Optimized Memory Representation for Algebraic Data Types

Gabriel RADANNE – Inria CASH/LIP
Laure Gonnord – Grenoble INP/LCIS & LIP/CASH

2022-2023

1 Context

Originally introduced in the 80s, Algebraic Data Types (ADT) [3, 1] have emerged in the last decade as an incredibly effective tool to model and manipulate data for programming. Algebraic Data Types are the combination of "Product types", which correspond to records, and "Sum types", which correspond to tagged unions, an extension of traditional enumerations. Algebraic Data Types are also provided with "pattern matching", an extension of `switch` which allow to deconstruct complex values conveniently and safely.

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 the data by ensuring via pattern-matching that its manipulation is well typed, exhaustive, and non-redundant.

— Optimize manipulation of the data thanks to the presence of richer constructs understood by the compiler.

Let us take the example of the `Option` algebraic data type, which indicates that a value can be present (the `Some` case) or not (the `None` case).

```
enum Option<T> { // Optional values of type T
  Some(T), // Some value of type T
  None, // No value
}
```

The type `Option<Int64>` represents an optional 64-bit integer and has a memory representation using up to two words : one word to distinguish between the two constructors and one word containing the integer. We now consider the type `Option<&T>` of optional pointers to a type T. Like machine integers, pointers occupy one word. However null pointers are forbidden in Rust, which means the value 0 is never used. Therefor, an optional pointer can use it to distinguish the `None` constructor which allow Rust to represent values of type `Option<&T>` with only one word. This recovers the efficiency of null pointers, without loosing safety. This concept is also used for other similar types such as file handles (for which -1 is forbidden).

While optimizing compilation exists for pattern matching [5, 7, 6], it often operates for fixed, usually non-optimized, memory representations. Optimizing transformations which improves the memory representation are so far seldom implemented by state-of-the-art compilers.

In the Ribbit project, we have developed a specification language to express bit-precise memory representation of Algebraic Data Types [2]. Our specification language supports a wide variety of tricks usually done manually by programmers, such as bit stealing, inlining, unpacking, or even unrolling of recursive structures.

2 Internship objective

The goal of this internship is to make Ribbit applicable to more context. We propose in particular two avenues for extensions.

LIP – UMR CNRS / ENS Lyon / UCB Lyon 1 / Inria – 69007 Lyon
E-mail : gabriel.radanne@inria.fr and Laure.Gonnord@grenoble-inp.fr

Direction 1 : Automatic generation of optimized representation In our current work, all specifications have to be written by hand. In practice, not all datatypes are performance-critical. Even in performance sensitive context, we would like users to delegate the initial memory representation to the compiler, to be refined later.

In this extensions, we propose to derive, from the source data type, an optimized memory representation, using modern optimisations techniques such as equality saturation [8]. Naturally, many optimisations criterion are possible (space, speed, indirections, ...), which requires some careful design. The candidate will start by computing some reasonably simple memory representation before exploring more aggressive optimisations.

Direction 2 : Ribbit and memory allocations In our current work, we assume memory is managed manually. Naturally, this is quite limiting. Memory management will incur additional constraints on the memory representation which remain to be determined (for instance, a garbage collector will require some metadata), but also numerous opportunities for optimisations, by emitting code that can reuse freed memory. The goal of this extension is to explore these questions for concrete memory management schemes. We will start by exploring the context of reference counting, following [4] to reuse memory aggressively during pattern matching.

3 Internship

The internship will take place in the CASH Team, LIP lab, in Lyon France.

Candidate profile The candidate should ideally be familiar with formal approaches in programming language design. They should also have taste for algorithmic design.

From the practical point of view, a basic experience in software programming and usage of collaborative tools such that `git`. Knowledge of OCaml is strongly recommended.

Références

[1] Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture, FPCA 1985, Nancy, France, September 16-19, 1985, Proceedings*, volume 201 of *Lecture Notes in Computer Science*, pages 368–381. Springer, 1985. doi : $10.1007/3\text{-}540\text{-}15975\text{-}4\backslash\_48$. URL https://doi.org/10.1007/3-540-15975-4_48.

[2] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Bit-Stealing Made Legal. In *ICFP 2023*, ICFP, Seattle (USA), United States, September 2023. doi : $10.1145/3607858$. URL https://inria.hal.science/hal-04165615.

[3] Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE : an experimental applicative language. In *Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980*, pages 136–143. ACM, 1980. doi : $10.1145/800087.802799$. URL https://doi.org/10.1145/800087.802799.

[4] Anton Lorenzen, Daan Leijen, and Wouter Swierstra. Fp² : Fully in-place functional programming. *Proc. ACM Program. Lang.*, 7(ICFP), aug 2023. doi : $10.1145/3607840$. URL https://doi.org/10.1145/3607840.

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

[6] Peter Sestoft. Ml pattern match compilation and partial evaluation. In *Partial Evaluation*, pages 446–464. Springer, 1996.

[7] Philip Wadler. Efficient compilation of pattern-matching. *The implementation of functional programming languages*, 1987.

[8] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg : Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL) :1–29, 2021. doi : $10.1145/3434304$. URL https://doi.org/10.1145/3434304.