Translation Verification of the OCaml pattern matching compiler

Francesco Mecca

1 TODO Scaletta [1/2]

- ☒ Abstract
- ☐ Introduction [0%]
  - ☐ Ocaml
  - ☐ Pattern matching
  - ☐ Translation Verification
  - ☐ Symbolic execution

Abstract

This dissertation presents an algorithm for the translation validation of the OCaml pattern matching compiler. Given the source representation of the target program and the target program compiled in untyped lambda form, the algoritmhm is capable of modelling the source program in terms of symbolic constraints on it's branches and apply symbolic execution on the untyped lambda representation in order to validate wheter the compilation produced a valid result. In this context a valid result means that for every input in the domain of the source program the untyped lambda translation produces the same output as the source program. The input of the program is modelled in terms of symbolic constraints closely related to the runtime representation of OCaml objects and the output consists of OCaml code blackboxes that are not evaluated in the context of the verification.

2 Introduction

2.1 TODO OCaml

Objective Caml (OCaml) is a dialect of the ML (Meta-Language) family of programming languages. OCaml shares many features with other dialects

of ML, such as SML and Caml Light, The main features of ML languages are the use of the Hindley-Milner type system that provides with respect to static type systems of traditional imperative and/or object oriented language such as C, C++ and Java many advantages such as:

- Parametric polymorphism: in certain scenarios a function can accept more than one type for the input parameters. For example a function that computes the lenght of a list doesn't need to inspect the type of the elements of the list and for this reason a List.length function can accept list of integers, list of strings and in general list of any type. Such languages offer polymorphic functions through subtyping at runtime only, while other languages such as C++ offer polymorphism through compile time templates and function overloading. With the Hindley-Milner type system each well typed function can have more than one type but always has a unique best type, called the *principal type*. For example the principal type of the List.length function is "For any a, function from list of a to int" and a is called the *type parameter*.

- Strong typing: Languages such as C and C++ allow the programmer to operate on data without considering its type, mainly through pointers. Other languages such as C# and Go allow type erasure so at runtime the type of the data can't be queried. In the case of programming languages using an Hindley-Milner type system the programmer is not allowed to operate on data by ignoring or promoting its type.

- Type Inference: the principal type of a well formed term can be inferred without any annotation or declaration.