Translation validation of a pattern-matching compiler

Francesco Mecca, Gabriel Scherer

Translation validation of a pattern-matching compiler

Francesco Mecca, Gabriel Scherer

August 16, 2020

Abstract

We propose an algorithm for the translation validation of a pattern matching compiler for a small subset of the OCaml pattern matching features. Given a source program and its compiled version the algorithm checks wheter the two are equivalent or produce a counter example in case of a mismatch.

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

1 Translation validation

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

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

```
(if li
   (let (tl =a (field 1 li))
      (if tl
         (let (y =a (field 0 tl))
            (makeblock 0 2 (makeblock 0 y)))
         (let (x =a (field 0 li))
            (makeblock 0 1 (makeblock 0 x)))))
   (makeblock 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 pattern matching compiler is a critical part of the OCaml compiler in terms of correctness because bugs typically result in wrong code production rather than 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. In the last five years there were (only) two known bugs in the OCaml pattern matching compiler; they resulted in silent wrong-code production, and were found long after they were introduced.

We would like to keep evolving the pattern matching compiler, either by using a new algorithm or by incremental refactorings. We want to verify the changed compiler to ensure that no bugs were introduced.

One could either verify the compiler implementation (full verification) or check each input-output pair (translation validation). We chose translation validation; it gives a weaker result but is much easier to adopt in a production compiler. The pattern-matching compiler is treated as a blackbox and proof only depends on our equivalence algorithm between source and target programs.

It would be very challenging to consider equivalence checking at the scale of a source-to-binary native compiler. In contrast, the pattern-matching compiler is a specific subsystem where we expect equivalence checking to be feasible in practice.

2 Decision Trees

Our algorithm translates both source and target programs into a common representation, *decision trees*. Here are the decision tree for the source and target example programs.

```
         Switch(Root)
          /       \
        ([])      (::)
        /            \
      Leaf         Switch(Root.1)
   (0, None)       /         \
                 ([])        (::)
                /              \
             Leaf              Leaf
      [x = Root.0]        [y = Root.1.0]
      (1, Some x)         (2, Some y)
```

```
            Switch(Root)
             /        \
       (= int 0)   (!= int 0)
         /               \
      Leaf          Switch(Root.1)
    (makeblock 0      /        \
     0 0a)           /          \
              (= int 0)     (!= int 0)
                /               \
             Leaf               Leaf
   [x = Root.0]            [y = Root.1.0]
   (makeblock 0            (makeblock 0
     1 (makeblock 0 x))      2 (makeblock 0 y))
```

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

Source conditions test the head constructor of the accessor, whereas target conditions test the low-level representation of values in Lambda code. For example, cons cells `x::xs` or tuples `(x,y)` are heap blocks (with tag 0), while the empty list `[]` is the immediate integer `0`.

In this simple example, the two decision trees perform exactly the same checks in the same order, so their equivalence is obvious. However, this is not the case in general, as the compiler may reorder checks and simplify away redundant checks.

Computing a source decision tree To compute the decision tree of a source program we use the standard approach of *matrix decomposition*. A pattern matrix is an intermediate datastructure that represents matching on several values in parallel. Each column of the matrix matches on a sub-value of the original scrutinee, denoted by an accessor a_i. For example, the matrix on the left corresponds roughly to the tuple pattern on the right:

$$\begin{array}{lllll} \ a_1 & a_2 & a_3 & & \\ (p_{1,1} & p_{1,2} & p_{1,3} & \to & e_1) \\ (p_{2,1} & p_{2,2} & p_{2,3} & \to & e_2) \end{array}$$

match (a_1, a_2, a_3) with
| ($p_{1,1}$, $p_{1,2}$, $p_{1,3}$) -> e_1
| ($p_{2,1}$, $p_{2,2}$, $p_{2,3}$) -> e_2

The central operation is to *decompose* a given matrix by looking at the variant constructors that occur in the first column – ($p_{1,1}$, $p_{2,1}$) in our example. For each constructor we compute a smaller *submatrix* containing the rows that match this constructor.

For example, the decomposition of the matrix on the left gives the two submatrices on the right, one for the `Some` constructor and one for `None`.

$$
\begin{array}{llllll}
a_1 & a_2 & a_3 & & \\
(\texttt{Some}(q) & p_{1,2} & p_{1,3} & \to & e_1) \\
(\_ & p_{2,2} & p_{2,3} & \to & e_2) \\
(\texttt{None} & p_{3,2} & p_{3,3} & \to & e_3)
\end{array}
\qquad
\begin{array}{l}
\begin{array}{llllll}
a_1.0 & a_2 & a_3 & & \\
(q & p_{1,2} & p_{1,3} & \to & e_1) \\
(\_ & p_{2,2} & p_{2,3} & \to & e_2)
\end{array}
\\[1em]
\begin{array}{lllll}
a_2 & a_3 & & \\
(p_{2,2} & p_{2,3} & \to & e_2) \\
(p_{3,2} & p_{3,3} & \to & e_3)
\end{array}
\end{array}
$$

We build a decomposition tree by repeating this decomposition step. At each step we emit a `Switch` node on the accessor of the first column. We have one sub-tree per submatrix, and the branch condition checks for the submatrix constructor.

We emit a leaf node when repeated decomposition reaches a matrix with empty rows (no columns).

Computing a target decision tree The target programs include the following Lambda constructs: `let, if, switch, Match_failure, catch, exit, field` and various comparison operations, guards. A simple symbolic execution engine traverses the target program, keeping 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.

For example, consider the Lambda fragment `if (= x 0) e1 e2)`. We build a `Switch` node with two children, one for `e1` and one for `e2`. We get the accessor corresponding to `x` in the symbolic environment, and the branch conditions of the two subtrees would correspond respectively to `= 0` (the domain $[0]$) and `!= 0` (the domain $[\texttt{min\_int}; -1] \cup [1; \texttt{max\_int}]$).

3 Checking equivalence

We now give a simplified sketch of our equivalence algorithm. See the appendices for more details.

To check the equivalence of a source and a target decision tree, we proceed recursively by case analysis. If we have two leaves, we check that the target right-hand-side is the compilation of the source right-hand-side, and that the captured environments are identical over their free variables. If we have a Switch node N and another tree T we check equivalence for each child of N. A child is a pair of a branch condition π_i and a subtree D_i. For every child (π_i, D_i) we *trim* T by killing the branches that are incompatible with π_i, and check that the trimmed tree is equivalent to D_i.

Contributions We have chosen a simple subset of the OCaml language and implemented a prototype equivalence checker along with a formal statement of correctness and proof sketches.

Our source language supports integers, lists, tuples and algebraic datatypes. Patterns support wildcards, constructors and literals, or-patterns $(p_1|p_2)$ and pattern variables. We also support `when` guards, which require the evaluation of expressions during matching. As they may perform side-effects, the evaluation of these guard expressions may not be reordered, erased or duplicated.

Our current implementation prototype can be found at:

https://github.com/FraMecca/inria-internship/.

Appendices

In these appendices we are trying to show the minimal amount of formalism to describe our equivalence algorithm in a precise way, and explain how we reason about its correctness – what the correctness statements are. We omit as many details as the explanation can afford (hopefully) to keep the document concise.

A A more formal setting

We will use, but not define by lack of space, the notion of source and target *programs* t_S and t_T, and *expressions* e_S and e_T. A source program t_S is a list of pattern-matching clauses (with an optional `when`-guard) of the form `|` p `(when` e_S`)? ->` e_S, and a target program t_T is a series of control-flow conditionals (`if, switch, Match_failure, catch, exit`), value access (`field`), variable bindings (`let`) and comparison operators in Lambda code, with arbitrary Lambda-code expressions e_T at the leaves.

We assume given an equivalence relation $e_S \approx_{\mathsf{expr}} e_T$ on leaf expressions. In our translation-validation setting, it suffices to relate each source expression to its compiled form – the compiler gives/computes this relation. We have to lift this relation on leaves into an equivalence procedure for (pattern-matching) programs.

$$e_S \approx_{\mathsf{expr}} e_T \ (\textit{assumed}) \qquad t_S(v_S),\ t_T(v_T),\ v_S \approx_{\mathsf{val}} v_T \ (\textit{omitted}) \qquad r_S \approx_{\mathsf{res}} r_T, R_S \approx_{\mathsf{run}} R_T \ (\textit{simple})$$

$$\begin{array}{llll} \textit{environment}\ \sigma(v) ::= [x_1 \mapsto v_1, \dots, v_n \mapsto v_n] & & \textit{matching result}\ \ r(v) ::= \mathsf{NoMatch} \mid \mathsf{Match}(\underline{e}(v)) \\ \textit{closed term}\ \ \underline{e}(v) ::= (\sigma(v), e) & & \textit{matching run}\ \ R(v) ::= (\underline{e}(v)_1, \dots, \underline{e}(v)_n), r(v) \end{array}$$

$$\frac{\forall x,\ \sigma_S(x) \approx_{\mathsf{val}} \sigma_T(x)}{\sigma_S \approx_{\mathsf{env}} \sigma_T} \qquad \frac{\sigma_S \approx_{\mathsf{env}} \sigma_T \qquad e_S \approx_{\mathsf{expr}} e_T}{(\sigma_S, e_S) \approx_{\mathsf{cl-expr}} (\sigma_T, e_T)} \qquad \frac{\forall v_S \approx_{\mathsf{val}} v_T, \quad t_S(v_S) \approx_{\mathsf{run}} t_T(v_T)}{t \approx_{\mathsf{prog}_S} t_T}$$

We use v_S and v_T for source and target values, and define a relation $v_S \approx_{\mathsf{val}} v_T$ to relate a source to a target value; this relation (omitted by lack of space) captures our knowledge of the OCaml value representation, for example it relates the empty list constructor `[]` to $\mathsf{int}\,0$. We can then define *closed* expressions $\underline{e}$, pairing a (source or target) expression with the environment σ captured by a program, and what it means to "run" a value against a program or a decision, written $t(v)$ and $D(v)$, which returns a trace $(\underline{e}_1, \dots, \underline{e}_n)$ of the executed guards and a *matching result* r.

Once formulated in this way, our equivalence algorithm must check the natural notion of input-output equivalence for matching programs, captured by the relation $t_S \approx_{\mathsf{prog}} t_T$.

B Decision trees

The parametrized grammar $D(\pi, e)$ describes the common structure of source and decision trees. We use π for the *conditions* on each branch, and a for our *accessors*, which give a symbolic description of a sub-value of the scrutinee. Source conditions π_S are just datatype constructors;

target conditions π_T are arbitrary sets of possible immediate-integer or block-tag values.

$$
\begin{array}{llcl}
\textit{decision trees} & D(\pi, e) & ::= & \mathsf{Leaf}(\underline{e}(a)) \\
 & & | & \mathsf{Failure} \\
 & & | & \mathsf{Switch}(a, (\pi_i, D_i)^{i \in I}, D_{\mathsf{fb}}) \\
 & & | & \mathsf{Guard}(\underline{e}(a), D_0, D_1) \\
\textit{accessors} & a & ::= & \mathsf{Root} \mid a.n \quad (n \in \mathbb{N})
\end{array}
$$

$$
\begin{array}{l}
\pi_S : \text{datatype constructors} \\
\pi_T \subseteq \{\mathsf{int}\, n \mid n \in \mathbb{Z}\} \uplus \{\mathsf{tag}\, n \mid n \in \mathbb{N}\} \\
\\
a(v_S), a(v_T), D_S(v_S), D_T(v_T) \quad (\textit{omitted})
\end{array}
$$

The tree $\mathsf{Leaf}(\underline{e})$ returns a leaf expression e in a captured environment σ mapping variables to accessors. $\mathsf{Failure}$ expresses match failure, when no clause matches the input value. $\mathsf{Switch}(a, (\pi_i, D_i)^{i \in I}, D_{\mathsf{fb}})$ has one subtree D_i for every head constructor that appears in the pattern matching clauses, and a fallback case for the constructors. $\mathsf{Guard}(\underline{e}, D_0, D_1)$ represents a `when`-guard on a closed expression $\underline{e}$, expected to be of boolean type, with sub-trees D_0 for the `true` case and D_1 for the `false` case.

We write $a(v)$ for the sub-value of the (source or target) value v that is reachable at the accessor a, and $D(v)$ for the result of running a value v against a decision tree D; this results in a (source or target) matching run $R(v)$, just like running the value against a program.

C From source programs to decision trees: matrix decomposition

We write $[\![t_S]\!]_S$ for the decision tree of the source program t_S. It satisfies the expected correctness statement:

$$\forall t_S, \forall v_S, \quad t_S(v_S) \approx_{\mathsf{run}} [\![t_S]\!]_S(v_S)$$

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

The decision tree of a source program is in fact defined by the more general operation of computing the decision tree of pattern matrix – by a matrix decomposition algorithm.

A pattern matrix with rows indexed over I and columns indexed over J is an object of the form $((a_j)^{j \in J}, ((p_{i,j})^{j \in J} \to e_i)^{i \in i})$ or, in more visual notation:

$$
\begin{array}{cccccc}
 & a_1 & \dots & a_j & & \\
(& p_{1,1} & \dots & p_{1,j} & \to & e_1) \\
 & & \dots & & & \\
(& p_{i,1} & \dots & p_{i,j} & \to & e_i)
\end{array}
$$

The usual intuition of a J-columns matrix m is that it matches on J input values simultaneously, with a natural *run* function $m(v_j)^{j \in J}$. This intuition does not suffice to formulate a correctness statement, because we cannot directly relate a matrix m matching on J inputs with a decision tree matching on a single input. To formulate the correctness statement, we remark that the J input values fed to a matrix are sub-values of a common value v, obtained through the accessors $(a_j)^{j \in J}$. For $m = ((a_j)^{j \in J}, \dots)$ we have the following correctness statement:

$$\forall t_S, \forall v_S, \quad \implies \quad m(a_j(v_S))^{j \in J} \approx_{\mathsf{run}} [\![m]\!]_S(v_S)$$

D From target programs to decision trees: symbolic execution

We write $[\![t_T]\!]_T$ for the decision tree of the target program t_T, satisfying a correctness statement similar to the source one:

$$\forall t_T, \forall v_T, \quad t_T(v_T) \approx_{\mathsf{run}} [\![t_T]\!]_T(v_T)$$

E Equivalence checking

During equivalence checking we traverse the two trees, recursively checking equivalence of pairs of subtrees. When we traverse a branch condition, we learn a condition on an accessor that restricts the set of possible input values that can flow in the corresponding subtree. We represent this in our algorithm as an *input domain* S of possible values (a mapping from accessors to target domains).

The equivalence checking algorithm $\mathsf{equiv}(S, D_S, D_T)$ takes an input domain S and a pair of source and target decision trees. In case the two trees are not equivalent, it returns a counter example.

It is defined exactly as a decision procedure for the provability of the judgment $(S \vdash_{[]} D_S \approx D_T)$, defined below in the general form $(S \vdash_G D_S \approx D_T)$ where G is a *guard queue*, indicating an imbalance between the guards observed in the source tree and in the target tree. (For clarity of exposition, the inference rules do not explain how we build the counter-example.)

$$\begin{array}{ccc} \textit{input space} & \textit{boolean result} & \textit{guard queues} \\ S \subseteq \{(v_S, v_T) \mid v_S \approx_{\mathsf{val}} v_T\} & b \in \{0,1\} & G ::= \ (t_1 = b_1), \ldots, (t_n = b_n) \end{array}$$

EMPTY

$$\frac{}{\emptyset \vdash_G D_S \approx D_T} \qquad \frac{}{S \vdash_{[]} \mathsf{Failure} \approx \mathsf{Failure}} \qquad \frac{t_S \approx_{\mathsf{expr}} t_T}{S \vdash_{[]} \mathsf{Leaf}(t_S) \approx \mathsf{Leaf}(t_T)}$$

EXPLODE-LEFT

$$\frac{\begin{array}{c}\forall i,\ (S \cap a = K_i) \vdash_G D_i \approx \mathsf{trim}(D_T, a = K_i) \\ (S \cap a \notin (K_i)^i) \vdash_G D_{\mathsf{fb}} \approx \mathsf{trim}(D_T, S(a) \cap a \notin (K_i)^i)\end{array}}{S \vdash_G \mathsf{Switch}(a, (K_i, D_i)^i, D_{\mathsf{fb}}) \approx D_T}$$

EXPLODE-RIGHT

$$\frac{D_S \in \mathsf{Leaf}(t), \mathsf{Failure} \qquad \forall i,\ (S \cap a \in \pi_i) \vdash_G D_S \approx D_i \qquad (S \cap a \notin (\pi_i)^i) \vdash_G D_S \approx D_{\mathsf{fb}}}{S \vdash_G D_S \approx \mathsf{Switch}(a, (\pi_i)^i D_i, D_{\mathsf{fb}})}$$

$$\frac{S \vdash_{G,(e_S=0)} D_0 \approx D_T \qquad S \vdash_{G,(e_S=1)} D_1 \approx D_T}{S \vdash_G \mathsf{Guard}(e_S, D_0, D_1) \approx D_T} \qquad \frac{e_S \approx_{\mathsf{expr}} e_T \qquad S \vdash_G D_S \approx D_b}{S \vdash_{(e_S=b),G} D_S \approx \mathsf{Guard}(e_T, D_0, D_1)}$$

The EMPTY rule states that two subtrees are equivalent when the input domain is empty. This is used when the two subtrees constrain the input in incompatible ways.

The explosion rules are used when a Switch node occurs on either side. Each child of this switch is tested for equality against the other tree. The branch condition of the child is used to refine the input domain. For example, if a source child is conditioned on the head constructor K, this child is checked in the restricted domain $S \cap a = K$, where $a = K$ is implicitly understood as the set of related value pairs $v_S \approx_{\mathsf{val}} v_T$ where the head constructor of $a(v)$ is K.

As an optimization, in the EXPLODE-LEFT rule we *trim* the target tree, by simplifying the tree in depth with the source condition. Trimming a target tree on a domain π computed from a source constructor K means mapping every branch condition π' of every node of the target tree to the intersection $\pi \cap \pi'$ when the accessors on both sides are equal, and removing the branches that result in an empty intersection. If the accessors are different, π' is left unchanged. Trimming avoids redundant work, because each target subtree removed (in one step) by trimming would have been traversed during the equivalence computation of each source subtree and their children, potentially many times.

We have only defined trimming of a target tree, not of a source tree: the branch condition on the source tree are just constructors, so they are less expressive than target domains and it is hard to define an intersection between the two. We restrict the EXPLODE-RIGHT rule to only work on terminal source trees, so that no duplicate work occurs.

There is no guarantee that the guards will appear at the same tree level on both sides: guards cannot be reordered (their evaluation may perform observable side-effects), but they can be permuted with (non-observable) switches. We store a *guard queue* G that tracks the guard conditions that we have traversed in the source tree, but not yet in the target tree, with the boolean result of each condition. Termination of the algorithm (in the Failure and Leaf rules) is successful only when the guards queue is empty. This ensures that both sides executed the same guards, in the same order.

The algorithm respects the following correctness statement:

$$\begin{aligned}
\mathsf{equiv}(S, D_S, D_T) = \mathsf{Yes} &\implies \forall v_S \approx_{\mathsf{val}} v_T \in S,\ D_S(v_S) \approx_{\mathsf{run}} D_T(v_T) \\
\mathsf{equiv}(S, D_S, D_T) = \mathsf{No}(v_S, v_T) &\implies v_S \approx_{\mathsf{val}} v_T \in S\ \wedge\ D_S(v_S) \not\approx_{run} D_T(v_T)
\end{aligned}$$

The inference rules above do not describe how to build a counter-example in the No case; our algorithm corresponds to a complete inference procedure that always succeeds, and builds a counter-example in the cases where the equivalence judgment does not hold. This is much heavier to describe in inference-rule notation, and best understood by looking at the code of our software prototype.

F Complete procedure

Putting all the pieces together, we have a sound and complete decision procedure to check the equivalence of a source program t_S and target program t_T. The correctness argument follows from composing together the correctness statements of each part.
(We write $\top$ for the full domain of all pairs $v_S \approx_{\mathsf{val}} v_T$.)

$$\mathsf{equiv}(t_S, t_T) \quad := \quad \mathsf{equiv}(\top, [\![t_S]\!]_S, [\![t_T]\!]_T)$$

$$\mathsf{equiv}(t_S, t_T) = \mathsf{Yes} \qquad\qquad \implies$$

$$\forall v_S \approx_{\mathsf{val}} v_T, \qquad t_S(v_S)\ \approx_{\mathsf{run}}\ [\![t_S]\!]_S(v_S)\ \approx_{\mathsf{run}}\ [\![t_T]\!]_T(v_T)\ \approx_{\mathsf{run}}\ (t_T)(v_T)$$

$$\mathsf{equiv}(t_S, t_T) = \mathsf{No}(v_S, v_T) \qquad\qquad \implies$$

$$v_S \approx_{\mathsf{val}} v_T \quad \wedge \qquad t_S(v_S)\ \approx_{\mathsf{run}}\ [\![t_S]\!]_S(v_S)\ \not\approx_{run}\ [\![t_T]\!]_T(v_T)\ \approx_{\mathsf{run}}\ (t_T)(v_T)$$