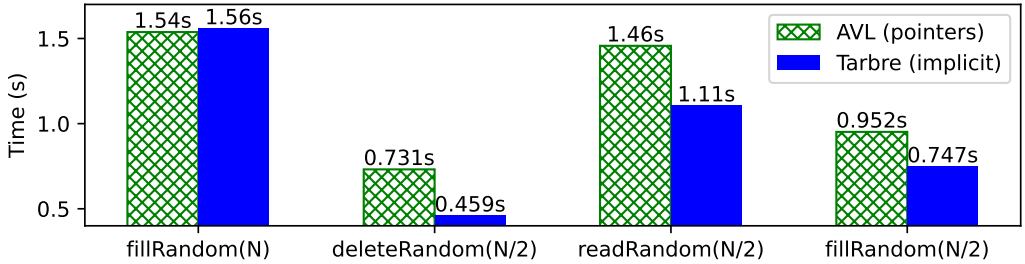


Figure: Subtree shifts

Benchmark on a Key-Value Store Scenario (N = 1M $\approx 2^{20}$)

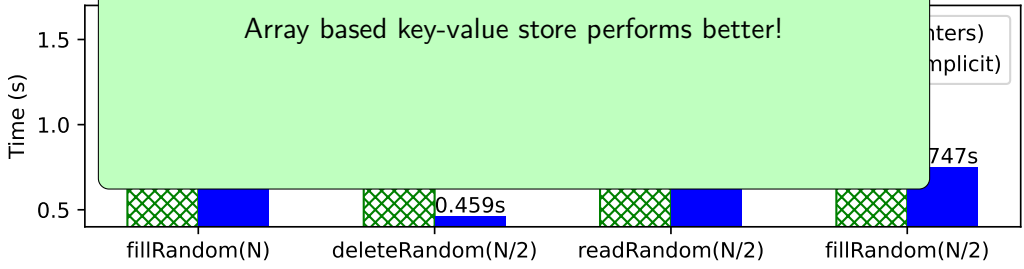
- A benchmark adapted from the database community
- Scenario:
  - Emulate a key-value database
  - Check the performance of pointers vs arrays.



Benchmark on a Key-Value Store Scenario (N = 1M $\approx 2^{20}$)

- A benchmark adapted from the database community
- Scen
  -
  -

Array based key-value store performs better!



Summary of the Contribution

- Implemented as a small library (calv)
- We tried to exploit the parallelism of the low-level operations
- It works!

Summary of the Contribution

- Implemented as a small library (calv)
- We tried to exploit the parallelism of the low-level operations
- It works!
- Hand Tuned

Summary of the Contribution

- Implemented as a small library (calv)
- We tried to exploit the parallelism of the low-level operations
- It works!
- Hand Tuned
- Improve parallelism
  - Improve granularity of low level operations

Summary of the Contribution

- Implemented as a small library (calv)
- We tried to exploit the parallelism of the low-level operations
- It works!
- Hand Tuned
- Improve parallelism
  - Improve granularity of low level operations

Let's try to automate!

How to compile ADTs reshaping efficiently?

Our Proposition: REW (A DSL to Compile Structural Transformations)

REW is small DSL to:

- Algebraic Data Types without sharing
- describe structural transformations on those through pattern matching.

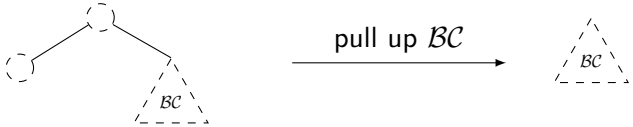
```
type tree = Empty | Node (tree,int,tree)

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
}
```

Figure: What REW looks like

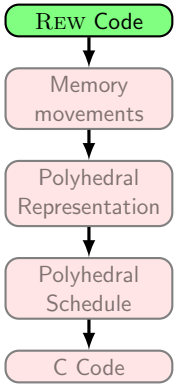
Compilation Steps

1. Give the specifications (describe the types & the transformations)
2. Figure out the movements
3. Compute the dependences between the movements
4. Reschedule the movements
5. Generate the code

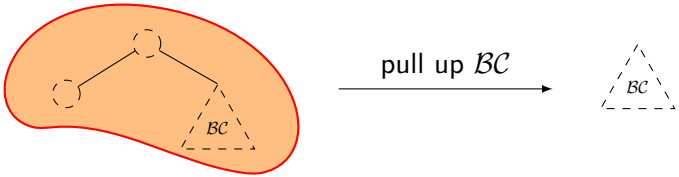
Step 1: Describe Structural Transformations



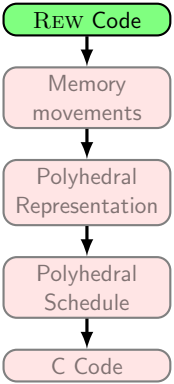

```
type tree = Empty | Node (tree,int,tree)

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
}
```



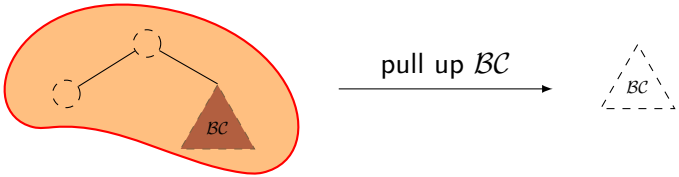
Step 1: Describe Structural Transformations



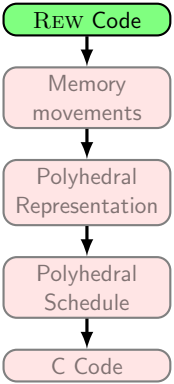

```
type tree = Empty | Node (tree,int,tree)

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
}
```



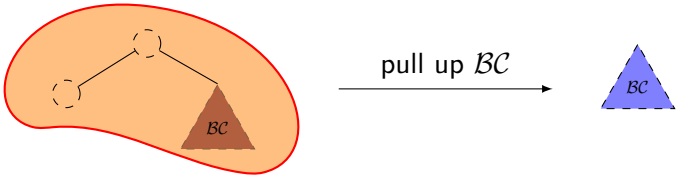
Step 1: Describe Structural Transformations



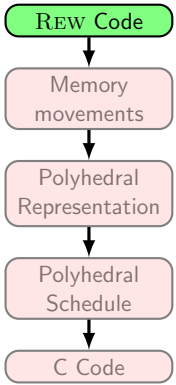

```
type tree = Empty | Node (tree,int,tree)

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
}
```



Step 1: Describe Structural Transformations

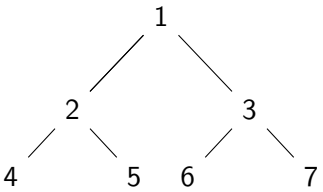



```
type tree = Empty | Node (tree,int,tree)

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 Paths

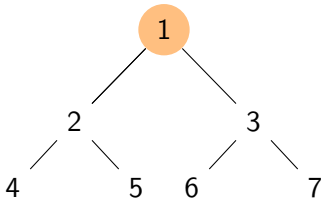
```
type tree = Empty | Node (tree,int,tree)
                    .0    .1    .2
```



ϵ

Our Notations for Paths

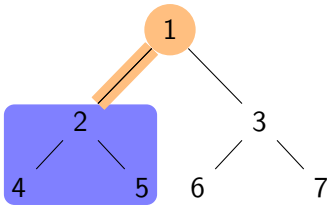
```
type tree = Empty | Node (tree,int,tree)
              .0      .1     .2
```



.1

Our Notations for Paths

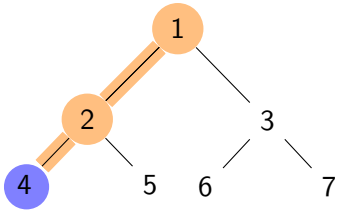
```
type tree = Empty | Node (tree,int,tree)
              .0      .1     .2
```



.0

Our Notations for Paths

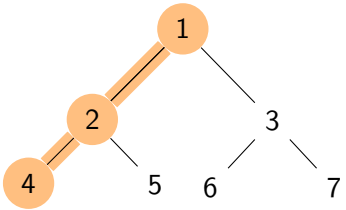
```
type tree = Empty | Node (tree,int,tree)
                    .0      .1     .2
```



$$.0.0 \equiv (.0)^2$$

Our Notations for Paths

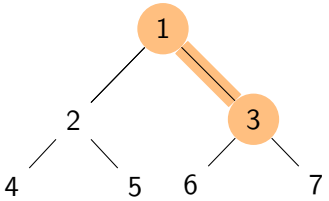
```
type tree = Empty | Node (tree,int,tree)
                      .0    .1    .2
```



$(.0)^2.1$

Our Notations for Paths

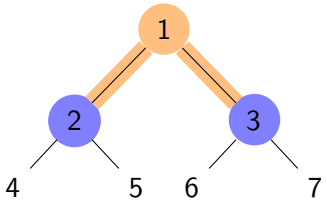
```
type tree = Empty | Node (tree,int,tree)
                    .0     .1    .2
```



.2.1

Our Notations for Paths

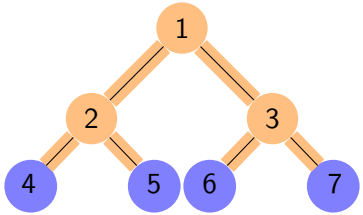
```
type tree = Empty | Node (tree,int,tree)
                          .0    .1    .2
```



φ

Our Notations for Paths

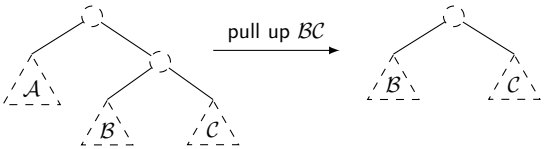
```
type tree = Empty | Node (tree,int,tree)
                          .0    .1    .2
```



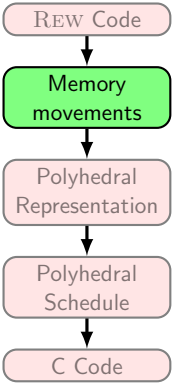
$\varphi.\varphi$

Step 2: Compute Subtree Movements

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

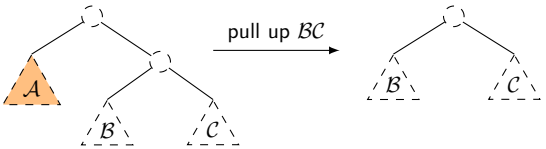


$$\langle\!| a : tree \mid .0 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| i : int \mid .1 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| b : tree \mid .2.0 \rightarrow .0 |\!\rangle$$
$$\langle\!| j : int \mid .2.1 \rightarrow .1 |\!\rangle$$
$$\langle\!| c : tree \mid .2.2 \rightarrow .2 |\!\rangle$$

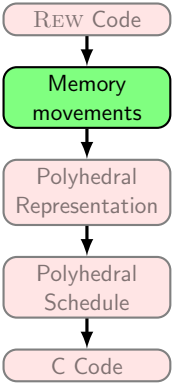


Step 2: Compute Subtree Movements

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

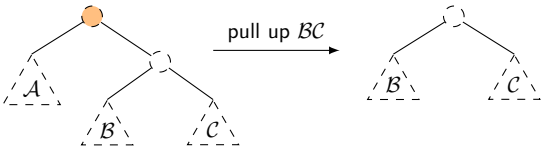


$$\langle\!| \, a : tree \mid .0 \rightarrow \emptyset \, |\!\rangle$$
$$\langle\!| \, i : int \mid .1 \rightarrow \emptyset \, |\!\rangle$$
$$\langle\!| \, b : tree \mid .2.0 \rightarrow .0 \, |\!\rangle$$
$$\langle\!| \, j : int \mid .2.1 \rightarrow .1 \, |\!\rangle$$
$$\langle\!| \, c : tree \mid .2.2 \rightarrow .2 \, |\!\rangle$$

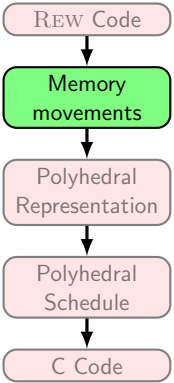


Step 2: Compute Subtree Movements

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

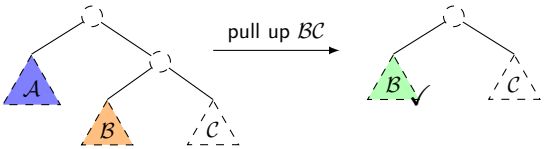


$$\langle\!| a : tree \mid .0 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| i : int \mid .1 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| b : tree \mid .2.0 \rightarrow .0 |\!\rangle$$
$$\langle\!| j : int \mid .2.1 \rightarrow .1 |\!\rangle$$
$$\langle\!| c : tree \mid .2.2 \rightarrow .2 |\!\rangle$$

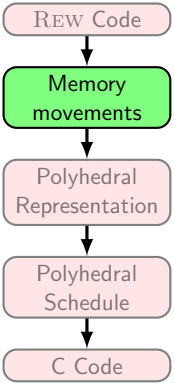


Step 2: Compute Subtree Movements

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

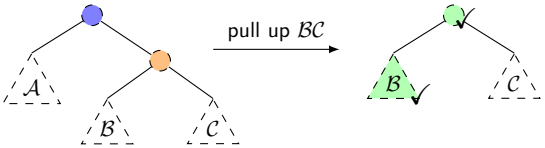


$$\langle\!| a : tree \mid .0 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| i : int \mid .1 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| b : tree \mid .2.0 \rightarrow .0 |\!\rangle$$
$$\langle\!| j : int \mid .2.1 \rightarrow .1 |\!\rangle$$
$$\langle\!| c : tree \mid .2.2 \rightarrow .2 |\!\rangle$$

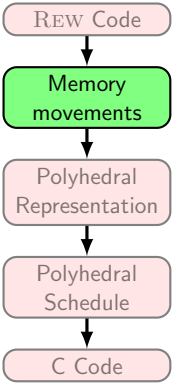


Step 2: Compute Subtree Movements

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

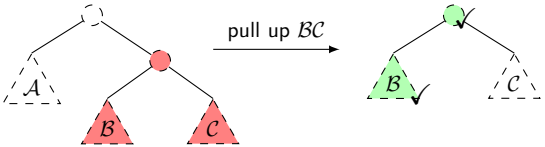


$$\langle\!| a : tree \mid .0 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| i : int \mid .1 \rightarrow \emptyset |\!\rangle$$
$$\langle\!| b : tree \mid .2.0 \rightarrow .0 |\!\rangle$$
$$\langle\!| j : int \mid .2.1 \rightarrow .1 |\!\rangle$$
$$\langle\!| c : tree \mid .2.2 \rightarrow .2 |\!\rangle$$

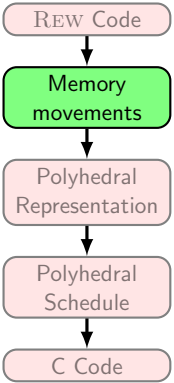


Step 2: Compute Subtree Movements

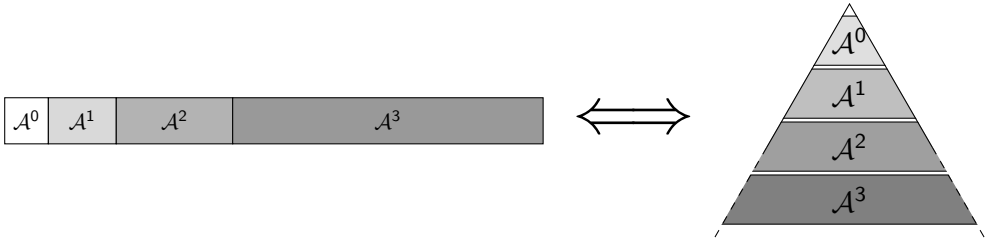
```
Node( a , i ,Node( b , j , c )) -> Node( b , j , c )
```



$$(\!| a : tree \mid .0 \rightarrow \emptyset |\!)$$
$$(\!| i : int \mid .1 \rightarrow \emptyset |\!)$$
$$(\!| b : tree \mid .2.0 \rightarrow .0 |\!)$$
$$(\!| j : int \mid .2.1 \rightarrow .1 |\!)$$
$$(\!| c : tree \mid .2.2 \rightarrow .2 |\!)$$

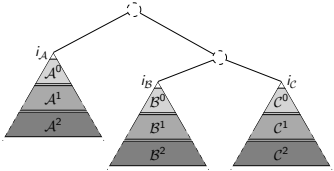


Use Layers to Subdivide Memory Movements

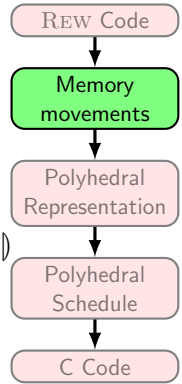


Step 2: Layer-aware movements (1/2)

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

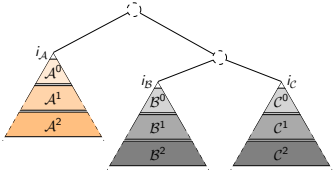


$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$

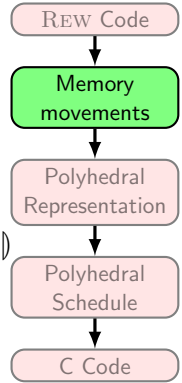


Step 2: Layer-aware movements (1/2)

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

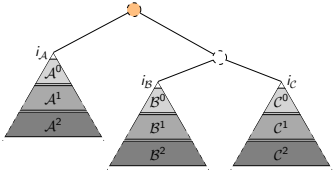


$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$

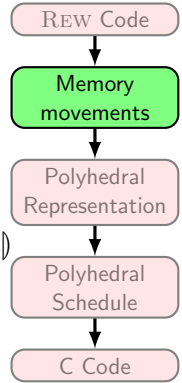


Step 2: Layer-aware movements (1/2)

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

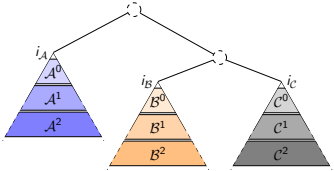


$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$

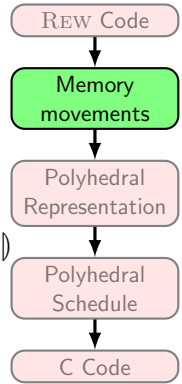


Step 2: Layer-aware movements (1/2)

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

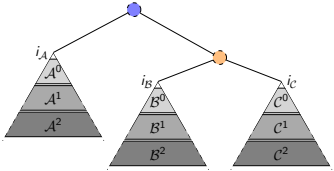


$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$

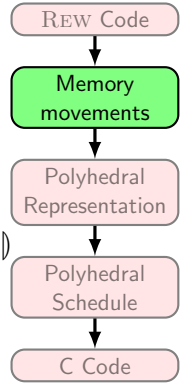


Step 2: Layer-aware movements (1/2)

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

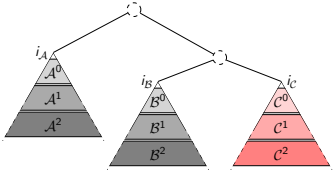


$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$

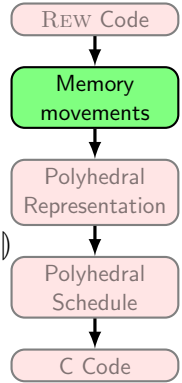


Step 2: Layer-aware movements (1/2)

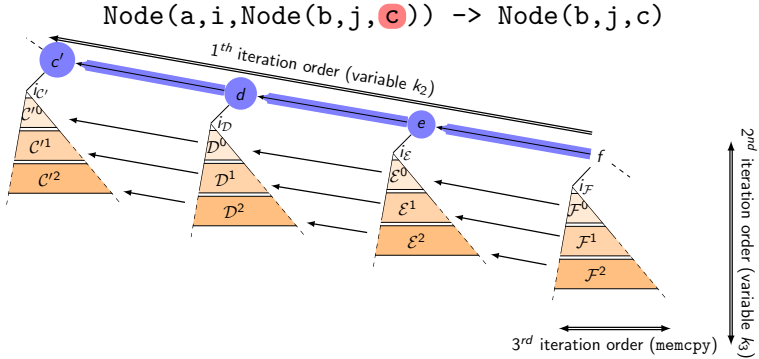
```
Node(a,i,Node(b,j,c)) -> Node(b,j,c)
```



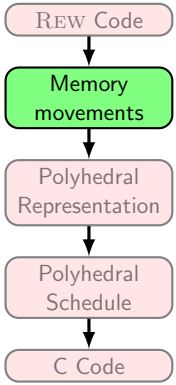
$$
\begin{array}{ll}
(\!| \ .0 \to \emptyset \ |\!) & \rightsquigarrow \forall k_0, (\!| \ .0.\varphi^{k_0} \to \emptyset \ |\!) \\
(\!| \ .1 \to \emptyset \ |\!) & \rightsquigarrow (\!| \ .1 \to \emptyset \ |\!) \\
(\!| \ .2.0 \to .0 \ |\!) & \rightsquigarrow \forall k_1, (\!| \ .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \ |\!) \\
(\!| \ .2.1 \to .1 \ |\!) & \rightsquigarrow (\!| \ .2.1 \to .1 \ |\!) \\
(\!| \ .2.2 \to .2 \ |\!) & \rightsquigarrow ?
\end{array}
$$



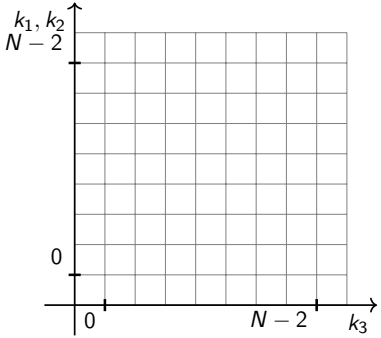
Step 2: Layer-aware movements (2/2)

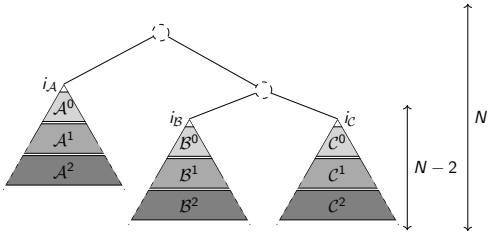


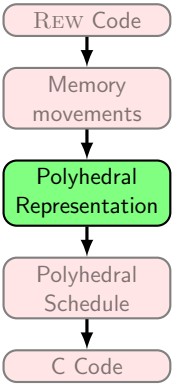
$$\langle\!| .2.2 \to .2 |\!\rangle \rightsquigarrow \begin{cases} \langle\!| .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 |\!\rangle & (c_1) \\ \langle\!| .2.2.2^{k_2}.0.\varphi^{k_3} \to .2.2^{k_2}.0.\varphi^{k_3} |\!\rangle & (c_0) \end{cases}$$



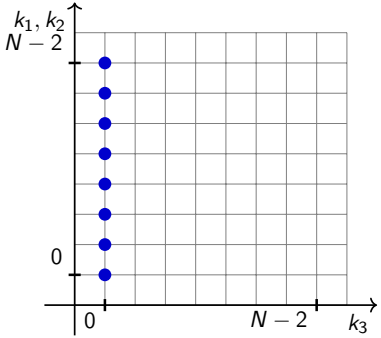
Step 3: Characterizing Memory Movements

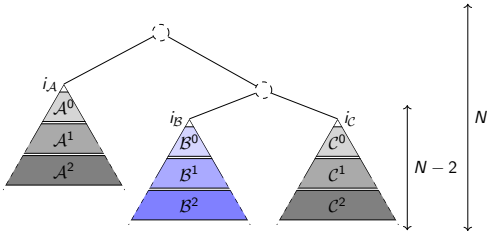




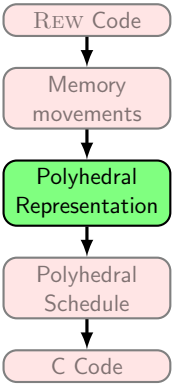


Step 3: Characterizing Memory Movements

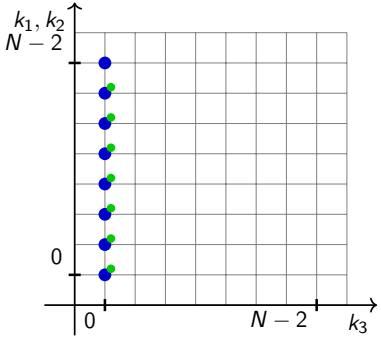


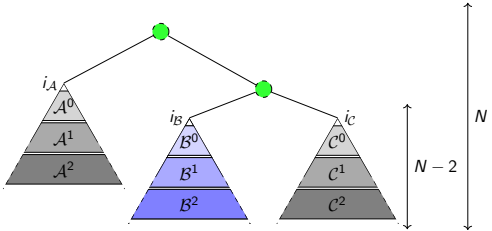


$$(\!|\, .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \,|\!) \qquad (b)$$



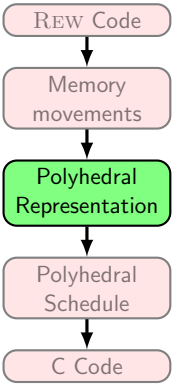
Step 3: Characterizing Memory Movements



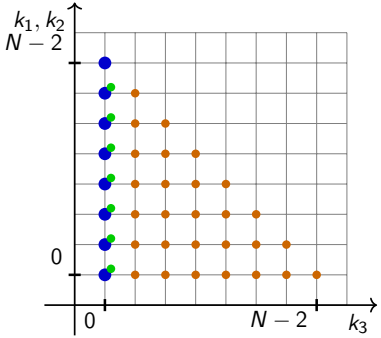


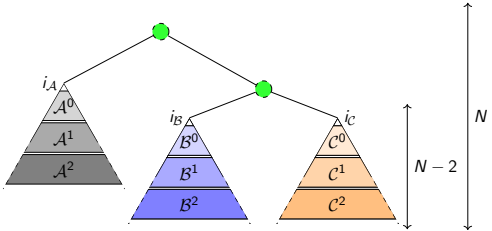
$$(\!|\, .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \,|\!) \qquad (b)$$

$$(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!) \qquad (c1)$$



Step 3: Characterizing Memory Movements

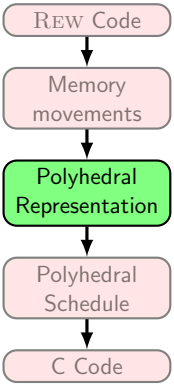




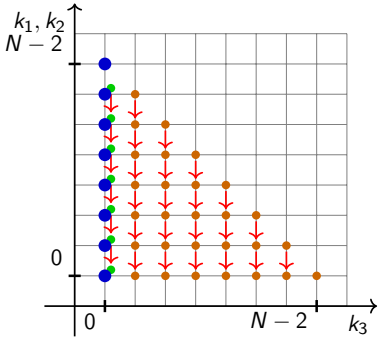
$$(\!|\, .2.0.\varphi^{k_1} \to .0.\varphi^{k_1} \,|\!) \qquad (b)$$

$$(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!) \qquad (c1)$$

$$(\!|\, .2.2.2^{k_2}.0.\varphi^{k_3} \to .2.2^{k_2}.0.\varphi^{k_3} \,|\!) \qquad (c0)$$

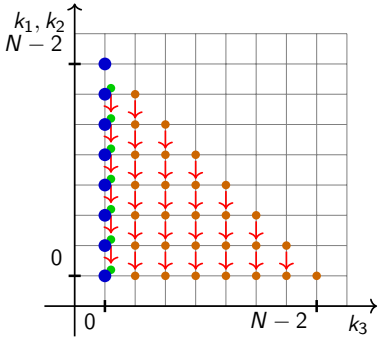


Polyhedral Like Dependencies: Intuition



Let's take the rule (c_0): $(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!)$

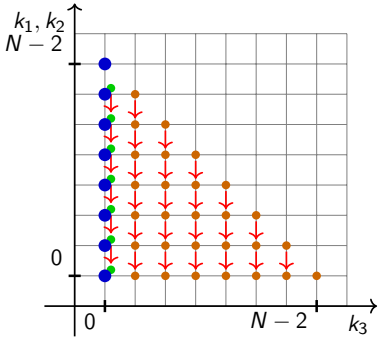
Polyhedral Like Dependencies: Intuition



Let's take the rule (c_0): $(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!)$

- When $k_2 = 1$, (c_0): $(\!|\, (.2)^3\, .1 \to (.2)^2.1 \,|\!)$
- When $k_2 = 2$, (c_0): $(\!|\, (.2)^4.1 \to (.2)^3.1 \,|\!)$

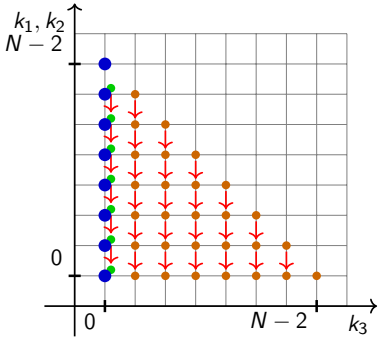
Polyhedral Like Dependencies: Intuition



Let's take the rule (c_0): $(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!)$

- When $k_2 = 1$, (c_0): $(\!|\, (.2)^3\,.1 \to (.2)^2.1 \,|\!)$
- When $k_2 = 2$, (c_0): $(\!|\, (.2)^4.1 \to (.2)^3.1 \,|\!)$

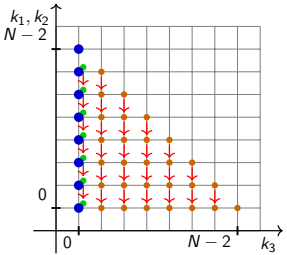
Polyhedral Like Dependencies: Intuition



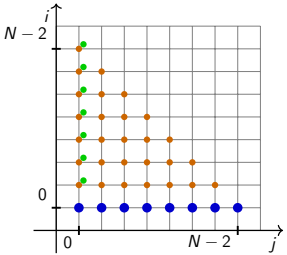
Let's take the rule (c_0): $(\!|\, .2.2.2^{k_2}.1 \to .2.2^{k_2}.1 \,|\!)$

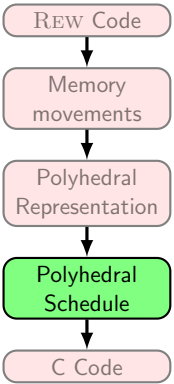
- When $k_2 = 1$, (c_0): $(\!|\, (.2)^3\, .1 \to (.2)^2.1 \,|\!)$
- When $k_2 = 2$, (c_0): $(\!|\, (.2)^4.1 \to (.2)^3.1 \,|\!)$
- More generally, $(c_0)(k)$ and $(c_0)(k+1)$ conflict.

Step 5: Polyhedral Model Based Scheduling

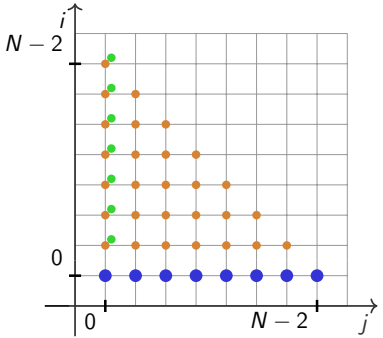


Scheduling by Farkas and ILP Solver

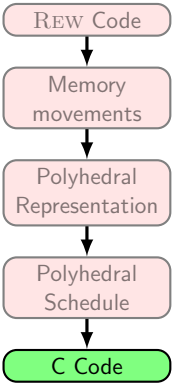




Step 6: From Schedule to Code

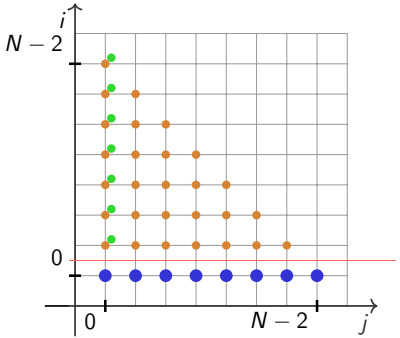


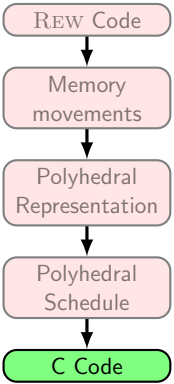
Application of Quilleré's Algorithm



Step 6: From Schedule to Code

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

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

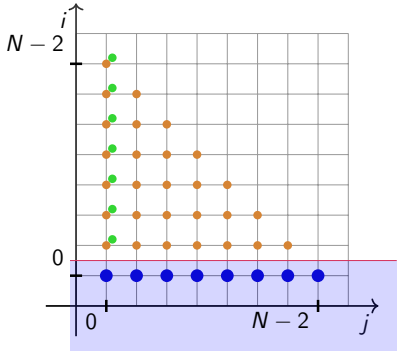




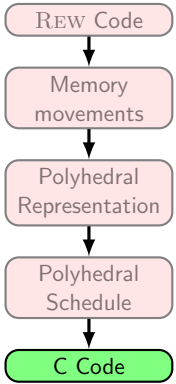
Application of Quilleré's Algorithm

Step 6: 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)
```

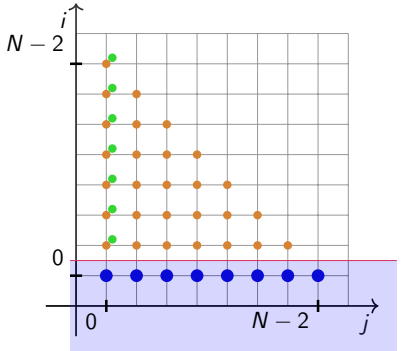


Application of Quilleré's Algorithm

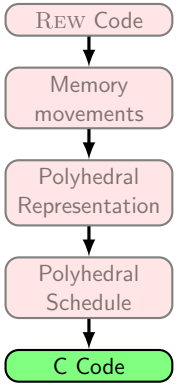


Step 6: From Schedule to Code

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

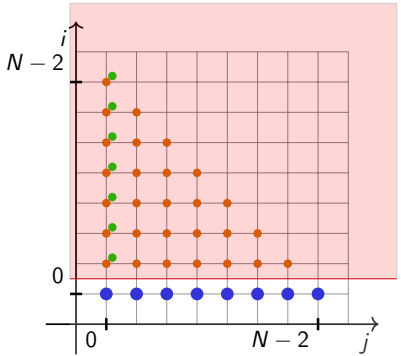


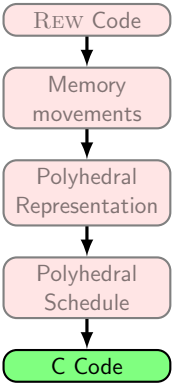
Application of Quilleré's Algorithm



Step 6: From Schedule to Code

```
for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)
    ⟪ .2.0.φ^j → .0.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)
```

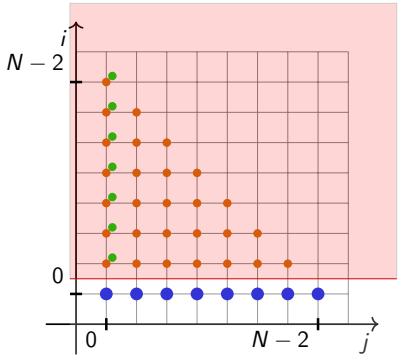


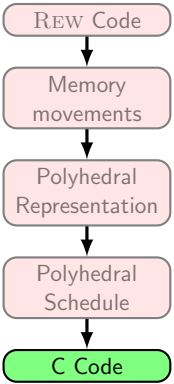


Application of Quilleré's Algorithm

Step 6: From Schedule to Code

```
for (i = 0 ; i <= 0 ; i += 1)
  for (j = 0 ; j <= N-2 ; j += 1)
    ⟪ . 2 . 0.φ^j → . 0.φ^j ⟫  // b
for (i = 1 ; i <= N-2 ; i+= 1)
  for (j = 0 ; j <= 0 ; j += 1)
    ⟪( . 2)^{i+1} . 1 → ( . 2)^i . 1⟫  //c1
  for (j = 0 ; j <= N - i - 2 ; j += 1)
    ⟪( . 2)^{i+1} . 0.φ^j → ( . 2)^i . 0.φ^j ⟫ //c0
```



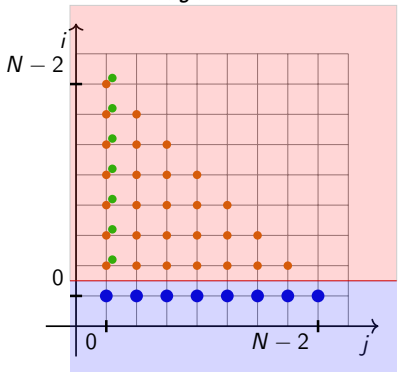


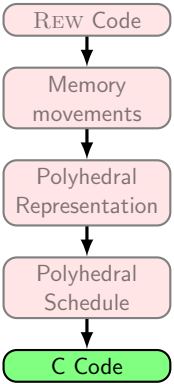
Application of Quilleré's Algorithm

Step 6: From Schedule to Code

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

```
for (j = 0 ; j <= N-2 ; j += 1)
    ⟪ . 2 . 0.φ^j → . 0.φ^j ⟫   // b
for (i = 1 ; i <= N-2 ; i+= 1)
   ⟪ ( . 2)^{i+1} . 1 → ( . 2)^i . 1 ⟫   //c1
   for (j = 0 ; j <= N - i - 2 ; j += 1)
      ⟪ ( . 2)^{i+1} . 0.φ^j → ( . 2)^i . 0.φ^j ⟫   //c0
```





Application of Quilleré's Algorithm

Conflict analysis: The Details

Theorem

Let $(\!| s_1 \to d_1 |\!)$ and $(\!| s_2 \to d_2 |\!)$ be two moves (not necessarily distinct). If we consider d_1 and s_2 as regular expressions.
Then,
$\mathcal{L}(d_1) \cap \mathcal{L}(s_2)$ is the conflict space and it can be characterized by affine inequalities.

Conflict analysis: Compute the dependences for all tuples of movements

Summary of the Contribution

- Promising technique to compile pattern matching-based structural transformations
- Extension of the polyhedral techniques to a different class of regular programs

Summary of the Contribution

- Promising technique to compile pattern matching-based structural transformations
- Extension of the polyhedral techniques to a different class of regular programs

Front end for ADTs which plugs into the polyhedral model

Conclusion & Contributions List

- Are Algebraic Data Types Regular Enough?
  - Yes, but the layout is important
  - Parallelization opportunities
  - Implemented as a small library
  - Better exploited through automation

Conclusion & Contributions List

- Are Algebraic Data Types Regular Enough?
  - Yes, but the layout is important
  - Parallelization opportunities
  - Implemented as a small library
  - Better exploited through automation
- How to Compile ADTs reshaping efficiently?
  - A framework to compile structural transformation
  - Reuse polyhedral ideas
  - Partial implementation by Gabriel

Conclusion & Contributions List

- Are Algebraic Data Types Regular Enough?
  - Yes, but the layout is important
  - Parallelization opportunities
  - Implemented as a small library
  - Better exploited through automation
- How to Compile ADTs reshaping efficiently?
  - A framework to compile structural transformation
  - Reuse polyhedral ideas
  - Partial implementation by Gabriel
- Another Model for the Input Language of the Polyhedral Model?
  - A framework to extend the polyhedral model
  - Based on a modified operational semantics
  - Enabling future non-exact extensions

Future Work

- Extend REW:
  - Add recursion, guards
  - Add support for functions
- Improve the compilation of REW
  - Generate parallel code
  - Improve parallel cache performance
- Continue to explore reformulation of the polyhedral model input language

Thank you for your attention

Questions?