



ADTs FOR FUN AND PROFIT

Laure GONNORD (Grenoble INP/UGA formerly Univ. Lyon) laure.gonnord@grenoble-inp.fr &

Gabriel RADANNE (Inria Lyon) gabriel.radanne@inria.fr

Algebraic Data Types (ADTs)



- A **Frog** is either Idle **or** Jumping to a given **Position** **or** Eating a **Bug** located **at** some **Position**.
- A **Bug** is either a Crawler **or** a Flyer.
- A **Position** is a 3D coordinate in space: x **and** y **and** z.

Algebraic Data Types (ADTs)

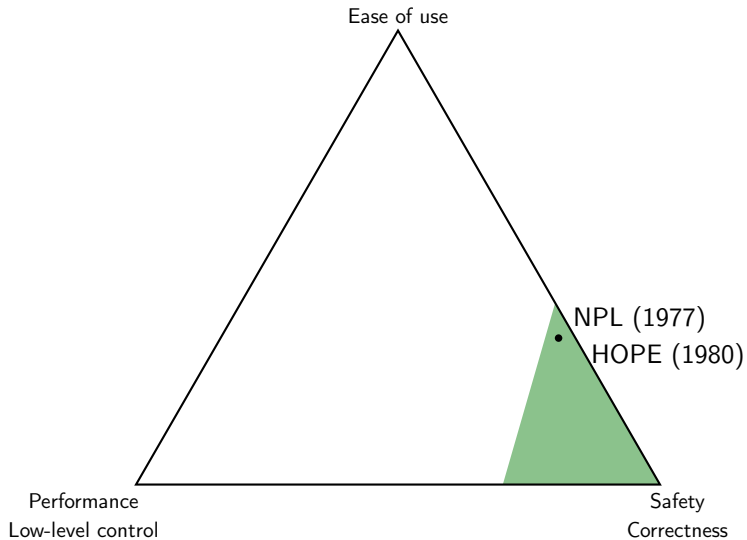


Position: x and y and z

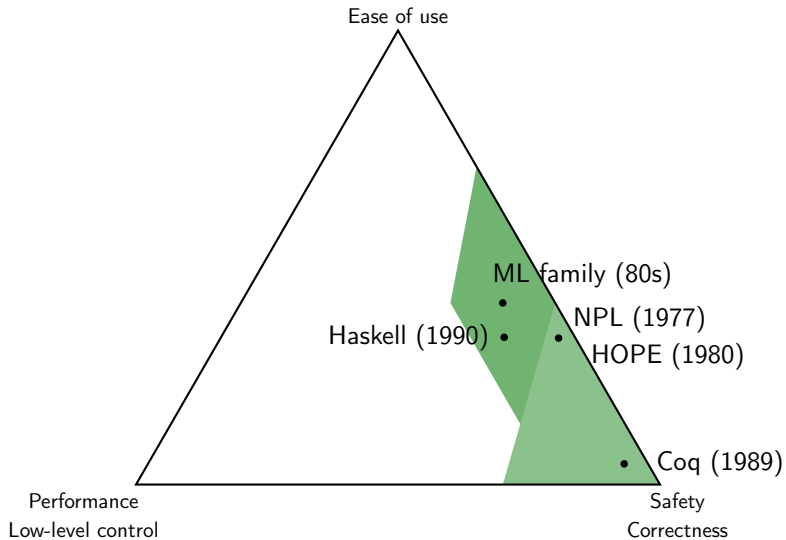
Bug: Crawler or Flyer

Frog: Idle or Jumping(Position) or Eating(Bug and Position)

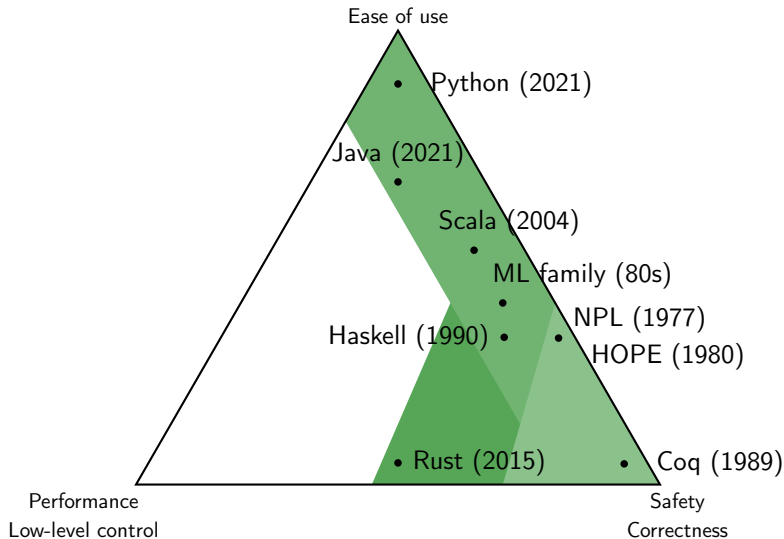
Algebraic Data Types (ADTs)



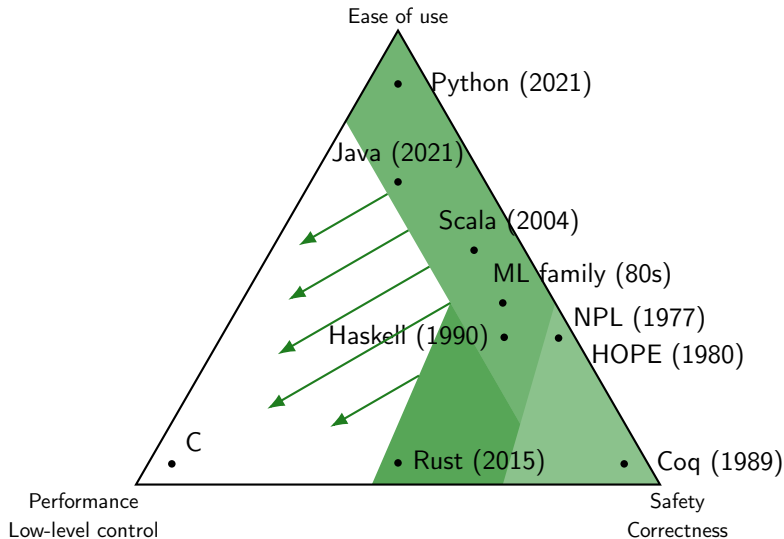
Algebraic Data Types (ADTs)



Algebraic Data Types (ADTs)



Algebraic Data Types (ADTs)



ADTs and pattern matching: what for?

Typing preservation, exhaustivity, non-redundancy

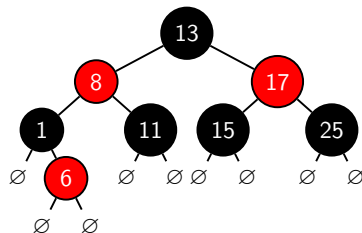
```
enum Color { Red, Black }
```

```
enum RBT {
```

```
    Node({c: Color, v: i64, l: RBT, r: RBT}),
```

```
    Empty
```

```
}
```



ADTs and pattern matching: what for?

Typing preservation, exhaustivity, non-redundancy

```
enum Color { Red, Black }
```

```
enum RBT {
```

```
  Node({c: Color, v: i64, l: RBT, r: RBT}),
```

```
  Empty
```

```
}
```

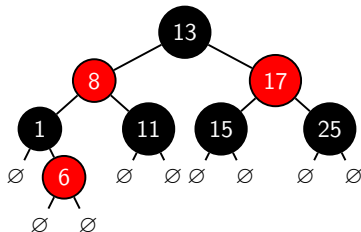
```
fn cardinal(t: RBT) -> u64 { match t {
```

```
  Empty => 0,
```

```
  Node({c: _, v: _, l: t1, r: t2}) =>
```

```
    1 + cardinal(t1) + cardinal(t2),
```

```
}}
```



ADTs and pattern matching: what for?

Typing preservation, exhaustivity, non-redundancy

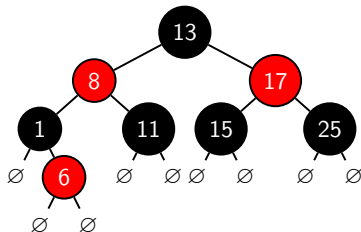
```
enum Color { Red, Black }
```

```
enum RBT {
```

```
  Node({c: Color, v: i64, l: RBT, r: RBT}),
```

```
  Empty
```

```
}
```



```
fn balance(c: Color, v: i64, l: RBT, r: RBT) -> RBT { match (c, v, l, r) {
  Black, z, Node({c: Red, v: y, l: Node({c: Red, v: x, l: a, r: b}), r: c}), d
| Black, z, Node({c: Red, v: x, l: a, r: Node({c: Red, v: y, l: b, r: c})}), d
| Black, x, a, Node({c: Red, v: z, l: Node({c: Red, v: y, l: b, r: c}), r: d})
| Black, x, a, Node({c: Red, v: y, l: b, r: Node({c: Red, v: z, l: c, r: d})})
=> Node({c: Red, v: y, l: Node({c: Black, v: x, l: a, r: b}),
        r: Node({c: Black, v: z, l: c, r: d})}),
| _ => Node({c, v, l, r}),
}}
```

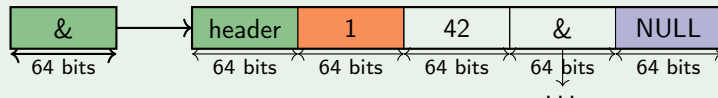
ADT : Memory Representation



How are things layed out in memory?

Naive memory layout

`Node({c: Black, v: 42, l: Node(...), r: Empty})` represented as



What about a more compact representation?



credits XKCD

RBTs in the Linux kernel - C data type

More data in less space!

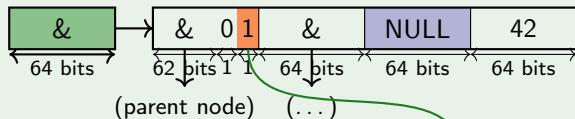
```
struct rb_node {
    unsigned long rb_parent_color;
    struct rb_node *rb_right, *rb_left;
};

struct rbt_i64_node {
    rb_node node;
    long value;
};

typedef rbt_i64_node* RBT;
```

Datatypes

`Node` ({c: `Black`, v: 42, l: `Node(...)`, r: `Empty`})



RBT linux: data manipulation code

Ok ... how do we implement that?

Data manipulation code

```
// Parent pointer accessor  
(struct rb_node)*(r->rb_parent_color & ~3);
```

RBT linux: data manipulation code

Ok ... how do we implement that?

Data manipulation code

```
// Parent pointer accessor
(struct rb_node)*(r->rb_parent_color & ~3);
```

Rebalancing Operation

```
if (rb_is_black(parent)) break;
gparent = rb_red_parent(parent); tmp = gparent->rb_left;
if (tmp && rb_is_red(tmp)) { /* Case 1 - color flips */
    rb_set_parent_color(tmp, gparent, RB_BLACK);
    rb_set_parent_color(parent, gparent, RB_BLACK);
    node = gparent; parent = rb_parent(node);
    rb_set_parent_color(node, parent, RB_RED);
    continue; }
tmp = parent->rb_left;
if (node == tmp) { /* Case 2 - right rotate at parent */
    tmp = node->rb_right;
    parent->rb_left = tmp;
    node->rb_right = parent;
    if (tmp) rb_set_parent_color(tmp, parent, RB_BLACK);
    rb_set_parent_color(parent, node, RB_RED);
    augment_rotate(parent, node); parent = node; tmp = node->
    rb_left; }
gparent->rb_right = tmp; /* Case 3 - left rotate at gparent */
parent->rb_left = gparent;
if (tmp) rb_set_parent_color(tmp, gparent, RB_BLACK);
__rb_rotate_set_parents(gparent, parent, root, RB_RED);
augment_rotate(gparent, parent);
break;
```

RBT linux: data manipulation code

Ok ... how do we implement that?

Data manipulation code

```
// Parent pointer accessor  
(struct rb_node)*(r->rb_parent_color & ~3);
```

Rebalancing Operation

```
if (rb_is_black(parent)) break;  
gparent = rb_red_parent(parent); tmp = gparent->rb_left;  
if (tmp && rb_is_red(tmp)) { /* Case 1 - color flips */  
    rb_set_parent_color(tmp, gparent, RB_BLACK);  
    rb_set_parent_color(parent, gparent, RB_BLACK);  
    node = gparent; parent = rb_parent(node);  
    rb_set_parent_color(node, parent, RB_RED);  
    continue; }  
tmp = parent->rb_left;  
if (node == tmp) { /* Case 2 - right rotate at parent */  
    tmp = node->rb_right;  
    parent->rb_left = tmp;  
    node->rb_right = parent;  
    if (tmp) rb_set_parent_color(tmp, parent, RB_BLACK);  
    rb_set_parent_color(parent, node, RB_RED);  
    augment_rotate(parent, node); parent = node; tmp = node->  
    rb_left; }  
gparent->rb_right = tmp; /* Case 3 - left rotate at gparent */  
parent->rb_left = gparent;  
if (tmp) rb_set_parent_color(tmp, gparent, RB_BLACK);  
__rb_rotate_set_parents(gparent, parent, root, RB_RED);  
augment_rotate(gparent, parent);  
break;
```

tedious

error-prone

RBT linux: data manipulation code

Ok ... how do we implement that?

Data manipulation code

```
// Parent pointer accessor  
(struct rb_node)*(r->rb_parent_color & ~3);
```

Question 1

How to specify custom layouts easily and safely?

Rebalancing Operation

```
if (rb_is_black(parent)) break;  
gparent = rb_red_parent(parent); tmp = gparent->rb_left;  
if (tmp && rb_is_red(tmp)) { /* Case 1 - color flips */  
    rb_set_parent_color(tmp, gparent, RB_BLACK);  
    rb_set_parent_color(parent, gparent, RB_BLACK);  
    node = gparent; parent = rb_parent(node);  
    rb_set_parent_color(node, parent, RB_RED);  
    continue; }  
tmp = parent->rb_left;  
if (node == tmp) { /* Case 2 - right rotate at parent */  
    tmp = node->rb_right;  
    parent->rb_left = tmp;  
    node->rb_right = parent;  
    if (tmp) rb_set_parent_color(tmp, parent, RB_BLACK);  
    rb_set_parent_color(parent, node, RB_RED);  
    augment_rotate(parent, node); parent = node->  
    rb_left; }  
gparent->rb_right = tmp; /* Case 3 - left rotate at gparent */  
parent->rb_left = gparent;  
if (tmp) rb_set_parent_color(tmp, gparent, RB_BLACK);  
__rb_rotate_set_parents(gparent, parent, root, RB_RED);  
augment_rotate(gparent, parent);  
break;
```

tedious

error-prone

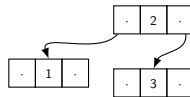
What about a *more compact* representation?



credits XKCD

A more linear representation for trees

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



- But pointers = no predictability
- We can exploit “regularity” and store them into arrays.

Breadth-first layout for trees



- 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

Question 2

How to generate efficient code for linearized representation automatically?

Our **research objectives**:

1. How to specify custom layouts easily and safely?
2. How to generate efficient code for linearized representation automatically?

I. Ribbiting Memory Representations

featuring Thaïs Baudon



Looking for Memory Representations

I have a **description** and **adequation** problem:

- I want to describe a memory representation ...
- ... which fits my type.

Looking for Memory Representations

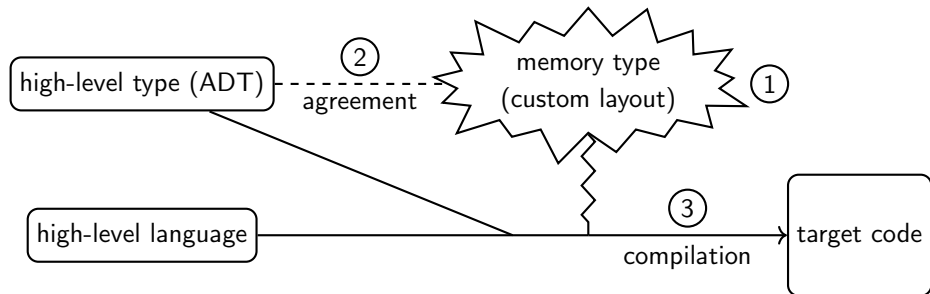
I have a **description** and **adequation** problem:

- I want to describe a memory representation ...
- ... which fits my type.

I make a new (description)
language !



Our approach



Ribbit

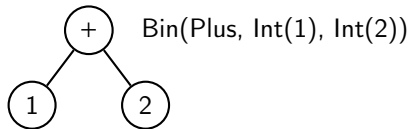
- standalone toy language
 - ADTs and pattern matching
 - memory layout specification with memory types
 - first-order, monomorphic, pure functional language
- compiler to custom target IR
- Ribbitulus: formal semantics and soundness proofs
- online demo: <https://gitlab.inria.fr/ribbit/ribbit>



Layout specification with memory types

Running example: arithmetic expressions

```
enum Op { Plus, Mult }
```



```
enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) }
```

Layout specification with memory types

Running example: arithmetic expressions

```
enum Op { Plus, Mult }
```

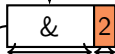


8 bits



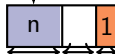
8 bits

```
enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) }
```



62 bits

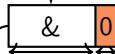
2



32b

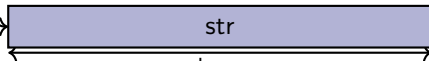
30

2

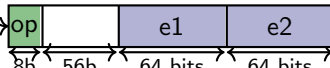


62 bits

2



large



8b

56b

64 bits

64 bits

Layout specification with memory types: Op

`enum Op { Plus, Mult }` } high-level type

represented as

`split {`
 `| 0 from Plus => (0)<8>`
 `| 1 from Mult => (1)<8>`
`}` } memory type



Layout specification with memory types: Op

```
enum Op { Plus, Mult }
```

} high-level type

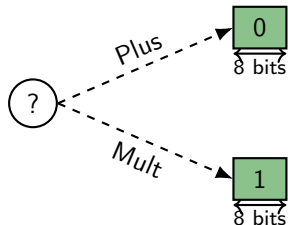
represented as

```
split {  
  | 0 from Plus => (0)<8>  
  | 1 from Mult => (1)<8>  
}
```

} memory type

provenance

discriminant value



Layout specification with memory types: Exp

```
enum Exp { Var (String), Int (i32), Bin (Op, Exp, Exp) }
```

represented as **discriminant location**

```
split .[0:2] {
```

```
}
```

Layout specification with memory types: Exp

```
enum Exp { Var( String ), Int( i32 ), Bin( Op , Exp , Exp ) }
```

represented as

```
split .[0:2] {
```

```
| 1 from Int(_) =>
```

```
| 2 from Var(_) =>
```

```
| 0 from Bin(_) =>
```

provenance

discriminant value

```
}
```

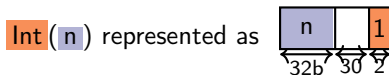
Layout specification with memory types: Exp

```
enum Exp { Var (String), Int (i32), Bin (Op, Exp, Exp) }
```

represented as

```
split .[0:2] {  
  | 1 from Int(_) => _<64> with [0:2]: (1)<2> with [32:32]: (.Int as i32)  
  | 2 from Var(_) =>  
  | 0 from Bin(_) =>  
}
```

Diagram illustrating the layout specification for the `Int` variant of the `Exp` enum. A bracket above the code indicates a "composite word". The expression `[0:2]: (1)<2>` is annotated with "bit ranges" and "[offset : width]". The expression `[32:32]: (.Int as i32)` is annotated with "fragment".



Layout specification with memory types: Exp

```
enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) }
```

represented as

```
split .[0:2] {
```

```
  | 1 from Int(_) => _<64> with [0:2]: (1)<2> with [32:32]: (.Int as i32)
```

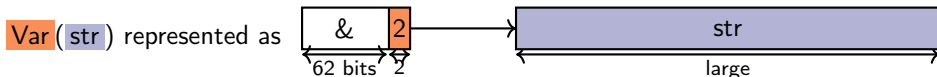
```
  | 2 from Var(_) => &<64>(.Var as String) with [0:2] : (2)<2>
```

```
  | 0 from Bin(_) =>
```

fragment

bit range
(address alignment bits)

```
}
```



Layout specification with memory types: Exp

```
enum Exp { Var (String), Int (i32), Bin (Op, Exp, Exp) }
```

represented as

```
split .[0:2] {
```

```
| 1 from Int(_) => _<64> with [0:2]: (1)<2> with [32:32]: (.Int as i32)
```

```
| 2 from Var(_) => &<64>( .Var as String ) with [0:2] : (2)<2>
```

```
| 0 from Bin(_) =>
```

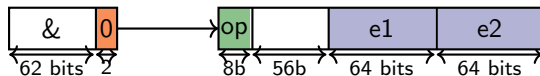
```
&<64>({{ (.Bin.0 as Op), _<56>, (.Bin.1 as Exp), (.Bin.2 as Exp) }})
```

```
with [0:2] : (0)<2>
```

```
}
```

fragments

Bin (op, e1, e2) represented as



Layout specification with memory types: Exp

```
enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) }
```

} ADT

represented as

```
split .[0:2] {
```

```
  | 1 from Int(_) => _<64> with [0:2]: (1)<2> with [32:32]: (.Int as i32)
```

```
  | 2 from Var(_) => &<64>(.Var as String) with [0:2] : (2)<2>
```

```
  | 0 from Bin(_) =>
```

```
    &<64>({{ (.Bin.0 as Op), _<56>, (.Bin.1 as Exp), (.Bin.2 as Exp) }})
```

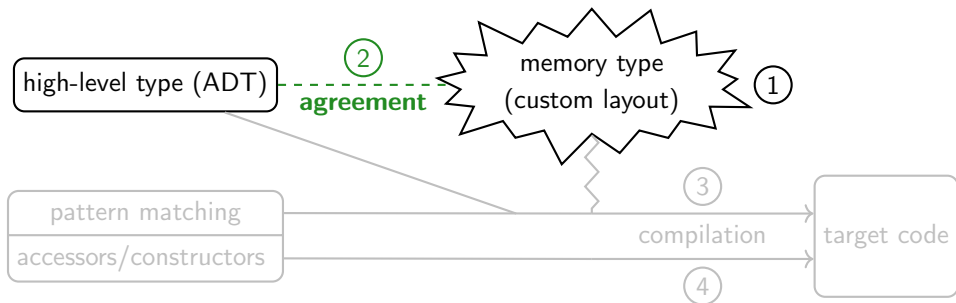
```
    with [0:2] : (0)<2>
```

```
}
```

}
memory
type

Is this a *good* memory layout for Exp?

Ensuring layout correctness



Ensuring layout correctness: agreement with ADTs

```
enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) } repr as
split .[0:2] {
  | 1 from Int(_) => _<64> with [0:2] : (1)<2> with [32:32] : (.Int as i32)
  | 2 from Var(_) => &<64>(.Var as String) with [0:2] : (2)<2>
  | 0 from Bin(_) => &<64>({{
    (.Bin.0 as Op), _<56>, (.Bin.1 as Exp), (.Bin.2 as Exp)
  }}) with [0:2] : (0)<2>
}
```

Agreement criteria

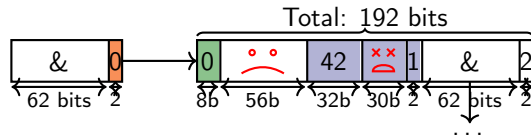
- **Distinguishability:** all constructors can be told apart using the memory type. → **splits**.
- **Coverage:** every part of the ADT appears within the memory type → **fragments**.
- **Fragment and branch coherence:** memory types in split branches and fragments agree with their respective subtypes.

Can we do better?

An optimized memory layout for Exp

```
Bin(Plus, Int(42), Var(...))
```

Naive memory representation

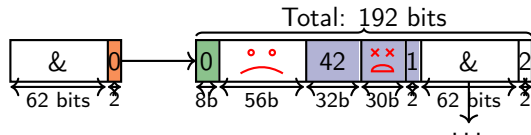


Waste of space!

An optimized memory layout for Exp

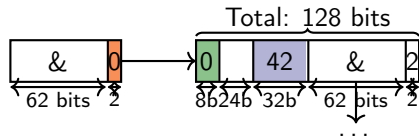
`Bin(Plus, Int(42), Var(...))`

Naive memory representation



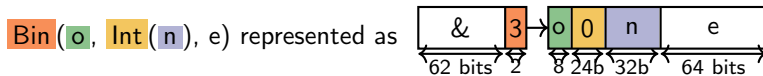
Waste of space!

Optimized memory representation

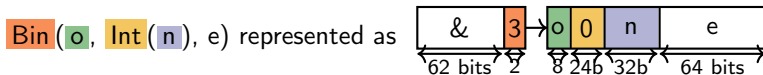


- Integer value **unboxed** from the `Int` constructor layout.
- Unboxed integer **inlined** in previously unused padding.

An optimized memory layout for Exp



An optimized memory layout for Exp



`enum Exp { Var(String), Int(i32), Bin(Op, Exp, Exp) } represented as`

`split .[0:2] {`

`| ...`

`| 0 from Bin(_, Bin_|Var(_), Bin_|Var(_)) => ...`

`| 3 from Bin(_, Int(_), _) | Bin(_, _, Int(_)) =>`

`&<64>(split .1 {`

`| 0 from Bin(_, Int(_), _) =>`

`{ { (.Bin.0 as Op), (0)<24>, (.Bin.0.Int as i32), (.Bin.1 as Exp) } }`

`| 1 from Bin(_, Bin_|Var(_), Int(_)) =>`

`{ { (.Bin.0 as Op), (1)<24>, (.Bin.1.Int as i32), (.Bin.0 as Exp) } }`

`} with [0:2] : (3)<2>`

`}`

What about the language?

Ribbit input language

- First-order functional language with **pattern matching**, **accessors** and **constructors**.
- Immutable, monomorphic ADTs.
- High-level elements are **layout-agnostic**.

Ribbit input language

- First-order functional language with **pattern matching**, **accessors** and **constructors**.
- Immutable, monomorphic ADTs.
- High-level elements are **layout-agnostic**.

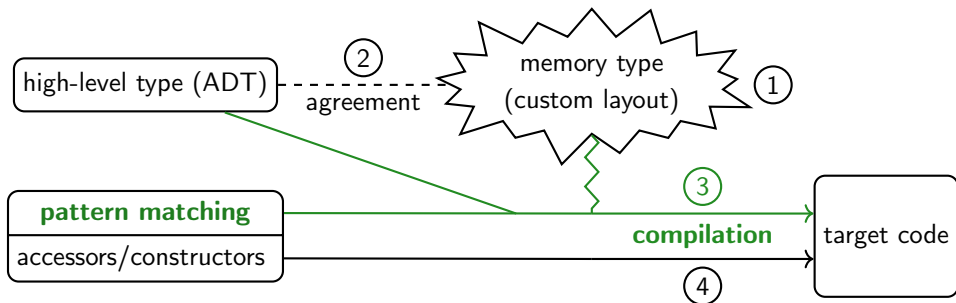
```
fn eval(e : Exp) -> Exp {  
  match e {  
    Int(_) | Var(_) => e,  
    Bin(op, e1, e2) => match (eval(e1), eval(e2)) {  
      Int(n1), Int(n2) => match op {  
        | Plus => Int(n1 + n2),  
        | Mult => Int(n1 * n2)  
      },  
      e1', e2' => Bin(op, e1', e2')  
    }  
  }  
}  
  
let exp = Bin(Plus, Int(1), Int(2));  
eval(exp)
```

Ribbit input language

- First-order functional language with **pattern matching**, **accessors** and **constructors**.
- Immutable, monomorphic ADTs.
- High-level elements are **layout-agnostic**.

```
fn eval(e : Exp) -> Exp {  
  match e {  
    Int(_) | Var(_) => e,  
    Bin(op, e1, e2) => match (eval(e1), eval(e2)) {  
      Int(n1), Int(n2) => match op {  
        | Plus => Int(n1 + n2),  
        | Mult => Int(n1 * n2)  
      },  
      e1', e2' => Bin(op, e1', e2')  
    }  
  }  
}  
  
let exp = Bin(Plus, Int(1), Int(2));  
eval(exp)
```

Compilation with custom memory layouts



Ribbit input language

```
fn eval(e : Exp) -> Exp {  
  match e {  
    Int(_) | Var(_) => e,  
    Bin(op, e1, e2) => match eval(e1), eval(e2) {  
      Int(n1), Int(n2) => match op {  
        | Plus => Int(n1 + n2),  
        | Mult => Int(n1 * n2)  
      },  
      e1', e2' => Bin(op, e1', e2')  
    }  
  }  
}  
  
let exp = Bin(Plus, Int(1), Int(2));  
eval(exp)
```

- High-level language constructs: **pattern matching**, **accessors** and **constructors**.
- Compiler emits equivalent layout-specific low-level code.

Ribbit input language

```
fn eval(e : Exp) -> Exp {  
  match e {  
    Int(_) | Var(_) => e,  
    Bin(_, _, _) =>  
      let op : Op = e.Bin.0; let e1 : Exp = e.Bin.1; let e2 : Exp = e.Bin.2;  
      let e1' : Exp = eval(e1); let e2' : Exp = eval(e2);  
      match e1', e2' {  
        Int(_), Int(_) =>  
          let n1 : i32 = e1'.Int; let n2 : i32 = e2'.Int;  
          match op { Plus => Int(n1 + n2), Mult => Int(n1 * n2) },  
        _, _ => Bin(op, e1', e2')  
      }  
    }  
}  
  
let exp = Bin(Plus, Int(1), Int(2));  
eval(exp)
```

- High-level language constructs: **pattern matching**, **accessors** and **constructors**.
- Compiler emits equivalent layout-specific low-level code.

Pattern matching compilation

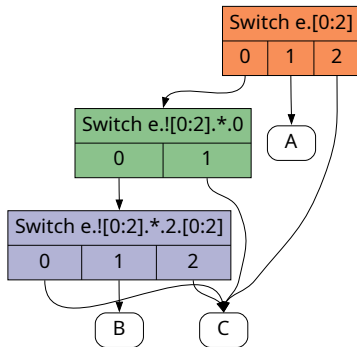
Input memory type

```
enum Exp { Var(String), Int(i32),  
          Bin(Op, Exp, Exp) }  
represented as split .[0:2] {  
  /* naive memory layout */  
}
```

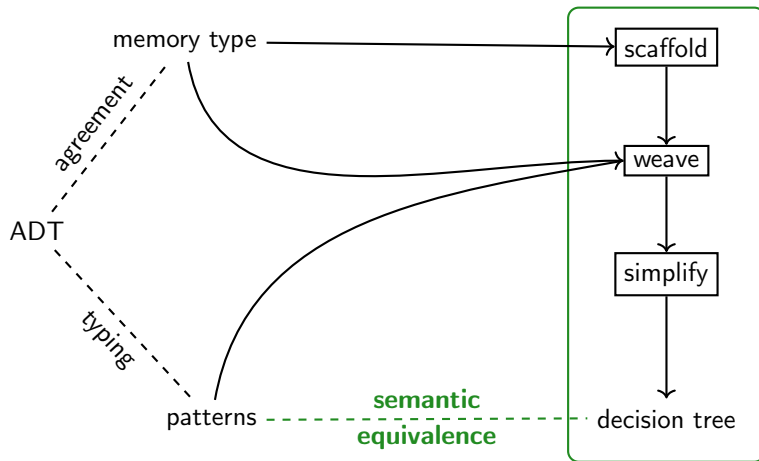
Input patterns

```
match e {  
  Int(_) => A,  
  Bin(Plus, Int(_), _) => B,  
  _ => C  
}
```

Output: layout-specific decision tree



Pattern matching compilation: overview



Pattern matching compilation: scaffolding and weaving

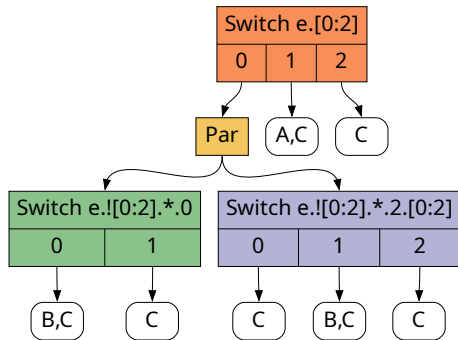
Memory type

```
split .[0:2] {  
  | 0 from Bin(_) => &<64>({{  
    (.Bin.0 as Op), _<56>,  
    (.Bin.1 as Exp), (.Bin.2 as Exp)  
  }}) with [0:2] : (0)<2>  
  | 1 from Int(_) => ...  
  | 2 from Var(_) => ... }  
}
```

Patterns

```
match e {  
  Int(_) => A,  
  Bin(Plus, Int(_), _) => B,  
  _ => C  
}
```

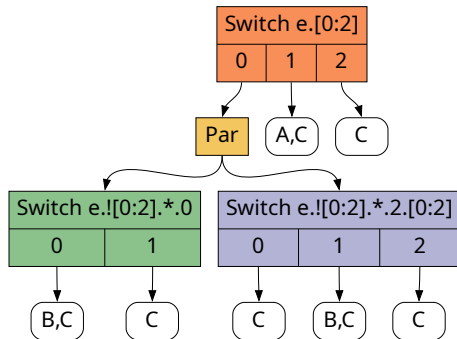
Memory tree



Pattern matching compilation: sequentialization

- non-deterministic choice
- use classic pattern matching compilation heuristics

Memory tree



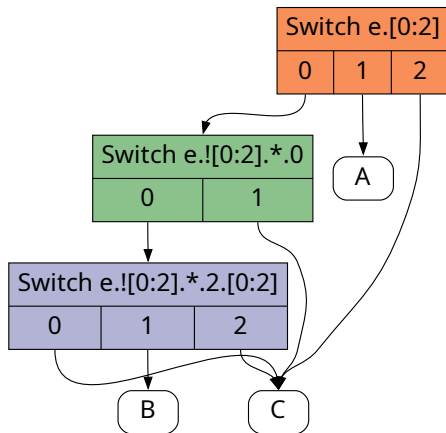
Pattern matching compilation: sequentialization

- non-deterministic choice
- use classic pattern matching compilation heuristics

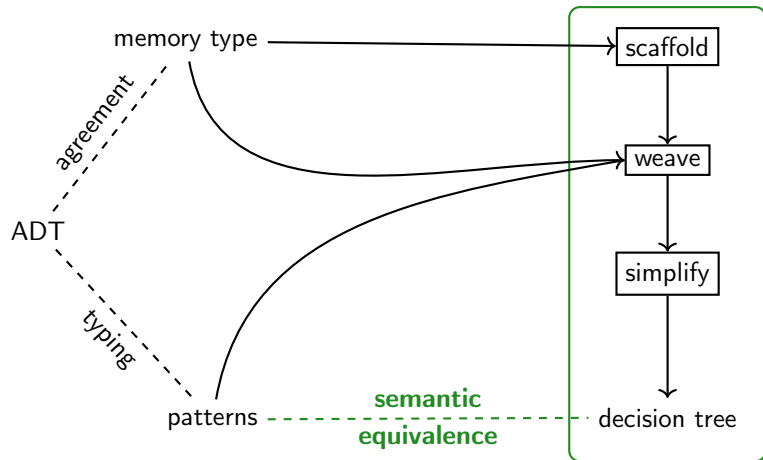
Input patterns

```
match e {  
  Int(_) => A,  
  Bin(Plus, Int(_), _) => B,  
  _ => C  
}
```

Final decision tree



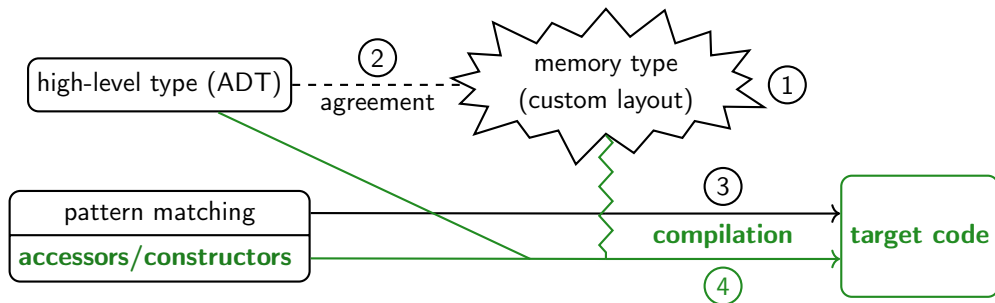
Pattern matching compilation: recap



Memory trees

- Capture dependencies
- Simplified into decision trees

Compilation of accessors and constructors



Compilation of accessors and constructors

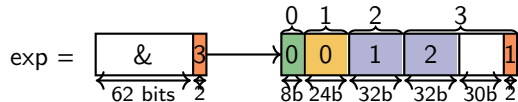
Input: desugared Ribbit program

```
fn eval(e : Exp) -> Exp {  
  match e {  
    Int(_) | Var(_) => e,  
    Bin(_, _, _) =>  
      let op : Op = e.Bin.0; let e1 : Exp = e.Bin.1; let e2 : Exp = e.Bin.2;  
      let e1' : Exp = eval(e1); let e2' : Exp = eval(e2);  
      match e1', e2' {  
        Int(_), Int(_) =>  
          let n1 : i32 = e1'.Int; let n2 : i32 = e2'.Int;  
          match op { Plus => Int(n1 + n2), Mult => Int(n1 * n2) },  
          _, _ => Bin(op, e1', e2')  
      }  
  }  
}  
  
let exp = Bin(Plus, Int(1), Int(2));  
eval(exp)
```

Let's try to execute it!

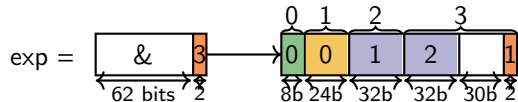
Expected behavior of target program (optimized layout)

```
let exp : Exp = Bin(Plus, Int(1), Int(2));
```



Expected behavior of target program (optimized layout)

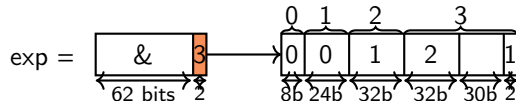
```
let exp : Exp = Bin(Plus, Int(1), Int(2));
```



```
eval(exp)
```

Expected behavior of target program (optimized layout)

```
let exp : Exp = Bin(Plus, Int(1), Int(2));
```

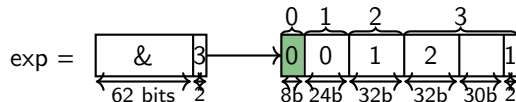


```
match e { Int(_) | Var(_) => e,
```

```
Bin( _, _, _ ) => // is this a binary operation?
```

Expected behavior of target program (optimized layout)

```
let exp : Exp = Bin(Plus, Int(1), Int(2));
```



```
op = exp.¬[0:2].*.0
```

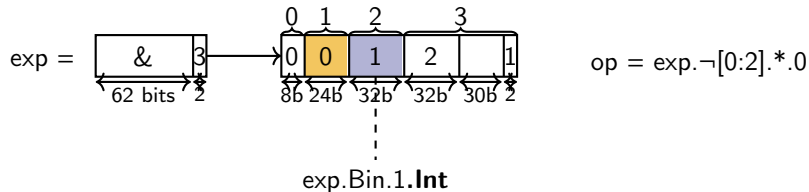
```
match e { Int(_) | Var(_) => e,
```

```
Bin(_, _, _) =>
```

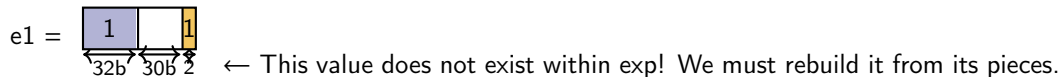
```
let op : Op = e.Bin.0; // extract binary operator
```

Expected behavior of target program (optimized layout)

```
let exp : Exp = Bin(Plus, Int(1), Int(2));
```



```
match e { Int(_) | Var(_) => e,  
Bin(_, _, _) =>  
let op : Op = e.Bin.0;  
let e1 : Exp = e.Bin.1; // extract first operand
```



Memorphisms: key points

- Complex memory layouts introduce unexpected complications
- Accessors sometimes require building a whole new value
- Compilation must emit low-level code to break down and rebuild memory contents
- Synthesize morphisms between different memory representations

→ complex compilation algorithm, relying on formal concepts from the Ribbitulus

Formal tidbits

Formal tidbit: the Ribbitulus

Source expressions

- Function calls, let-bindings, **pattern matching**

- **Pivot expressions** $(u : \tau \text{ as } \widehat{\tau})$

$(\text{Bin}(\text{Mult}, e.\text{Bin}.1, e.\text{Bin}.2) : \text{Exp} \text{ as } \widehat{\text{Exp}}_{\text{opt}})$

term

ADT

memory type

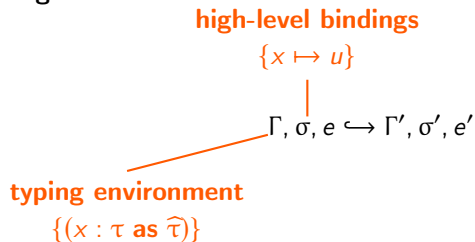
Formal tidbit: the Ribbitulus

Source expressions

- Function calls, let-bindings, **pattern matching**
- **Pivot expressions** $(u : \tau \text{ as } \widehat{\tau})$

High-level semantics

- Classic semantics on ML-like terms
- Pivots are normal forms



Formal tidbit: the Ribbitulus

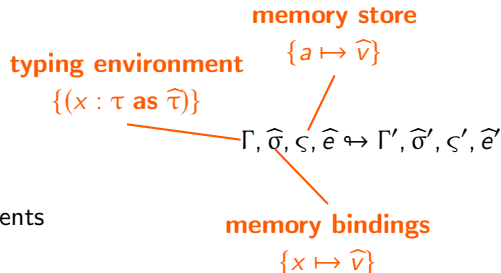
Source expressions

- Function calls, let-bindings, **pattern matching**
- **Pivot expressions** $(u : \tau \text{ as } \widehat{\tau})$

High-level semantics

Memory-level semantics

- Equipped with a memory model
- Pivots guide manipulation of memory contents



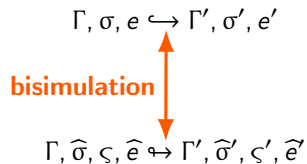
Formal tidbit: the Ribbitulus

Source expressions

- Function calls, let-bindings, **pattern matching**
- **Pivot expressions** $(u : \tau \text{ as } \widehat{\tau})$

High-level semantics

Memory-level semantics

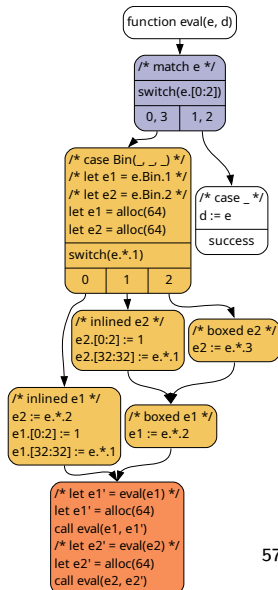


Target Intermediate Representation

- Just low-level enough for our needs
- Control-Flow Graph in **Destination-Passing Style**

Features

- Explicit **memory manipulation** instructions:
allocate, read, write
- Switch nodes supersede decision trees
- **Memory locations** $\approx$ unaligned pointers
 - Read-only locations: already-built values
 - Writable locations: building in progress
- Same memory model as source (memory-level) semantics



Compilation of pivot expressions

Compilation of pivot expressions

Key idea: go back and forth between two states.

- I am emitting code to build something in memory of shape $X \rightarrow$ **rebuild**
- I am looking for something in memory representing $Y \rightarrow$ **seek**

seek

rebuild

Compilation of pivot expressions

seek

rebuild

```
match e { Int(_) | Var(_) => e,  
Bin(_, _, _) =>  
let e1 = (e.Bin.1 : Exp as  $\widehat{\text{Exp}}_{\text{opt}}$ )
```

Existing value: e

- $(e : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}}) \in \Gamma$
- Of the form $\text{Bin}(_, _, _)$

Desired value: e1

$(e.\text{Bin}.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$

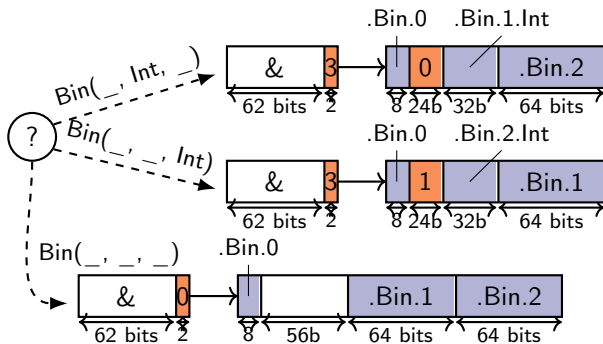
Compilation of pivot expressions

Seeking e.Bin.1... Position statically unknown!

seek

rebuild

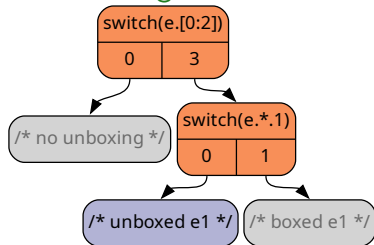
Existing value: e



Desired value: e1

(e.Bin.1 : Exp as $\widehat{\text{Exp}}_{\text{opt}}$)

Emitted target code



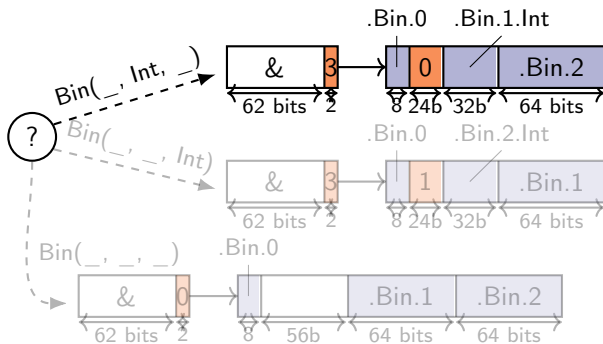
Compilation of pivot expressions

seek

rebuild

Seeking e.Bin.1... Position statically unknown!

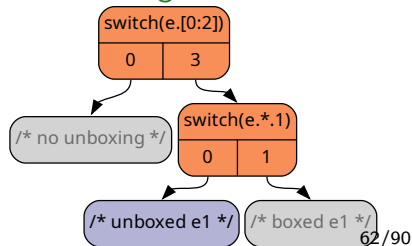
Existing value: e



Desired value: e1

- $(e.Bin.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$
- Of the form `Int(_)`

Emitted target code



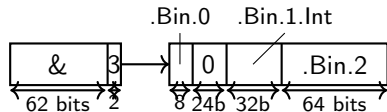
Compilation of pivot expressions

[seek](#)[rebuild](#)

Seeking e.Bin.1... Not found!

Existing value: e

- $(e : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}}) \in \Gamma$
- Of the form $\text{Bin}(_, \text{Int}(_), _)$



Desired value: e1

- $(e.\text{Bin}.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$
- Of the form $\text{Int}(_)$

Compilation of pivot expressions

seek

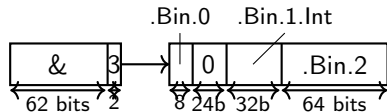
rebuild

Rebuilding e.Bin.1.

- **Constant parts:** allocate and initialize memory
- **Fragment:** I need (e.Bin.1.Int as i32)

Existing value: e

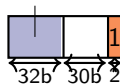
- $(e : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}}) \in \Gamma$
- Of the form $\text{Bin}(_, \text{Int}(_), _)$



Desired value: e1

- $(e.Bin.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$
- Of the form $\text{Int}(_)$

$e1.Int = e.Bin.1.Int$



Compilation of pivot expressions

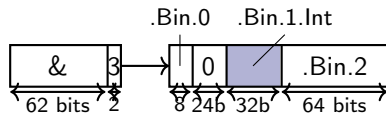
seek

rebuild

Seeking `e.Bin.1.Int`... Found at position `.[0:2].*.2` in memory!

Existing value: `e`

- $(e : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}}) \in \Gamma$
- Of the form `Bin(, Int(, ))`

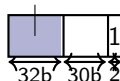


```
/* unboxed e1 */  
let e1 = alloc(64)  
e1.[0:2] := 1  
e1.[32:32] := e1.[0:2].*.2
```

Desired value: `e1`

- $(e.\text{Bin}.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$
- Of the form `Int(,)`

`e1.Int = e.Bin.1.Int`



Compilation of pivot expressions

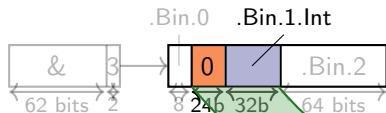
Emitted target code witnesses the memorphism.

seek

rebuild

Existing value: e

- $(e : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}}) \in \Gamma$
- Of the form $\text{Bin}(_, \text{Int}(_), _)$

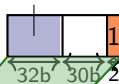


```
/* unboxed e1 */  
let e1 = alloc(64)  
e1.[0:2] := 1  
e1.[32:32] := e1.[0:2].*.2
```

Desired value: $e1$

- $(e.\text{Bin}.1 : \text{Exp as } \widehat{\text{Exp}}_{\text{opt}})$
- Of the form $\text{Int}(_)$

$e1.\text{Int} = e.\text{Bin}.1.\text{Int}$



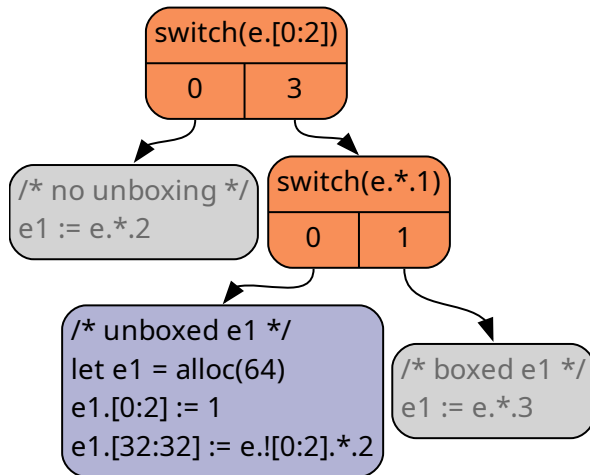
memorphism

Compilation of pivot expressions

Emitted target code witnesses the memorphism.

seek

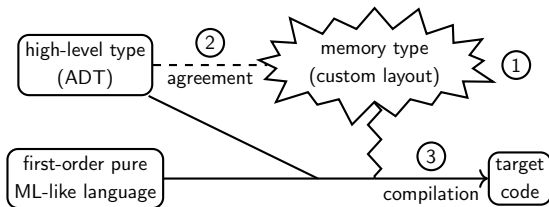
rebuild



Key points

- Compile pivot expressions to target in DPS
- Synthesize memorphisms between memory layouts
- Alternatively **seek** and **rebuild** values in memory
- Proof of correctness by weak simulation

Custom Memory Representation with Ribbit



- expressive DSL for memory layouts
- compilation to custom target IR
- memory-level semantics, soundness and correctness proofs

Developed during the thesis of **Thais Baudon**:

Algebraic Data Types for High-Performance Computing

<https://ribbit.gitlabpages.inria.fr/ribbit>

II. Tree Linearization

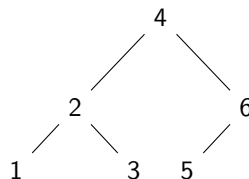
featuring Paul Iannetta



Par Ji-Elle - Travail personnel, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=10833582>

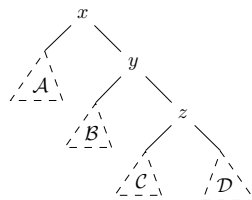
An aggressive version for AVLs: tarbres

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

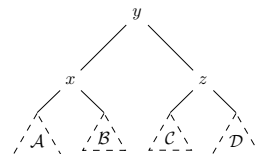


Operations:

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

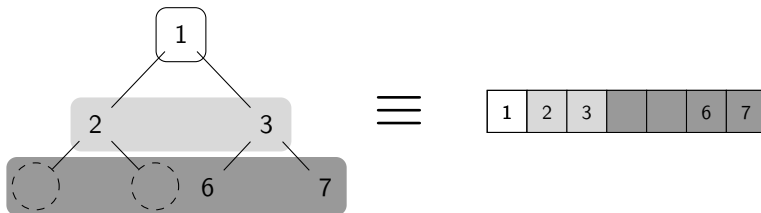


(a) A binary tree T



(b) T after L-rotation

Recall the breadth-first encoding: “Tarbres”



Some remarks:

- Cache oblivious community has designed many byzantine layouts
- Most famous: Van Em Boas Layout.

Question 2 (recall)

How to generate efficient code this linearized representation automatically?

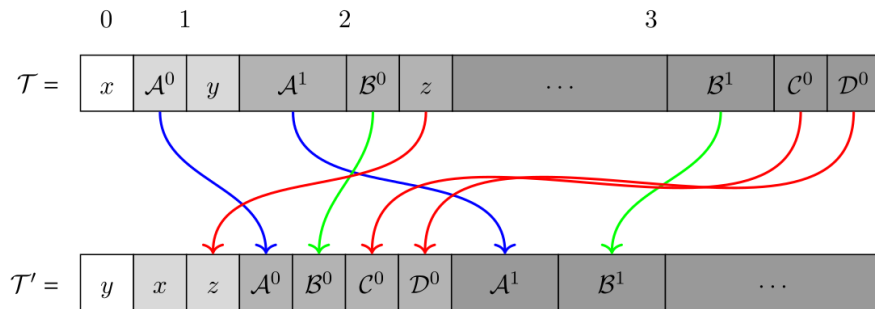
Our objective

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)

“Non intuitive” process:

- Our new rotations will be algorithmically **more costly**, but the cache. . .
- For databases applications: $\#reads \gg \#writes$.

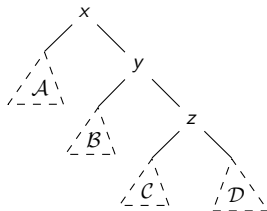
Rotations for Tarbres: not so easy



AVL Rebalancing as a sequence of elementary operations

Fact

Rotations can be expressed as a sequence of low-level operations that are simpler to optimize.

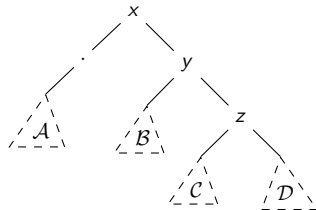


- Original Unbalanced State

AVL Rebalancing as a sequence of elementary operations

Fact

Rotations can be expressed as a sequence of low-level operations that are simpler to optimize.

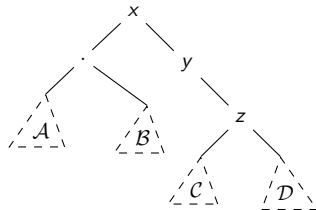


- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$

AVL Rebalancing as a sequence of elementary operations

Fact

Rotations can be expressed as a sequence of low-level operations that are simpler to optimize.

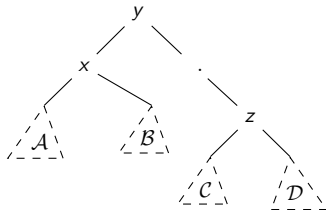


- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$
- Shift Left $\mathcal{B}$

AVL Rebalancing as a sequence of elementary operations

Fact

Rotations can be expressed as a sequence of low-level operations that are simpler to optimize.

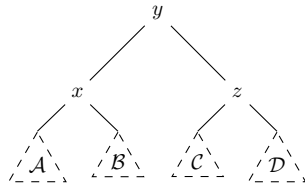


- Original Unbalanced State
- Pull Down (Left) $\mathcal{A}$
- Shift Left $\mathcal{B}$
- Rename

AVL Rebalancing as a sequence of elementary operations

Fact

Rotations can be expressed as a sequence of low-level operations that are simpler to optimize.



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

How to compile tarbres efficiently ?

How to compile tarbres efficiently ?
 $\simeq$ How to reshape ADTs shaped as arrays ?

Looking for efficient code

I have a **description** and **optimisation** problem:

- I want to (simply) describe operations on Tarbres;
- ...and compile them efficiently

Looking for efficient code

I have a **description** and **optimisation** problem:

- I want to (simply) describe operations on Tarbres;
- ...and compile them efficiently

I make a new language and
code generation !



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

REW is small DSL to:

- Algebraic Data Types without sharing
- Fixed (implicit) encoding as *tarbres*
- 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  
}
```

Compilation Steps

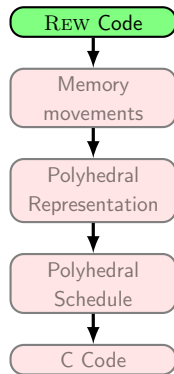
1. Give the specifications (describe the types & the transformations)
2. Figure out the movements
3. Compute the dependences between the movements
4. Reschedule the movements
5. Generate the code

Step 1: Describe Structural Transformations

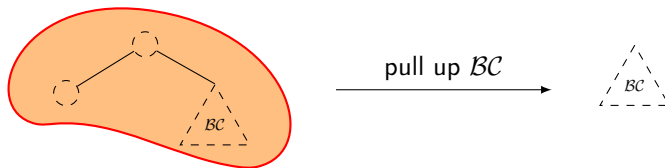


```
type tree = Empty | Node (tree,int,tree)
```

```
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  
}
```

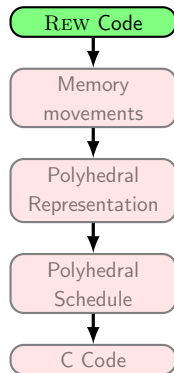


Step 1: Describe Structural Transformations

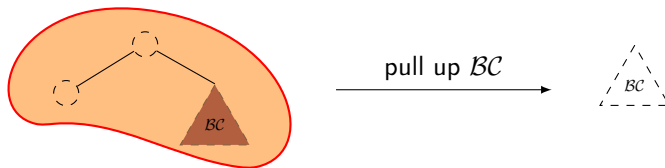


```
type tree = Empty | Node (tree,int,tree)
```

```
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  
}
```

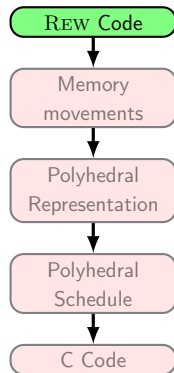


Step 1: Describe Structural Transformations

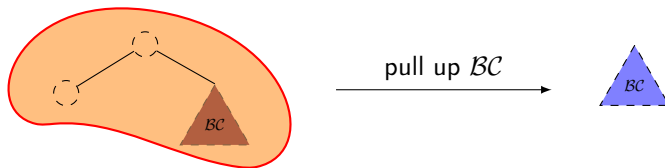


```
type tree = Empty | Node (tree,int,tree)
```

```
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  
}
```

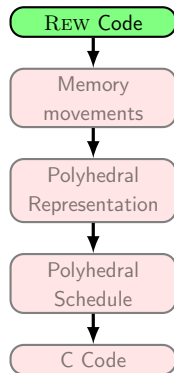


Step 1: Describe Structural Transformations



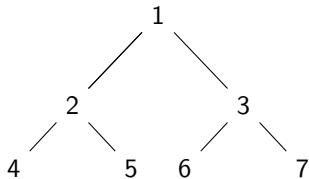
```
type tree = Empty | Node (tree,int,tree)
```

```
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  
}
```



A notation for paths

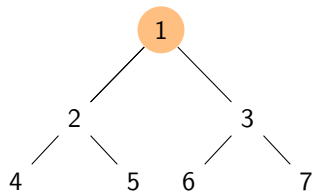
```
type tree = Empty | Node (tree,int,tree)  
                        .0  .1  .2
```



€

A notation for paths

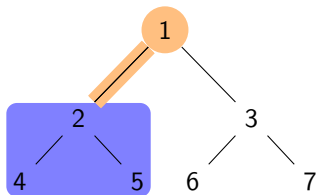
```
type tree = Empty | Node (tree, int, tree)  
                        .0  .1  .2
```



.1

A notation for paths

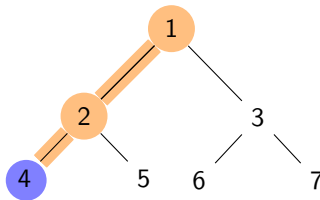
```
type tree = Empty | Node (tree, int, tree)  
                        .0  .1  .2
```



.0

A notation for paths

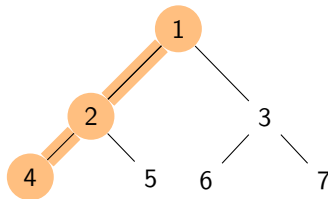
```
type tree = Empty | Node (tree, int, tree)  
                        .0  .1  .2
```



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

A notation for paths

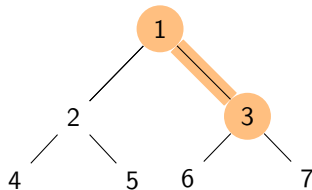
```
type tree = Empty | Node (tree,int,tree)  
                        .0  .1  .2
```



$(.0)^2.1$

A notation for paths

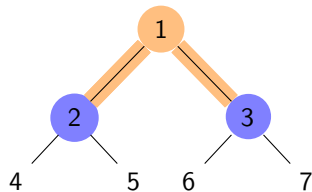
```
type tree = Empty | Node (tree, int, tree)  
                        .0  .1  .2
```



.2.1

A notation for paths

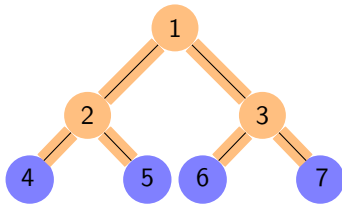
```
type tree = Empty | Node (tree,int,tree)  
                        .0  .1  .2
```



φ

A notation for paths

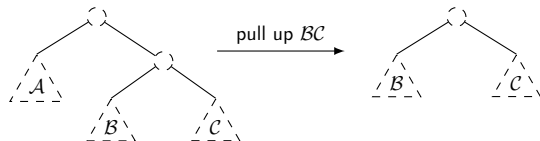
```
type tree = Empty | Node (tree, int, tree)  
                        .0  .1  .2
```



$\varphi.\varphi$

Step 2: Compute Subtree Movements

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



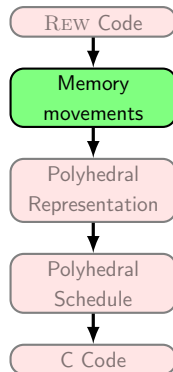
$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

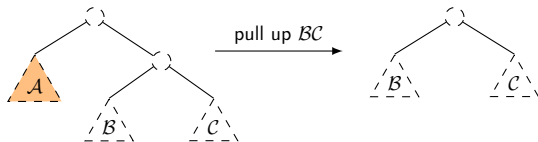
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$



Step 2: Compute Subtree Movements

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



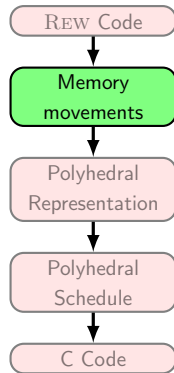
$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

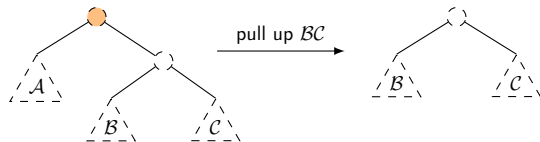
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$



Step 2: Compute Subtree Movements

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



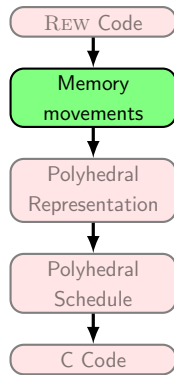
$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

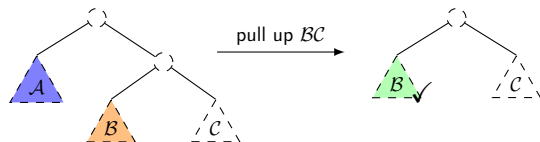
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$



Step 2: Compute Subtree Movements

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



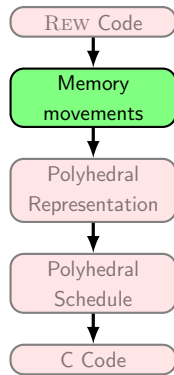
$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

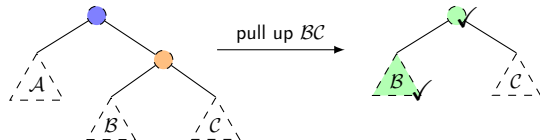
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$



Step 2: Compute Subtree Movements

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



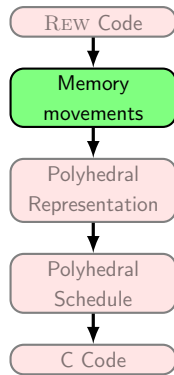
$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

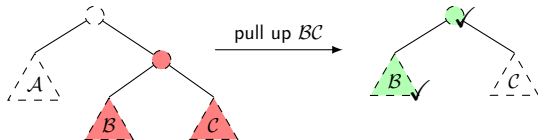
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$



Step 2: Compute Subtree Movements

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



$\langle a : \text{tree} \mid .0 \rightarrow \emptyset \rangle$

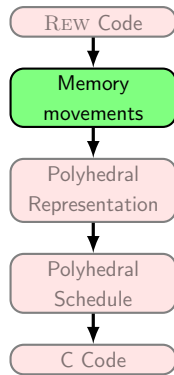
$\langle i : \text{int} \mid .1 \rightarrow \emptyset \rangle$

$\langle b : \text{tree} \mid .2.0 \rightarrow .0 \rangle$

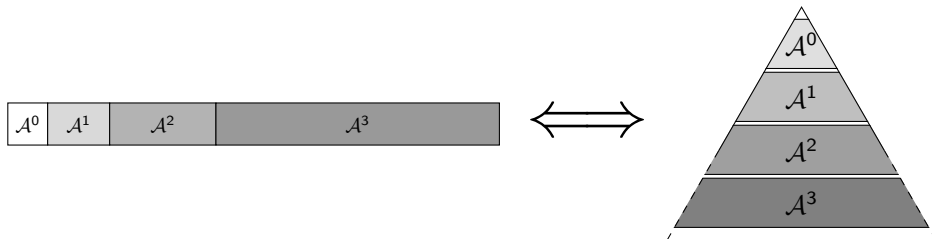
$\langle j : \text{int} \mid .2.1 \rightarrow .1 \rangle$

$\langle c : \text{tree} \mid .2.2 \rightarrow .2 \rangle$

Oups, there is a **conflict!**, why ?



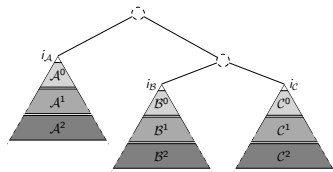
Step 2: Use Layers to Subdivide Memory Movements



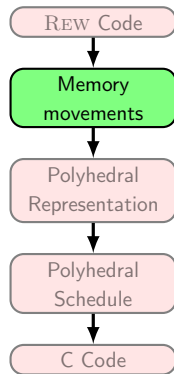
Remark : layer k at a certain position is denoted by $pos.\varphi^k$

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

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

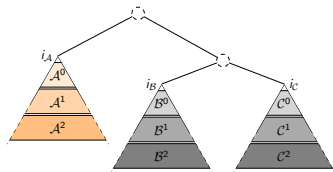


$$\begin{aligned}
 \langle .0 \rightarrow \emptyset \rangle &\rightsquigarrow \forall k_0, \langle .0.\varphi^{k_0} \rightarrow \emptyset \rangle \\
 \langle .1 \rightarrow \emptyset \rangle &\rightsquigarrow \langle .1 \rightarrow \emptyset \rangle \\
 \langle .2.0 \rightarrow .0 \rangle &\rightsquigarrow \forall k_1, \langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \\
 \langle .2.1 \rightarrow .1 \rangle &\rightsquigarrow \langle .2.1 \rightarrow .1 \rangle \\
 \langle .2.2 \rightarrow .2 \rangle &\rightsquigarrow ?
 \end{aligned}$$

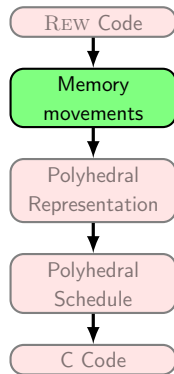


Step 2: Layer-aware movements (1/2)

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

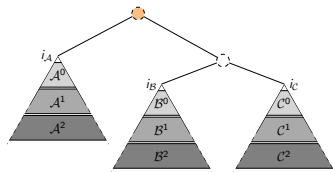


$$\begin{aligned}
 \langle .0 \rightarrow \emptyset \rangle &\rightsquigarrow \forall k_0, \langle .0.\varphi^{k_0} \rightarrow \emptyset \rangle \\
 \langle .1 \rightarrow \emptyset \rangle &\rightsquigarrow \langle .1 \rightarrow \emptyset \rangle \\
 \langle .2.0 \rightarrow .0 \rangle &\rightsquigarrow \forall k_1, \langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \\
 \langle .2.1 \rightarrow .1 \rangle &\rightsquigarrow \langle .2.1 \rightarrow .1 \rangle \\
 \langle .2.2 \rightarrow .2 \rangle &\rightsquigarrow ?
 \end{aligned}$$

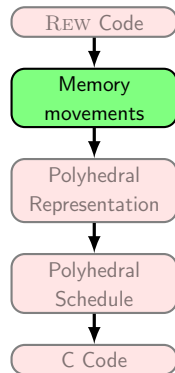


Step 2: Layer-aware movements (1/2)

$\text{Node}(a, \mathbf{i}, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c)$

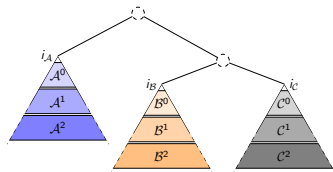


$$\begin{aligned}
 \langle .0 \rightarrow \emptyset \rangle &\rightsquigarrow \forall k_0, \langle .0.\varphi^{k_0} \rightarrow \emptyset \rangle \\
 \langle \mathbf{.1} \rightarrow \mathbf{\emptyset} \rangle &\rightsquigarrow \langle \mathbf{.1} \rightarrow \mathbf{\emptyset} \rangle \\
 \langle .2.0 \rightarrow .0 \rangle &\rightsquigarrow \forall k_1, \langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \\
 \langle .2.1 \rightarrow .1 \rangle &\rightsquigarrow \langle .2.1 \rightarrow .1 \rangle \\
 \langle .2.2 \rightarrow .2 \rangle &\rightsquigarrow ?
 \end{aligned}$$



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

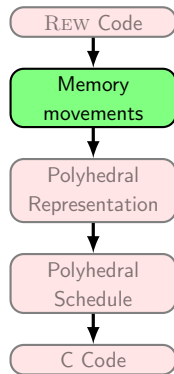
$\text{Node}(a, i, \text{Node}(b, j, c)) \rightarrow \text{Node}(b, j, c)$



$$\begin{aligned}
 (.0 \rightarrow \emptyset) &\rightsquigarrow \forall k_0, (.0.\varphi^{k_0} \rightarrow \emptyset) \\
 (.1 \rightarrow \emptyset) &\rightsquigarrow (.1 \rightarrow \emptyset) \\
 (.2.0 \rightarrow .0) &\rightsquigarrow \forall k_1, (.2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1}) \\
 (.2.1 \rightarrow .1) &\rightsquigarrow (.2.1 \rightarrow .1) \\
 (.2.2 \rightarrow .2) &\rightsquigarrow ?
 \end{aligned}$$

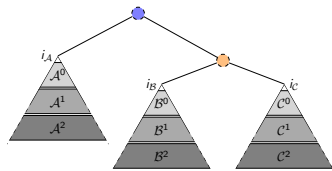
Important remark:

$(b) : k_1 : (.2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1})$ is in fact a memcopy/memmv (of layer number k_1 of .2.0 into .0, layer k_1).

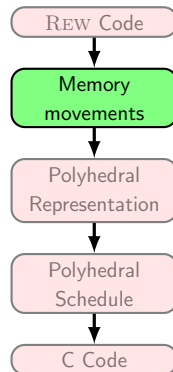


Step 2: Layer-aware movements (1/2)

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

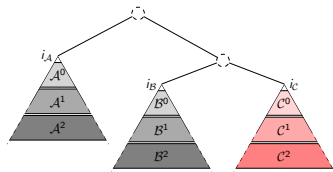


$$\begin{aligned}
 \langle .0 \rightarrow \emptyset \rangle &\rightsquigarrow \forall k_0, \langle .0.\varphi^{k_0} \rightarrow \emptyset \rangle \\
 \langle .1 \rightarrow \emptyset \rangle &\rightsquigarrow \langle .1 \rightarrow \emptyset \rangle \\
 \langle .2.0 \rightarrow .0 \rangle &\rightsquigarrow \forall k_1, \langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \\
 \langle .2.1 \rightarrow .1 \rangle &\rightsquigarrow \langle .2.1 \rightarrow .1 \rangle \\
 \langle .2.2 \rightarrow .2 \rangle &\rightsquigarrow ?
 \end{aligned}$$

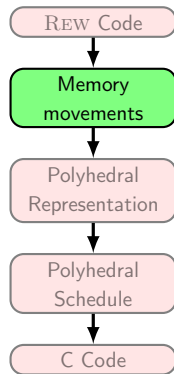


Step 2: Layer-aware movements (1/2)

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

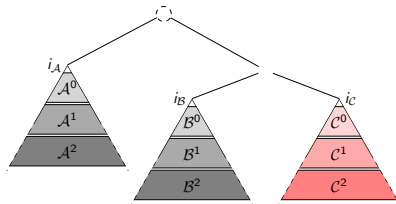


$$\begin{aligned}
 \langle .0 \rightarrow \emptyset \rangle &\rightsquigarrow \forall k_0, \langle .0.\varphi^{k_0} \rightarrow \emptyset \rangle \\
 \langle .1 \rightarrow \emptyset \rangle &\rightsquigarrow \langle .1 \rightarrow \emptyset \rangle \\
 \langle .2.0 \rightarrow .0 \rangle &\rightsquigarrow \forall k_1, \langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \\
 \langle .2.1 \rightarrow .1 \rangle &\rightsquigarrow \langle .2.1 \rightarrow .1 \rangle \\
 \langle .2.2 \rightarrow .2 \rangle &\rightsquigarrow ?
 \end{aligned}$$

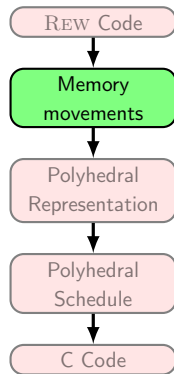


Step 2: Layer-aware movements (2/2)

$\text{Node}(a, i, \text{Node}(b, j, \mathbf{c})) \rightarrow \text{Node}(b, j, c)$

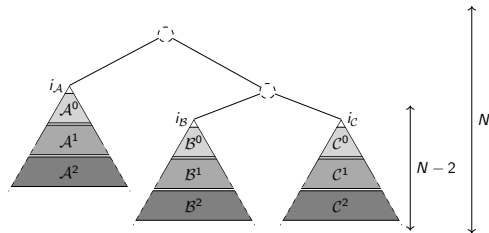
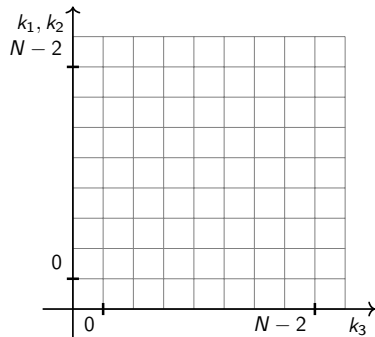


$$\langle .2.2 \rightarrow .2 \rangle \rightsquigarrow \begin{cases} \langle .2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1 \rangle & (c_1) \\ \langle .2.2.2^{k_2}.0.\varphi^{k_3} \rightarrow .2.2^{k_2}.0.\varphi^{k_3} \rangle & (c_0) \end{cases}$$

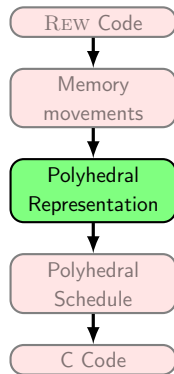


There are still conflicts, but computations can be done “layer by layer”.

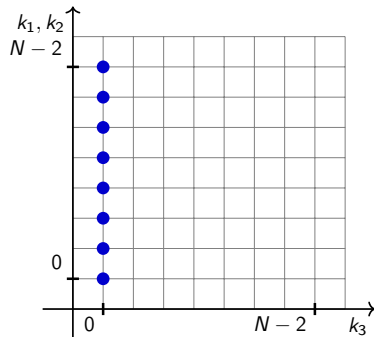
Step 3: Characterizing Memory Movements



A point is a memcopy/memmv
(of a layer)

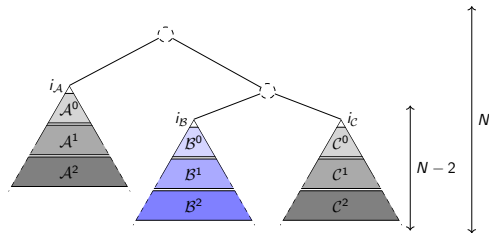


Step 3: Characterizing Memory Movements

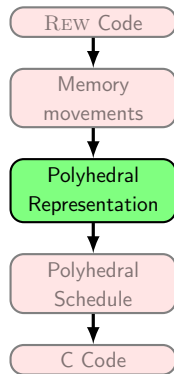


A point is a memcopy/memmv
(of a layer)

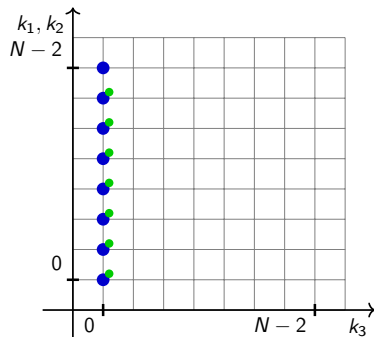
$(\mid .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \mid)$



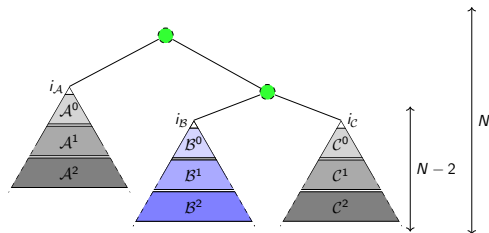
(b)



Step 3: Characterizing Memory Movements

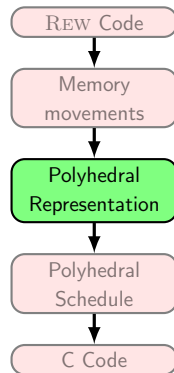


A point is a memcopy/memmv
(of a layer)

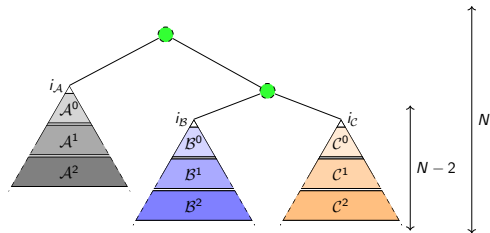
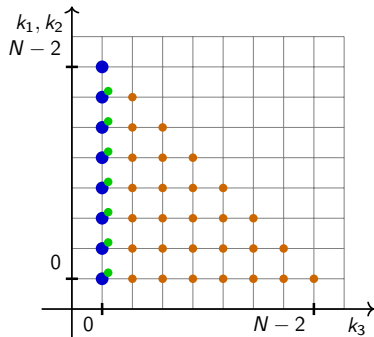


$$\langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \quad (b)$$

$$\langle .2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1 \rangle \quad (c1)$$



Step 3: Characterizing Memory Movements

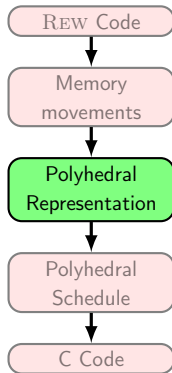


A point is a memcopy/memmv
(of a layer)

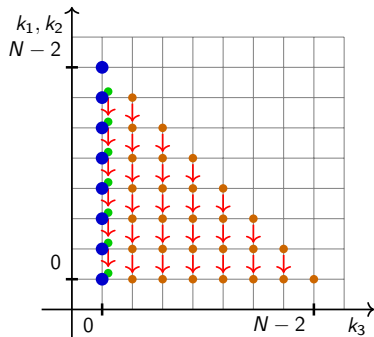
$$\langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle \quad (b)$$

$$\langle .2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1 \rangle \quad (c1)$$

$$\langle .2.2.2^{k_2}.0.\varphi^{k_3} \rightarrow .2.2^{k_2}.0.\varphi^{k_3} \rangle \quad (c0)$$

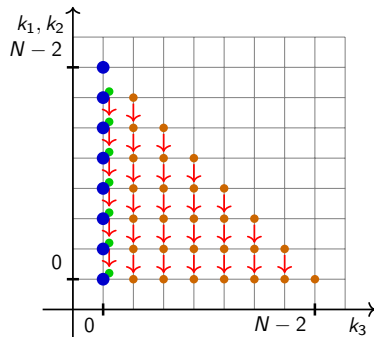


Step 3: Polyhedral Like Dependencies: Intuition



Let's take the rule $(c_0): \langle .2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1 \rangle$

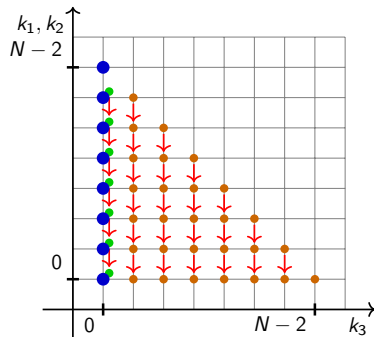
Step 3: Polyhedral Like Dependencies: Intuition



Let's take the rule $(c_0): \langle (.2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1) \rangle$

- When $k_2 = 1$, $(c_0): \langle (.2)^3 .1 \rightarrow (.2)^2 .1 \rangle$
- When $k_2 = 2$, $(c_0): \langle (.2)^4 .1 \rightarrow (.2)^3 .1 \rangle$

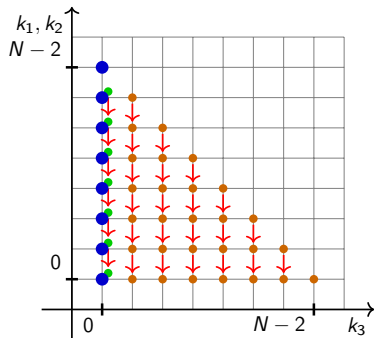
Step 3: Polyhedral Like Dependencies: Intuition



Let's take the rule $(c_0): \langle (.2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1) \rangle$

- When $k_2 = 1$, $(c_0): \langle (.2)^3 .1 \rightarrow (.2)^2 .1 \rangle$
- When $k_2 = 2$, $(c_0): \langle (.2)^4 .1 \rightarrow (.2)^3 .1 \rangle$

Step 3: Polyhedral Like Dependencies: Intuition



Let's take the rule $(c_0): \langle (.2.2.2^{k_2}.1 \rightarrow .2.2^{k_2}.1) \rangle$

- When $k_2 = 1$, $(c_0): \langle (.2)^3 .1 \rightarrow (.2)^2 .1 \rangle$
- When $k_2 = 2$, $(c_0): \langle (.2)^4 .1 \rightarrow (.2)^3 .1 \rangle$
- More generally, $(c_0)(k)$ and $(c_0)(k+1)$ conflict.

Step 3 : Computing the Dependencies

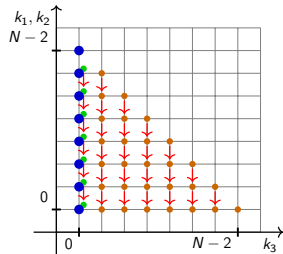
Theorem

Let $\langle s_1 \rightarrow d_1 \rangle$ and $\langle s_2 \rightarrow d_2 \rangle$ be two moves (not necessarily distinct). Consider d_1 and s_2 as regular expressions.

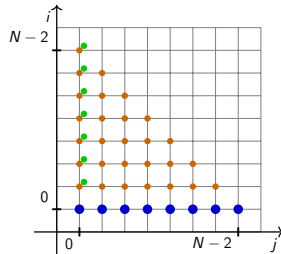
Then,

$\mathcal{L}(d_1) \cap \mathcal{L}(s_2)$ is the conflict space and it can be characterized by **computable** affine inequalities.

Step 4: Polyhedral Model Based Scheduling



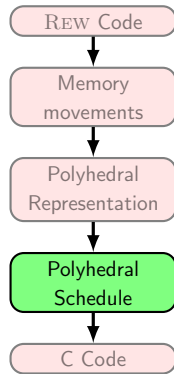
Scheduling
($\simeq$ ILP)



The output of this phase is a schedule:

each computation is given a **multidimensional timestamp** (i, j) .

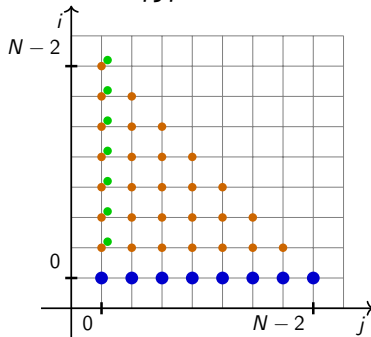
Ex: the memcopy/memmv: (b) : $\langle .2.0.\varphi^{k_1} \rightarrow .0.\varphi^{k_1} \rangle$ is done at date $(0, k_1)$.



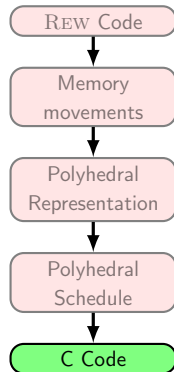
Step 5: From Schedule to Code

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

```
for (j = 0 ; j <= N-2 ; j += 1)
   $\llbracket .2.0.\varphi^j \rightarrow .0.\varphi^j \rrbracket$  // b
for (i = 1 ; i <= N-2 ; i += 1)
   $\llbracket (.2)^{i+1}.1 \rightarrow (.2)^i.1 \rrbracket$  //c1
  for (j = 0 ; j <= N - i - 2 ; j += 1)
     $\llbracket (.2)^{i+1}.0.\varphi^j \rightarrow (.2)^i.0.\varphi^j \rrbracket$  //c0
```



Application of polyhedral code generation: nested for loops **with memcopy**



Summary of the second contribution

- A custom layout can expose parallelization/cache-friendly optimisation opportunities **as regularities**.
- Partial implementation as a small library
- Extension of the polyhedral techniques.

From the ADT point of view:

- A fixed layout (no custom specification of it)
- But more aggressive structural transformations.

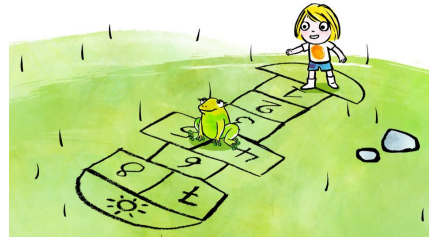
Developed during the thesis of **Paul Iannetta**:

Compiling Trees : Combining Data Layouts and the
Polyhedral Model

Let us conclude.

Future work

- An equational formulation of the compilation algorithms of Ribbit
- A meta-theory based on initial algebras.
- Ribbit in Rust



(cc "Petit Malabar" de Patrick VOLVE)

And a fusion of the two approaches: Ribbit with arrays and linearization.

Backup slides

Performance impact of memory representation

Synthetic red-black tree rebalancing benchmark in the (first) Ribbit prototype.

memory layout	avg. pattern matching exec. time	size of memory structure
OCaml-like	5.45 ns	156×64 bits
Linux-like	4.54 ns	94×64 bits

Optimized memory layouts can:

- Improve memory footprint
- Reduce costly operations (e.g., pointer dereference) → compile pattern matching and other expressions to efficient low-level code

Cf <https://hal.science/hal-04165615v1>

Implementation AVL as Tarbres

- 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/>

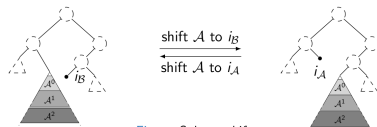


Figure: Subtree shifts

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

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

