Translation Verification of the OCaml 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.

1.1 Motivation

Pattern matching in computer science is a widely employed technique for describing computation as well as deduction. Pattern matching is central in many programming languages such as the ML family languages, Haskell and Scala, different model checkers, such as Murphi, and proof assistants such as Coq and Isabelle. Recently the C++ community is considering[1] adding the support for pattern matching in the compiler. The work done in this thesis provides a general method that is agnostic with respect to the compiler implementation and the language used.

The work focused on the OCaml pattern matching compiler that 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 the current code base. For this reason we want to verify that future 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. Much effort has been spent on this topic and whole compilers have been proven either manually[2, 3, 4] or using a proof assistant[5, 6, 7]. 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.

1.2 The Pattern Matching Compiler

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

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

Given as input to the pattern matching compiler, this snippet of code gets translated into the Lambda intermediate representation of the OCaml compiler. The Lambda representation of a program is shown by calling the `ocamlc` compiler with the `-drawlambda` flag. In this example we renamed the variables assigned in order to ease the understanding of the tests that are performed when the code is translated into the Lambda form.

```
(function scrutinee
    (if scrutinee ;;; true when scrutinee (list) not empty
    (let (tail =a (field 1 scrutinee/81)) ;;; assignment
        (if tail
        (let
            y =a (field 0 tail))
            ;;; y is the first element of the tail
            (makeblock 0 2 (makeblock 0 y)))
            ;;; allocate memory for tuple (2, Some y)
        (let (x =a (field 0 scrutinee))
            ;;; x is the head of the scrutinee
```

```
           (makeblock 0 1 (makeblock 0 x)))))
           ;;; allocate memory for tuple (1, Some x)
   [0: 0 0a]))) ;;; low level representatio of (0, None)
```

1.3 Our approach

Our algorithm translates both source and target programs into a common representation that we call `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.

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

`(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 blocks `x::xs` or tuples `(x,y)` are memory blocks with tag 0.

The following parametrized grammar $D(\pi, e)$ describes the common structure of source and decision trees. We denote as π 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 low level OCaml values. Expressions e_S and e_T are arbitrary OCaml expressions that are lowered by the compiler into lambda expressions.

$$\begin{array}{lrcl} \textit{environment} & \sigma(v) & ::= & [x_1 \mapsto v_1, \dots, v_n \mapsto v_n] \\ \textit{closed term} & \underline{e}(v) & ::= & (\sigma(v), e) \\ \textit{accessors} & a & ::= & \mathsf{Root} \mid a.n \quad (n \in \mathbb{N}) \end{array}$$

π_S : datatype constructors
$\pi_T \subseteq \{\mathsf{int}\, n \mid n \in \mathbb{Z}\} \uplus \{\mathsf{tag}\, n \mid n \in \mathbb{N}\}$

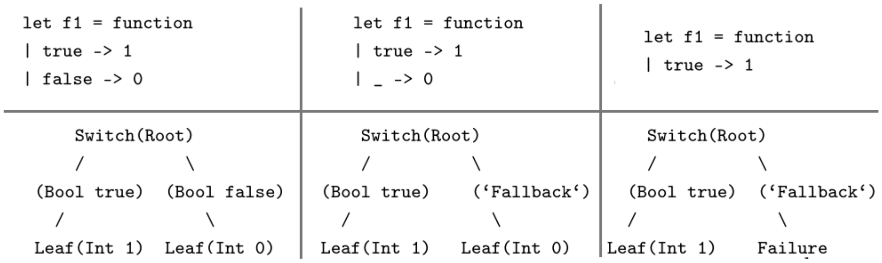
$a(v_S), a(v_T), D_S(v_S), D_T(v_T) \quad (\textit{omitted})$

$$
\begin{aligned}
\textit{decision trees } D(\pi, e) ::= \; & \mathsf{Leaf}(\underline{e}(a)) \\
& \mid \mathsf{Failure} \\
& \mid \mathsf{Switch}(a, (\pi_i, D_i)^{i \in I}, D_{\mathsf{fb}}) \\
& \mid \mathsf{Guard}(\underline{e}(a), D_0, D_1) \\
& \mid \mathit{Unreachable}
\end{aligned}
$$

The tree $\mathsf{Leaf}(\underline{e})$ returns a leaf expression e in a captured environment σ mapping variables to accessors.

$\mathsf{Failure}$ expresses a match failure that occurs when no clause matches the input value.

$\mathsf{Switch}(a, (\pi_i, D_i)^{i \in I}, D_{fallback})$ has one subtree D_i for every head constructor that appears in the pattern matching clauses, and a fallback case that is used when at least one variant of the constructor doesn't appear in the clauses. The presence of the fallback case does not imply non-exhaustive match clauses.



```
let f1 = function          let f1 = function          let f1 = function
| true -> 1                | true -> 1                | true -> 1
| false -> 0               | _ -> 0
```

```
     Switch(Root)               Switch(Root)               Switch(Root)
      /         \                /         \                /         \
(Bool true)  (Bool false)  (Bool true)   ('Fallback')  (Bool true)  ('Fallback')
   /              \           /              \          /               \
Leaf(Int 1)  Leaf(Int 0)   Leaf(Int 1)   Leaf(Int 0)  Leaf(Int 1)     Failure
```

As we can see from these simple examples in which we pattern match on a boolean constructor the fallback node in the second case implicitly covers the path in which the value is equal to false while in the third case the failure terminal signals the presence of non-exahustive clauses.

$\mathsf{Guard}(\underline{e}, D_0, D_1)$ 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.

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 first 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 D_i. For every child (π_i, D_i) we reduce

T by killing all the branches that are incompatible with π_i and check that the reduced tree is equivalent to D_i.

1.4 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 such as $(p_1|p_2)$ and pattern variables. In particular Or-patterns provide a more compact way to group patterns that point to the same expression.

```
match w with
| p1 -> expr                match w with
| p2 -> expr                |p1 |p2 |p3 -> expr
| p3 -> expr
```

We also support `when` guards, which are interesting as they introduce the evaluation of expressions during matching.

We write $\llbracket \mathrm{t}_S \rrbracket_S$ to denote the translation of the source program (the set of pattern matching clauses) into a decision tree, computed by a matrix decomposition algorithm (each column decomposition step gives a `Switch` node). It satisfies the following correctness statement:

$$\forall t_s, \forall v_s, \quad t_s(v_s) = \llbracket t_s \rrbracket_s (v_s)$$

The correctness statement intuitively states that for every source program, for every source value that is well-formed input to a source program, running the program t_S against the input value v_S is the same as running the compiled source program $\llbracket \mathrm{t}_S \rrbracket$ (that is a decision tree) against the same input value v_S.

1.5 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 are black boxes of OCaml code that branches the execution of the symbolic engine. Whenever a guard is met, we emit a Guard node that contains two subtrees, one for each boolean value that can result from the evaluation of the `guard condition` at runtime. The symbolic engine explores both paths because we will see later that for the equivalence checking the computation of the guard condition can be skipped. In comparison with the source decision trees, `Unreachable` nodes are never emitted.

We write $[\![t_T]\!]_T$ to denote the translation of a target program t_T into a decision tree of the target program t_T, satisfying the following correctness statement that is simmetric to the correctness statement for the translation of source programs:

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

1.6 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. Our algorithm respects the following correctness statement:

$$\begin{aligned} \mathsf{equiv}(S, D_S, D_T) = \mathsf{Yes} \ \wedge \ D_T \ \mathsf{covers}\ S &\implies \forall v_S \approx v_T \in S,\ D_S(v_S) = D_T(v_T) \\ \mathsf{equiv}(S, D_S, D_T) = \mathsf{No}(v_S, v_T) \ \wedge \ D_T \ \mathsf{covers}\ S &\implies v_S \approx v_T \in S \ \wedge \ D_S(v_S) \neq D_T(v_T) \end{aligned}$$

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 and for this reason a List.length function can accept lists of integers, lists of strings and in general lists of any type. JVM languages offer polymorphic functions and classes 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[10]. Other languages such as Swift and Java performs type erasure[9, 8] 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 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 reifycated through functors[11].

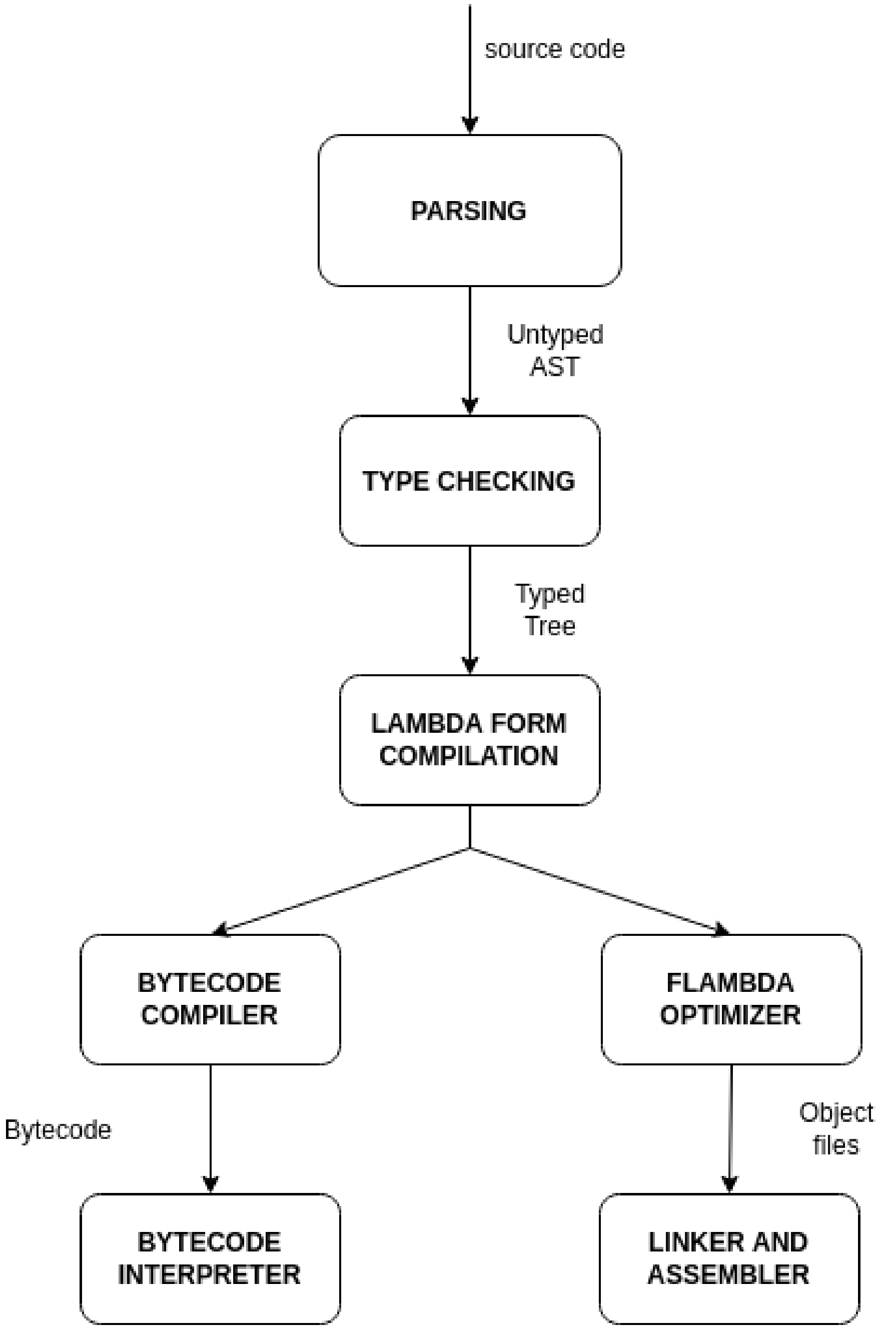
2.2 Compiling OCaml code

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*[12]
- 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[13]. 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[14] before being translated into assembly.

This is an overview of the different compiler steps.



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 *block* or *box*. We can abstract the type of OCaml runtime values as the following:

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

where a one bit tag is used to distinguish between Constant or Block. 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[17] 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 -*drawlambda* flag or in a more compact form 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 allows to allocate low level OCaml values and every constant constructor of 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 blocks 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 defines 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:

- Arithmetic operations: +, -, *, /, % (modulo), neg (unary negation)
- Bitwise operations: &, |, ˆ, «, » (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 "blue"
| Green -> print "green"
| _ -> print "white or black"
```

Pattern matching clauses provide 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

```
begin match tuple with
| (Some _, Some _) -> print "Pair of optional types"
| (Some _, None) | (None, Some _) -> print "Pair of optional types, one of which is null"
| (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 pattern matching compiler emits a warning when a pattern is not exhaustive or some patterns are shadowed by precedent ones.

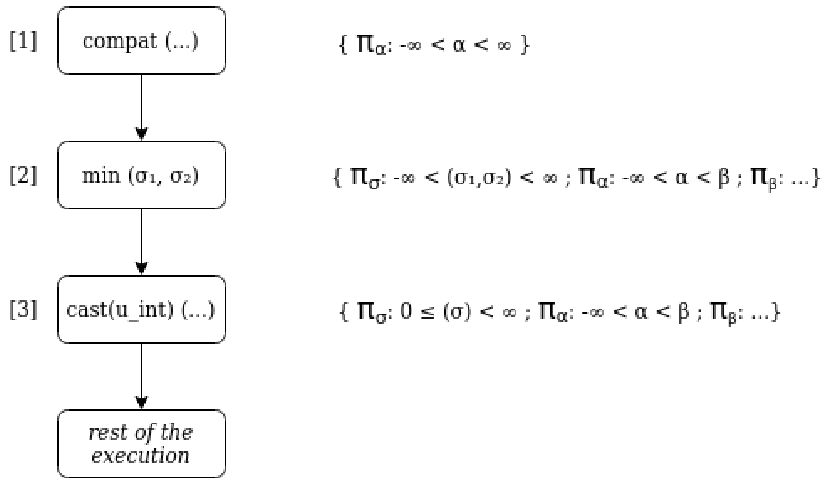
2.6 Symbolic execution

Symbolic execution is a widely used techniques in the field of computer security. It allows to analyze different execution paths of a program simultanously while tracking which inputs trigger the execution of different parts of the program. Inputs are modelled symbolically rather than taking "concrete" values. A symbolic execution engine keeps track of expressions and variables in terms of these symbolic symbols and attaches logical constraints to every branch that is being followed. Symbolic execution engines are used to track bugs by modelling the domain of all possible inputs of a program, detecting infeasible paths, dead code and proving that two code segments are equivalent.

Let's take as example this signedness bug that was found in the FreeBSD kernel[15] and allowed, when calling the *getpeername* function, to read portions of kernel memory.

```
int compat;
{
    struct file *fp;
    register struct socket *so;
    struct sockaddr *sa;
    int len, error;

    ...

    len = MIN(len, sa->sa_len);    /* [1] */
    error = copyout(sa, (caddr_t)uap->asa, (u_int)len);
    if (error)
goto bad;

    ...

bad:
    if (sa)
FREE(sa, M_SONAME);
    fdrop(fp, p);
    return (error);
}
```

The tree of the execution is presented below. It is built by evaluating the function by consider the integer variable *len* the symbolic variable α, sa->sa_len the symbolic variable β and π indicates the set of constraints on the symbolic variables. The input values to the functions are identified by σ.


[1]
compat (...)
$\{ \Pi_{\alpha}: -\infty < \alpha < \infty \}$
[2]
min (σ_1, σ_2)
$\{ \Pi_{\sigma}: -\infty < (\sigma_1,\sigma_2) < \infty ; \Pi_{\alpha}: -\infty < \alpha < \beta ; \Pi_{\beta}: ...\}$
[3]
cast(u_int) (...)
$\{ \Pi_{\sigma}: 0 \leq (\sigma) < \infty ; \Pi_{\alpha}: -\infty < \alpha < \beta ; \Pi_{\beta}: ...\}$
rest of the execution

We can see that at step 3 the set of possible values of the scrutinee α is bigger than the set of possible values of the input σ to the *cast* directive, that is: $\pi_\alpha \nsubseteq \pi_\sigma$. For this reason the *cast* may fail when α is *len* negative number, outside the domain π_σ. In C this would trigger undefined behaviour (signed overflow) that made the exploit possible.

1. Symbolic Execution in terms of Hoare Logic Every step of the evaluation in a symbolic engine can be modelled as the following transition:

   $$(\pi_\sigma, (\pi_i)^i) \rightarrow (\pi'_\sigma, (\pi'_i)^i)$$

   if we express the π transitions as logical formulas we can model the execution of the program in terms of Hoare Logic. The state of the computation is a Hoare triple {P}C{Q} where P and Q are respectively the *precondition* and the *postcondition* that constitute the assertions of the program. C is the directive being executed. The language of the assertions P is:

   $$\mathrm{P} ::= \mathrm{true} \mid \mathrm{false} \mid \mathrm{a} < \mathrm{b} \mid \mathrm{P}_1 \wedge \mathrm{P}_2 \mid \mathrm{P}_1 \vee \mathrm{P}_2 \mid \neg\, \mathrm{P}$$

   where a and b are numbers. In the Hoare rules assertions could also take the form of

   $$\mathrm{P} ::= \forall\ \mathrm{i.}\ \mathrm{P} \mid \exists\ \mathrm{i.}\ \mathrm{P} \mid \mathrm{P}_1 \Rightarrow \mathrm{P}_2$$

   where i is a logical variable, but assertions of these kinds increases the complexity of the symbolic engine. \ Execution follows the following inference rules:

   - Empty statement :

     $$\frac{}{\{P\}skip\{P\}}$$

   - Assignment statement : The truthness of P[a/x] is equivalent to the truth of {P} after the assignment.

     $$\frac{}{\{P[a/x]\}x := a\{P\}}$$

   - Composition : c_1 and c_2 are directives that are executed in order; {Q} is called the *mid condition*.

     $$\frac{\{P\}c_1\{R\}, \{R\}c_2\{Q\}}{\{P\}c_1; c_2\{Q\}}$$

- Conditional :

$$\frac{\{P \wedge b\}c_1\{Q\}, \{P \wedge \neg b\}c_2\{Q\}}{\{P\}\text{if b then } c_1 \text{ else } c_2\{Q\}}$$

- Loop : {P} is the loop invariant. After the loop is finished P holds and ¬b caused the loop to end.

$$\frac{\{P \wedge b\}c\{P\}}{\{P\}\text{while b do c}\{P \wedge \neg b\}}$$

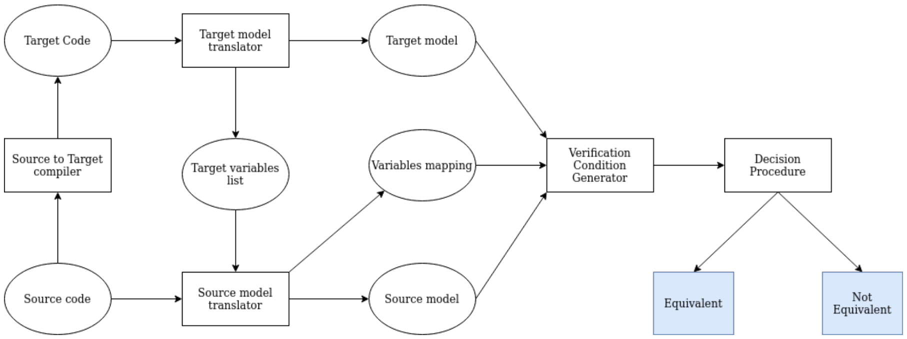
Even if the semantics of symbolic execution engines are well defined, the user may run into different complications when applying such analysis to non trivial codebases[19] For example, depending on the domain, loop termination is not guaranteed. Even when termination is guaranteed, looping causes exponential branching that may lead to path explosion or state explosion. Reasoning about all possible executions of a program is not always feasible and in case of explosion usually symbolic execution engines implement heuristics to reduce the size of the search space[**?**].

2.7 Translation Validation

Translators, such as compilers and code generators, are huge pieces of software usually consisting of multiple subsystem and constructing an actual specification of a translator implementation for formal validation is a very long task. Moreover, different translators implement different algorithms, so the correctness proof of a translator cannot be generalized and reused to prove another translator. Translation validation is an alternative to the verification of existing translators that consists of taking the source and the target (compiled) program and proving *a posteriori* their semantic equivalence.

2.7.1 Translation Validation as Transation Systems

There are many successful attempts at translation validation of code translators[21] and to a less varying degree of compilers[22]. Pnueli et al. provide a computational model based on synchronous transition systems to prove a translation verification tool based on the following model.



The description of the computational models resembles closely the one in [21]. A synchronous transition system (STS) A = (V, Θ, ρ) where

- V: is a finite set of variables; $\sum_v$ is the set of all states over V
- Θ: a satisfiable assertion over the state variables of A, representing its initial state
- ρ: a transition relation computed as an assertion ρ(V, V') that relates a state s$\in\sum_v$ to the successor s'$\in\sum_v$

A computation of A is an infinite sequence σ = (s$_0$, s$_1$, s$_2$, . . .) where

$$\forall \mathrm{i}\in\mathbb{N}\ \mathrm{s}_i\in\sum_v$$
$$\mathrm{s}_0\models\Theta$$
$$\forall \mathrm{i}\in\mathbb{N}\ (\mathrm{s}_i,\ \mathrm{s}_{\mathrm{i+1+}})\models\rho$$

We can give a notion of correct implementation of the Source to Target compiler in the translation validation settings by using the concept of refinement between STS. Let A = (V$_\mathrm{A}$, Θ$_\mathrm{A}$, ρ_A, E$_\mathrm{A}$) and C = (V$_\mathrm{C}$, Θ$_\mathrm{C}$, ρ_C, E$_\mathrm{C}$) be an abstract and a concrete STS where E$_\mathrm{A}\subseteq$V$_\mathrm{A}$ and E$_\mathrm{C}\subseteq$V$_\mathrm{C}$ and are externally observable variables. We call a projection s$^\mathrm{F}$ the state s projected on the subset F$\subseteq$V. An observation of a STS is any infinite sequence of the form (s$_0{}^\mathrm{E}$, s$_1{}^\mathrm{E}$, s$_2{}^\mathrm{E}$, . . .) for a path σ = (s$_0$, s$_1$, s$_2$, . . .). We say that the concrete STS C refines the abstract STS A if Obs(C)$\subseteq$Obs(A). If we can prove a correct mapping of state variables at the target and source levels (as highlighted by the figure) by a function map: V$_\mathrm{A}\mapsto$V$_\mathrm{C}$ we can use inductively prove equivalence using a simulation relation:

$$\bigwedge\nolimits_{s\in V_A^S} s = map(s) \wedge \Theta_C \Rightarrow \Theta_A \text{ (initial condition)}$$
$$\bigwedge\nolimits_{i\in V_A^I} i \wedge \bigwedge\nolimits_{s\in V_A^S} s = map(s) \wedge \rho_A \wedge \rho_C \Rightarrow$$
$$\bigwedge\nolimits_{i\in V_A^S} s' \wedge \bigwedge\nolimits_{o\in V_A^O} o' = map(o') \text{ (step)}$$

Proof is out of scope for this thesis. Our work uses bisimulation to prove equivalence.

3 Translation Validation of the Pattern Matching Compiler

3.1 Accessors

OCaml encourages widespread usage of composite types to encapsulate data. Composite types include *discriminated unions*, of which we have seen different use cases, and *records*, that are a form of *product types* such as *structures* in C.

```
struct Shape {
    int centerx;
    int centery;
    enum ShapeKind kind;
    union {
        struct { int side; };
        struct { int length, height; };
        struct { int radius; };
    };
};
```

```
type shape = {
    x:int;
    y:int;
    kind:shapekind
}
and shapekind
      | Square of int
      | Rect of int * int
      | Circle of int
```

Primitive OCaml datatypes include aggregate types in the form of *tuples* and *lists*. Other aggregate types are built using module functors[16]. Low level *Lambda* untyped constructors of the form

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

provides enough flexibility to encode source higher kinded types. This shouldn't surprise because the *Lambda* language consists of s-expressions. The *field* operation allows to address a *Block* value; the expressions *(field 0 x)* and *(field 1 x)* are equivalent to the Lisp primitives *(car x)* and *(cdr x)* respectively.

```
let value =  1 :: 2 :: 3 :: []
```

```
(field 0 x) = 1
(field 0 (field 1 x)) = 2
(field 0 (field 1 (field 1 x)) = 3
(field 0 (field 1 (field 1 (field 1 x)) = []
```

We can represent the concrete value of a higher kinded type as a flat list of blocks. In the prototype we call this "view" into the value of a datatype the *accessor a*.

$$a ::= \text{Here} \mid n.a$$

Accessors have some resemblance with the low level *Block* values, such as the fact that both don't encode type informations; for example the accessor of a list of integers is structurally equivalent to the accessor of a tuple containing the same elements.

We can intuitively think of the *accessor* as the access path to a value that can be reached by deconstructing the scrutinee. At the source level *accessors* are constructed by inspecting the structure of the patterns at hand. At the target level *accessors* are constructed by compressing the steps taken by the symbolic engine during the evaluation of a value. *Accessors* don't store any kind of information about the concrete value of the scrutinee. Accessors respect the following invariants:

$$\begin{aligned} &\mathrm{v(Here)} = \mathrm{v} \\ &\mathrm{K(v}_i)^i\mathrm{(k.a)} = \mathrm{v}_k\mathrm{(a)} \text{ if } \mathrm{k} \in [0;\mathrm{n}[\end{aligned}$$

We will see in the following chapters how at the source level and the target level a value $\mathrm{v_S}$ and a value $\mathrm{v_T}$ can be deconstructed into a value vector $(v_i)^{i\in I}$ of which we can access the root using the *Here* accessor and we can inspect the k-th element using an accessor of the form $k.a$.

3.2 Source program

The OCaml source code of a pattern matching function has the following form:

$$\begin{aligned} &\text{match variable with} \\ &|\ \text{pattern}_1 \to \text{expr}_1 \\ &|\ \text{pattern}_2 \text{ when guard} \to \text{expr}_2 \\ &|\ \text{pattern}_3 \text{ as var} \to \text{expr}_3 \\ &\vdots \\ &|\ \mathrm{p}_n \to \text{expr}_n \end{aligned}$$

Patterns could or could not be exhaustive.

Pattern matching code could also be written using the more compact form:

$$\begin{aligned} &\text{function} \\ &|\ \text{pattern}_1 \to \text{expr}_1 \\ &|\ \text{pattern}_2 \text{ when guard} \to \text{expr}_2 \\ &|\ \text{pattern}_3 \text{ as var} \to \text{expr}_3 \\ &\vdots \\ &|\ \mathrm{p}_n \to \text{expr}_n \end{aligned}$$

This BNF grammar describes formally the grammar of the source program:

```
start ::= "match" id "with" patterns | "function" patterns
patterns ::= (pattern0|pattern1) pattern1+
;; pattern0 and pattern1 are needed to distinguish the first case in which
;; we can avoid writing the optional vertical line
pattern0 ::= clause
pattern1 ::= "|" clause
clause ::= lexpr "->" rexpr
lexpr ::= rule (ε|condition)
rexpr ::= _code ;; arbitrary code
rule ::= wildcard|variable|constructor_pattern| or_pattern
wildcard ::= "_"
variable ::= identifier
constructor_pattern ::= constructor (rule|ε) (assignment|ε)
constructor ::= int|float|char|string|bool |unit
                |record|exn|objects|ref |list|tuple|array|variant|parameterized_variant ;; data types
or_pattern ::= rule ("|" wildcard|variable|constructor_pattern)+
condition ::= "when" b_guard
assignment ::= "as" id
b_guard ::= ocaml_expression ;; arbitrary code
```

The source program is parsed using the ocaml-compiler-libs[?] library. The result of parsing, when successful, results in a list of clauses and a list of type declarations. Every clause consists of three objects: a left-hand-side that is the kind of pattern expressed, an option guard and a right-hand-side expression. Patterns are encoded in the following way:

pattern	type
_	Wildcard
p_1 as x	Assignment
$c(p_1,p_2,\ldots,p_n)$	Constructor
$(p_1 \| p_2)$	Orpat

Once parsed, the type declarations and the list of clauses are encoded in the form of a matrix that is later evaluated using a matrix decomposition algorithm.

Patterns are of the form

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

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	$\forall v$
$(p_1 \mid p_2)$	$\preccurlyeq$	v	iff $p_1 \preccurlyeq v$ or $p_2 \preccurlyeq v$
$c(p_1, p_2, \ldots, p_a)$	$\preccurlyeq$	$c(v_1, v_2, \ldots, v_a)$	iff $(p_1, p_2, \ldots, p_a) \preccurlyeq (v_1, v_2, \ldots, v_a)$
$(p_1, p_2, \ldots, p_a)$	$\preccurlyeq$	$(v_1, v_2, \ldots, v_a)$	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.

During compilation by the translator, expressions at the right-hand-side are compiled into Lambda code and are referred as lambda code actions l_i.

We define the *pattern matrix* P as the matrix $|m \times n|$ where m is bigger or equal than the number of clauses in the source program and n is equal to the arity of the constructor with the gratest arity.

$$P = \begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \\ p_{2,1} & p_{2,2} & \cdots & p_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ p_{m,1} & p_{m,2} & \cdots & p_{m,n} \end{pmatrix}$$

Every row of P is called a pattern vector $\vec{p_i} = (p_1, p_2, \ldots, p_n)$; in every instance of P pattern vectors appear normalized on the length of the longest pattern vector. Considering the pattern matrix P we say that the value vector $\vec{v} = (v_1, v_2, \ldots, v_i)$ matches the pattern vector p_i in P if and only if the following two conditions are satisfied:

- $p_{i,1}, p_{i,2}, \cdots, p_{i,n} \preccurlyeq (v_1, v_2, \ldots, v_n)$
- $\forall j < i$ $p_{j,1}, p_{j,2}, \cdots, p_{j,n} \npreccurlyeq (v_1, v_2, \ldots, v_n)$

In other words given the pattern vector of the i-th row p_i, we say that p_i matches $\vec{v}$ if every element v_k of $\vec{v}$ is an instance of the corresponding sub-patten $p_{i,k}$ and none of the pattern vectors of the previous rows matches.

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

- Pattern 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

Wit the support of two auxiliary functions

- tail of an ordered family

$$\mathrm{tail}((x_i)^{i \in I}) := (x_i)^{i \neq \min(I)}$$

- first non-⊥ element of an ordered family

$$\begin{aligned} &\mathrm{First}((x_i)^i) := \bot && \text{if } \forall i,\ x_i = \bot \\ &\mathrm{First}((x_i)^i) := x_{\min\{i \mid x_i \neq \bot\}} && \text{if } \exists i,\ x_i \neq \bot \end{aligned}$$

we now define what it means to run a pattern row against a value vector of the same length, that is $(p_i)^i(v_i)^i$

p_i	v_i	$\mathrm{result}_{\mathrm{pat}}$
$\varnothing$	$(\varnothing)$	$[]$
$(\_,\ \mathrm{tail}(p_i)^i)$	(v_i)	$\mathrm{tail}(p_i)^i(\mathrm{tail}(v_i)^i)$
$(x,\ \mathrm{tail}(p_i)^i)$	(v_i)	$\sigma[x \mapsto v_0]$ if $\mathrm{tail}(p_i)^i(\mathrm{tail}(v_i)^i) = \sigma$
$(K(q_j)^j,\ \mathrm{tail}(p_i)^i)$	$(K(v'_j)^j, \mathrm{tail}(v_j)^j)$	$((q_j)^j + \mathrm{tail}(p_i)^i)((v'_j)^j + \mathrm{tail}(v_i)^i)$
$(K(q_j)^j,\ \mathrm{tail}(p_i)^i)$	$(K'(v'_l)^l, \mathrm{tail}(v_j)^j)$	$\bot$ if $K \neq K'$
$(q_1 \mid q_2,\ \mathrm{tail}(p_i)^i)$	$(v_i)^i$	$\mathrm{First}((q_1, \mathrm{tail}(p_i)^i)(v_i)^i,\ (q_2, \mathrm{tail}(p_i)^i)(v_i)^i)$

A source program t_S is a collection of pattern clauses pointing to blackbox *bb* terms. Running a program t_S against an input value v_S, written $t_S(v_S)$ produces a result *r* belonging to the following grammar:

$$\begin{aligned} &t_S ::= (p \to bb)^{i \in I} \\ &t_S(v_S) \to r \\ &r ::= \text{guard list * (Match } (\sigma, bb) \mid \text{NoMatch} \mid \text{Absurd)} \end{aligned}$$

We can define what it means to run an input value v_S against a source program t_S:

$$\begin{aligned} t_S(v_S) &:= \text{NoMatch} \quad \text{if } \forall i,\ p_i(v_S) = \bot \\ t_S(v_S) &= \{\ \text{Absurd if } bb_{i_0} = .\ \text{(refutation clause)} \\ &\qquad \text{Match } (\sigma, bb_{i_0}) \text{ otherwise} \\ &\qquad \text{where } i_o = \min\{i \mid p_i(v_S) \neq \bot\} \end{aligned}$$

Expressions of type *guard* and *bb* are treated as blackboxes of OCaml code. A sound approach for treating these blackboxes would be to inspect the OCaml compiler during translation to Lambda code and extract the blackboxes compiled in their Lambda representation. This would allow to test for structural equality with the counterpart Lambda blackboxes at the target level. Given that this level of introspection is currently not possibile because the OCaml compiler performs the translation of the pattern clauses in a single pass, we decided to restrict the structure of blackboxes to the following (valid) OCaml code:

```
external guard : 'a -> 'b = "guard"
external observe : 'a -> 'b = "observe"
```

We assume the existence of these two external functions *guard* and *observe* with a valid type that lets the user pass any number of arguments to them. All the guards are of the form `guard <arg> <arg> <arg>`, where the <arg> are expressed using the OCaml pattern matching language. Similarly, all the right-hand-side expressions are of the form `observe <arg> <arg> ...` with the same constraints on arguments.

```
(* declaration of an algebraic and recursive datatype t *)
type t = K1 | K2 of t

let _ = function
  | K1 -> observe 0
  | K2 K1 -> observe 1
  | K2 x when guard x -> observe 2 (* guard inspects the x variable *)
  | K2 (K2 x) as y when guard x y -> observe 3
  | K2 _ -> observe 4
```

We note that the right hand side of *observe* is just an arbitrary value and in this case just enumerates the order in which expressions appear. This oversimplification of the structure of arbitrary code blackboxes allows us to test for structural equivalence without querying the compiler during the *TypedTree* to *Lambda* translation phase.

```
  let _ = function
    | K1 -> lambda$_0$
    | K2 K1 -> lambda$_1$
    | K2 x when lambda-guard$_0$ -> lambda$_2$
    | K2 (K2 x) as y when lambda-guard$_1$ -> lambda$_3$
    | K2 _ -> lambda$_4$
```

3.2.1 Matrix decomposition of pattern clauses

We define a new object, the *clause matrix* P → L of size |m x n+1| that associates pattern vectors $\vec{p_i}$ to lambda code action l$_i$.

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

The initial input of the decomposition 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 the *clause matrix* P → L. 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 an empty sequence and 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:

1. 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

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

That means in every lambda action l_i in the P' → L' matrix there is a binding of x_1 to the variable that appears on the pattern $p_{i,1}$. When there are wildcard patterns, bindings are omitted the lambda action l_i remains unchanged.

2. Constructor rule: if all patterns in the first column of P are constructors patterns of the form $k(q_1, q_2, \ldots, q_{n'})$ we define a new matrix, the specialized clause matrix S, by applying the following transformation on every row p:

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

Patterns of the form $q_{i,j}$ matches on the values of the constructor and we define the variables $y_1, y_2, \ldots, y_a$ 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
```

1. 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:

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

results in three rows added to the clause matrix

$$p_{i,1}, p_{i,2}, ..., p_{i,m} \to l$$

$$q_{i,1}, p_{i,2}, ..., p_{i,m} \to l$$

$$r_{i,1}, p_{i,2}, ..., p_{i,m} \to l$$

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

It is important to note that the application of the decomposition algorithm converges. This intuition can be verified by defining the size of the clause matrix P → L as equal to the length of the longest pattern vector $\vec{p_i}$ and the length of a pattern vector as the number of symbols that appear in the clause. While it is very easy to see that the application of rules 1) and 4) produces new matrices of length equal or smaller than the original clause matrix, we can show that:

- with the application of the constructor rule the pattern vector $\vec{p_i}$ loses one symbol after its decomposition:

$$|(\mathrm{p_{i,1}}\ (\mathrm{q_1},\ \mathrm{q_2},\ \ldots,\ \mathrm{q_{n'}}),\ \mathrm{p_{i,2}},\ \mathrm{p_{i,3}},\ \ldots,\ \mathrm{p_{i,n}})| = \mathrm{n} + \mathrm{n'}$$
$$|(\mathrm{q_{i,1}},\ \mathrm{q_{i,2}},\ \ldots,\ \mathrm{q_{i,n'}},\ \mathrm{p_{i,2}},\ \mathrm{p_{i,3}},\ \ldots,\ \mathrm{p_{i,n}})| = \mathrm{n} + \mathrm{n'} - 1$$

- with the application of the orpat rule, we add one row to the clause matrix P → L but the length of a row containing an Or-pattern decreases

$$|P \to L| = \Big|\begin{pmatrix} (p_{1,1}|q_{1,1}) & p_{1,2} & \cdots & p_{1,n} \to l_1 \\ \vdots & \vdots & \ddots & \vdots \to \vdots \end{pmatrix}\Big| = n + 1$$

$$|P' \to L'| = \Big|\begin{pmatrix} p_{1,1} & p_{1,2} & \cdots & p_{1,n} \to l_1 \\ q_{1,1} & p_{1,2} & \cdots & p_{1,n} \to l_1 \\ \vdots & \vdots & \ddots & \vdots \to \vdots \end{pmatrix}\Big| = n$$

In our prototype the source matrix m_S is defined as follows

$$\mathrm{SMatrix\ m}_S := (\mathrm{a}_j)^{\mathrm{j \in J}},\ ((\mathrm{p_{ij}})^{\mathrm{j \in J}} \to \mathrm{bb}_i)^{\mathrm{i \in I}}$$

3.3 Target translation

The target program of the following general form is parsed using a parser generated by Menhir[24], a LR(1) parser generator for the OCaml programming language. Menhir compiles LR(1) a grammar specification, in this case a subset of the Lambda intermediate language, down to OCaml code. This is the grammar of the target language[**?**] (TODO: check menhir grammar)

```
start ::= sexpr
sexpr ::= variable | string | "(" special_form ")"
string ::= "\"" identifier "\"" ;; string between doublequotes
variable ::= identifier
special_form ::= let|catch|if|switch|switch-star|field|apply|isout
let ::= "let" assignment sexpr ;; (assignment sexpr)+ outside of pattern match code
assignment ::= "function" variable variable+ ;; the first variable is the identifier of the function
field ::= "field" digit variable
apply ::= ocaml_lambda_code ;; arbitrary code
catch ::= "catch" sexpr with sexpr
with ::= "with" "(" label ")"
exit ::= "exit" label
switch-star ::= "switch*" variable case*
switch::= "switch" variable case* "default:" sexpr
case ::= "case" casevar ":" sexpr
casevar ::= ("tag"|"int") integer
if ::= "if" bexpr sexpr sexpr
bexpr ::= "(" ("!="|"="\vert{}">"|"<="|">"|"<") sexpr digit | field ")"
label ::= integer
```

The prototype doesn't support strings.

The AST built by the parser is traversed and evaluated by the symbolic execution engine. Given that the target language supports jumps in the form of "catch - exit" blocks the engine tries to evaluate the instructions inside the blocks and stores the result of the partial evaluation into a record. When a jump is encountered, the information at the point allows to finalize the evaluation of the jump block. In the environment the engine also stores bindings to values and functions. For performance reasons the compiler performs integer addition and subtraction on variables that appears inside a *switch* expression in order to have values always start from 0. Let's see an example of this behaviour:

```
let f x = match x with
  | 3 -> "3"
  | 4 -> "4"
  | 5 -> "5"
  | 6 -> "6"
  | _ -> "_"

(let
  (f/80 =
     (function x/81
       (catch
         (let (switcher/83 =a (-3+ x/81))
           (if (isout 3 switcher/83) (exit 1)
             (switch* switcher/83
              case int 0: "3"
              case int 1: "4"
              case int 2: "5"
              case int 3: "6")))
        with (1) "_")))
  (makeblock 0 f/80))
```

The prototype takes into account such transformations and at the end of the symbolic evaluation it traverses the result in order to "undo" such optimization and have accessors of the variables match their intended value directly.

3.4 Decision Trees

We have already given the parametrized grammar for decision trees and we will now show how a decision tree is constructed from source and target programs.

$$\begin{array}{lrll}
\textit{decision trees } D(\pi, e) & ::= & \mathsf{Leaf}(\underline{e}(a)) & \\
 & & \mid \mathsf{Failure} & \pi_S : \text{datatype constructors} \\
 & & \mid \mathsf{Switch}(a, (\pi_i, D_i)^{i \in I}, D)_{fallback} & \pi_T \subseteq \{\mathsf{int}\, n \mid n \in \mathbb{Z}\} \uplus \{\mathsf{tag}\, n \mid n \in \mathbb{N}\} \\
 & & \mid \mathsf{Guard}(\underline{e}(a), D_0, D_1) & \\
 & & \mid \mathit{Unreachable} & a(v_S), a(v_T), D_S(v_S), D_T(v_T) \\
\textit{accessors} \quad a & ::= & \mathsf{Root} \;\mid\; a.n \quad (n \in \mathbb{N}) &
\end{array}$$

While the branches of a decision tree represents intuitively the possible paths that a program can take, branch conditions π_S and π_T represents the shape of possible values that can flow along that path.

3.4.1 From source programs to decision trees

Let's consider some trivial examples:

function true -> 1

is translated to

Switch ([(true, Leaf 1)], Failure)

while

```
function
| true -> 1
| false -> 2
```

will be translated to

```
Switch ([(true, Leaf 1); (false, Leaf 2)])
```

It is possible to produce Unreachable examples by using refutation clauses (a "dot" in the right-hand-side)

```
function
| true -> 1
| false -> 2
| _ -> .
```

that gets translated into

```
Switch ([(true, Leaf 1); (false, Leaf 2)], Unreachable)
```

We trust this annotation, which is reasonable as the type-checker verifies that it indeed holds. We'll see that while we can build Unreachable nodes from source programs, in the lambda target there isn't a construct equivalent to the refutation clause.

Guard nodes of the tree are emitted whenever a guard is found. Guards node contains a blackbox of code that is never evaluated and two branches, one that is taken in case the guard evaluates to true and the other one that contains the path taken when the guard evaluates to false. We say that a translation of a source program to a decision tree is correct when for every possible input, the source program and its respective decision tree produces the same result

$$\forall \mathrm{v}_S,\ \mathrm{t}_S(\mathrm{v}_S) = \llbracket \mathrm{t}_S \rrbracket_S(\mathrm{v}_S)$$

We define the decision tree of source programs $\llbracket \mathrm{t}_S \rrbracket$ in terms of the decision tree of pattern matrices $\llbracket \mathrm{m}_S \rrbracket$ by the following:

$$\llbracket ((\mathrm{p}_i \to \mathrm{bb}_i)^{\mathrm{i}\in\mathrm{I}} \rrbracket := \llbracket (\mathrm{Here}),\ (\mathrm{p}_i \to \mathrm{bb}_i)^{\mathrm{i}\in\mathrm{I}}\ \rrbracket$$

Decision tree computed from pattern matrices respect the following invariant:

$$\forall \mathrm{v}\ (\mathrm{v}_i)^{\mathrm{i}\in\mathrm{I}} = \mathrm{v}(\mathrm{a}_i)^{\mathrm{i}\in\mathrm{I}} \to \mathrm{m}(\mathrm{v}_i)^{\mathrm{i}\in\mathrm{I}} = \llbracket \mathrm{m} \rrbracket(\mathrm{v}) \text{ for } \mathrm{m} = ((\mathrm{a}_i)^{\mathrm{i}\in\mathrm{I}},\ (\mathrm{r}_i)^{\mathrm{i}\in\mathrm{I}})$$

The invariant conveys the fact that OCaml pattern matching values can be deconstructed into a value vector and if we can correctly inspect the value vector elements using the accessor notation we can build a decision tree ⟦m⟧ from a pattern matrix that are equivalent when run against the value at hand.

We proceed to show the correctness of the invariant by a case analysys.

Base cases:

1. [| ∅, $(\varnothing \rightarrow \text{bb}_i)^i$ |] ≡ Leaf bb_i where i := min(I), that is a decision tree [|m|] defined by an empty accessor and empty patterns pointing to blackboxes bb_i. This respects the invariant because a source matrix in the case of empty rows returns the first expression and $(Leaf bb)(v) := Match\ bb$
2. [| $(\text{a}_j)^j$, ∅ |] ≡ Failure, as it is the case with the matrix decomposition algorithm

Regarding non base cases: Let's first define some auxiliary functions

- The index family of a constructor: $Idx(K) := [0; arity(K)[$
- head of an ordered family (we write x for any object here, value, pattern etc.): $head((x_i)^{i \in I}) = x_{min(I)}$
- tail of an ordered family: $tail((x_i)^{i \in I}) := (x_i)^{i \neq min(I)}$
- head constructor of a value or pattern:

$$\begin{aligned}&\text{constr}(\text{K}(\text{x}_i)^i) = \text{K}\\&\text{constr}(\_) = \bot\\&\text{constr}(\text{x}) = \bot\end{aligned}$$

- first non-⊥ element of an ordered family:

$$\begin{aligned}&\text{First}((\text{x}_i)^i) := \bot &&\text{if } \forall \text{i}, \text{x}_i = \bot\\&\text{First}((\text{x}_i)^i) := \text{x\_min}\{\text{i} \mid \text{x}_i \neq \bot\} &&\text{if } \exists \text{i}, \text{x}_i \neq \bot\end{aligned}$$

- definition of group decomposition:

```
let $\mathrm{constrs}((\mathrm{p}_i)^{\mathrm{i} \in \mathrm{I}}) = \{ \mathrm{K} \mid \mathrm{K} = \mathrm{constr}(\mathrm{p}_i), \mathrm{i} \in \mathrm{I} \}$
let Groups(m) where $\mathrm{m} = ((\mathrm{a}_i)^i\ ((\mathrm{p}_{ij})^i \to \mathrm{e}_j)^{ij}) =$
    let $(\mathrm{K}_k)^k = \mathrm{constrs}(\mathrm{p}_{i0})^i$ in
    ( $\mathrm{K}_k \to$
        $((\mathrm{a}_{0 \cdot l})^l + \mathrm{tail}(\mathrm{a}_i)^i)$
        (
        if $\mathrm{p}_{oj}$ is $\mathrm{K}_k(\mathrm{q}_l)$ then
            $(\mathrm{q}_l)^l + \mathrm{tail}(\mathrm{p}_{ij})^i \to \mathrm{e}_j$
        if $\mathrm{p}_{oj}$ is _ then
            $(\_)^l + \mathrm{tail}(\mathrm{p}_{ij})^i \to \mathrm{e}_j$
        else $\perp$
        $)^j$
    ), (
        $\mathrm{tail}(\mathrm{a}_i)^i, (\mathrm{tail}(\mathrm{p}_{ij})^i \to \mathrm{e}_j$ if $\mathrm{p}_{0j}$ is _ else $\perp)^j$
    )
```

Groups(m) is an auxiliary function that decomposes a matrix m into submatrices, according to the head constructor of their first pattern. Groups(m) returns one submatrix m_r for each head constructor K that occurs on the first row of m, plus one "wildcard submatrix" $\mathrm{m}_{\mathrm{wild}}$ that matches on all values that do not start with one of those head constructors.
Intuitively, m is equivalent to its decomposition in the following sense: if the first pattern of an input vector (v_i)^i starts with one of the head constructors K_k, then running (v_i)^i against m is the same as running it against the submatrix $\mathrm{m}_{\mathrm{K}_k}$; otherwise (when its head constructor is not one of $(\mathrm{K}_k)^k$) it is equivalent to running it against the wildcard submatrix.

We formalize this intuition as follows

1. Lemma (Groups): Let m be a matrix with

   $$\mathrm{Groups(m)} = (\mathrm{k}_r \to \mathrm{m}_r)\hat{}\mathrm{k},\ \mathrm{m}_{\mathrm{wild}}$$

   For any value vector $(v_i)^l$ such that $v_0 = k(v'_l)^l$ for some constructor k, we have:

   $$\begin{array}{l} \text{if } \mathrm{k} = \mathrm{k}_k \text{ for some k then} \\ \quad \mathrm{m}(\mathrm{v}_i)^i = \mathrm{m}_k((\mathrm{v_l}')^l + (\mathrm{v_i})^{\mathrm{i} \in \mathrm{I} \backslash \{0\}}) \\ \text{else} \\ \quad \mathrm{m}(\mathrm{v}_i)^i = \mathrm{m}_{\mathrm{wild}}(\mathrm{v}_i)^{\mathrm{i} \in \mathrm{I} \backslash \{0\}} \end{array}$$

2. Proof: Let m be a matrix $((\mathrm{a}_i)^i, ((\mathrm{p}_{ij})^i \to \mathrm{e}_j)^j)$ with

$$\mathrm{Groups(m)} = (\mathrm{K}_k \to \mathrm{m}_k)^k, \mathrm{m}_{\mathrm{wild}}$$

Below we are going to assume that m is a simplified matrix such that the first row does not contain an or-pattern or a binding to a variable.

Let $(\mathrm{v}_i)^i$ be an input matrix with $\mathrm{v}_0 = \mathrm{K}_v(\mathrm{v'}_{\mathrm{l}})^l$ for some constructor K_v. We have to show that:

- if $\mathrm{K}_k = \mathrm{K}_v$ for some $\mathrm{K}_k \in \mathrm{constrs}(\mathrm{p}_{0j})^j$, then

$$\mathrm{m}(\mathrm{v}_i)^i = \mathrm{m}_k((\mathrm{v'}_l)^l + \mathrm{tail}(\mathrm{v}_i)^i)$$

- otherwise $\mathrm{m}(\mathrm{v}_i)^i = \mathrm{m}_{\mathrm{wild}}(\mathrm{tail}(\mathrm{v}_i)^i)$

Let us call (r_{kj}) the j-th row of the submatrix m_k, and $\mathrm{r}_{j\mathrm{wild}}$ the j-th row of the wildcard submatrix $\mathrm{m}_{\mathrm{wild}}$.

Our goal contains same-behavior equalities between matrices, for a fixed input vector $(\mathrm{v}_i)^i$. It suffices to show same-behavior equalities between each row of the matrices for this input vector. We show that for any j,

- if $\mathrm{K}_k = \mathrm{K}_v$ for some $\mathrm{K}_k \in \mathrm{constrs}(\mathrm{p}_{0j})^j$, then

$$(\mathrm{p}_{ij})^i(\mathrm{v}_i)^i = \mathrm{r}_{kj}((\mathrm{v'}_l)^l + \mathrm{tail}(\mathrm{v}_i)^i$$

- otherwise

$$(\mathrm{p}_{ij})^i(\mathrm{v}_i)^i = \mathrm{r}_{j\mathrm{wild}}\ \mathrm{tail}(\mathrm{v}_i)^i$$

In the first case (K_v is K_k for some $\mathrm{K}_k \in \mathrm{constrs}(\mathrm{p}_{0j})^j$), we have to prove that

$$(\mathrm{p}_{ij})^i(\mathrm{v}_i)^i = \mathrm{r}_{kj}((\mathrm{v'}_l)^l + \mathrm{tail}(\mathrm{v}_i)^i$$

By definition of m_k we know that r_{kj} is equal to

```
if $\mathrm{p}_{oj}$ is $\mathrm{K}_k(\mathrm{q}_l)$ then
  $(\mathrm{q}_l)^l + \mathrm{tail}(\mathrm{p}_{ij})^i \to \mathrm{e}_j$
if $\mathrm{p}_{oj}$ is _ then
  $(\_)^l + \mathrm{tail}(\mathrm{p}_{ij})^i \to \mathrm{e}_j$
else $\bot$
```

By definition of $(\mathrm{p}_i)^i(\mathrm{v}_i)^i$ we know that $(\mathrm{p}_{ij})^i(\mathrm{v}_i)$ is equal to

$(\mathrm{K}(\mathrm{q}_j)^j,\ \mathrm{tail}(\mathrm{p}_{ij})^i)\ (\mathrm{K}(\mathrm{v'}_l)^l, \mathrm{tail}(\mathrm{v}_i)^i) := ((\mathrm{q}_j)^j + \mathrm{tail}(\mathrm{p}_{ij})^i)((\mathrm{v'}_l)^l + \mathrm{tail}(\mathrm{v}_i)^i)$
$(\_,\ \mathrm{tail}(\mathrm{p}_{ij})^i)\ (\mathrm{v}_i) := \mathrm{tail}(\mathrm{p}_{ij})^i(\mathrm{tail}(\mathrm{v}_i)^i)$
$(\mathrm{K}(\mathrm{q}_j)^j,\ \mathrm{tail}(\mathrm{p}_{ij})^i)\ (\mathrm{K'}(\mathrm{v'}_l)^l, \mathrm{tail}(\mathrm{v}_j)^j) := \bot$ if K $\neq$ K'

We prove this first case by a second case analysis on p_{0j}.

TODO

In the second case (K_v is distinct from K_k for all $\mathrm{K}_k \in \mathrm{constrs}(\mathrm{p}_{oj})^j$), we have to prove that

$$(\mathrm{p}_{ij})^i(\mathrm{v}_i)^i = \mathrm{r}_{j\mathrm{wild}}\ \mathrm{tail}(\mathrm{v}_i)^i$$

3.4.2 From target programs to target decision trees

Symbolic Values during symbolic evaluation have the following form

$$\mathrm{v}_T ::= \mathrm{Block}(\mathrm{tag} \in \mathbb{N},\ (\mathrm{v}_i)^{\mathrm{i}\in\mathrm{I}}) \mid \mathrm{n} \in \mathbb{N}$$

The result of the symbolic evaluation is a target decision tree D_T

$$\mathrm{D}_T ::= \mathrm{Leaf\ bb} \mid \mathrm{Switch}(\mathrm{a},\ (\pi_i \to \mathrm{D}_i)^{\mathrm{i}\in\mathrm{S}}\ ,\ \mathrm{D?}) \mid \mathrm{Failure}$$

Every branch of the decision tree is "constrained" by a domain π_T that intuitively tells us the set of possible values that can "flow" through that path.

$$\mathrm{Domain}\ \pi = \{\ \mathrm{n}|\mathrm{n}{\in}\mathbb{N}\ \mathrm{x}\ \mathrm{n}|\mathrm{n}{\in}\mathrm{Tag}{\subseteq}\mathbb{N}\ \}$$

π_T conditions are refined by the engine during the evaluation; at the root of the decision tree the domain corresponds to the set of possible values that the type of the function can hold. D? is the fallback node of the tree that is taken whenever the value at that point of the execution can't flow to any other subbranch. Intuitively, the π_{fallback} condition of the D? fallback node is

$$\pi_{fallback} = \neg \bigcup_{i\in I} \pi_i$$

The fallback node can be omitted in the case where the domain of the children nodes correspond to set of possible values pointed by the accessor at that point of the execution

$$domain(v_s(a)) = \bigcup_{i \in I} pi_i$$

We say that a translation of a target program to a decision tree is correct when for every possible input, the target program and its respective decision tree produces the same result

$$\forall \mathrm{v}_T,\ \mathrm{t}_T(\mathrm{v}_T) = [\![\mathrm{t}_T]\!]_T(\mathrm{v}_T)$$

3.5 Equivalence checking

3.5.1 Introductory remarks

We assume as given an equivalence relation $e_S \approx_{\mathsf{expr}} e_T$ on expressions (that are contained in the leaves of the decision trees). As highlighted before, in the prototype we use a simple structural equivalence between *observe* expressions but ideally we could query the OCaml compiler and compare the blackboxes in Lambda form. We already defined what it means to run a value v_S against a program t_S:

$$\begin{array}{l}
\mathrm{t}_S(\mathrm{v}_S) := \text{NoMatch} \quad \text{if } \forall \text{i},\ \mathrm{p}_i(\mathrm{v}_S) = \bot \\
\mathrm{t}_S(\mathrm{v}_S) = \{\ \text{Absurd if } \mathrm{bb}_{\mathrm{i}_0} = .\ \text{(refutation clause)} \\
\qquad\qquad\quad \text{Match } (\sigma,\ \mathrm{bb}_{\mathrm{i}_0}) \text{ otherwise} \\
\qquad\qquad\quad \text{where } \mathrm{i}_o = \min\{\mathrm{i} \mid \mathrm{p}_i(\mathrm{v}_S) \neq \bot\}
\end{array}$$

and simmetrically, to run a value v_T against a target program t_T:

We denote as $v_S \approx_{\mathsf{val}} v_T$ the equivalence relation between a source value v_S and a target value v_T. The equivalence relation is proven by structural induction.

$$\overset{\textsc{integers}}{\dfrac{}{i \approx_{\mathsf{val}} (i)}} \qquad \overset{\textsc{boolean}}{\dfrac{}{false \approx_{\mathsf{val}} (0)}} \qquad \overset{\textsc{boolean}}{\dfrac{}{true \approx_{\mathsf{val}} (1)}} \qquad \overset{\textsc{unit value}}{\dfrac{}{() \approx_{\mathsf{val}} (0)}} \qquad \overset{\textsc{Empty List}}{\dfrac{}{[] \approx_{\mathsf{val}} (0)}}$$

$$\overset{\textsc{List}}{\dfrac{v_S \approx_{\mathsf{val}} v_T \qquad v'_S \approx_{\mathsf{val}} v'_T}{[v_S; v'_S] \approx_{\mathsf{val}} (block\ \ v_T\ \ v'_T)}} \qquad \overset{\textsc{Tuple}}{\dfrac{v_S \approx_{\mathsf{val}} v_T \qquad v'_S \approx_{\mathsf{val}} v'_T}{(v_S; v'_S) \approx_{\mathsf{val}} (block\ \ v_T\ \ v'_T)}}$$

$$\overset{\textsc{Record}}{\dfrac{v_S \approx_{\mathsf{val}} v_T \qquad v'_S \approx_{\mathsf{val}} v'_T}{\{v_S; v'_S\} \approx_{\mathsf{val}} (block\ \ v_T\ \ v'_T)}} \qquad \overset{\textsc{constant constructor}}{\dfrac{K_i \in Variant}{K_i \approx_{\mathsf{val}} (i)}}$$

$$\overset{\textsc{Variant}}{\dfrac{K_i \in Variant \qquad v_S \approx_{\mathsf{val}} v_T}{K_i v_S \approx_{\mathsf{val}} (block\ \ (tag\ \ i)\ \ v_T)}}$$

The relation $v_S \approx_{\mathsf{val}} v_T$ 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 $(\underline{e}_1, \ldots, \underline{e}_n)$ of the executed guards and a *matching result* r.

$$e_S \approx_{\mathsf{expr}} e_T \ (\mathit{assumed}) \qquad\qquad t_S(v_S),\ t_T(v_T),\ v_S \approx_{\mathsf{val}} v_T$$

$$r_S \approx_{\mathsf{res}} r_T, R_S \approx_{\mathsf{run}} R_T \ (\mathit{simple})$$

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

$$\frac{\forall x,\ \sigma_S(x) \approx_{\mathsf{val}} \sigma_T(x)}{\sigma_S \approx_{\mathsf{env}} \sigma_T} \qquad\qquad \frac{\sigma_S \approx_{\mathsf{env}} \sigma_T \qquad e_S \approx_{\mathsf{expr}} e_T}{(\sigma_S, e_S) \approx_{\mathsf{cl-expr}} (\sigma_T, e_T)}$$

$$\frac{\forall v_S \approx_{\mathsf{val}} v_T, \quad t_S(v_S) \approx_{\mathsf{run}} t_T(v_T)}{t_S \approx_{\mathsf{prog}} t_T}$$

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

3.5.2 The equivalence checking algorithm

During the equivalence checking phase 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.)

$$\begin{array}{cc} \textit{input space} & \textit{boolean result} \\ S \subseteq \{(v_S, v_T) \mid v_S \approx_{\mathsf{val}} v_T\} & b \in \{No, Yes\} \end{array}$$

$$\begin{array}{c} \textit{guard queues} \\ G ::= \ (t_1 = b_1), \ldots, (t_n = b_n) \end{array}$$

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

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

If the source decision tree (left hand side) is a terminal while the target decisiorn 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.

$$\textsc{explode-right}\quad \frac{\begin{array}{c} D_S \in \mathsf{Leaf}(t), \mathsf{Failure} \\ \forall i,\ (S \cap a \in \pi_i) \vdash_G D_S \approx D_i \qquad (S \cap a \notin (\pi_i)^i) \vdash_G D_S \approx D_{fallback} \end{array}}{S \vdash_G D_S \approx \mathsf{Switch}(a, (\pi_i)^i D_i, D_{\mathsf{fb}})}$$

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.

$$\textsc{explode-right}\quad \frac{\begin{array}{c} D_S \in \mathsf{Leaf}(t), \mathsf{Failure} \\ \forall i,\ (S \cap a \in \pi_i) \vdash_G D_S \approx D_i \qquad (S \cap a \notin (\pi_i)^i) \vdash_G D_S \approx D_{\mathsf{fb}} \end{array}}{S \vdash_G D_S \approx \mathsf{Switch}(a, (\pi_i)^i D_i, D_{\mathsf{fb}})}$$

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.

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

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

Our equivalence-checking algorithm is a exactly decision procedure for the provability of the judgment $\mathsf{equiv}(S, C_S, C_T)$, defined by the previous inference rules. Running a program t_S or its translation $[\![\mathrm{t}_S]\!]$ against an input v_S produces as a result r in the following way:

$$(\ [\![\mathrm{t}_S]\!]_S(\mathrm{v}_S) \equiv \mathrm{D}_S(\mathrm{v}_S)\ ) \to \mathrm{r}$$
$$\mathrm{t}_S(\mathrm{v}_S) \to \mathrm{r}$$

Likewise

$(\ [\![\mathrm{t}_T]\!]_T(\mathrm{v}_T) \equiv \mathrm{D}_T(\mathrm{v}_T)\ ) \to \mathrm{r}$
$\mathrm{t}_T(\mathrm{v}_T) \to \mathrm{r}$
result r ::= guard list * (Match blackbox | NoMatch | Absurd)
guard ::= blackbox.

Having defined equivalence between two inputs of which one is expressed in the source language and the other in the target language, $v_S \approx_{\mathsf{val}} v_T$, we can define the equivalence between a couple of programs or a couple of decision trees

$$t_S \approx_{\mathsf{prog}} t_T := \forall v_S \approx_{\mathsf{val}} v_T, t_S(v_S) = t_T(v_T)$$
$$D_S \approx D_T := \forall v_S \approx_{\mathsf{val}} v_T, D_S(v_S) = D_T(v_T)$$

The result of the proposed equivalence algorithm is *Yes* or *No(v_S, v_T)*. In particular, in the negative case, v_S and v_T are a couple of possible counter examples for which the decision trees produce a different result.

In the presence of guards we can say that two results are equivalent modulo the guards queue, written $r_1 \simeq gs\ r_2$, when:

$$(\mathrm{gs}_1,\ \mathrm{r}_1) \simeq_{\mathsf{gs}} (\mathrm{gs}_2,\ \mathrm{r}_2) \Leftrightarrow (\mathrm{gs}_1,\ \mathrm{r}_1) = (\mathrm{gs}_2 + \mathrm{gs},\ \mathrm{r}_2)$$

We say that D_T covers the input space *S*, written *covers(D_T, S)* when every value $\mathrm{v}_S \in \mathrm{S}$ is a valid input to the decision tree D_T. (TODO: rephrase) Given an input space *S* and a couple of decision trees, where the target decision tree D_T covers the input space *S* we can define equivalence:

$$equiv(S, C_S, C_T, gs) = Yes \land covers(C_T, S) \to \forall v_S \approx_{\mathsf{val}} v_T \in S, C_S(v_S) \simeq_{gs} C_T(v_T)$$

Similarly we say that a couple of decision trees in the presence of an input space S are *not* equivalent in the following way:

$$equiv(S, C_S, C_T, gs) = No(v_S, v_T) \land covers(C_T, S) \to v_S \approx_{\mathsf{val}} v_T \in S \land C_S(v_S) \neq_{gs} C_T(v_T)$$

Corollary: For a full input space S, that is the universe of the target program:

$$equiv(S, [\![t_S]\!]_S, [\![t_T]\!]_T, \emptyset) = Yes \Leftrightarrow t_S \approx_{\mathsf{prog}} t_T$$

3.5.3 The trimming lemma

The trimming lemma allows to reduce the size of a decision tree given an accessor $a \to \pi$ relation (TODO: expand)

$$\forall \mathrm{v}_T \in (\mathrm{a}{\to}\pi),\ \mathrm{D}_T(\mathrm{v}_T) = \mathrm{D}_{\mathrm{t/a}\to\pi}(\mathrm{v}_T)$$

We prove this by induction on C_T:

- D_T = $\mathrm{Leaf}_{\mathrm{bb}}$: when the decision tree is a leaf terminal, the result of trimming on a Leaf is the Leaf itself

$$\mathrm{Leaf}_{\mathrm{bb/a}\to\pi}(\mathrm{v}) = \mathrm{Leaf}_{\mathrm{bb}}(\mathrm{v})$$

- The same applies to Failure terminal

$$\mathrm{Failure}_{/\mathrm{a}\to\pi}(\mathrm{v}) = \mathrm{Failure}(\mathrm{v})$$

- When D_T = $\mathrm{Switch(b, }(\pi{\to}\mathrm{D}_i)^i)_{/\mathrm{a}\to\pi}$ then we look at the accessor a of the subtree D_i and we define π_i' = π_i if a≠b else $\pi_i \cap \pi$ Trimming a switch node yields the following result:

$$\mathrm{Switch(b, }(\pi{\to}\mathrm{D}_i)^{\mathrm{i}\in\mathrm{I}})_{/\mathrm{a}\to\pi} := \mathrm{Switch(b, }(\pi\text{'}_i{\to}\mathrm{D}_{\mathrm{i/a}\to\pi})^{\mathrm{i}\in\mathrm{I}})$$

For the trimming lemma we have to prove that running the value v_T against the decision tree D_T is the same as running v_T against the tree $\mathrm{D}_{\mathrm{trim}}$ that is the result of the trimming operation on D_T

$$\mathrm{D}_T(\mathrm{v}_T) = \mathrm{D}_{\mathrm{trim}}(\mathrm{v}_T) = \mathrm{Switch(b, }(\pi_i\text{'}{\to}\mathrm{D}_{\mathrm{i/a}\to\pi})^{\mathrm{i}\in\mathrm{I}})(\mathrm{v}_T)$$

We can reason by first noting that when $\mathrm{v}_T \notin (\mathrm{b}{\to}\pi_i)^i$ the node must be a Failure node. In the case where $\exists$k | $\mathrm{v}_T \in (\mathrm{b}{\to}\pi_k)$ then we can prove that

$$D_{k/a\to\pi}(v_T) = \text{Switch}(b,\ (\pi_i\text{'}\to D_{i/a\to\pi})^{i\in I})(v_T)$$

because when a $\neq$ b then π_k'$= \pi_k$ and this means that $v_T \in \pi_k$' while when a = b then π_k'$=(\pi_k \cap \pi)$ and $v_T \in \pi_k$' because:

- by the hypothesis, $v_T \in \pi$
- we are in the case where $v_T \in \pi_k$

So $v_T \in \pi_k$' and by induction

$$D_k(v_T) = D_{k/a\to\pi}(v_T)$$

We also know that $\forall v_T \in (b \to \pi_k) \to D_T(v_T) = D_k(v_T)$ By putting together the last two steps, we have proven the trimming lemma.

3.5.4 Equivalence checking

The equivalence checking algorithm takes as parameters an input space *S*, a source decision tree D_S and a target decision tree D_T:

$$\text{equiv}(S,\ D_S,\ D_T) \to \text{Yes} \mid \text{No}(v_S,\ v_T)$$

When the algorithm returns Yes and the input space is covered by D_S we can say that the couple of decision trees are the same for every couple of source value v_S and target value v_T that are equivalent.

$$\text{equiv}(S,\ D_S,\ D_T) = \text{Yes and cover}(D_T,\ S) \to \forall\ v_S \simeq v_T \in S \land D_S(v_S) = D_T(v_T)$$

In the case where the algorithm returns No we have at least a couple of counter example values v_S and v_T for which the two decision trees outputs a different result.

$$\text{equiv}(S,\ D_S,\ C_T) = \text{No}(v_S,v_T) \text{ and cover}(D_T,\ S) \to \forall\ v_S \simeq v_T \in S \land D_S(v_S) \neq D_T(v_T)$$

We define the following

$$\begin{aligned}&\text{Forall(Yes)} = \text{Yes}\\&\text{Forall(Yes::l)} = \text{Forall(l)}\\&\text{Forall}(\text{No}(v_S,v_T)\text{::\_}) = \text{No}(v_S,v_T)\end{aligned}$$

There exists and are injective:

$$\mathrm{int(k)} \in \mathbb{N}\ (\mathrm{arity(k)} = 0)$$
$$\mathrm{tag(k)} \in \mathbb{N}\ (\mathrm{arity(k)} > 0)$$
$$\pi(\mathrm{k}) = \{\mathrm{n} |\ \mathrm{int(k)} = \mathrm{n}\}\ \mathrm{x}\ \{\mathrm{n} |\ \mathrm{tag(k)} = \mathrm{n}\}$$

where k is a constructor.

We proceed by case analysis:

1. in case of unreachable:

$$\mathrm{D}_S(\mathrm{v}_S) = \mathrm{Absurd(Unreachable)} \neq \mathrm{D}_T(\mathrm{v}_T)\ \forall \mathrm{v}_S, \mathrm{v}_T$$

1. In the case of an empty input space

$$\mathrm{equiv}(\varnothing,\ \mathrm{D}_S,\ \mathrm{D}_T) := \mathrm{Yes}$$

   and that is trivial to prove because there is no pair of values (v_S, v_T) that could be tested against the decision trees. In the other subcases S is always non-empty.

2. When there are *Failure* nodes at both sides the result is *Yes*:

$$\mathrm{equiv(S,\ Failure,\ Failure)} := \mathrm{Yes}$$

   Given that ∀v, Failure(v) = Failure, the statement holds.

3. When we have a Leaf or a Failure at the left side:

$$\mathrm{equiv(S,\ Failure\ as\ D}_S\mathrm{,\ Switch(a,\ (}\pi_i \rightarrow \mathrm{D}_{Ti})^{\mathrm{i}\in\mathrm{I}})) := \mathrm{Forall(equiv(\ S}\cap\mathrm{a}\rightarrow\pi(\mathrm{k}_i)),\ \mathrm{D}_S,\ \mathrm{D}_{Ti})^{\mathrm{i}\in\mathrm{I}})$$
$$\mathrm{equiv(S,\ Leaf\ bb}_S\mathrm{\ as\ D}_S\mathrm{,\ Switch(a,\ (}\pi_i \rightarrow \mathrm{D}_{Ti})^{\mathrm{i}\in\mathrm{I}})) := \mathrm{Forall(equiv(\ S}\cap\mathrm{a}\rightarrow\pi(\mathrm{k}_i)),\ \mathrm{D}_S,\ \mathrm{D}_{Ti})^{\mathrm{i}\in\mathrm{I}})$$

   Our algorithm either returns Yes for every sub-input space $\mathrm{S}_i := \mathrm{S}\cap(\mathrm{a}\rightarrow\pi(\mathrm{k}_i))$ and subtree C_{Ti}

$$\mathrm{equiv}(\mathrm{S}_i,\ \mathrm{D}_S,\ \mathrm{D}_{Ti}) = \mathrm{Yes}\ \forall\mathrm{i}$$

   or we have a counter example v_S, v_T for which

$$v_S \approx_{\mathsf{val}} v_T \in S_k \wedge D_S(v_S) \neq D_{Tk}(v_T)$$

   then because $v_T \in (a \rightarrow \pi_k) \rightarrow D_T(v_T) = D_{Tk}(v_T)$ and $v \approx_{\mathsf{val}S} v_T \in S \wedge D_S(v_S) \neq D_T(v_T)$ we can say that

$$\mathrm{equiv}(\mathrm{S}_i,\ \mathrm{C}_S,\ \mathrm{C}_{Ti}) = \mathrm{No}(\mathrm{v}_S,\ \mathrm{v}_T)\ \mathrm{for\ some\ minimal\ k}\in\mathrm{I}$$

4. When we have a Switch on the right we define π_{fallback} as the domain of values not covered but the union of the constructors k_i

$$\pi_{fallback} = \neg \bigcup_{i\in I} \pi(k_i)$$

Our algorithm proceeds by trimming

$\mathrm{equiv(S,\ Switch(a,\ (k}_i \rightarrow \mathrm{D}_{Si})^{\mathrm{i\in I}}, \mathrm{D_{sf}),\ D}_T) :=$
$\mathrm{Forall(equiv(\ S}\cap(\mathrm{a}\rightarrow\pi(\mathrm{k}_i)^{\mathrm{i\in I}}), \mathrm{D}_{Si}, \mathrm{D}_{\mathrm{t/a}\rightarrow\pi(\mathrm{k}_i)})^{\mathrm{i\in I}} + \mathrm{equiv(S}\cap(\mathrm{a}\rightarrow\pi_n), \mathrm{D}_S, \mathrm{D}_{\mathrm{a}\rightarrow\pi_{\text{fallback}}}))$

The statement still holds and we show this by first analyzing the *Yes* case:

$$\mathrm{Forall(equiv(\ S}\cap(\mathrm{a}\rightarrow\pi(\mathrm{k}_i)^{\mathrm{i\in I}}), \mathrm{D}_{Si}, \mathrm{D}_{\mathrm{t/a}\rightarrow\pi(\mathrm{k}_i)})^{\mathrm{i\in I}} = \mathrm{Yes}$$

The constructor k is either included in the set of constructors k_i:

$$\mathrm{k} \mid \mathrm{k}\in(\mathrm{k}_i)^i \wedge \mathrm{D}_S(\mathrm{v}_S) = \mathrm{D}_{Si}(\mathrm{v}_S)$$

We also know that

$$(1)\ \mathrm{D}_{Si}(\mathrm{v}_S) = \mathrm{D}_{\mathrm{t/a}\rightarrow\pi_i}(\mathrm{v}_T)$$
$$(2)\ \mathrm{D}_{\mathrm{T/a}\rightarrow\pi_i}(\mathrm{v}_T) = \mathrm{D}_T(\mathrm{v}_T)$$

(1) is true by induction and (2) is a consequence of the trimming lemma. Putting everything together:

$$\mathrm{D}_S(\mathrm{v}_S) = \mathrm{D}_{Si}(\mathrm{v}_S) = \mathrm{D}_{\mathrm{T/a}\rightarrow\pi_i}(\mathrm{v}_T) = \mathrm{D}_T(\mathrm{v}_T)$$

When the $\mathrm{k}\notin(\mathrm{k}_i)^i$ [TODO]

The auxiliary Forall function returns *No(v_S, v_T)* when, for a minimum k,

$$\mathrm{equiv(S}_k, \mathrm{D}_{Sk}, \mathrm{D}_{\mathrm{T/a}\rightarrow\pi_k} = \mathrm{No(v}_S, \mathrm{v}_T)$$

Then we can say that

$$\mathrm{D}_{Sk}(\mathrm{v}_S) \neq \mathrm{D}_{\mathrm{t/a}\rightarrow\pi_k}(\mathrm{v}_T)$$

that is enough for proving that

$$\mathrm{D}_{Sk}(\mathrm{v}_S) \neq (\mathrm{D}_{\mathrm{t/a}\rightarrow\pi_k}(\mathrm{v}_T) = \mathrm{D}_T(\mathrm{v}_T))$$

4 Examples

In this section we discuss some examples given as input and output of the prototype tool. Source and target files are taken from the internship git repository[25].

example0.ml

```
external observe : 'a -> 'b = "observe"

let mm = function
  | 2 -> observe 2
  | 3 -> observe 3
  | 4 -> observe 4
  | _ -> observe 5
```

We can see from this first source file the usage of the *observe* directive. The following is the target file generated by the OCaml compiler.

example0.lambda

```
(setglobal Example0!
  (let
    (mm/81 =
       (function param/82
         (catch
           (let (switcher/85 =a (-2+ param/82))
             (if (isout 2 switcher/85) (exit 1)
               (switch* switcher/85
                case int 0: (observe 2)
                case int 1: (observe 3)
                case int 2: (observe 4))))
          with (1) (observe 5))))
    (makeblock 0 mm/81)))
```

The prototype tool states that the compilation was successful and the two programs are equivalent.

example0.trace

```
Target program decision tree
Switch ({ var=AcAdd(-2 AcRoot=param/82); dom=Int [-inf; -1] [3; +inf] v Tag _; }) =
        Leaf=VConstant:5
```

```
Switch ({ var=AcAdd(-2 AcRoot=param/82); dom=Int [0; 2]; }) =
        Switch ({ var=AcAdd(-2 AcRoot=param/82); dom=Int 0; }) =
                Leaf=VConstant:2
        Switch ({ var=AcAdd(-2 AcRoot=param/82); dom=Int 1; }) =
                Leaf=VConstant:3
        Switch ({ var=AcAdd(-2 AcRoot=param/82); dom=Int 2; }) =
                Leaf=VConstant:4
        Fallback=None
Fallback=None


Source program decision tree
Switch AcRoot:{
        Int 3 ->
                Leaf=’Int 3 ’

        Int 4 ->
                Leaf=’Int 4 ’

        Int 2 ->
                Leaf=’Int 2 ’
} Fallback: Leaf=’Int 5 ’

The two programs are equivalent.
```

The following example shows how the prototype handles ADT.

example3.ml

```
external observe : ’a -> ’b = "observe"

type t = K1 of int | K2 of bool | K3

let a = fun t -> match t with
  | K1 1 -> observe 1 1
  | K1 2 -> observe 1 2
  | K1 _ -> observe 1 ()
  | K2 true -> observe 2 true
  | K2 false -> observe 2 false
  | K3 -> observe 3
```

example3.lambda

```
(setglobal Example3!
  (let
    (a/85 =
       (function t/86
         (switch* t/86
          case int 0: (observe 3)
          case tag 0:
           (let (*match*/90 =a (field 0 t/86))
             (catch
               (if (!= *match*/90 1)
                 (if (!= *match*/90 2) (exit 1) (apply (observe 1) 2))
                 (apply (observe 1) 1))
              with (1) (apply (observe 1) 0a)))
          case tag 1:
           (let (*match*/91 =a (field 0 t/86))
             (if (!= *match*/91 0) (apply (observe 2) 1a)
               (apply (observe 2) 0a))))))
    (makeblock 0 a/85)))
```

example3.trace

```
Target program decision tree
Switch ({ var=AcRoot=t/86; dom=Int 0; }) =
        Leaf=VConstant:3
Switch ({ var=AcRoot=t/86; dom=Tag 0; }) =
        Switch ({ var=AcField(0 AcRoot=t/86); dom=Int [-inf; 0] [2; +inf] v Tag _; }) =
                Switch ({ var=AcField(0 AcRoot=t/86); dom=Int [-inf; 1] [3; +inf] v Tag _; }) =
                        Leaf=VConstant:1, VConstant:0
                Switch ({ var=AcField(0 AcRoot=t/86); dom=Int 2; }) =
                        Leaf=VConstant:1, VConstant:2
                Fallback=None
        Switch ({ var=AcField(0 AcRoot=t/86); dom=Int 1; }) =
                Leaf=VConstant:1, VConstant:1
        Fallback=None
Switch ({ var=AcRoot=t/86; dom=Tag 1; }) =
        Switch ({ var=AcField(0 AcRoot=t/86); dom=Int [-inf; -1] [1; +inf] v Tag _; }) =
                Leaf=VConstant:2, VConstant:1
        Switch ({ var=AcField(0 AcRoot=t/86); dom=Int 0; }) =
                Leaf=VConstant:2, VConstant:0
        Fallback=None
Fallback=None

Source program decision tree
Switch AcRoot:{
        Variant K2 ->
                Switch AcRoot.0:{
                Bool false ->
                        Leaf='Int 2 Bool false '

                Bool true ->
                        Leaf='Int 2 Bool true '
        } Fallback: Unreachable

        Variant K1 ->
                Switch AcRoot.0:{
                Int 2 ->
                        Leaf='Int 1 Int 2 '

                Int 1 ->
                        Leaf='Int 1 Int 1 '
        } Fallback: Leaf='Int 1 Unit '

        Variant K3 ->
                Leaf='Int 3 '
} Fallback: Unreachable

The two programs are equivalent.
```

This trivial pattern matching code shows how guards are handled.

guards2.ml

```
[@@@warning "-20"]
external observe : 'a -> 'b = "observe"
external guard : 'a -> 'b = "guard"

let ff = function
  | x when guard () -> observe 1
  | _ when guard 1 -> observe 2
  | _ -> observe 3
```

guards2.lambda

```
(setglobal Guards2!
  (let
      (ff/82 =
             (function x/83
                      (if (guard 0a) (observe 1)
                                 (if (guard 1) (observe 2) (observe 3)))))
                                     (makeblock 0 ff/82)))
```

guards2.trace

```
Target program decision tree
Guard (VConstant:0):
guard(true) =
        Leaf=VConstant:1
guard(false) =
        Guard (VConstant:1):
        guard(true) =
                Leaf=VConstant:2
        guard(false) =
                Leaf=VConstant:3

Source program decision tree
Switch AcRoot:{
} Fallback: Guard (Unit ) =
        guard(true) =
                Leaf=’Int 1 ’
        guard(false) =
                Guard (Int 1 ) =
                guard(true) =
                        Leaf=’Int 2 ’
                guard(false) =
                        Leaf=’Int 3 ’

The two programs are equivalent.
```

The following source code shows the usage of the OCaml refutation clause.

example9.ml

```
[@@@warning "-20"]
external observe : ’a -> ’b = "observe"

let test = function
  | true -> observe 0
  | false -> observe 1
  | _ -> .
    (* Unreachable; if this annotation was incorrect,
       the OCaml compiler would error at pattern-checking-time *)
```

example9.lambda

```
(setglobal Example9!
  (let
    (test/81 =
       (function param/82 (if (!= param/82 0) (observe 0) (observe 1))))
    (makeblock 0 test/81)))
```

example9.trace

```
Target program decision tree
Switch ({ var=AcRoot=param/82; dom=Int [-inf; -1] [1; +inf] v Tag _; }) =
        Leaf=VConstant:0
Switch ({ var=AcRoot=param/82; dom=Int 0; }) =
        Leaf=VConstant:1
Fallback=None



Source program decision tree
Switch AcRoot:{
        Bool false ->
                Leaf='Int 1 '

        Bool true ->
                Leaf='Int 0 '
} Fallback: Unreachable

The two programs are equivalent.
```

In this example the tool correctly handles *Failure* nodes on both decision trees.

example9bis.ml

```
[@@@warning "-20"]
external observe : 'a -> 'b = "observe"
```

```
let test = function
  | true -> observe 0
  (* we expect a Match_failure node for 'false' in the lambda representation *)
```

example9bis.lambda

```
(setglobal Example9bis!
  (let
    (test/81 =
       (function param/82
         (catch (if (!= param/82 0) (observe 0) (exit 1)) with (1)
           (raise
             (makeblock 0 (global Match_failure/18!)
               [0: "example9bis.ml" 4 11])))))
    (makeblock 0 test/81)))
```

example9bis.trace

```
Target program decision tree
Switch ({ var=AcRoot=param/82; dom=Int [-inf; -1] [1; +inf] v Tag _; }) =
        Leaf=VConstant:0
Switch ({ var=AcRoot=param/82; dom=Int 0; }) =
        Failure
    Fallback=None


Source program decision tree
Switch AcRoot:{
        Bool true ->
                Leaf='Int 0 '
} Fallback: Failure

The two programs are equivalent.
```

5 Conclusions

In recent years bugs in the OCaml pattern matching compiler have been infrequent: only one in the last four years[26]. The main motivation for this work is the ongoing efforts of the OCaml core developers towards towards refactoring the OCaml pattern matching compiler[27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39]. The prototype presented in this dissertation shows a sound approach for the verification of a correct compilation. Compared to similar works in the field, this work differs from Necula et al.[22] for the absence of false positives cases. The work of Kirchner et al.[40] is the most similar to this dissertation but has the drawbacks of introducing a new *IL*, requires a full fledged SAT solver (Zenon) and does not handle guards. The internship was focused on providing a working MVP. In order for the work to be integrated into the OCaml compiler toolchain the prototype needs to handle all primitive datatypes. While string and floats are trivial to add to the current implementation, OCaml supports extensible sum types[41] that may require a lot of effort to be successfully integrated into the tool. Extensible sum types are a product type whose set of values can be extended by declaring new constructors at runtime. Pattern matching on extensible sum types requires a wildcard pattern to handle unknown variant constructors. Exceptions were the first type of extensible sum type implemented in the language. The prototype uses *observe* and *guard* directives as a shortcut to avoid depending on the compiler. In order to support real-world code we need to instrument the compiler-libs to provide a two pass compilation where guards and leaf expressions can be compiled to Lambda code indipendently from the rest of the pattern match code.

References

[1] Sergei Murzin, Michael Park, David Sankel, Dan Sarginson. *P1371R1: Pattern Matching Proposal for C++.* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1371r1.pdf

[2] John McCarthy and James Painter. *Correctness of a compiler for arithmetic expressions.* Proceedings Symposium in Applied Mathematics, Vol. 19, pages 33–41. AMS, 1967.

[3] F. Lockwood Morris. *Advice on structuring compilers and proving them correct.* 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pages 144–152. ACM Press, 1973

[4] David Lacey, Neil D. Jones, Eric Van Wyk, and Carl Christian Frederiksen. *Proving correctness of compiler optimizations by temporal logic.* 29th ACM Symposium on Principles of Programming Languages, pages 283–294. ACM Press, 2002.

[5] Leroy Xavier. *Formal verification of a realistic compiler.* Communications of the ACM 52.7 (2009): 107-115.

[6] Berghofer, Stefan, and Martin Strecker. *Extracting a formally verified, fully executable compiler from a proof assistant.* Electronic Notes in Theoretical Computer Science 82.2 (2004): 377-394.

[7] Lochbihler, Andreas. *Verifying a compiler for Java threads.* European Symposium on Programming. Springer, Berlin, Heidelberg, 2010.

[8] Nayebi, Fatih. *Swift Functional Programming.* Packt Publishing Ltd, 2017.

[9] Radenski, Atanas, Jeff Furlong, Vladimir Zanev. *The Java 5 generics compromise orthogonality to keep compatibility.* Journal of Systems and Software 81.11 (2008): 2069-2078.

[10] Alexandrescu, Andrei. *Modern C++ design: generic programming and design patterns applied.* Addison-Wesley, 2001.

[11] Yallop, Jeremy, and Oleg Kiselyov. *First-class modules: hidden power and tantalizing promises.* Workshop on ML. Vol. 2. 2010.

[12] Augustsson, Lennart. *Compiling pattern matching.* Conference on Functional Programming Languages and Computer Architecture. Springer, Berlin, Heidelberg, 1985.

[13] Dolan, Stephen. *Malfunctional programming.* ML Workshop. 2016.

[14] OCaml core developers. *OCaml Manual* http://caml.inria.fr/pub/docs/manual-ocaml/flambda.html

[15] https://www.cvedetails.com/cve/CVE-2002-0973/

[16] Charguéraud, A., Filliâtre, J. C., Pereira, M., Pottier F. (2017). *VOCAL–A Verified OCAml Library.* https://hal.inria.fr/hal-01561094/

[17] Yaron Minsky, Anil Madhavapeddy, Jason Hickey *Real World OCaml: Functional Programming for the masses.* https://dev.realworld.org/compiler-backend.html

[18] Syme, Don, Gregory Neverov, James Margetson. *Extensible pattern matching via a lightweight language extension.* Proceedings of the 12th ACM SIGPLAN international conference on Functional programming (2007).

[19] Baldoni, Roberto and Coppa, Emilio and D'Elia, Daniele Cono and Demetrescu, Camil and Finocchi, Irene. *A Survey of Symbolic Execution Techniques.* ACM Computing Surveys (CSUR), 51(3), 1-39.

[20] Cadar, Cristian, and Koushik Sen. *Symbolic execution for software testing: three decades later.* Communications of the ACM 56.2 (2013): 82-90.

[21] Amir Pnueli, Ofer Shtrichman, Michael Siegel. *Translation Validation: from SIGNAL to C.* Correct System Design. Springer, Berlin, Heidelberg, 1999. 231-255.

[22] Necula, George C. *Translation validation for an optimizing compiler.* Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (2000).

[23] https://github.com/janestreet/ocaml-compiler-libs

[24] Régis-Gianas, François Pottier Yann. *Menhir Reference Manual.*

[25] https://github.com/FraMecca/inria-internship

[26] https://github.com/ocaml/ocaml/issues/7390

[27] https://github.com/ocaml/ocaml/issues/7390

[28] https://github.com/ocaml/ocaml/pull/8768

[29] https://github.com/ocaml/ocaml/pull/8850

[30] https://github.com/ocaml/ocaml/pull/9447

[31] https://github.com/ocaml/ocaml/pull/9464

[32] https://github.com/ocaml/ocaml/pull/9493

[33] https://github.com/ocaml/ocaml/pull/9646

[34] https://github.com/ocaml/ocaml/pull/9608

[35] https://github.com/ocaml/ocaml/pull/9599

[36] https://github.com/ocaml/ocaml/pull/9563

[37] https://github.com/ocaml/ocaml/pull/9520

[38] https://github.com/ocaml/ocaml/pull/9511

[39] https://github.com/ocaml/ocaml/pull/9647

[40] Kirchner, Claude, Pierre-Etienne Moreau, and Antoine Reilles. *Formal validation of pattern matching code*. Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming. 2005.

[41] OCaml core developers. *OCaml Manual* https://caml.inria.fr/pub/docs/manual-ocaml/extensiblevariants.html