Knit&Frog: Pattern matching compilation for custom memory representations (doctoral session)

Thaïs Baudon[1], Gabriel Radanne[2], and Laure Gonnord[3]

[1]ENS Lyon/LIP
[2]Inria/LIP
[3]Grenoble-INP/LCIS & LIP

Abstract

Initially present only in functional languages such as OCaml and Haskell, Algebraic Data Types have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions though *pattern-matching*. Numerous approaches have been designed to compile rich pattern matching to cleverly designed, efficient decision trees. However, these approaches are specific to a choice of *internal memory representation* which must accommodate garbage-collection and polymorphism.

ADTs now appear in languages more liberal in their memory representation such as Rust. Notably, Rust is now introducing more and more optimizations of the memory layout of Algebraic Data Types. As memory representation and compilation are interdependent, it raises the question of pattern matching compilation in the presence of non-regular, potentially customized, memory layouts.

In this article, we present Knit&Frog, a framework to compile pattern-matching for monomorphic ADTs, *parametrized* by an arbitrary memory representation. We propose a novel way to describe choices of memory representation along with a validity condition under which we prove the correctness of our compilation scheme. The approach is implemented in a prototype tool `ribbit`.

1 Introduction

Algebraic Data Types (ADTs) are an essential tool to model data and information. They allow to group together information in a consistent way through the use of records, also called product types, and to organize options through the use of variants, also called sum types. Product and sum types together allow to enforce invariants present in the domain under study and provide convenient tools to inspect these data-types through pattern matching.

Combined, these features offers numerous advantages:

- Model data in a way that is close to the programmer's intuition, abstracting away the details of the memory representation of said data.
- Safely handle data by ensuring via pattern-matching that its manipulation is well-typed, exhaustive and non-redundant.
- Optimize manipulation of data thanks to rich constructs understood by the compiler.

Despite these promises, Algebraic Data Types were initially only present in functional programming languages such as OCaml and Haskell. Recently, they have gained a foothold in more mainstream languages such as Typescript, Scala, Rust and even soon Java. They are however still lacking in high-performance lower level languages. One difficulty to language designers who want to add pattern matching to their language is that compiling a rich pattern matching language to efficient code is a non-trivial task, which is not commonly available in shared compiler frameworks such as LLVM. Indeed,

such frameworks only provide optimizations for C-like switches on integers (or integer-like enumerations). Additionally, existing works on pattern matching (Maranget, 2008; Wadler, 1987; Sestoft, 1996) provide very efficient compilation schemes, but are geared towards memory representations found in GC-managed functional languages such as OCaml and Haskell: uniform representations with liberal usage of boxing. Highly non-uniform data representations such as the ones found in C++ do not easily fit.

More generally, the descriptive nature of ADTs should enable compilers to aggressively optimize the representation of terms. The simplest example is the `Option` type, which is either `Some value` or `None`. If the value in question is an integer from 0 to 10, `None` can easily be represented as 11. This optimization is regularly done by programmers manually, forgoing the guarantees provided by languages. More complex optimizations on nested and rich data-types are even more error-prone. These transformations could easily be done automatically by the compiler. This trove of optimization potential has been brushed on in recent versions of Rust, but remains largely unexplored.

Finally, on top of the previously mentioned difficulty of adapting pattern matching compilation schemes to irregular memory representations, the correctness of such compilation schemes is also delicate to establish when the terms can be arbitrarily shaped.

The wider goal of this thesis is to enable the use of ADTs in high-performance applications. This requires optimized, possibly custom memory representations for ADTs, either constructed automatically or user-provided. However, current pattern matching compilation approaches mandate a fixed representation (often designed for high-level, garbage-collected languages) and thus lack the required flexibility. In addition, a memory-aware pattern matching compilation approach could enable further performance gains by tweaking the decision tree according to individual representation quirks. This requires a more flexible pattern matching compilation approach, able to handle various memory representations.

In this article, we lay the groundwork for highly optimized pattern matching in any language with arbitrary non-uniform memory representation. We present Knit&Frog, a compilation framework for rich pattern languages with arbitrary memory representation of monomorphic terms:

- We provide a characterization of memory representations for terms of Algebraic Data Types as two main operations: **Knit** which builds a term into its memory representation, and **Frog** which deconstructs the memory representation to identify the underlying term.
- We provide a compilation scheme parameterized by the memory representation. This scheme is adapted from Maranget (2008), the best implementation of pattern matching known thus far.
- We provide a sufficient condition for a memory representation to yield itself to pattern matching and prove end-to-end correctness of our compilation scheme in this case.

We first give some motivation examples of memory representations and pattern-matching compilation as motivation, before hinting at the scientific content of our presentation.

2 Algebraic Data Types and their memory representation

Let us now explore the different memory representation choices for Algebraic Data Types (ADTs, for short), and how their values are manipulated. In this section, we present various examples of types and programs and discuss how they are currently represented in existing languages, and how they might be. For illustration, we will look at the output of two languages which implement ADTs: Rust and OCaml. While these languages have some common lineage, they have a very different attitude towards code emission: OCaml is a GC-managed language which factors predictability and regularity. Rust on the other hand favors performance and absolute control over low-level details. These differences result in drastically different choices in memory representation, hence the need for a sophisticated pattern matching compiler that is flexible enough to take these differences into account. To illustrate this, let's consider an example of code manipulating *red-black trees*.

```
1 //Colors for the Red-Black Trees
2 enum Color { Red, Black, }
3
4 //Red-Black Trees of content T
5 enum RBT<T> {
6   Node (Color, T, &RBT<T>, &RBT<T>),
7   Empty,
8 }
```

```
1 match c, v, t1, t2 {
2   Black, z, &Node(Red, y, &Node(Red, x, a, b), c), d
3   | Black, z, &Node(Red, x, a, &Node(Red, y, b, c)), d
4   | Black, x, a, &Node(Red, z, &Node(Red, y, b, c), d)
5   | Black, x, a, &Node(Red, y, b, &Node(Red, z, c, d))
6   => Node(Red, y, &Node(Black, x, a, b), &Node(Black, z, c, d)),
7   a, b, c, d => Node (a, b, c, d),
8 }
```

Figure 1: Example of Red-Black tree and their balancing operation in Rust

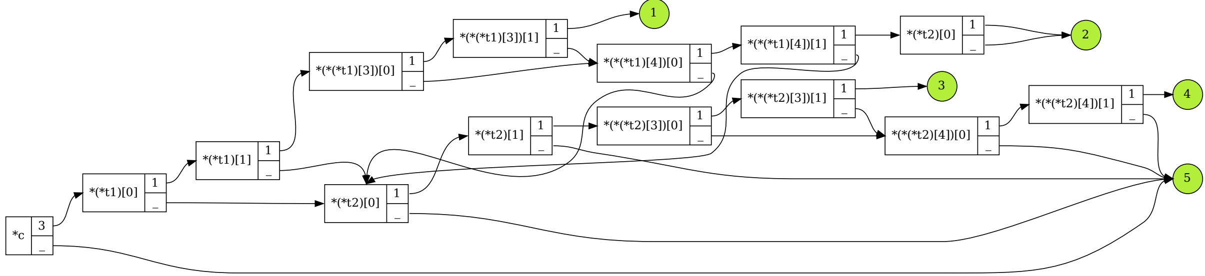

Figure 2: Output of our `ribbit` compiler for the code of Fig. 1

This output is a (decision tree) whose root is on the left. Decision (square) nodes compute values from their input (low-level representation), then branch depending on this value. Round nodes denote the final returned value of the procedure (the numbering of the matched branch).

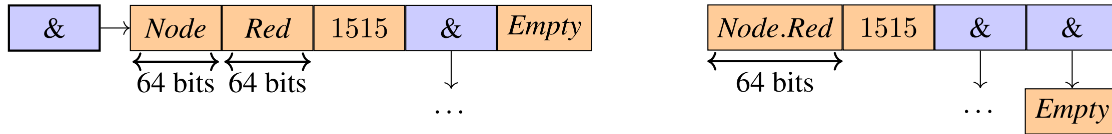

Figure 3: OCaml and Rust memory representations of $\mathcal{T}$=`Node(Red, 1515, &..., &Empty)`

Such optimized decision trees should still respect, and even take advantage of, the memory representation. A Rust version is depicted in Fig. 1. This type definition expresses trees as recursive data structures, and uses a tuple for the various fields in the `Node` case. For instance, $\mathcal{T}$=`Node(Red, 1515, &Node(Black, 42, &Empty,&Empty), &Empty)` is a tree of type `RBT<Int>`.

Red-Black trees famously rely on a fairly complex balancing step, which redistributes the colors depending on the internal invariant of the data structure. Thanks to nested pattern matching, this step can be expressed very compactly, as shown in Fig. 1. This pattern matching inspects four arguments at the same time: the current color `c`, the current value `v` and the sub-trees `t1` and `t2`.

Since this pattern matching is at the core of a performance-sensitive data structure, we naturally want it to be as efficient as possible. This is why many pattern matching implementations come with clever heuristics and techniques to output optimized decision trees (Kosarev u. a., 2020; Maranget, 2008; Sestoft, 1996). The resulting code is highly non-trivial, as can be seen in Fig. 2.

The OCaml-equivalent type of `RBT<u64>` would use 6 memory words per `Node`, as illustrated in Fig. 3. However, type `RBT<u64>` is represented by the Rust compiler using only 4 memory words per `Node`: a node is then made of a word with a bit marking it as non-0 (to avoid confusion with `None`) and the color bit, followed by the 64-bit integer and the two pointers. It is however possible to be even more compact: the Linux kernel uses a hand-crafted representation of Red-Black trees that exploits bit-stealing to use one less word (Wilke, 2016): since pointers are word-aligned, colors can be stored in the lower bits of each pointer. These complex manually-tuned optimizations are common in low-level C programming, but come at a high cost: programmers forgo the use of modern safer constructs such as Algebraic Data Types and pattern matching. Furthermore, they now need to design the decision tree themselves, and make sure their choice of representation contains enough information to distinguish, for instance, between the

`Empty` and `Node` cases. Moreover, the emission of code for such mangled objects in memory is delicate (notably, the bit-stealing technique in question is undefined behavior in the C standard).

Ideally, we would like to still use high-level pattern languages, while the compilation process adapts and exploits the optimized representation. Such optimized representation would be found automatically by the compiler, or specified by the programmer, transparently from the rest of the code which only mentions constructors of the Algebraic Data Type. We make a first step towards this goal by developing Knit&Frog, a pattern matching compilation technique **parameterized by the memory representation**.

3 Complete article

In the complete paper, we make the following contributions:

1. a source pattern language featuring variables and or-patterns and a target decision tree language with switches on memory expressions that include pointer dereferencing operations and array accesses;
2. a formal definition of memory representations consisting of three ingredients: a mapping $\text{Repr}^\bullet$ from values to memory values; a function $\text{Knit}^\bullet$ that rebuilds the type-appropriate representation of a subterm from the memory representation of a parent term and a function $\text{Frog}^\bullet$ that decodes a memory value to discriminate between different variants of a type;
3. a compilation scheme inspired by Maranget (2008) parametrized by these three ingredients;
4. a validity criterion ensuring $\text{Knit}^\bullet$ and $\text{Frog}^\bullet$ correctly manipulate memory representations of values and a proof of correctness of our algorithm;
5. several examples of both toy and realistic memory representations;
6. `ribbit`, a prototype implementation of our approach.

The complete article is currently under submission; an ArXiV link will be included in the final version of this extended abstract.

During our presentation, we will detail our approach to formalizing memory representations and how to compile pattern matching accordingly, with illustrating examples. We will finish our presentation with future perspective, in particular our plan to generate new optimized representations of Algebraic Data Types which fulfill our validity criterion.

References

[Kosarev u. a. 2020] Kosarev, Dmitry ; Lozov, Petr ; Boulytchev, Dmitry: Relational Synthesis for Pattern Matching. In: S. Oliveira, Bruno C. d. (Hrsg.): *Processings of the 18th Programming Languages and Systems Asian Symposium, APLAS 2020, Fukuoka, Japan, 2020* Bd. 12470, Springer, 2020, S. 293–310. – URL https://doi.org/10.1007/978-3-030-64437-6_15

[Maranget 2008] Maranget, Luc: Compiling pattern matching to good decision trees. In: Sumii, Eijiro (Hrsg.): *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, ACM, 2008, S. 35–46. – URL https://doi.org/10.1145/1411304.1411311

[Sestoft 1996] Sestoft, Peter: ML pattern match compilation and partial evaluation. In: *Partial Evaluation*. Springer, 1996, S. 446–464

[Wadler 1987] Wadler, Philip: Efficient compilation of pattern-matching. In: *The implementation of functional programming languages* (1987)

[Wilke 2016] Wilke, Pierre: *Formally verified compilation of low-level C code. (Compilation formellement vérifiée de code C de bas-niveau)*, University of Rennes 1, France, Dissertation, 2016. – URL https://tel.archives-ouvertes.fr/tel-01483676