Type declaration example: red-black trees (RBT)

```plaintext
Color = Red & Black
RBT = Empty + Node (Color, i32, &RBT, &RBT)
```

- Type-safety, non-redundancy and exhaustivity checks.
- Concise expression of complex functions through pattern-matching.

Pattern matching on RBTs: example

```plaintext
match c, v, t1, t2 {
    Black, z, Node(Red, y, Node(Red, x, a, b, c), d) | Black, z, Node(Red, x, a, Node(Red, y, b, c)), d |
    Black, x, a, Node(Red, z, Node(Red, y, b, c), d) | Black, x, a, Node(Red, y, b, Node(Red, z, c, d)),
    τ => Node(Red, y, Node(Black, x, a, b), Node(Black, z, c, d)), a, b, c, d => Node(a, b, c, d),
}
```

- Available in different kinds of languages: OCaml, Haskell, Scala, Rust...
- Yet complex structures are used in HPC without ADTs!

Memory representations

- Pattern matching compilation is representation-dependent.
- Current compilation approaches mandate a fixed representation.
- HPC requires efficient, customizable memory representations.

Internal representation of red-black trees (Repr_reinit)

Example for Node(1515, &...).

OCaml representation: uniform and composable but inefficient.

<table>
<thead>
<tr>
<th>&amp;</th>
<th>Node</th>
<th>Red</th>
<th>1515</th>
<th>&amp;</th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>64 bits</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Rust representation: more compact, less indirect but less uniform.

<table>
<thead>
<tr>
<th>Node</th>
<th>Red</th>
<th>1515</th>
<th>&amp;</th>
<th>&amp;</th>
<th>Empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bits</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Specify efficient memory representations for ADTs.
- Compile and optimize pattern matching for such representations.

Implementation: ribbit

2.5k LoC in OCaml.

- Input pattern language → evaluation, compilation.
- Compiles to decision trees (switches on memory values).
- Defining a new representation typically takes 50 LoC.

Based on state-of-the-art pattern matching compilation.

To compile this matching (cf. steps on the right), we use:

- Knit(n) emits code that rebuilds the representation of a subterm from its position in the parent term.
- Frog(n)(Knit)(h) emits code that destructs a memory value and branches to its associated decision tree.

Finally, we obtain the decision tree below.

Produced code for the example

```plaintext
switch(head constructor of v0) {
    case None : 0;
    case Some :
        switch(head constructor of Some(C)) [
            case A : 0;
            case B : 1;
            case C : 2 + i32 representation of Some(C)(v0);
        ]
}
```