Translation Verification of the pattern matching compiler

Francesco Mecca

1 Introduction

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

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

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

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

Given as input to the pattern matching compiler, this snippet of code gets translated into the Lambda intermediate representation of the OCaml com-

piler. The Lambda representation of a program is shown by calling the `ocamlc` compiler with `-drawlambda` flag.

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

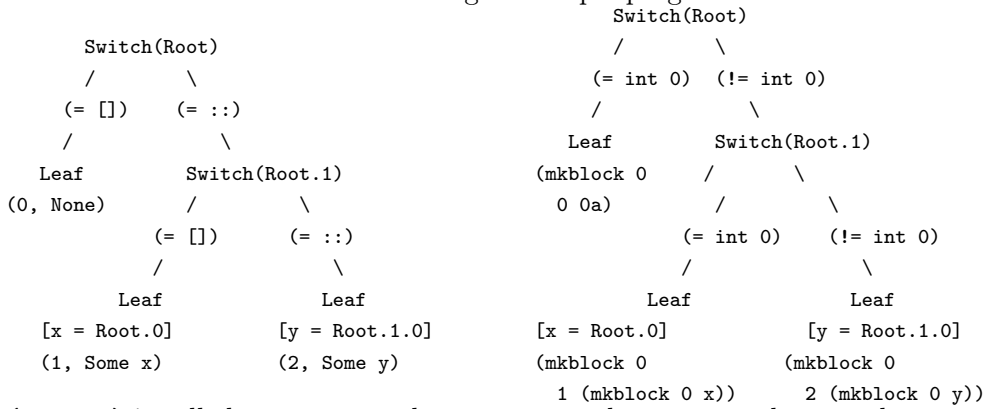
The OCaml pattern matching compiler is a critical part of the OCaml compiler in terms of correctness because bugs typically would result in wrong code production rather than triggering compilation failures. Such bugs also are hard to catch by testing because they arise in corner cases of complex patterns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

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

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

1.1 Our approach

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



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

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

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

1.2 From source programs to decision trees

Our source language supports integers, lists, tuples and all algebraic datatypes. Patterns support wildcards, constructors and literals, Or-patterns such as

$(p_1|p_2)$ and pattern variables. We also support `when` guards, which are interesting as they introduce the evaluation of expressions during matching. Decision trees have nodes of the form:

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

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

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

$$\forall t_s, \forall v_s, \quad t_s(v_s) = [\![t_s]\!]_s(v_s)$$

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

1.3 From target programs to decision trees

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

Guards result in branching. In comparison with the source decision trees, `Unreachable` nodes are never emitted.

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

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

1.4 Equivalence checking

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

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

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

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

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

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

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

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

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

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

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

The equivalence checking algorithm deals with guards by storing a queue. A guard blackbox is pushed to the queue whenever the algorithm encounters a Guard node on the right, while it pops a blackbox from the queue whenever a Guard node appears on the left hand side. The algorithm stops with failure if the popped blackbox and the and blackbox on the left hand Guard node are different, otherwise in continues by exploding to two subtrees, one in which the guard condition evaluates to true, the other when it evaluates to false. Termination of the algorithm is successful only when the guards queue is empty.

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

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

2 Background

2.1 OCaml

Objective Caml (OCaml) is a dialect of the ML (Meta-Language) family of programming that features with other dialects of ML, such as SML and Caml Light. The main features of ML languages are the use of the Hindley-Milner type system that provides many advantages with respect to static type systems of traditional imperative and object oriented language such as C, C++ and Java, such as:

- Polymorphism: in certain scenarios a function can accept more than one type for the input parameters. For example a function that computes the length of a list doesn't need to inspect the type of the elements of the list and for this reason a List.length function can accept lists of integers, lists of strings and in general lists of any type. Such languages offer polymorphic functions through subtyping at runtime only, while other languages such as C++ offer polymorphism through compile time templates and function overloading. With the Hindley-Milner type system each well typed function can have more than one type but always has a unique best type, called the *principal type*. For example the principal type of the List.length function is "For any a, function from list of a to int" and a is called the *type parameter*.

- Strong typing: Languages such as C and C++ allow the programmer to operate on data without considering its type, mainly through pointers. Other languages such as C# and Go allow type erasure so at runtime the type of the data can't be queried. In the case of programming languages using an Hindley-Milner type system the programmer is not allowed to operate on data by ignoring or promoting its type.

- Type Inference: the principal type of a well formed term can be inferred without any annotation or declaration.

- Algebraic data types: types that are modeled by the use of two alge-

braic operations, sum and product. A sum type is a type that can hold of many different types of objects, but only one at a time. For example the sum type defined as $A + B$ can hold at any moment a value of type A or a value of type B. Sum types are also called tagged union or variants. A product type is a type constructed as a direct product of multiple types and contains at any moment one instance for every type of its operands. Product types are also called tuples or records. Algebraic data types can be recursive in their definition and can be combined.

Moreover ML languages are functional, meaning that functions are treated as first class citizens and variables are immutable, although mutable statements and imperative constructs are permitted. In addition to that features an object system, that provides inheritance, subtyping and dynamic binding, and modules, that provide a way to encapsulate definitions. Modules are checked statically and can be reifycated through functors.

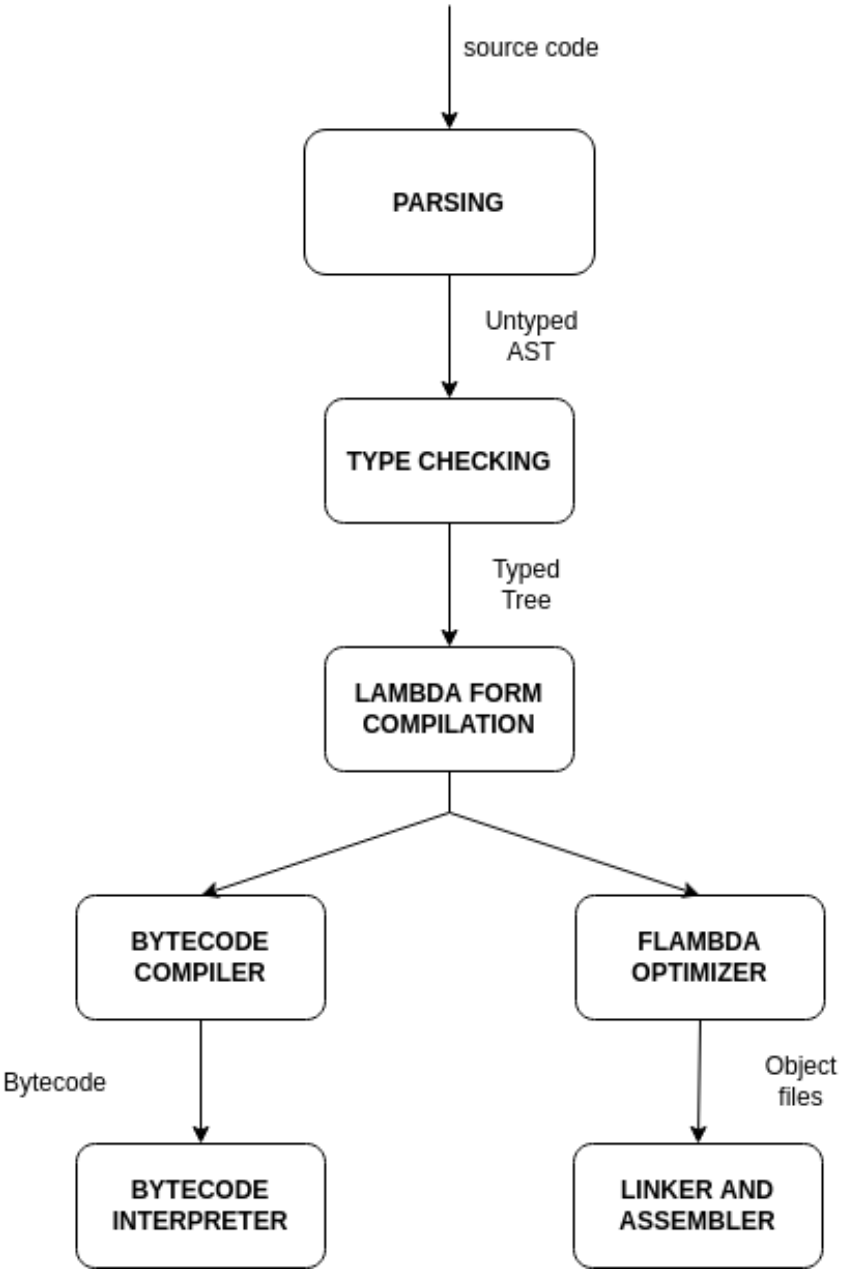
2.2 Compiling OCaml code

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

- type inference, using the classical *Algorithm W*
- perform subtyping and gathers type information from the module system
- ensures that the code obeys the rule of the OCaml type system

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

This is an overview of the different compiler steps.


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

2.3 Memory representation of OCaml values

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

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

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

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

2.4 Lambda form compilation

A Lambda code target file is produced by the compiler and consists of a single s-expression. Every s-expression consist of *(*, a sequence of elements separated by a whitespace and a closing *)*. Elements of s-expressions are:

- Atoms: sequences of ascii letters, digits or symbols
- Variables
- Strings: enclosed in double quotes and possibly escaped

- S-expressions: allowing arbitrary nesting

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

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


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

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

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

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

map every pattern matching clauses to its target expression. Values are addressed by a unique name.

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

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

In the case of types with a smaller number of variants, the pattern matching compiler may avoid the overhead of computing a jump table. This example also highlights the fact that non constant constructor are mapped to cons cell that are accessed using the *tag* directive.

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

In the Lambda language are several numeric types:

- integers: that us either 31 or 63 bit two's complement arithmetic depending on system word size, and also wrapping on overflow
- 32 bit and 64 bit integers: that use 32-bit and 64-bit two's complement arithmetic with wrap on overflow

- big integers: offer integers with arbitrary precision
- floats: that use IEEE754 double-precision (64-bit) arithmetic with the addition of the literals *infinity*, *neg_ infinity* and *nan*.

The are various numeric operations defined:

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

1. Functions

   Functions are defined using the following syntax, and close over all bindings in scope: (lambda (arg1 arg2 arg3) BODY) and are applied using the following syntax: (apply FUNC ARG ARG ARG) Evaluation is eager.

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

   The Lambda form supports many other directives such as *strinswitch* that is constructs aspecialized jump tables for string, integer range comparisons and so on. These construct are explicitely undocumented because the Lambda code intermediate language can change across compiler releases.

2.5 Pattern matching

Pattern matching is a widely adopted mechanism to interact with ADT. C family languages provide branching on predicates through the use of if statements and switch statements. Pattern matching on the other hands express predicates through syntactic templates that also allow to bind on data structures of arbitrary shapes. One common example of pattern matching

is the use of regular expressions on strings. provides pattern matching on ADT and primitive data types. The result of a pattern matching operation is always one of:

- this value does not match this pattern"
- this value matches this pattern, resulting the following bindings of names to values and the jump to the expression pointed at the pattern.

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

match color with
| Red -> print "red"
| Blue -> print "red"
| Green -> print "red"
| _ -> print "white or black"
```

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

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

Parenthesized patterns, such as the third one in the previous example, matches the same value as the pattern without parenthesis.

The same could be done with tuples

```
begin match tuple with
| (Some _, Some _) -> print "Pair of optional types"
| (Some _, None) | (None, Some _) -> print "Pair of optional types, one of which is nul
| (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.

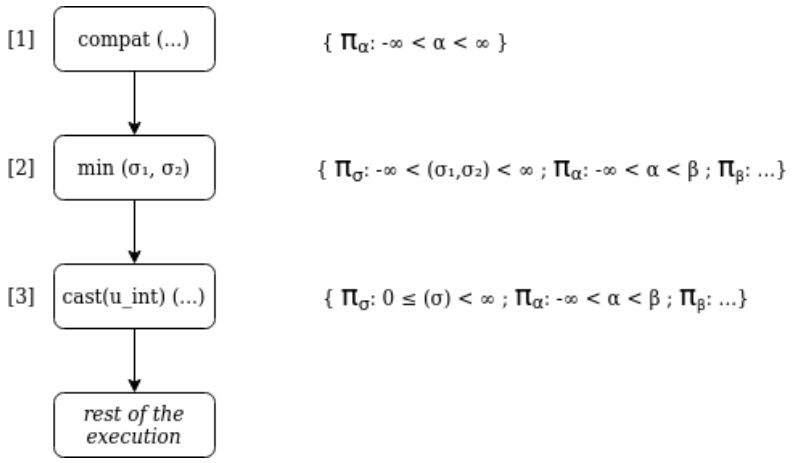
2.6 Symbolic execution

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 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 when the function is evaluated considering *int len* our symbolic variable α, sa->sa_len as symbolic variable β and π as the set of constraints on a symbolic variable:



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 exploitation possible.

Every step of evaluation can be modelled as the following transition:

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

if we express the π constraints as logical formulas we can model the execution of the program in terms of Hoare Logic. 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 rules of Hoare logic:

- 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 *midcondition*.

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

- Conditional :

$$\frac{\{P \wedge b\}c_1\{Q\}, \{P \wedge \not b\}c_2\{Q\}}{\{P\}\text{if b then } c_1 \text{ else } c_2\{Q\}}$$

- Loop : {P} is the loop invariant. After the loop is finished P holds and $\neg eg$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. 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 translators 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.

- ☐ Techniques for translation validation
- ☐ What does semantically equivalent mean
- ☐ What happens when there is no semantic equivalence
- ☐ Translation validation through symbolic execution

3 Translation validation of the Pattern Matching Compiler

3.1 Source program

The algorithm takes as its input a source program and translates it into an algebraic data structure which type we call *decision_ tree*.

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

Unreachable, Leaf of source_expr and Failure are the terminals of the three. We distinguish

- Unreachable: statically it is known that no value can go there
- Failure: a value matching this part results in an error
- Leaf: a value matching this part results into the evaluation of a source black box of code

The algorithm doesn't support type-declaration-based analysis to know the list of constructors at a given type. 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.

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 true.

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

$$
\begin{array}{l}
\text{match variable with} \\
\mid \text{pattern}_1 \rightarrow \text{expr}_1 \\
\mid \text{pattern}_2 \text{ when guard} \rightarrow \text{expr}_2 \\
\mid \text{pattern}_3 \text{ as var} \rightarrow \text{expr}_3 \\
\vdots \\
\mid \text{p}_n \rightarrow \text{expr}_n
\end{array}
$$

and can include any expression that is legal for the OCaml compiler, such as *when* guards and assignments. Patterns could or could not be exhaustive.

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

$$
\begin{array}{l}
\text{function} \\
\mid \text{pattern}_1 \rightarrow \text{expr}_1 \\
\mid \text{pattern}_2 \text{ when guard} \rightarrow \text{expr}_2 \\
\mid \text{pattern}_3 \text{ as var} \rightarrow \text{expr}_3 \\
\vdots \\
\mid \text{p}_n \rightarrow \text{expr}_n
\end{array}
$$

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 ;;
;; rules
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|parameter
or_pattern ::= rule ("|" wildcard|variable|constructor_pattern)+
condition ::= "when" bexpr
assignment ::= "as" id
bexpr ::= _code ;; arbitrary code

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

During compilation by the translators, expressions are compiled into Lambda code and are referred as lambda code actions l_i.

The entire pattern matching code is represented as a clause matrix that

associates rows of patterns ($p_{i,1}$, $p_{i,2}$, ..., $p_{i,n}$) to lambda code action l^i

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

The pattern p matches a value v, written as p $\preccurlyeq$ v, when one of the following rules apply

_	$\preccurlyeq$	v	$\forall$v
x	$\preccurlyeq$	v	$\forall$v
(p_1 | p_2)	$\preccurlyeq$	v	iff $p_1 \preccurlyeq$ v or $p_2 \preccurlyeq$ v
c(p_1, p_2, ..., p_a)	$\preccurlyeq$	c(v_1, v_2, ..., v_a)	iff (p_1, p_2, ..., p_a) $\preccurlyeq$ (v_1, v_2, ..., v_a)
(p_1, p_2, ..., p_a)	$\preccurlyeq$	(v_1, v_2, ..., v_a)	iff $p_i \preccurlyeq v_i$ $\forall$i $\in$ [1..a]

When a value v matches pattern p we say that v is an *instance* of p.

Considering the pattern matrix P we say that the value vector $\vec{v} = (v_1, v_2, \ldots, v_i)$ matches the line number i in P if and only if the following two conditions are satisfied:

- $p_{i,1}$, $p_{i,2}$, $\cdots$, $p_{i,n} \preccurlyeq (v_1, v_2, \ldots, v_i)$
- $\forall j < i$ $p_{j,1}$, $p_{j,2}$, $\cdots$, $p_{j,n} \not\preccurlyeq (v_1, v_2, \ldots, v_i)$

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

- Patter p is less precise than pattern q, written p $\preccurlyeq$ q, when all instances of q are instances of p
- Pattern p and q are equivalent, written p $\equiv$ q, when their instances are the same
- Patterns p and q are compatible when they share a common instance

3.1.1 Parsing of the source program

The source program of the following general form is parsed using a parser generated by Menhir, a LR(1) parser generator for the OCaml programming language. Menhir compiles LR(1) a grammar specification, in this case the OCaml pattern matching grammar, down to OCaml code.

match variable with
| pattern$_1$ -> e$_1$
| pattern$_2$ -> e$_2$
$\vdots$
| p$_m$ -> e$_m$

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$_2$,...,p$_n$)	Constructor
(p$_1$| p$_2$)	Orpat

Guards and right-hand-sides are treated as a blackbox of OCaml code. A sound approach for treating these blackbox 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 equality with the respective blackbox at the target level. Given that this level of introspection is currently not possibile, 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 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.

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

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

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.

3.1.2 Matrix decomposition of pattern clauses

The initial input of the decomposition algorithm C consists of a vector of variables $\vec{x} = (\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n)$ of size n where n is the arity of the type of x and a clause matrix P $\rightarrow$ L of width n and height m. 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 empty

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

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

   That means in every lambda action l_i there is a binding of x_1 to the variable that appears on the pattern $p_{i,1}$. Bindings are omitted for wildcard patterns and 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, ..., q_n) we define a new matrix, the specialized clause matrix S, by applying the following transformation on every row p:

   ```
   for every c ∈ Set of constructors
       for i ← 1 .. m
           let k_i ← constructor_of(p_{i,1})
           if k_i = c then
               p ← 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 new fresh variables y_1, y_2, ..., y_a so that the lambda action l_i becomes

```
(let (y1 (field 0 x1))
     (y2 (field 1 x1))
     ...
     (ya (field (a−1) x1))
     li)
```

and the result of the compilation for the set of constructors $\{\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_k\}$ is:

```
switch x1 with
case c1: l1
case c2: l2
...
case ck: lk
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:

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

results in three rows added to the clause matrix

$$p_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

$$q_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

$$r_{i,1}, p_{i,2}, ..., p_{i,m} \rightarrow l_i$$

2. Mixture rule: When none of the previous rules apply the clause matrix $\mathrm{P} \rightarrow \mathrm{L}$ is split into two clause matrices, the first $\mathrm{P}_1 \rightarrow \mathrm{L}_1$ that is the largest prefix matrix for which one of the three previous rules apply, and $\mathrm{P}_2 \rightarrow \mathrm{L}_2$ containing the remaining rows. The algorithm is applied to both matrices.

3.2 Target translation

TODO

3.3 Equivalence checking

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

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

Our equivalence-checking algorithm $\mathsf{equiv}(S, C_S, C_T)G$ is a exactly decision procedure for the provability of the judgment $(\mathsf{equiv}(S, C_S, C_T)G)$,

defined below.

$$
\begin{array}{rl}
 & \textit{constraint trees} \\
C ::= & \mathsf{Leaf}(t) \\
 \mid & \mathsf{Failure} \\
 \mid & \mathsf{Switch}(a, (\pi_i, C_i)^i, C_{\mathsf{fb}}) \\
 \mid & \mathsf{Guard}(t, C_0, C_1)
\end{array}
\qquad
\begin{array}{rl}
 & \textit{boolean result} \\
b & \in \{0, 1\}
\end{array}
$$

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

$$
\begin{array}{c}
\textit{input space} \\
S \subseteq \{(v_S, v_T) \mid v_S \approx_{\mathsf{val}} v_T\}
\end{array}
$$

$$
\frac{}{\mathsf{equiv}(\emptyset, C_S, C_T)G}
\qquad
\frac{}{\mathsf{equiv}(S, \mathsf{Failure}, \mathsf{Failure})[]}
$$

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

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

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

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

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

4 Correctness of the algorithm

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:

$$(\llbracket \mathrm{t}_S \rrbracket_S(\mathrm{v}_S) = \mathrm{C}_S(\mathrm{v}_S)) \to \mathrm{r}$$
$$\mathrm{t}_S(\mathrm{v}_S) \to \mathrm{r}$$

Likewise

$$(\llbracket \mathrm{t}_T \rrbracket_T(\mathrm{v}_T) = \mathrm{C}_T(\mathrm{v}_T)) \to \mathrm{r}$$
$$\mathrm{t}_T(\mathrm{v}_T) \to \mathrm{r}$$

where result r ::= guard list * (Match blackbox | NoMatch | Absurd) and guard ::= blackbox.

Having defined equivalence between two inputs of which one is expressed in the source language and the other in the target language $\mathrm{v}_S \simeq \mathrm{v}_T$ (TODO define, this talks about the representation of source values in the target)

we can define the equivalence between a couple of programs or a couple of decision trees

$$\mathrm{t}_S \simeq \mathrm{t}_T := \forall \mathrm{v}_S \simeq \mathrm{v}_T,\ \mathrm{t}_S(\mathrm{v}_S) = \mathrm{t}_T(\mathrm{v}_T)$$
$$\mathrm{C}_S \simeq \mathrm{C}_T := \forall \mathrm{v}_S \simeq \mathrm{v}_T,\ \mathrm{C}_S(\mathrm{v}_S) = \mathrm{C}_T(\mathrm{v}_T)$$

The proposed equivalence algorithm that works on a couple of decision trees is returns either *Yes* or *No(v_S, v_T)* where v_S and v_T are a couple of possible counter examples for which the constraint trees produce a different result.

4.1 Statements

Theorem. 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

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

Likewise, for the target language:

$$\forall \mathrm{v}_T,\ \mathrm{t}_T(\mathrm{v}_T) = \llbracket \mathrm{t}_T \rrbracket_T(\mathrm{v}_T)$$

Definition: 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:

$$(\mathrm{gs}_1, \mathrm{r}_1) \simeq \mathrm{gs}\ (\mathrm{gs}_2, \mathrm{r}_2) \Leftrightarrow (\mathrm{gs}_1, \mathrm{r}_1) = (\mathrm{gs}_2 +\!\!+ \mathrm{gs}, \mathrm{r}_2)$$

Definition: we say that C_T covers the input space S, written /covers(C_T, S) when every value $\mathrm{v}_S \in \mathrm{S}$ is a valid input to the decision tree C_T. (TODO: rephrase)

Theorem: Given an input space S and a couple of decision trees, where the target decision tree C_T covers the input space S, we say that the two decision trees are equivalent when:

$$\mathrm{equiv}(\mathrm{S}, \mathrm{C}_S, \mathrm{C}_T, \mathrm{gs}) = \mathrm{Yes} \wedge \mathrm{covers}(\mathrm{C}_T, \mathrm{S}) \rightarrow \forall \mathrm{v}_S \simeq \mathrm{v}_T \in \mathrm{S},\ \mathrm{C}_S(\mathrm{v}_S) \simeq \mathrm{gs}\ \mathrm{C}_T(\mathrm{v}_T)$$

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

$$\mathrm{equiv}(\mathrm{S}, \mathrm{C}_S, \mathrm{C}_T, \mathrm{gs}) = \mathrm{No}(\mathrm{v}_S, \mathrm{v}_T) \wedge \mathrm{covers}(\mathrm{C}_T, \mathrm{S}) \rightarrow \mathrm{v}_S \simeq \mathrm{v}_T \in \mathrm{S} \wedge \mathrm{C}_S(\mathrm{v}_S) \not\simeq \mathrm{gs}\ \mathrm{C}_T(\mathrm{v}_T)$$

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

$$\mathrm{equiv}(\mathrm{S}, [\![\mathrm{t}_S]\!]_S, [\![\mathrm{t}_T]\!]_T, \varnothing) = \mathrm{Yes} \Leftrightarrow \mathrm{t}_S \simeq \mathrm{t}_T$$

1. Proof of the correctness of the translation from source programs to source decision trees

   We define a source term t_S as a collection of patterns pointing to blackboxes

   $$\mathrm{t}_S ::= (\mathrm{p} \rightarrow \mathrm{bb})^{\mathrm{i} \in \mathrm{I}}$$

   A pattern is defined as either a constructor pattern, an or pattern or a constant pattern

   $$\mathrm{p} ::= \quad \mathrm{K}(\mathrm{p}_i)^i,\ \mathrm{i} \in \mathrm{I} \quad (\mathrm{p} \quad \mathrm{q}) \quad \mathrm{n} \in \mathbb{N}$$

A decision tree is defined as either a Leaf, a Failure terminal or an intermediate node with different children sharing the same accessor a and an optional fallback. Failure is emitted only when the patterns don't cover the whole set of possible input values S. The fallback is not needed when the user doesn't use a wildcard pattern. %%% Give example of thing

$$\begin{array}{ll} \mathrm{C}_S ::= \text{Leaf bb} & \text{Switch(a, } (\mathrm{K}_i \to \mathrm{C}_i)^{\mathrm{i}\in\mathrm{S}} \text{ , C?)} \\ \text{a} ::= \text{Here} & \text{n.a} \\ \mathrm{v}_S ::= \mathrm{K}(\mathrm{v}_i)^{\mathrm{i}\in\mathrm{I}} & \mathrm{n} \in \mathbb{N} \end{array}$$

We define the decomposition matrix m_S as

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

We define the decision tree of source programs $[\![\mathrm{t}_S]\!]$ in terms of the decision tree of pattern matrices $[\![\mathrm{m}_S]\!]$ by the following: $[\![((\mathrm{p}_i \to \mathrm{bb}_i)^{\mathrm{i}\in\mathrm{I}}]\!]$ $:= [\![(\text{Root}), (\mathrm{p}_i \to \mathrm{bb}_i)^{\mathrm{i}\in\mathrm{I}}]\!]$

decision tree computed from pattern matrices respect the following invariant:

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

where

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

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

Base cases:

(a) $[|\ \varnothing, (\varnothing \to \mathrm{bb}_i)^i\ |] :=$ Leaf bb_i where $\mathrm{i} := \min(\mathrm{I})$, that is a decision tree [|ms|] defined by an empty accessor and empty patterns pointing to blackboxes bb_i. This respects the invariant because a decomposition matrix in the case of empty rows returns the first expression and we known that (Leaf bb)(v) := Match bb

(b) $[\![(\mathrm{a}_j)^j, \varnothing]\!] :=$ Failure

Regarding non base cases: Let's first define

$$\begin{aligned}
&\text{let Idx(k)} := [0; \text{arity(k)}[\\
&\text{let First}(\varnothing) := \bot \\
&\text{let First}((\mathrm{a}_j)^j) := \mathrm{a}_{\min(\mathrm{j} \in \mathrm{J} \neq \varnothing)}
\end{aligned}$$

$$m := ((a_i)^i((p_{ij})^i \to e_j)^{ij})$$

$$(k_k)^k := headconstructor(p_{i0})^i$$

$$Groups(m) := (k_k \to ((a)_{0.l})^{l \in Idx(k_k)} +\!+\!+ (a_i)^{i \in I \setminus \{0\}}), (if p_{0j} isk(q_l) then (q_l)^{l \in Idx(k_k)} +\!+\!+ (p_{ij})^{i \in I \setminus \{} \tag{1}$$

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 decompositionin the following sense: if the first pattern of an input vector (v_i)^i starts with one of the head constructors k, then running (v_i)^i against m is the same as running it against the submatrix m_k; otherwise (its head constructor is none of the k) it is equivalent to running it against the wildcard submatrix.

We formalize this intuition as follows: Lemma (Groups): Let

$$m$$

be a matrix with

$$Groups(m) = (k_r \to m_r)^k, m_{wild}$$

. For any value vector

$$(v_i)^l$$

such that

$$v_0 = k(v'_l)^l$$

for some constructor k, we have:

$$if k = k_k for some k then m(v_i)^i = m_k((v'_l)^l +++ (v_i)^{i \in I \setminus \{0\}}) else m(v_i)^i = m_{wild}(v_i)^{i \in I \setminus \{0\}}$$

2. Proof: Let

$$m$$

be a matrix with

$$Group(m) = (k_r \to m_r)^k, m_{wild}$$

. Let

$$(v_i)^i$$

be an input matrix with

$$v_0 = k(v'_l)^l$$

for some k. We proceed by case analysis:

- either k is one of the k_k for some k
- or k is none of the $(k_k)^k$

Both $m(v_i)^i$ and $m_k(v_k)^k$ are defined as the first matching result of a family over each row r_j of a matrix

We know, from the definition of Groups(m), that m_k is

$$((a)0.l)^{l \in Idx(k_k)} +++ (a_i)^{i \in I \setminus \{0\}}), (if p_{0j} is k(q_l) then (q_l)^l +++ (p_{ij})^{i \in I \setminus \{0\}} \to e_j if p_{0j} is_t hen(^l_) +++ (p_i$$

By definition, $m(v_i)^i$ is $m(v_i)^i = First(r_j(v_i)^i)^j$ for m = $((a_i)^i, (r_j)^j)$ $(p_i)^i$ $(v_i)^i$ = { if $p_0 = k(q_l)^l$, $v_0 = k'(v'_k)^k$, k=Idx(k') and l=Idx(k) if $k \neq k'$ then $\perp$ if k = k' then $((q_l)^l + (p_i)^{i \in I \setminus \ \backslash 0 \backslash})$ $((v'_k)^k + (v_i)^{i \in I \setminus \ \backslash 0 \backslash})$ if $p_0 = (q_1|q_2)$ then First($(q_1 {p_i}^{i \in I \ \setminus \ \backslash 0 \backslash})$ ${v_i}^{i \in I \ \setminus \ \backslash 0 \backslash}$, $(q_2 {p_i}^{i \in I \ \setminus \ \backslash 0 \backslash})$ ${v_i}^{i \in I \ \setminus \ \backslash 0 \backslash}$) }

For this reason, if we can prove that

$$\forall \mathrm{j},\ \mathrm{r}_j(\mathrm{v}_i)^i = \mathrm{r'}_j((\mathrm{v'}_k)^k \mathbin{+\!\!+} (\mathrm{v}_i)^i)$$

it follows that

$$\mathrm{m}(\mathrm{v}_i)^i = \mathrm{m}_k((\mathrm{v'}_k)^k \mathbin{+\!\!+} (\mathrm{v}_i)^i)$$

from the above definition.

We can also show that $\mathrm{a}_i = {\mathrm{a}_{0.\mathrm{l}}}^l + \mathrm{a}_{\mathrm{i}\in\mathrm{I}\setminus\ \setminus 0\setminus}$ because $\mathrm{v}(\mathrm{a}_0) = \mathrm{K}(\mathrm{v}(\mathrm{a})\{0.\mathrm{l}\})^l)$

4.2 Proof of equivalence checking

4.2.1 The trimming lemma

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

$$\forall \mathrm{v}_T \in (\mathrm{a}{\rightarrow}\pi),\ \mathrm{C}_T(\mathrm{v}_T) = \mathrm{C}_{\mathrm{t}/\mathrm{a}\rightarrow\pi(\mathrm{k}_i)}(\mathrm{v}_T)$$

We prove this by induction on C_T: a. $\mathrm{C}_T = \mathrm{Leaf}_{\mathrm{bb}}$: when the decision tree is a leaf terminal, we know that

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

That means that the result of trimming on a Leaf is the Leaf itself b. The same applies to Failure terminal

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

c. When $\mathrm{C}_T = \mathrm{Switch}(\mathrm{b},\ (\pi{\rightarrow}\mathrm{C}_i)^i)_{/\mathrm{a}\rightarrow\pi}$ then we look at the accessor a of the subtree C_i and we define $\pi_i\mathrm{'} = \pi_i$ if $\mathrm{a}{\neq}\mathrm{b}$ else $\pi_i{\cap}\pi$ Trimming a switch node yields the following result:

$$\mathrm{Switch}(\mathrm{b},\ (\pi{\rightarrow}\mathrm{C}_i)^i)_{/\mathrm{a}\rightarrow\pi} := \mathrm{Switch}(\mathrm{b},\ (\pi'_i{\rightarrow}\mathrm{C}_{\mathrm{i}/\mathrm{a}\rightarrow\pi})^i)$$

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

$$C_T(v_T) = C_{\text{trim}}(v_T) = \text{Switch}(b, (\pi_i' \rightarrow C_{i/a\rightarrow\pi})^i)(v_T)$$

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

$$C_{k/a\rightarrow\pi}(v_T) = \text{Switch}(b, (\pi_i' \rightarrow C_{i/a\rightarrow\pi})^i)(v_T)$$

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

$$C_k(v_T) = C_{k/a\rightarrow\pi}(v_T)$$

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

4.2.2 Equivalence checking

The equivalence checking algorithm takes as parameters an input space S, a source decision tree C_S and a target decision tree C_T:

$$\text{equiv}(S, C_S, C_T) \rightarrow \text{Yes} \quad \text{No}(v_S, v_T)$$

When the algorithm returns Yes and the input space is covered by C_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.

$$\text{equiv}(S, C_S, C_T) = \text{Yes and cover}(C_T, S) \rightarrow \forall\ v_S \simeq v_T \in S \wedge C_S(v_S) = C_T(v_T)$$

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.

$$\text{equiv}(S, C_S, C_T) = \text{No}(v_S,v_T) \text{ and } \text{cover}(C_T, S) \rightarrow \forall\ v_S \simeq v_T \in S \wedge C_S(v_S) \neq C_T(v_T)$$

We define the following

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

There exists and are injective:

$$\begin{aligned}
&\text{int}(k) \in \mathbb{N}\ (\text{arity}(k) = 0)\\
&\text{tag}(k) \in \mathbb{N}\ (\text{arity}(k) > 0)\\
&\pi(k) = \{n|\ \text{int}(k) = n\}\ x\ \{n|\ \text{tag}(k) = n\}
\end{aligned}$$

where k is a constructor.

We proceed by case analysis:

1. in case of unreachable:

$$C_S(v_S) = \text{Absurd}(\text{Unreachable}) \neq C_T(v_T)\ \forall v_S,v_T$$

1. In the case of an empty input space

$$\text{equiv}(\varnothing, C_S, C_T) := \text{Yes}$$

   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*:

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

   Given that $\forall v$, Failure(v) = Failure, the statement holds.

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

equiv(S, Failure as C_S, Switch(a, $(\pi_i \rightarrow C_{Ti})^i$)) := Forall(equiv(S$\cap$a$\rightarrow\pi(k_i)$), C_S, $C_{Ti})^i$)
equiv(S, Leaf bb_S as C_S, Switch(a, $(\pi_i \rightarrow C_{Ti})^i$)) := Forall(equiv(S$\cap$a$\rightarrow\pi(k_i)$), C_S, $C_{Ti})^i$)

The algorithm either returns Yes for every sub-input space S_i := S$\cap$(a$\rightarrow\pi(k_i)$)
and subtree C_{Ti}

$$\text{equiv}(S_i, C_S, C_{Ti}) = \text{Yes } \forall i$$

or we have a counter example v_S, v_T for which

$$v_S \simeq v_T \in S_k \wedge c_S(v_S) \neq C_{Tk}(v_T)$$

then because

$$v_T \in (a \rightarrow \pi_k) \rightarrow C_T(v_T) = C_{Tk}(v_T) \ ,$$
$$v_S \simeq v_T \in S \wedge C_S(v_S) \neq C_T(v_T)$$

we can say that

$$\text{equiv}(S_i, C_S, C_{Ti}) = \text{No}(v_S, v_T) \text{ for some minimal } k \in I$$

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

$$\pi_n = \neg(\bigcup \pi(k_i)^i)$$

The algorithm proceeds by trimming

equiv(S, Switch(a, $(k_i \rightarrow C_{Si})^i$, C_{sf}), C_T) :=
Forall(equiv(S$\cap$(a$\rightarrow\pi(k_i)^i$), C_{Si}, $C_{t/a\rightarrow\pi(k_i)})^i$ + equiv(S$\cap$(a$\rightarrow\pi(k_i)$), C_S, $C_{a\rightarrow\pi_n}$))

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

$$\text{Forall(equiv(} S\cap(a\rightarrow\pi(k_i)^i), C_{Si}, C_{t/a\rightarrow\pi(k_i)})^i = \text{Yes}$$

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

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

We also know that

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

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

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

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,

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

Then we can say that

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

that is enough for proving that

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