Translation Verification of the OCaml pattern matching compiler

Francesco Mecca

1 TODO Scaletta [1/2]

- ☒ Abstract
- ⊟ Background [20%]
  - ☒ Ocaml
  - ☐ Lambda code [0%]
    - ☐ Untyped lambda form
    - ☐ OCaml specific instructions
  - ⊟ Pattern matching [50%]
    - ☒ Introduzione
    - ☐ Compilation to lambda form
  - ☐ Translation Verification
  - ☐ Symbolic execution
- ☐ Translation verification of the Pattern Matching Compiler
  - ☐ Source translation
    - ☐ Formal Grammar
    - ☐ Compilation of source patterns
  - ☐ Target translation
    - ☐ Formal Grammar
    - ☐ Symbolic execution of the lambda target
  - ☐ Equivalence between source and target
  - ☐ Practical results

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 Background

2.1 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. **TODO** Pattern matching [37%]

   ☐ capisci come mettere gli esempi uno accanto all'altro

   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 is a mechanism for destructuring and analyzing data structures for the presence of values simbolically represented as tokens. One common example of pattern matching is the use of regular expressions on strings. OCaml provides pattern matching on ADT, primitive data types.

   ☒ Esempio enum, C e Ocaml

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

begin match color with
```

```
| Red -> print "red"
| Blue -> print "red"
| Green -> print "red"
```

OCaml provides tokens to express data destructoring

☒ Esempio destructor list

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

☒ Esempio destructor tuples

```
begin match tuple with
| (Some _, Some _) -> print "Pair of optional types"
| (Some _, None) -> print "Pair of optional types, last null"
| (None, Some _) -> print "Pair of optional types, first null"
| (None, None) -> print "Pair of optional types, both null"
```

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

☐ Esempio binding e guards

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

☐ Un altro esempio con destructors e tutto i lresto

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

- ☐ Esempio informale

It can be described more formally through a BNF grammar.

- ☐ BNF
- ☐ Come funziona il pattern matching?

2. **TODO** 1.2.1 Pattern matching compilation to lambda code

   - ☐ Da tabella a matrice

   Formally, pattern are defined as follows:

pattern	Patterns
_	wildcard
x	variable
$c(p_1,p_2,\ldots,p_n)$	constructor pattern
$(p_1 \mid p_2)$	or-pattern

Values are defined as follows:

values	Values
$c(v_1, v_2, \ldots, v_n)$	constructor value

The entire pattern matching code can be represented as a clause matrix that associates rows of patterns $(p_{i,1}, p_{i,2}, \ldots, p_{i,n})$ 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]

We can also say that v is an *instance* of p.

When we consider the pattern matrix P we say that the value vector $\vec{v} = (v_1, v_2, \ldots, v_i)$ 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

2.2 1.2.1.1 Initial state of the compilation

Given a source of the following form:

#+BEGIN_SRC ocaml match x with

p1 -> e1
p2 -> e2

...

p_m -> e_m

#+END_SRC ocaml

the initial input of the algorithm consists of a vector of variables $\vec{x} = (x_1, x_2, \ldots, x_n)$ of size n and a clause matrix $P \to L$ of width n and height m.

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