Translation Verification of the OCaml pattern matching compiler

Francesco Mecca

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.

1 Background

1.1 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 many advantages with respect to static type systems of traditional imperative and object oriented language such as C, C++ and Java, such as:

- 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 lists of integers, lists of strings and in general lists 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.

- Algebraic data types: types that are modelled by the use of two algebraic operations, sum and product. A sum type is a type that can hold of many different types of objects, but only one at a time. For example the sum type defined as $A + B$ can hold at any moment a value of type A or a value of type B. Sum types are also called tagged union or variants. A product type is a type constructed as a direct product of multiple types and contains at any moment one instance for every type of its operands. Product types are also called tuples or records. Algebraic data types can be recursive in their definition and can be combined.

Moreover ML languages are functional, meaning that functions are treated as first class citizens and variables are immutable, although mutable statements and imperative constructs are permitted. In addition to that OCaml features an object system, that provides inheritance, subtyping and dynamic binding, and modules, that provide a way to encapsulate definitions. Modules are checked statically and can be reificated through functors.

1.2 Lambda form compilation

OCaml provides compilation in form of a byecode executable with an optionally embeddable interpreter and a native executable that could be statically linked to provide a single file executable.

After the OCaml typechecker has proven that the program is type safe, the OCaml compiler lower the code to *Lambda*, an s-expression based language that assumes that its input has already been proved safe. On the *Lambda* representation of the source program, the compiler performes a series of optimization passes before translating the lambda form to assembly code.

1. OCaml datatypes

   Most native data types in OCaml, such as integers, tuples, lists, records, can be seen as instances of the following definition

   ```
   type t = Nil | One of int | Cons of int * t
   ```

   that is a type *t* with three constructors that define its complete signature. Every constructor has an arity. Nil, a constructor of arity 0, is called a constant constructor.

2. Lambda form types A lambda form target file produced by the ocaml compiler consists of a single s-expression. Every s-expression consist of *(*, a sequence of elements separated by a whitespace and a closing *)*. Elements of s-expressions are:

   - Atoms: sequences of ascii letters, digits or symbols
   - Variables
   - Strings: enclosed in double quotes and possibly escaped
   - S-expressions: allowing arbitrary nesting

   There are several numeric types:

   - integers: that us either 31 or 63 bit two's complement arithmetic depending on system word size, and also wrapping on overflow
   - 32 bit and 64 bit integers: that use 32-bit and 64-bit two's complement arithmetic with wrap on overflow
   - big integers: offer integers with arbitrary precision
   - floats: that use IEEE754 double-precision (64-bit) arithmetic with the addition of the literals *infinity*, *neg_ infinity* and *nan*.

   The are varios numeric operations defined:

- Arithmetic operations: +, -, *, /, % (modulo), neg (unary negation)
- Bitwise operations: &, |, ˆ, «, » (zero-shifting), a» (sign extending)
- Numeric comparisons: <, >, <=, >=, ==

3. Functions

   Functions are defined using the following syntax, and close over all bindings in scope: (lambda (arg1 arg2 arg3) BODY) and are applied using the following syntax: (apply FUNC ARG ARG ARG) Evaluation is eager.

4. Bindings The atom *let* introduces a sequence of bindings: (let BINDING BINDING BINDING ... BODY)

5. Other atoms TODO: if, switch, stringswitch... TODO: magari esempi

1.3 Pattern matching

Pattern matching is a widely adopted mechanism to interact with ADT. C family languages provide branching on predicates through the use of if statements and switch statements. Pattern matching on the other hands express predicates through syntactic templates that also allow to bind on data structures of arbitrary shapes. One common example of pattern matching is the use of regular expressions on strings. OCaml provides pattern matching on ADT and primitive data types. The result of a pattern matching operation is always one of:

- this value does not match this pattern"
- this value matches this pattern, resulting the following bindings of names to values and the jump to the expression pointed at the pattern.

```
type color = | Red | Blue | Green | Black | White

match color with
| Red -> print "red"
| Blue -> print "red"
| Green -> print "red"
| _ -> print "white or black"
```

OCaml provides tokens to express data destructoring. For example we can examine the content of a list with patten matching

```
begin match list with
| [ ] -> print "empty list"
| element1 :: [ ] -> print "one element"
| (element1 :: element2) :: [ ] -> print "two elements"
| head :: tail-> print "head followed by many elements"
```

Parenthesized patterns, such as the third one in the previous example, matches the same value as the pattern without parenthesis.

The same could be done with tuples

```
begin match tuple with
| (Some _, Some _) -> print "Pair of optional types"
| (Some _, None) | (None, Some _) -> print "Pair of optional types, one of which is nul
| (None, None) -> print "Pair of optional types, both null"
```

The pattern pattern$_1$ | pattern$_2$ represents the logical "or" of the two patterns pattern$_1$ and pattern$_2$. A value matches pattern$_1$ | pattern$_2$ if it matches pattern$_1$ or pattern$_2$.

Pattern clauses can make the use of *guards* to test predicates and variables can captured (binded in scope).

```
begin match token_list with
| "switch"::var::"{"::rest -> ...
| "case"::":"::var::rest when is_int var -> ...
| "case"::":"::var::rest when is_string var -> ...
| "}"::[ ] -> ...
| "}"::rest -> error "syntax error: " rest
```

Moreover, the OCaml pattern matching compiler emits a warning when a pattern is not exhaustive or some patterns are shadowed by precedent ones.

In general pattern matching on primitive and algebraic data types takes the following form.

```
match variable with
| pattern_1 -> expr_1
| pattern_2 when guard -> expr_2
| pattern_3 as var -> expr_3
⋮
| p_n -> expr_n
```

It can be described more formally through a BNF grammar.

```
pattern ::= value-name
| _   ;; wildcard pattern
| constant  ;; matches a constant value
| pattern as  value-name  ;; binds to value-name
| ( pattern ) ;; parenthesized pattern
| pattern |  pattern ;; or-pattern
| constr  pattern ;; variant pattern
| [ pattern  { ; pattern }  [ ; ] ] ;; list patterns
| pattern ::  pattern   ;; lists patterns using cons operator (::)
| [| pattern  { ; pattern }  [ ; ] |] ;; array pattern
| char-literal ..  char-literal   ;; match on a range of characters
| { field  [: typexpr]  [= pattern] { ; field  [: typexpr]  [= pattern] } \
  [; _ ] [ ; ] } ;; patterns that match on records
```

1.4 Symbolic execution

1.5 Translation validation

Translators, such as translators and code generators, are huge pieces of software usually consisting of multiple subsystem and constructing an actual specification of a translator implementation for formal validation is a very long task. Moreover, different translators implement different algorithms, so the correctness proof of a translator cannot be generalized and reused to prove another translator. Translation validation is an alternative to the verification of existing translators that consists of taking the source and the target (compiled) program and proving *a posteriori* their semantic equivalence.

☐ Techniques for translation validation

☐ What does semantically equivalent mean

☐ What happens when there is no semantic equivalence

☐ Translation validation through symbolic execution