

Translation validation of a pattern-matching compiler

Francesco Mecca (University of Turin), Gabriel Scherer (INRIA)

August 22, 2020



Checking a pattern-matching compiler

From pattern-matching to simple control-flow.

Not simple: tradeoffs in code speed vs. code size.

Bugs in the compiler: silent wrong-code production.

Painful to detect and diagnose.

In OCaml, three bugs in the last few years.

Afraid to change the compiler.

We want to catch such bugs at compile-time.

Translation-validation: check each source-target pair at compile-time.

Work In Progress: simple patterns + `when`-guards.

Cannot reproduce the bugs yet.

Extensible approach: symbolic execution.

Automated solvers?

Encode patterns (`Foo 42 :: rest`) as formulas over access paths,
Delegate equivalence checking to a solver.

Kirchner, Moreau, and Reilles (2005) use first-order logic and Zenon.

Downsides:

- hard to guess the robustness of solvers on those problems
- hard to scale when pattern-matching is interleaved with arbitrary evaluation:
when guards, pattern guards (Haskell, Successor ML), etc.

Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In *PPDP*, 2005.

Example: source and target programs

Target program: exactly the OCaml `-drawlambda` output.

```
type 'a option =  
  | None  
  | Some of 'a  
  
let mm test ret input =  
  match input with  
  | Some x when test x -> ret x  
  | Some 42 -> ret 42  
  | _ -> ret 0
```

Example: source and target programs

Target program: exactly the OCaml `-drawlambda` output.

```
type 'a option =
| None
| Some of 'a

let mm test ret input =
  match input with
  | Some x when test x -> ret x
  | Some 42 -> ret 42
  | _ -> ret 0

(mm = (function test ret input
      (catch
        (if input
          (let (x =a (field 0 input))
            (if (apply test x)
              (apply ret x)
              (if (!= x 42)
                (exit 1)
                (apply ret 42))))))
      (exit 1))
  with (1)
      (apply ret 0))))
```

Example: source and target programs

Target program: exactly the OCaml `-drawlambda` output.

```
type 'a option =
| None
| Some of 'a

let mm test ret input =
  match input with
  | Some x when test x -> ret x
  | Some 42 -> ret 42
  | _ -> ret 0

(mm = (function test ret input
      (catch
        (if input
          (let (x =a (field 0 input))
            (if (apply test x)
              (apply ret x)
              (if (!= x 42)
                (exit 1)
                (apply ret 42))))))
      (exit 1))
  with (1)
      (apply ret 0))))
```

Pattern-matching. Arbitrary expressions :

Example: source and target programs

Target program: exactly the OCaml `-drawlambda` output.

```
type 'a option =
| None
| Some of 'a

let mm test ret input =
  match input with
  | Some x when test x -> ret x
  | Some 42 -> ret 42
  | _ -> ret 0

(mm = (function test ret input
      (catch
        (if input
          (let (x =a (field 0 input))
            (if (apply test x)
              (apply ret x)
              (if (≠ x 42)
                (exit 1)
                (apply ret 42))))))
        (exit 1))
      with (1)
        (apply ret 0))))
```

Pattern-matching. Arbitrary expressions : only in guards and leaves.

Example: source and target programs

Target program: exactly the OCaml `-drawlambda` output.

```
type 'a option =
| None
| Some of 'a

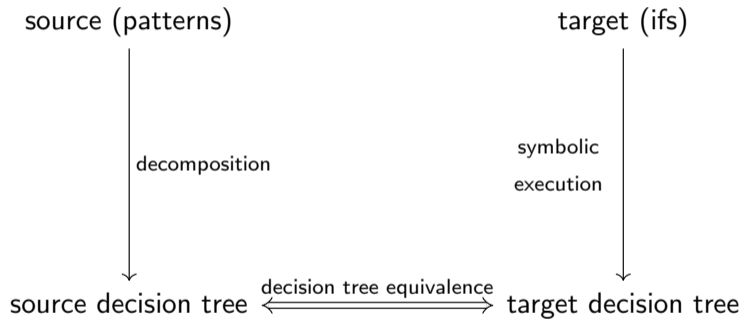
let mm test ret input =
  match input with
  | Some x when test x -> ret x
  | Some 42 -> ret 42
  | _ -> ret 0

(mm = (function test ret input
      (catch
        (if input
          (let (x =a (field 0 input))
            (if (apply test x)
              (apply ret x)
              (if (≠ x 42)
                (exit 1)
                (apply ret 42))))))
        (exit 1))
      with (1)
        (apply ret 0))))
```

Pattern-matching. Arbitrary expressions : only in guards and leaves.

Use the compiler as an oracle on those; check equivalence on the rest.

Our approach



Common representation: decision trees

match input with

| Some x when test x -> ret x

| Some 42 -> ret 42

| _ -> ret 0

Common representation: decision trees

match input with

| Some x when test x -> ret x

| Some 42 -> ret 42

| _ -> ret 0

```
      Switch(Root)
      / None      \ Some
Leaf              Guard
[](ret 0)        [x = Root.0](test x)
                  / true           \ false
                  Leaf              Switch(Root.0)
                  [x = Root.0](ret x) / 42           \ ¬42
                                      Leaf           Leaf
                                      [](ret 42)    [](ret 0)
```

+ Failure

Common representation: decision trees

match input with

| Some x when test x -> ret x

| Some 42 -> ret 42

| _ -> ret 0

```
          Switch(Root)
        /  None    \  Some
      Leaf          Guard
    [](ret 0)      [x = Root.0](test x)
                  / true           \ false
                Leaf              Switch(Root.0)
                [x = Root.0](ret x) / 42           \ ¬42
                                      Leaf          Leaf
                                      [](ret 42)   [](ret 0)
```

+ Failure

Source decision trees test language-level values (None, Some).

Target decision trees test low-level representations (int 0, tag 0).

Equivalence: specification

Heterogeneous equivalence of decision trees:

related source/target values give related results. $(\perp \mid (\sigma, e))$

In particular: tests on accessors may be split and reordered.

But: guards must be checked in the exact same order.

(side-effects: observable evaluation order)

Equivalence: naive

Source/target leaves with compatible path conditions must return the same result.

$$S \vdash D_S \approx D_T \qquad \text{input space} \\ S \subseteq \{(v_S, v_T) \mid v_S \approx_{\text{val}} v_T\}$$

Naive rules:

$$\frac{\forall i, (S \cap a = K_i) \vdash D_i \approx D_T}{S \vdash \text{Switch}(a, (K_i, D_i)^i) \approx D_T} \qquad \frac{\forall i, (S \cap a \in \pi_i) \vdash D_S \approx D_i}{S \vdash D_S \approx \text{Switch}(a, (\pi_i, D_i)^i)}$$

$$\frac{}{\emptyset \vdash D_S \approx D_T} \qquad \frac{S \neq \emptyset \quad t_S \approx_{\text{expr}} t_T}{S \vdash \text{Leaf}(t_S) \approx \text{Leaf}(t_T)} \qquad \frac{S \neq \emptyset}{S \vdash \text{Failure} \approx \text{Failure}}$$

Equivalence: trimming

For each source switch condition, we can *trim* the tree right away.

Shares work. (hb^h rather than b^{2h})

Naive rules:

$$\frac{\forall i, (S \cap a = K_i) \vdash D_i \approx D_T}{S \vdash \text{Switch}(a, (K_i, D_i)^i) \approx D_T}$$

$$\frac{\forall i, (S \cap a \in \pi_i) \vdash D_S \approx D_i}{S \vdash D_S \approx \text{Switch}(a, (\pi_i)^i D_i)}$$

Our rules:

$$\frac{\forall i, (S \cap a = K_i) \vdash D_i \approx \text{trim}(D_T, a = K_i)}{S \vdash \text{Switch}(a, (K_i, D_i)^i) \approx D_T}$$

$$\frac{D_S \in \text{Leaf}(-), \text{Failure} \quad \forall i, (S \cap a \in \pi_i) \vdash D_S \approx D_i}{S \vdash D_S \approx \text{Switch}(a, (\pi_i)^i D_i)}$$

Equivalence: guards

Keep a queue of guards encountered in the source but not in the target yet.

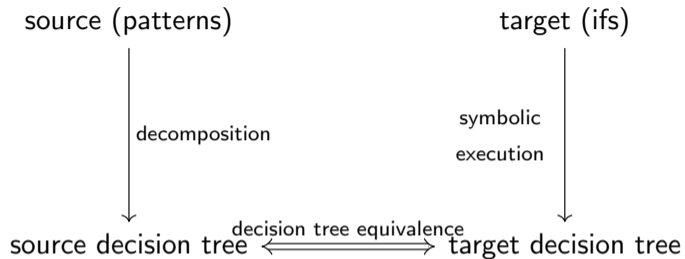
Full judgment: $S \vdash_G D_S \approx D_T$

$$\frac{S \vdash_{G,(e_S=0)} D_0 \approx D_T \quad S \vdash_{G,(e_S=1)} D_1 \approx D_T}{S \vdash_G \text{Guard}(e_S, D_0, D_1) \approx D_T}$$

$$\frac{S \neq \emptyset \quad e_S \approx_{\text{expr}} e_T \quad S \vdash_G D_S \approx D_b}{S \vdash_{(e_S=b),G} D_S \approx \text{Guard}(e_T, D_0, D_1)}$$

Switch rules preserve the guard queue,
non-empty leaf rules require an empty queue.

Conclusion



Work in progress. Future work:

- Exceptions / extensible constructors:
symbolic names with (in)equality assumptions.
- Mutable fields:
forget path conditions on potential mutation.
- Compiler integration.