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:

- 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.

- 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. 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 i
| (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
```

2. 1.2.1 Pattern matching compilation to lambda code

   During compilation, patterns are in the form

pattern	type of pattern
_	wildcard
x	variable
c(p_1,p_2,…,p_n)	constructor pattern
(p_1| p_2)	or-pattern

Expressions are compiled into lambda code and are referred as lambda code actions.

The entire pattern matching code can be represented as a clause matrix that associates rows of patterns ($p_{i,1}$, $p_{i,2}$, …, $p_{i,n}$) to lambda code action l^i

$$(P \to L) = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} & \to l_1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} & \to l_2 \\ \vdots & \vdots & \ddots & \vdots & \to \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} & \to l_m \end{pmatrix}$$

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.

The pattern *p* matches a value *v*, written as p $\preccurlyeq$ v, when one of the following rules apply

_	$\preccurlyeq$	v	$\forall$v
x	$\preccurlyeq$	v	$\forall$v
(p$_1$ |\ p$_2$)	$\preccurlyeq$	v	iff p$_1$ $\preccurlyeq$ v or p$_2$ $\preccurlyeq$ v
c(p$_1$, p$_2$, ..., p$_a$)	$\preccurlyeq$	c(v$_1$, v$_2$, ..., v$_a$)	iff (p$_1$, p$_2$, ..., p$_a$) $\preccurlyeq$ (v$_1$, v$_2$, ..., v$_a$)
(p$_1$, p$_2$, ..., p$_a$)	$\preccurlyeq$	(v$_1$, v$_2$, ..., v$_a$)	iff p$_i$ $\preccurlyeq$ v$_i$ $\forall$i $\in$ [1..a]

When a value v matches pattern p we say that v is an *instance* of p.

Considering the pattern matrix P we say that the value vector $\vec{v}$ = (v$_1$, v$_2$, ..., v$_i$) matches the line number i in P if and only if the following two conditions are satisfied:

- $p_{i,1}, p_{i,2}, \cdots, p_{i,n} \preccurlyeq (v_1, v_2, \ldots, v_i)$
- $\forall j < i \; p_{j,1}, p_{j,2}, \cdots, p_{j,n} \not\preccurlyeq (v_1, v_2, \ldots, v_i)$

We can define the following three relations with respect to patterns:

- Patter p is less precise than pattern q, written p $\preccurlyeq$ q, when all instances of q are instances of p
- Pattern p and q are equivalent, written p $\equiv$ q, when their instances are the same
- Patterns p and q are compatible when they share a common instance

(a) Initial state of the compilation
Given a source of the following form:

match variable with
| pattern$_1$ -> e$_1$
| pattern$_2$ -> e$_2$
$\vdots$
| p$_m$ -> e$_m$

the initial input of the algorithm C consists of a vector of variables $\vec{x}$ = (x$_1$, x$_2$, ..., x$_n$) of size n where n is the arity of the type of x and a clause matrix P $\rightarrow$ L of width n and height m. That is:

$$C((\vec{x} = (x_1, x_2, ..., x_n), \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \rightarrow l_1 \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \rightarrow l_2 \\ \vdots & \vdots & \ddots & \vdots \rightarrow \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \rightarrow l_m \end{pmatrix})$$

The base case C$_0$ of the algorithm is the case in which the $\vec{x}$ is empty, that is $\vec{x} = ()$, then the result of the compilation C$_0$ is l$_1$

$$C_0((), \begin{pmatrix} \rightarrow l_1 \\ \rightarrow l_2 \\ \rightarrow \vdots \\ \rightarrow l_m \end{pmatrix})) = l_1$$

When $\vec{x} \neq ()$ then the compilation advances using one of the following four rules:

i. Variable rule: if all patterns of the first column of P are wildcard patterns or bind the value to a variable, then

$$C(\vec{x}, P \rightarrow L) = C((x_2, x_3, ..., x_n), P' \rightarrow L')$$

where

$$\begin{pmatrix} p_{1,2} & \cdots & p_{1,n} & \rightarrow & (let & y_1 & x_1) & l_1 \\ p_{2,2} & \cdots & p_{2,n} & \rightarrow & (let & y_2 & x_1) & l_2 \\ \vdots & \ddots & \vdots & \rightarrow & \vdots & \vdots & \vdots & \vdots \\ p_{m,2} & \cdots & p_{m,n} & \rightarrow & (let & y_m & x_1) & l_m \end{pmatrix}$$

That means in every lambda action l$_i$ there is a binding of x$_1$ to the variable that appears on the pattern \$p$_{i,1}$. Bindings are omitted for wildcard patterns and the lambda action l$_i$ remains unchanged.

ii. Constructor rule: if all patterns in the first column of P are constructors patterns of the form k(q$_1$, q$_2$, . . . , q$_n$) we define a new matrix, the specialized clause matrix S, by applying the following transformation on every row p:

```
for every c ∈ Set of constructors
    for i ← 1 .. m
        let k_i ← constructor_of($p_{i,1}$)
        if k_i = c then
            p ← $q_{i,1}$, $q_{i,2}$, ..., $q_{i,n'}$, $p_{i,2}$, $p_{i,3}$, ..., $p_{i,n}$
```

Patterns of the form $q_{i,j}$ matches on the values of the constructor and we define new fresh variables y$_1$, y$_2$, . . . , y$_a$ so that the lambda action l$_i$ becomes

```
(let (y_1 (field 0 x_1))
     (y_2 (field 1 x_1))
     ...
     (y_a (field (a−1) x_1))
     l_i)
```

and the result of the compilation for the set of constructors {c_1, c_2, . . . , c_k} is:

```
switch x_1 with
case c_1: l_1
case c_2: l_2
...
case c_k: l_k
default: exit
```

i. Orpat rule: there are various strategies for dealing with or-patterns. The most naive one is to split the or-patterns. For example a row p containing an or-pattern:

$$(p_{i,1}|q_{i,1}|r_{i,1}), p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

results in three rows added to the clause matrix

$$p_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

$$q_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

$$r_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

ii. Mixture rule: When none of the previous rules apply the clause matrix P → L is splitted into two clause matrices, the first $P_1 \rightarrow L_1$ that is the largest prefix matrix for which one of the three previous rules apply, and $P_2 \rightarrow L_2$ containing the remaining rows. The algorithm is applied to both matrices.