Translation Verification of the pattern matching compiler

Francesco Mecca

1 Introduction

This dissertation presents an algorithm for the translation validation of the OCaml pattern matching compiler. Given a source program and its compiled version the algorithm checks whether the two are equivalent or produce a counter example in case of a mismatch. For the prototype of this algorithm we have chosen a subset of the OCaml language and implemented a prototype equivalence checker along with a formal statement of correctness and its proof. The prototype is to be included in the OCaml compiler infrastructure and will aid the development.

Our equivalence algorithm works with decision trees. Source patterns are converted into a decision tree using a matrix decomposition algorithm. Target programs, described in the Lambda intermediate representation language of the OCaml compiler, are turned into decision trees by applying symbolic execution.

A pattern matching compiler turns a series of pattern matching clauses into simple control flow structures such as `if, switch`, for example:

```
match x with
| [] -> (0, None)
| x::[] -> (1, Some x)
| _::y::_ -> (2, Some y)
```

```
(if scrutinee
   (let (field_1 =a (field 1 scrutinee))
      (if field_1
         (let
            (field_1_1 =a (field 1 field_1)
             x =a (field 0 field_1))
            (makeblock 0 2 (makeblock 0 x)))
         (let (y =a (field 0 scrutinee))
```

```
                (makeblock 0 1 (makeblock 0 y)))))
    [0: 0 0a])
```

The code on the right is in the Lambda intermediate representation of the OCaml compiler. The Lambda representation of a program is shown by calling the `ocamlc` compiler with `-drawlambda` flag.

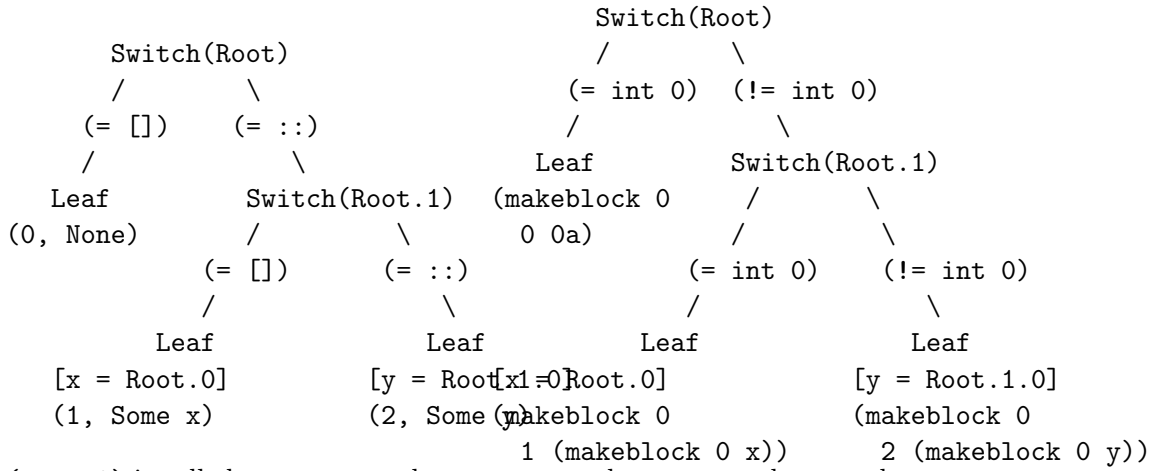
The OCaml pattern matching compiler is a critical part of the OCaml compiler in terms of correctness because bugs typically would result in wrong code production rather than triggering compilation failures. Such bugs also are hard to catch by testing because they arise in corner cases of complex patterns which are typically not in the compiler test suite or most user programs.

The OCaml core developers group considered evolving the pattern matching compiler, either by using a new algorithm or by incremental refactoring of its code base. For this reason we want to verify that new implementations of the compiler avoid the introduction of new bugs and that such modifications don't result in a different behavior than the current one.

One possible approach is to formally verify the pattern matching compiler implementation using a machine checked proof. Another possibility, albeit with a weaker result, is to verify that each source program and target program pair are semantically correct. We chose the latter technique, translation validation because is easier to adopt in the case of a production compiler and to integrate with an existing code base. The compiler is treated as a black-box and proof only depends on our equivalence algorithm.

1.1 Our approach

%% replace common TODO Our algorithm translates both source and target programs into a common representation, decision trees. Decision trees where chosen because they model the space of possible values at a given branch of execution. Here are the decision trees for the source and target example program.



`(Root.0)` is called an *accessor*, that represents the access path to a value that can be reached by deconstructing the scrutinee. In this example `Root.0` is the first subvalue of the scrutinee.

Target decision trees have a similar shape but the tests on the branches are related to the low level representation of values in Lambda code. For example, cons cells `x::xs` or tuples `(x,y)` are blocks with tag 0.

To check the equivalence of a source and a target decision tree, we proceed by case analysis. If we have two terminals, such as leaves in the previous example, we check that the two right-hand-sides are equivalent. If we have a node N and another tree T we check equivalence for each child of N, which is a pair of a branch condition π_i and a subtree C_i. For every child (π_i, C_i) we reduce T by killing all the branches that are incompatible with π_i and check that the reduced tree is equivalent to C_i.

1.2 From source programs to decision trees

Our source language supports integers, lists, tuples and all algebraic datatypes. Patterns support wildcards, constructors and literals, or-patterns $(p_1|p_2)$ and pattern variables. We also support `when` guards, which are interesting as they introduce the evaluation of expressions during matching. Decision trees have nodes of the form:

```
type decision_tree =
 | Unreachable
 | Failure
 | Leaf of source_expr
 | Guard of source_expr * decision_tree * decision_tree
 | Switch of accessor * (constructor * decision_tree) list * decision_tree
```

In the `Switch` node we have one subtree for every head constructor that appears in the pattern matching clauses and a fallback case for other values. The branch condition π_i expresses that the value at the switch accessor starts with the given constructor. `Failure` nodes express match failures for values that are not matched by the source clauses. `Unreachable` is used when we statically know that no value can flow to that subtree.

We write $[\![t_S]\!]_S$ for the decision tree of the source program t_S, computed by a matrix decomposition algorithm (each column decomposition step gives a `Switch` node). It satisfies the following correctness statement:

$$\forall t_S, \forall v_S, \quad t_S(v_S) = [\![t_S]\!]_S(v_S)$$

Running any source value v_S against the source program gives the same result as running it against the decision tree.

1.3 From target programs to decision trees

The target programs include the following Lambda constructs: `let, if, switch, Match_failure, catch, exit, field` and various comparison operations, guards. The symbolic execution engine traverses the target program and builds an environment that maps variables to accessors. It branches at every control flow statement and emits a `Switch` node. The branch condition π_i is expressed as an interval set of possible values at that point. In comparison with the source decision trees, `Unreachable` nodes are never emitted. Guards result in branching. In comparison with the source decision trees, `Unreachable` nodes are never emitted.

We write $[\![t_T]\!]_T$ for the decision tree of the target program t_T, satisfying the following correctness statement:

$$\forall t_T, \forall v_T, \quad t_T(v_T) = [\![t_T]\!]_T(v_T)$$

1.4 Equivalence checking

The equivalence checking algorithm takes as input a domain of possible values S and a pair of source and target decision trees and in case the two trees are not equivalent it returns a counter example. The algorithm respects the following correctness statement:

$$\mathsf{equiv}(S, C_S, C_T) = \mathsf{Yes} \ \wedge \ C_T \ \mathsf{covers} \ S \implies \forall v_S \approx v_T \in S, \ C_S(v_S) = C_T(v_T)$$
$$\mathsf{equiv}(S, C_S, C_T) = \mathsf{No}(v_S, v_T) \ \wedge \ C_T \ \mathsf{covers} \ S \implies v_S \approx v_T \in S \ \wedge \ C_S(v_S) \neq C_T(v_T)$$

The algorithm proceeds by case analysis. Inference rules are shown. If S is empty the results is Yes.

$$\frac{}{\mathsf{equiv}(\emptyset, C_S, C_T)G}$$

If the two decision trees are both terminal nodes the algorithm checks for content equality.

$$\frac{}{\mathsf{equiv}(S, \mathsf{Failure}, \mathsf{Failure})[]}$$

$$\frac{t_S \approx_{\mathsf{term}} t_T}{\mathsf{equiv}(S, \mathsf{Leaf}(t_S), \mathsf{Leaf}(t_T))[]}$$

If the source decision tree (left hand side) is a terminal while the target decistion tree (right hand side) is not, the algorithm proceeds by *explosion* of the right hand side. Explosion means that every child of the right hand side is tested for equality against the left hand side.

$$\frac{\begin{array}{c} C_S \in \mathsf{Leaf}(t), \mathsf{Failure} \\ \forall i,\ \mathsf{equiv}((S \wedge a \in D_i), C_S, C_i)G \qquad \mathsf{equiv}((S \wedge a \notin (D_i)^i), C_S, C_{\mathsf{fb}})G \end{array}}{\mathsf{equiv}(S, C_S, \mathsf{Switch}(a, (D_i)^i C_i, C_{\mathsf{fb}}))G}$$

When the left hand side is not a terminal, the algorithm explodes the left hand side while trimming every right hand side subtree. Trimming a left hand side tree on an interval set dom_S computed from the right hand side tree constructor means mapping every branch condition dom_T (interval set of possible values) on the left to the intersection of dom_T and dom_S when the accessors on both side are equal, and removing the branches that result in an empty intersection. If the accessors are different, dom_T is left unchanged.

$$\frac{\begin{array}{c} \forall i,\ \mathsf{equiv}((S \wedge a = K_i), C_i, \mathsf{trim}(C_T, a = K_i))G \\ \mathsf{equiv}((S \wedge a \notin (K_i)^i), C_{\mathsf{fb}}, \mathsf{trim}(C_T, a \notin (K_i)^i))G \end{array}}{\mathsf{equiv}(S, \mathsf{Switch}(a, (K_i, C_i)^i, C_{\mathsf{fb}}), C_T)G}$$

The equivalence checking algorithm deals with guards by storing a queue. A guard blackbox is pushed to the queue whenever the algorithm encounters a Guard node on the right, while it pops a blackbox from the queue whenever a Guard node appears on the left hand side. The algorithm stops with failure

if the popped blackbox and the and blackbox on the left hand Guard node are different, otherwise in continues by exploding to two subtrees, one in which the guard condition evaluates to true, the other when it evaluates to false. Termination of the algorithm is successful only when the guards queue is empty.

$$\frac{\mathsf{equiv}(S, C_0, C_T)G, (t_S = 0) \qquad \mathsf{equiv}(S, C_1, C_T)G, (t_S = 1)}{\mathsf{equiv}(S, \mathsf{Guard}(t_S, C_0, C_1), C_T)G}$$

$$\frac{t_S \approx_{\mathsf{term}} t_T \qquad \mathsf{equiv}(S, C_S, C_b)G}{\mathsf{equiv}(S, C_S, \mathsf{Guard}(t_T, C_0, C_1))(t_S = b), G}$$

2 Background

2.1 OCaml

Objective Caml (OCaml) is a dialect of the ML (Meta-Language) family of programming that 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 length 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 modeled 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 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 reifycated through functors.

2.2 Compiling OCaml code

The OCaml compiler provides compilation of source files in form of a bytecode executable with an optionally embeddable interpreter or as a native executable that could be statically linked to provide a single file executable. Every source file is treated as a separate *compilation unit* that is advanced through different states. The first stage of compilation is the parsing of the input code that is trasformed into an untyped syntax tree. Code with syntax errors is rejected at this stage. After that the AST is processed by the type checker that performs three steps at once:

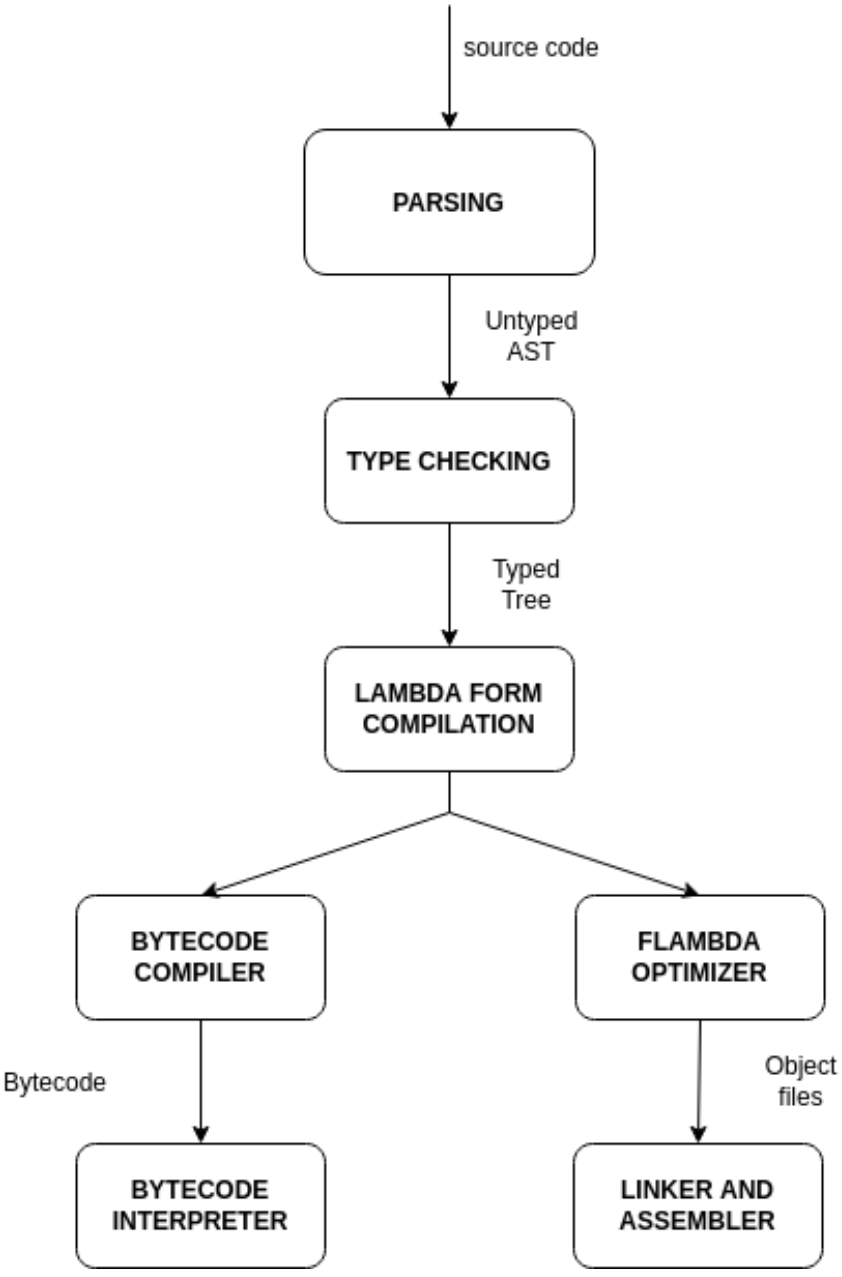
- type inference, using the classical *Algorithm W*

- perform subtyping and gathers type information from the module system

- ensures that the code obeys the rule of the OCaml type system

At this stage, incorrectly typed code is rejected. In case of success, the untyped AST in trasformed into a *Typed Tree*. After the typechecker has proven that the program is type safe, the compiler lower the code to *Lambda*, an s-expression based language that assumes that its input has already been proved safe. After the Lambda pass, the Lambda code is either translated into bytecode or goes through a series of optimization steps performed by the *Flambda* optimizer before being translated into assembly.

This is an overview of the different compiler steps.


source code
PARSING
Untyped
AST
TYPE CHECKING
Typed
Tree
LAMBDA FORM
COMPILATION
BYTECODE
COMPILER
FLAMBDA
OPTIMIZER
Bytecode
Object
files
BYTECODE
INTERPRETER
LINKER AND
ASSEMBLER

2.3 Memory representation of OCaml values

An usual OCaml source program contains few to none explicit type signatures. This is possible because of type inference that allows to annotate the AST with type informations. However, since the OCaml typechecker guarantes that a program is well typed before being transformed into Lambda code, values at runtime contains only a minimal subset of type informations needed to distinguish polymorphic values. For runtime values, OCaml uses a uniform memory representation in which every variable is stored as a value in a contiguous block of memory. Every value is a single word that is either a concrete integer or a pointer to another block of memory, that is called *cell* or *box*. We can abstract the type of OCaml runtime values as the following:

```
type t = Constant | Cell of int * t
```

where a one bit tag is used to distinguish between Constant or Cell. In particular this bit of metadata is stored as the lowest bit of a memory block.

Given that all the OCaml target architectures guarantee that all pointers are divisible by four and that means that two lowest bits are always 00 storing this bit of metadata at the lowest bit allows an optimization. Constant values in OCaml, such as integers, empty lists, Unit values and constructors of arity zero (*constant* constructors) are unboxed at runtime while pointers are recognized by the lowest bit set to 0.

2.4 Lambda form compilation

A Lambda code target file is produced by the compiler and 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

The Lambda form is a key stage where the compiler discards type informations and maps the original source code to the runtime memory model described. In this stage of the compiler pipeline pattern match statements are analyzed and compiled into an automata.

```
type t = | Foo | Bar | Baz | Fred

let test = function
  | Foo -> "foo"
  | Bar -> "bar"
  | Baz -> "baz"
  | Fred -> "fred"
```

The Lambda output for this code can be obtained by running the compiler with the *-dlambda* flag:

```
(setglobal Prova!
  (let
    (test/85 =
       (function param/86
 (switch* param/86
  case int 0: "foo"
  case int 1: "bar"
  case int 2: "baz"
  case int 3: "fred")))
    (makeblock 0 test/85)))
```

As outlined by the example, the *makeblock* directive is responsible for allocating low level OCaml values and every constant constructor ot the algebraic type t is stored in memory as an integer. The *setglobal* directives declares a new binding in the global scope: Every concept of modules is lost at this stage of compilation. The pattern matching compiler uses a jump table to map every pattern matching clauses to its target expression. Values are addressed by a unique name.

```
type t = | English of p | French of q
type p = | Foo | Bar
type q = | Tata| Titi
type t = | English of p | French of q

let test = function
  | English Foo -> "foo"
  | English Bar -> "bar"
  | French Tata -> "baz"
  | French Titi -> "fred"
```

In the case of types with a smaller number of variants, the pattern matching compiler may avoid the overhead of computing a jump table. This example

also highlights the fact that non constant constructor are mapped to cons cell that are accessed using the *tag* directive.

```
(setglobal Prova!
  (let
    (test/89 =
       (function param/90
 (switch* param/90
  case tag 0: (if (!= (field 0 param/90) 0) "bar" "foo")
  case tag 1: (if (!= (field 0 param/90) 0) "fred" "baz"))))
    (makeblock 0 test/89)))
```

In the Lambda language 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 various numeric operations defined:

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

1. 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.

2. Other atoms The atom *let* introduces a sequence of bindings at a smaller scope than the global one: (let BINDING BINDING BINDING ... BODY)

   The Lambda form supports many other directives such as *strinswitch* that is constructs aspecialized jump tables for string, integer range

comparisons and so on. These construct are explicitely undocumented because the Lambda code intermediate language can change across compiler releases.

2.5 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. 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"
```

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