Translation Verification of the pattern matching compiler

Francesco Mecca

Abstract

This dissertation presents an algorithm for the translation validation of the pattern matching compiler. Given the source representation of the target program and the target program compiled in untyped lambda form, the algoritmhm is capable of modelling the source program in terms of symbolic constraints on it's branches and apply symbolic execution on the untyped lambda representation in order to validate wheter the compilation produced a valid result. In this context a valid result means that for every input in the domain of the source program the untyped lambda translation produces the same output as the source program. The input of the program is modelled in terms of symbolic constraints closely related to the runtime representation of objects and the output consists of OCaml code blackboxes that are not evaluated in the context of the verification.

1 Background

1.1

Objective Caml () is a dialect of the ML (Meta-Language) family of programming languages. shares many 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 lenght 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 modelled 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 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 reificated through functors.

1.2 Lambda form compilation

provides compilation in form of a byecode executable with an optionally embeddable interpreter and a native executable that could be statically linked to provide a single file executable.

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. On the *Lambda* representation of the source program, the compiler performes a series of optimization passes before translating the lambda form to assembly code.

1. datatypes

   Most native data types in , such as integers, tuples, lists, records, can be seen as instances of the following definition

   ```
   type t = Nil | One of int | Cons of int * t
   ```

   that is a type *t* with three constructors that define its complete signature. Every constructor has an arity. Nil, a constructor of arity 0, is called a constant constructor.

2. Lambda form types A lambda form target file produced by the compiler 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

   There 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 varios numeric operations defined:

   - Arithmetic operations: +, -, *, /, % (modulo), neg (unary negation)

- Bitwise operations: &, |, ˆ, «, » (zero-shifting), a» (sign extending)
- Numeric comparisons: <, >, <=, >=, ==

3. 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.

4. Bindings The atom *let* introduces a sequence of bindings: (let BINDING BINDING BINDING ... BODY)

5. Other atoms TODO: if, switch, stringswitch... TODO: magari esempi

1.3 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 patten 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 nu
| (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.

1.4 Symbolic execution

1.5 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

1.6 Translation validation of the Pattern Matching Compiler

1. Source program The algorithm takes as its input a source program and translates it into an algebraic data structure called *constraint_ tree.*

```
type constraint_tree =
  | Unreachable
  | Failure
  | Leaf of source_expr
  | Guard of source_blackbox * constraint_tree * constraint_tree
  | Node of accessor * (constructor * constraint_tree) list * constraint_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 blackbox 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
```

[ ] Converti a disegni

Is translated to

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

while

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

will give

Node ([(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 Node ([(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.

[ ] Finisci con Node [ ] Spiega del fallback [ ] rivedi compilazione per tenere in considerazione il tuo albero invece che le lambda [ ] Specifica che stesso algoritmo usato per compilare a lambda, piu‘ optimizations

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

```
match variable with
| pattern$_1$ -> expr$_1$
| pattern$_2$ when guard -> expr$_2$
| pattern$_3$ as var -> expr$_3$
⋮
| p$_n$ -> expr$_n$
```

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

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

```
function
| pattern$_1$ -> expr$_1$
| pattern$_2$ when guard -> expr$_2$
| pattern$_3$ as var -> expr$_3$
⋮
| p$_n$ -> 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 ($\varepsilon$|condition)
rexpr ::= _code ;; arbitrary code

rule ::= wildcard|variable|constructor_pattern|or_pattern ;;

;; rules
wildcard ::= "_"
variable ::= identifier
```

```
constructor_pattern ::= constructor (rule|$\varepsilon$) (assignment|$\varepsilon$

constructor ::= int|float|char|string|bool
                |unit|record|exn|objects|ref
                |list|tuple|array
                |variant|parameterized_variant ;;  data types

or_pattern ::=  wildcard|variable|constructor_pattern ("|" wildcard|variable|const

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$_2$,...,p$_n$)	constructor pattern
(p$_1$| 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$_{\mathrm{i,1}}$, p$_{\mathrm{i,2}}$, ..., p$_{\mathrm{i,n}}$) 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 $\vec{v} = (\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_i)$ matches the line number i in P if and only if the following two conditions are satisfied:

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

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

- 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

(a) Initial state of the compilation
Given a source of the following form:

match variable with
| $\mathrm{pattern}_1$ -> e_1
| $\mathrm{pattern}_2$ -> e_2
⋮
| p_m -> e_m

the initial input of the 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, that is $\vec{x} = ()$, then 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:

i. Variable rule: if all patterns of the first column of P are wildcard patterns or bind the value to a variable, then

$$C(\vec{x}, P \to L) = C((x_2, x_3, ..., x_n), P' \to L')$$

where

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

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.

ii. 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 ∈ 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 $y_1, y_2, \ldots, y_a$ 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 $\{c_1, c_2, \ldots, c_k\}$ is:

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

i. 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

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

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

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

ii. Mixture rule: When none of the previous rules apply the clause matrix P $\rightarrow$ L is splitted 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.