Compiler Intermediate Representation for Algebraic Data Types

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

2022-2023

1 Context

Algebraic Data Types In the last few decades, Algebraic Data Types (ADT) have emerged 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).

Such optimizing transformations are so far seldom implemented by state-of-the-art compilers. Our long term project is to develop these transformations and make them available to language designers.

MLIR – Scaling compiler infrastructure for domain specific computation LLVM [1] is the defacto standard compiler framework for many languages (ranging from C and Rust, to Haskell). It allow compiler writers and language designers to harness its numerous builtin compiler transformation, and to write new ones.

MLIR [2] is a general purpose compiler internal representation, developed on top of the LLVM architecture, which features *dialects*. Its goal is to enable many different programming language constructs, allowing compiler writer to choose the one they want. Dialects are then lowered progressively towards the LLVM intermediate representation.

1. https://llvm.org/

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

2 Internship objective

In the last few years, we have been working on expressive and optimized memory representation for Algebraic Data Types[1], in order to make them more appealing to High Performance programmers.

To pursue this goal further, we want to provide an MLIR dialect for Algebraic Data Type which provide language constructs to build and pattern matching Algebraic Data Types, and the algorithm to compile such pattern matching to lower representations.

During the internship, the intern will gain working knowledge of algorithm to compiler pattern matching of Algebraic Data Types, propose language constructs for ADT in MLIR, and implement them.

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`.

Références

[1] Thaïs Baudon, Gabriel Radanne, and Laure Gonnord. Knit&Frog : Pattern matching compilation for custom memory representations (doctoral session). In *AFADL 2022 - 21ème journées Approches Formelles dans l'Assistance au Développement de Logiciels*, Vannes, France, June 2022. URL https://hal.inria.fr/hal-03676356.

[2] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir : Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14. IEEE, 2021.