1 Translation validation of the Pattern Matching Compiler

1.1 Source program

Our 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

Our 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:

match variable with
| $\text{pattern}_1 \rightarrow \text{expr}_1$
| pattern_2 when guard $\rightarrow \text{expr}_2$
| pattern_3 as var $\rightarrow \text{expr}_3$
$\vdots$
| $\text{p}_n \rightarrow \text{expr}_n$

Patterns could or could not be exhaustive.

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

function
| pattern$_1$ $\rightarrow$ expr$_1$
| pattern$_2$ when guard $\rightarrow$ expr$_2$
| pattern$_3$ as var $\rightarrow$ expr$_3$
$\vdots$
| p$_n$ $\rightarrow$ expr$_n$

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|parameter
or_pattern ::= rule ("|" wildcard|variable|constructor_pattern)+
condition ::= "when" bexpr
assignment ::= "as" id
bexpr ::= _code ;; 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
$(p_1\| p_2)$	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 translators, 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 x n| where m 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 $\vec{p_i} = (\mathrm{p}_1, \mathrm{p}_2, \ldots, \mathrm{p}_n)$; 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 $\vec{v} = (\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_i)$ matches the pattern vector p_i in P if and only if the following two conditions are satisfied:

- $\mathrm{p_{i,1}}, \mathrm{p_{i,2}}, \cdots, \mathrm{p_{i,n}} \preccurlyeq (\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_i)$
- $\forall \mathrm{j} < \mathrm{i}\ \mathrm{p_{j,1}}, \mathrm{p_{j,2}}, \cdots, \mathrm{p_{j,n}} \not\preccurlyeq (\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_i)$

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

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

- first non-$\perp$ element of an ordered family

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

we now define what it means to run a pattern row against a value vector of the same length, that is $(\mathrm{p}_i)^i(\mathrm{v}_i)^i$

p_i	v_i	$\mathrm{result}_{\mathrm{pat}}$
$\varnothing$	$(\varnothing)$	$[]$
$(\_,\ \mathrm{tail}(\mathrm{p}_i)^i)$	(v_i)	$\mathrm{tail}(\mathrm{p}_i)^i(\mathrm{tail}(\mathrm{v}_i)^i)$
$(\mathrm{x},\ \mathrm{tail}(\mathrm{p}_i)^i)$	(v_i)	$\sigma[\mathrm{x}\mapsto\mathrm{v}_0]$ if $\mathrm{tail}(\mathrm{p}_i)^i(\mathrm{tail}(\mathrm{v}_i)^i) = \sigma$
$(\mathrm{K}(\mathrm{q}_j)^j,\ \mathrm{tail}(\mathrm{p}_i)^i)$	$(\mathrm{K}(\mathrm{v'}_j)^j, \mathrm{tail}(\mathrm{v}_j)^j)$	$((\mathrm{q}_j)^j + \mathrm{tail}(\mathrm{p}_i)^i)((\mathrm{v'}_j)^j + \mathrm{tail}(\mathrm{v}_i)^i)$
$(\mathrm{K}(\mathrm{q}_j)^j,\ \mathrm{tail}(\mathrm{p}_i)^i)$	$(\mathrm{K'}(\mathrm{v'}_l)^l, \mathrm{tail}(\mathrm{v}_j)^j)$	$\perp$ if $\mathrm{K} \neq \mathrm{K'}$
$(\mathrm{q}_1 \mid \mathrm{q}_2,\ \mathrm{tail}(\mathrm{p}_i)^i)$	$(\mathrm{v}_i)^i$	$\mathrm{First}((\mathrm{q}_1,\mathrm{tail}(\mathrm{p}_i)^i)(\mathrm{v}_i)^i,\ (\mathrm{q}_2,\mathrm{tail}(\mathrm{p}_i)^i)(\mathrm{v}_i)^i)$

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

$$\mathrm{t}_S ::= (\mathrm{p} \to \mathrm{bb})^{\mathrm{i}\in\mathrm{I}}$$
$$\mathrm{t}_S(\mathrm{v}_S) \to \mathrm{r}$$
$$\mathrm{r} ::= \text{guard list} * (\text{Match bb} \mid \text{NoMatch} \mid \text{Absurd})$$

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

$$\mathrm{t}_S(\mathrm{v}_S) := \perp \qquad \text{if } \forall \mathrm{i},\ \mathrm{p}_i(\mathrm{v}_S) = \perp$$
$$\mathrm{First}((\mathrm{x}_i)^i) := \mathrm{x\_min}\{\mathrm{i} \mid \mathrm{x}_i \neq \perp\} \qquad \text{if } \exists \mathrm{i},\ \mathrm{x}_i \neq \perp$$
$$\mathrm{t}_S(\mathrm{v}_S) = \text{Absurd if } \mathrm{bb\_min}\{\mathrm{p}_i \to \mathrm{bb}_i \mid \mathrm{p}_i(\mathrm{v}_S) \neq \perp\} = \textit{refutation clause}$$
$$\mathrm{t}_S(\mathrm{v}_S) = \text{Match } \mathrm{bb\_min}\{\mathrm{p}_i \to \mathrm{bb}_i \mid \mathrm{p}_i(\mathrm{v}_S) \neq \perp\}$$

[...] Big part that I think doesn't need revision from you [...]

In our prototype we make use of accessors to encode stored values.

```
let value =  1 :: 2 :: 3 :: []
(* that can also be written *)
let value = []
    |> List.cons 3
    |> List.cons 2
    |> List.cons 1
```

```
(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)) =
```

An *accessor* *a* represents the access path to a value that can be reached by deconstructing the scrutinee.

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

The above example, in encoded form:

```
Here = 1
1.Here = 2
1.1.Here = 3
1.1.1.Here = []
```

In our prototype the source matrix m_S is defined as follows

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

Source matrices are used to build source decision trees C_S. 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

$$\text{C}_S ::= \text{Leaf bb} \mid \text{Switch}(\text{a}, (\text{K}_i \to \text{C}_i)^{\text{i}\in\text{S}}, \text{C?}) \mid \text{Failure} \mid \text{Unreachable}$$
$$\text{v}_S ::= \text{K}(\text{v}_i)^{\text{i}\in\text{I}} \mid \text{n} \in \mathbb{N}$$

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

$$\forall \text{v}_S,\ \text{t}_S(\text{v}_S) = [\![\text{t}_S]\!]_S(\text{v}_S)$$

We define the decision tree of source programs $[\![\text{t}_S]\!]$ in terms of the decision tree of pattern matrices $[\![\text{m}_S]\!]$ by the following:

$$[\![((\text{p}_i \to \text{bb}_i)^{\text{i}\in\text{I}}]\!] := [\![(\text{Here}),\ (\text{p}_i \to \text{bb}_i)^{\text{i}\in\text{I}}\ ]\!]$$

Decision tree computed from pattern matrices respect the following invariant:

$$\forall \text{v}\ (\text{v}_i)^{\text{i}\in\text{I}} = \text{v}(\text{a}_i)^{\text{i}\in\text{I}} \to [\![\text{m}]\!](\text{v}) = \text{m}(\text{v}_i)^{\text{i}\in\text{I}} \text{ for m} = ((\text{a}_i)^{\text{i}\in\text{I}},\ (\text{r}_i)^{\text{i}\in\text{I}})$$
$$\text{v}(\text{Here}) = \text{v}$$
$$\text{K}(\text{v}_i)^i(\text{k.a}) = \text{v}_k(\text{a}) \text{ if k} \in [0;\text{n}[$$

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

Base cases:

1. [| ∅, $(\varnothing \to \mathrm{bb}_i)^i$ |] ≡ Leaf bb_i where i := min(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 source matrix in the case of empty rows returns the first expression and (Leaf bb)(v) := Match bb

2. [| $(\mathrm{a}_j)^j$, ∅ |] ≡ Failure

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

- The index family of a constructor

$$\mathrm{Idx}(\mathrm{K}) := [0; \mathrm{arity}(\mathrm{K})[$$

- head of an ordered family (we write x for any object here, value, pattern etc.)

$$\mathrm{head}((\mathrm{x}_i)^{\mathrm{i} \in \mathrm{I}}) = \mathrm{x\_min(I)}$$

- tail of an ordered family

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

- head constructor of a value or pattern

$$\mathrm{constr}(\mathrm{K}(\mathrm{x}_i)^i) = \mathrm{K}$$
$$\mathrm{constr}(\_) = \bot$$
$$\mathrm{constr}(\mathrm{x}) = \bot$$

- first non-⊥ element of an ordered family

$$\mathrm{First}((\mathrm{x}_i)^i) := \bot \qquad \text{if } \forall \mathrm{i},\ \mathrm{x}_i = \bot$$
$$\mathrm{First}((\mathrm{x}_i)^i) := \mathrm{x\_min}\{\mathrm{i} \mid \mathrm{x}_i \neq \bot\} \qquad \text{if } \exists \mathrm{i},\ \mathrm{x}_i \neq \bot$$

- definition of group decomposition:

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

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" m_{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 m_{K_k}; otherwise (its head constructor $\notin (\text{K}_k)^k$) 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

$$\text{Groups(m)} = (\text{k\_r} \to \text{m\_r})\text{\^{}k},\ \text{m}_{\text{wild}}$$

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

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

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 $K_k \in \text{constrs}(p_{0j})^j$), we have to prove that

$$(p_{ij})^i(v_i)^i = r_{kj}((v'_l)^l + \text{tail}(v_i)^i$$

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

$$\begin{array}{l}
\text{if } p_{oj} \text{ is } K_k(q_l) \text{ then} \\
\quad (q_l)^l + \text{tail}(p_{ij})^i \to e_j \\
\text{if } p_{oj} \text{ is _ then} \\
\quad (_)^l + \text{tail}(p_{ij})^i \to e_j \\
\text{else } \bot
\end{array}$$

By definition of $(p_i)^i(v_i)^i$ we know that $(p_{ij})^i(v_i)$ is equal to

$$\begin{array}{l}
(K(q_j)^j,\ \text{tail}(p_{ij})^i)\ (K(v'_l)^l,\text{tail}(v_i)^i) := ((q_j)^j + \text{tail}(p_{ij})^i)((v'_l)^l + \text{tail}(v_i)^i) \\
(_,\ \text{tail}(p_{ij})^i)\ (v_i) := \text{tail}(p_{ij})^i(\text{tail}(v_i)^i) \\
(K(q_j)^j,\ \text{tail}(p_{ij})^i)\ (K'(v'_l)^l,\text{tail}(v_j)^j) := \bot \text{ if } K \neq K'
\end{array}$$

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 $K_k \in \text{constrs}(p_{oj})^j$), we have to prove that

$$(p_{ij})^i(v_i)^i = r_{j\,\text{wild}}\ \text{tail}(v_i)^i$$

TODO

1.2 Target translation

The target 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 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. Integer additions and subtractions are simplified in a second pass. The result of the symbolic evaluation is a target decision tree C_T

$$C_T ::= \mathrm{Leaf}\ bb \mid \mathrm{Switch}(a, (\pi_i \to C_i)^{i\in S}, C?) \mid \mathrm{Failure}$$
$$v_T ::= \mathrm{Cell}(tag \in \mathbb{N}, (v_i)^{i\in I}) \mid n \in \mathbb{N}$$

Every branch of the decision tree is "constrained" by a domain

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

Intuitively, the π condition at every branch tells us the set of possible values that can "flow" through that path. π 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. C? 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 C? fallback node is

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

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

$$\text{domain}(\text{v}_S(\text{a})) = \bigcup_{\text{i}\in\text{I}}\pi_i$$

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

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

1.3 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. Our 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.

$$\begin{array}{rl} & \textit{constraint trees} \\ C ::= & \mathsf{Leaf}(t) \\ & \mid \mathsf{Failure} \\ & \mid \mathsf{Switch}(a, (\pi_i, C_i)^i, C_{\mathsf{fb}}) \\ & \mid \mathsf{Guard}(t, C_0, C_1) \end{array} \qquad \begin{array}{rl} & \textit{boolean result} \\ b & \in \{0, 1\} \\ & \textit{guard queues} \\ G ::= & (t_1 = b_1), \ldots, (t_n = b_n) \end{array}$$

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

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

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

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

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

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

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

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:

$$(\ [\![t_S]\!]_S(v_S) \equiv C_S(v_S)\ ) \to r$$
$$t_S(v_S) \to r$$

Likewise

$(\ [\![t_T]\!]_T(v_T) \equiv C_T(v_T)\ ) \to r$

$t_T(v_T) \to r$

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, $v_S \simeq v_T$, we can define the equivalence between a couple of programs or a couple of decision trees

$$t_S \simeq t_T := \forall v_S \simeq v_T,\ t_S(v_S) = t_T(v_T)$$
$$C_S \simeq C_T := \forall v_S \simeq v_T,\ C_S(v_S) = C_T(v_T)$$

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:

$$(gs_1,\ r_1) \simeq gs\ (gs_2,\ r_2) \Leftrightarrow (gs_1,\ r_1) = (gs_2 \mathbin{++} gs,\ r_2)$$

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

$$\text{equiv}(S,\ C_S,\ C_T,\ gs) = \text{Yes} \wedge \text{covers}(C_T,\ S) \to \forall v_S \simeq v_T \in S,\ C_S(v_S) \simeq gs\ C_T(v_T)$$

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

$$\text{equiv}(S,\ C_S,\ C_T,\ gs) = \text{No}(v_S,v_T) \wedge \text{covers}(C_T,\ S) \to v_S \simeq v_T \in S \wedge C_S(v_S) \not\simeq gs\ C_T(v_T)$$

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

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

1.3.1 The trimming lemma

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

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

We prove this by induction on C_T:

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

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

- The same applies to Failure terminal

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

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

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

For the trimming lemma we have to prove that running the value v_T against the decision tree C_T is the same as running v_T against the tree C_{trim} that is the result of the trimming operation on C_T

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

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

$$\text{C}_{\text{k/a}\rightarrow\pi}(\text{v}_T) = \text{Switch}(\text{b}, (\pi_i\text{'}\rightarrow\text{C}_{\text{i/a}\rightarrow\pi})^{\text{i}\in\text{I}})(\text{v}_T)$$

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

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

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

1.3.2 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:

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

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.

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

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.

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

We define the following

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

There exists and are injective:

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

where k is a constructor.

We proceed by case analysis:

1. in case of unreachable:

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

1. In the case of an empty input space

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

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

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

$$\begin{aligned}
&\mathrm{equiv(S,\ Failure\ as\ C}_S\mathrm{,\ Switch(a,\ (}\pi_i \to \mathrm{C}_{Ti})^{\mathrm{i}\in\mathrm{I}})) := \mathrm{Forall(equiv(\ S}\cap\mathrm{a}\to\pi(\mathrm{k}_i)),\ \mathrm{C}_S,\ \mathrm{C}_{Ti})^{\mathrm{i}\in\mathrm{I}})\\
&\mathrm{equiv(S,\ Leaf\ bb}_S\mathrm{\ as\ C}_S\mathrm{,\ Switch(a,\ (}\pi_i \to \mathrm{C}_{Ti})^{\mathrm{i}\in\mathrm{I}})) := \mathrm{Forall(equiv(\ S}\cap\mathrm{a}\to\pi(\mathrm{k}_i)),\ \mathrm{C}_S,\ \mathrm{C}_{Ti})^{\mathrm{i}\in\mathrm{I}})
\end{aligned}$$

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

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

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

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

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_{fallback} as the domain of values not covered but the union of the constructors k_i

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

Our algorithm proceeds by trimming

$$\begin{aligned}&\text{equiv}(S,\ \text{Switch}(a,\ (k_i \rightarrow C_{Si})^{i \in I},\ C_{sf}),\ C_T) :=\\ &\text{Forall}(\text{equiv}(\ S \cap (a \rightarrow \pi(k_i)^{i \in I}),\ C_{Si},\ C_{t/a \rightarrow \pi(k_i)})^{i \in I} + \text{equiv}(S \cap (a \rightarrow \pi_n),\ C_S,\ C_{a \rightarrow \pi_{\text{fallback}}}))\end{aligned}$$

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

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

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

$$k \mid k \in (k_i)^i \wedge C_S(v_S) = C_{Si}(v_S)$$

We also know that

$$\begin{aligned}&(1)\ C_{Si}(v_S) = C_{t/a \rightarrow \pi_i}(v_T)\\ &(2)\ C_{T/a \rightarrow \pi_i}(v_T) = C_T(v_T)\end{aligned}$$

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

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

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,

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

Then we can say that

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

that is enough for proving that

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