OCaml front end for the Stainless verifier

Julie Giunta

School of Computer and Communication Sciences
Master Semester Project Report

Supervisor
Dragana Milovancevic
EPFL / LARA

Supervisor
Prof. Viktor Kunčak
EPFL / LARA

June 2021

OCaml front end for the Stainless verifier

Julie Giunta
EPFL, Switzerland
julie.giunta@epfl.ch

Abstract

OCaml is a programming language supporting functional, imperative and object-oriented styles. One might want to verify OCaml programs using Stainless, a tool for verifying Scala programs. To this extent, a front end is needed to translate OCaml to Scala.

1 Introduction

The goal of this project was to implement an OCaml front end for the Stainless verifier. This front end allows the parsing of an OCaml program which is then translated in Scala, in such a way that it can be used within Stainless. For this purpose, we:

1. Retrieve the types of the variables that are part of the interface of the OCaml program, using the *ocamlc* compiler(1).

2. Insert the retrieved types in the OCaml program.

3. Parse the OCaml program using Camlp5(2).

4. Print the OCaml program in Scala using our own printer that overwrites Camlp5's printer.

5. Wrap the translated program in an object and add the import statements.

These different steps will be explained with more details in section 3.

2 Camlp5

Camlp5 is a preprocessor and pretty-printer for OCaml programs. It provides provides parsing and printing tools too.

Since reinventing the wheel is often a bad idea, we decided to use Camlp5 to parse the OCaml program so that we did not have to create our own grammar and parser. To perform the translation, we decided to overwrite part of Camlp5's printer. Our printer handles nodes of type *expr*, *patt*, *ctyp* and *str_item*, which are described in the Pcaml module(3).

Our system follows this logic: first, we parse the OCaml program using Camlp5's parser. We obtain a syntax tree(4) that has type MLast. We print that syntax tree using our own printer.

3 Implementation

The front end is composed of three main files:

- A shell script that executes the steps described in section 1. With more details, those steps are:

  1. Retrieve the interface of the OCaml file to translate using the **ocamlc -i** command.
  2. Compile the Scala program that inserts types in the OCaml program with the **scalac** command.
  3. Run the Scala program. The program receives as input the paths to the OCaml file to translate, the interface of the OCaml file obtained in step 1 and the file to write the output to.
  4. Compile *pr_scala.ml*, the printer written in OCaml with the command:

     ```
     $ ocamlfind ocamlc -syntax
         camlp5r -package camlp5
         -linkpkg pr_scala.ml
     ```

  5. Run the OCaml program to translate the file obtained in step 3 to Scala. The command used is:

     ```
     $ camlp5o pr_o.cmo
        ./pr_scala.cmo $
        OCAML_INPUT_WITH_TYPES > $
        SCALA_OUTPUT_FILE
     ```

  6. Wrap the translated program in an object and insert the import statements with the **sed** command.
  7. Delete all the unnecessary files created during the operation with the **rm** command.

- A Scala program that parses the interface of the OCaml file and inserts the types found in the interface in the OCaml program.

- An OCaml program that overwrites part of Camlp5's printer. This program defines how OCaml's expressions, patterns, types and structures in general should be printed in Scala.

The code can be found here: `https://github.com/GiuntaJ/OCaml-front-end-for-the-Stainless-verifier`

3.1 Structure items

Structure items are the main statements of an OCaml implementation file (.ml file). Our printer handles the translation of types declarations, exceptions declarations, variables definitions and expressions. If the nodes of the parsed tree does not match any of those cases, the printer fails.

3.1.1 Type declarations

It is possible to define custom data types in OCaml(5).

These types can be aliases of another type. We translate this case using the keyword **type**, so '**type** alias = char;;' becomes '**type** Alias = Char'.

A custom data type might have multiple values. If this is the case, we translate the new type as an abstract class and each value is translated as a case object or case class that extends the new type. We use case objects for types that do not carry data and case classes otherwise. Since this data is not labeled in OCaml, we gave arbitrary names to the parameters. For example,

```
type simple = A | B
and with_data =
  | A2
  | B2 of int
  | C2 of char * int;;
```

becomes:

```
sealed abstract class Simple {}
case object A extends Simple {}
case object B extends Simple {}

sealed abstract class With_data {}
case object A2 extends With_data {}
case class B2(param0: Int63) extends
    With_data {}
case class C2(param0: Char, param1:
    Int63) extends With_data {}
```

Newly defined types might be polymorphic. When this is the case, we retrieve each type parameter and construct a generic class, as shown below:

```
type 'a polymorphic =
  | A3
  | B3 of 'a * 'a polymorphic;;
```

is translated to:

```
sealed abstract class Polymorphic[A] {}
case class A3[A]() extends
    Polymorphic[A] {}
case class B3[A](param0: A, param1:
    Polymorphic[A]) extends
    Polymorphic[A] {}
```

Finally, records are translated as case classes with labeled parameters. If a parameter is **mutable**, we use the keyword **var** to signal it. With the input:

```
type 'a record =
  {first : 'a list;
   mutable second : char};;
```

we retrieve as output:

```
sealed case class Record[A](first:
    List[A], var second: Char) {}
```

3.1.2 Exceptions

In OCaml, you can create custom exceptions. They can carry data. To translate these exceptions, we create case classes that extends **Exception** from Stainless. Here are some examples of inputs:

```
exception Failure_simple;;
exception Failure_string of string;;
exception Failure_complex of int *
    string;;
```

that give the following outputs:

```
sealed case class Failure_simple()
    extends Exception {}
sealed case class
    Failure_string(param0: String)
    extends Exception {}
sealed case class Failure_complex(
     param0: Int63,
     param1: String
) extends Exception {}
```

3.1.3 Variables

Variables are top-level let bindings in OCaml. They can be recursive if they contain the keyword **rec** and they may have multiple arguments. We decided to translate variables that are not recursive nor have arguments with the Scala keyword **val**. Recursive variables and variables that have arguments (functions) are translated using the keyword **def**. Mutable variables are defined using the keyword **var** (see 3.2.7 and 3.2.8). It is also possible to define multiple variables at the same time with the keyword **and**. We translate each of these variables separately and create a new **val**, **var** or **def** for each of them. For functions, it is possible to use generic types. When this is the case we retrieve those types and put them into brackets just after the function name. For example, if the input is:

```
let value_non_rec = 'a';;
let rec value_rec = 'b';;
let fun_non_rec x y = x :: y;;
let rec fun_rec n =
  if n <= 1 then 1
  else fun_rec (n - 1);;
let first = 'a' and second = "b" and
    third = 3;;
```

the output will be:

```
val value_non_rec: Char = 'a'
def value_rec: Char = 'b'
def fun_non_rec[A](x: A, y: List[A]):
    List[A] = { x :: y }
def fun_rec(n: Int63): Int63 = { if (n
    <= 1) 1 else fun_rec(n - 1) }
```

```
val first: Char = 'a'
val second: String = "b"
val third: Int63 = 3
```

3.1.4 Expressions

Simple expressions are translated using *pr_expr*, the printer for expressions, as described in section 3.2

3.2 Expressions

The parts of an OCaml program that actually do something are expressions. After parsing the input file, the AST will contain nodes of type *expr*. To translate these nodes, we override *pr_expr*, the printer for expressions. The expressions for which the translation was implemented are described below. If the program contains an expression that was not implemented, a failure is raised.

3.2.1 Basic expressions

For integers, floats, chars, strings and identifiers (which includes true, false, '()' and '[]'), we translate OCaml as follows. With input:

```
let x_int = 10;;
let x_int_big = 4611686018427387903;;
let x_float = 1.;;
let x_char = 'a';;
let x_string = "abc";;
let x_true = true;;
let x_false = false;;
let x_unit = ();;
let x_nil = [];;
```

we obtain:

```
val x_int: Int63 = 10
val x_int_big: Int63 = BitVectors.
    fromLong(4611686018427387903L).
    narrow[Int63]
val x_float: Double = 1.0
val x_char: Char = 'a'
val x_string: String = "abc"
val x_true: Boolean = true
val x_false: Boolean = false
val x_unit: Unit = ()
val x_nil: List[A] = Nil()
```

We need to perform a special translation for integer literals that are greater that Int.MaxValue from Scala since it is not accepted. We need to transform then into a **Long** and then into an **Int63**.

Doubles are not accepted in Stainless but they can be compiled with Scala.

'*Nil()*' should have type **Nil** or **List[Nothing]**. Since it is a really specific case, that is not really used in general, we let the user determine which type it should have themselves.

For identifiers, since OCaml does not have the same reserved keywords as Scala and their identifiers might contain the symbol **'**, we change identifiers that clash with Scala while translating. To do so, we keep track of all the identifiers we see and when we see an identifier that corresponds to a reserved keyword from Scala or that contains **'**, we change that identifier. To do so, we replace all the **'** by _ and add a number at the end of the identifier. We verify that this new identifier does not already exist and if it exist, we add a new number at the end of the identifier. When we have found a unique identifier, we register the mapping in a list. So:

```
let special_id' = 'b';;
let duplicate_id_0 = special_id';;
let duplicate_id' = 'c';;
let var = "abc";;
```

becomes:

```
val special_id_0: Char = 'b'
val duplicate_id_0: Char = special_id_0
val duplicate_id_01: Char = 'c'
val var0: String = "abc"
```

Translation for accessing the nth element of a string and concatenating strings are also available, but only when you compile the program with Scala and not Stainless since Stainless lacks some support for strings. For example, with input:

```
let string_concat = "abc" ^ "def";;
let string_access = string_concat.[4];;
```

we get:

```
val string_concat: String = "abc" +
    "def"
val string_access: Char =
    string_concat.apply(4)
```

3.2.2 Boolean operations

The front end support the translation of '||' and '&&' as well as the deprecated 'or' and '&'. For the logical or, for either of these inputs:

```
let or_bool = true || false;;
let or_bool = true or false;;
```

we will retrieve this output:

```
val or_bool: Boolean = true || false
```

For the logical and, for either of these inputs:

```
let and_bool = true && false;;
let and_bool_deprecated = true &
    false;;
```

we will retrieve:

```
val and_bool: Boolean = true && false
```

3.2.3 Comparisons

The less than ('<'), less equal ('<='), greater than ('>') and greater equal ('>=') operators are the same in OCaml and Scala.

For equality checks we translate '=', which is used for structural equality in OCaml(6) to '==', which is the Scala equivalent. The not equal for structural equality ('<>') is translated '! ='. For physical equality(6), we translate OCaml's '==' to 'eq' and '! =' to 'ne', which are the Scala equivalents. Unfortunately, 'eq' and 'ne' are not available in Stainless but they can be compiled with Scala.

Here are some examples:

```
let less = 0 < 1;;
let less_equal = 0 <= 1;;
let greater = 0 > 1;;
let greater_equal = 0 >= 1;;
let equal = 0 = 1;;
let not_equal = 0 <> 1;;
let array = [|1; 2|];;
let equal_ref = array == [|1; 2|];;
let not_equal_ref = [|1; 2|] != array;;
```

we will retrieve:

```
val less: Boolean = 0 < 1
val less_equal: Boolean = 0 <= 1
val greater: Boolean = 0 > 1
val greater_equal: Boolean = 0 >= 1
val equal: Boolean = 0 == 1
val not_equal: Boolean = 0 != 1
var array: Array[Int63] = Array(1, 2)
val equal_ref: Boolean = array eq
    Array(1, 2)
val not_equal_ref: Boolean = Array(1,
    2) ne array
```

3.2.4 Mathematical operations

The front end supports unary operators for OCaml integers and floats ('−', '−.', '∼ +' and '∼ +.'). Those operators are translated to '−' and '+', for integers and floats. The translation of this input:

```
let unary_minus_int = -1 + 1;;
let unary_plus_int = ~+ 1 - 1;;
let unary_minus_float = -. 1.0 +. 1.0;;
let unary_plus_float = ~+. 1.0 -. 1.0;;
```

will return:

```
val unary_minus_int_2: Int63 = -(1) + 1
val unary_plus_int: Int63 = +(1) - 1
val unary_minus_float: Double = -(1.0)
    + 1.0
val unary_plus_float: Double = +(1.0)
    - 1.0
```

Doubles are not accepted in Stainless but they are available in Scala. Some problems may arise with **Int63** too.

For both integers and floats, addition ('+', '+.'), subtraction ('−', '−.'), multiplication ('∗', '∗.') and division ('/', '/.') operators are supported. They are respectively translated to '+', '−', '∗' and '/'. For example, the input:

```
let plus_int = x_int + 1;;
let minus_float = x_float -. 1.5;;
let mul_int = x_int * 2;;
let divide_float = x_float /. 1.5;;
```

will produce the output:

```
val plus_int: Int63 = x_int + 1
val minus_float: Double = x_float - 1.5
val mul_int: Int63 = x_int * 2
val divide_float: Double = x_float /
    1.5
```

For integers, modulo ('mod') and bitwise operations (and ('land'), or ('lor'), xor ('lxor'), logical shift left ('lsl'), logical shift right ('lsr') and arithmetic shift left ('asr')) are available. They are respectively translated to '%', '&', '|', 'ˆ', '<<', '>>'and '>>>'. The input:

```
let mod_int = x_int mod 3;;
let land_int = x_int land 8;;
let lor_int = x_int lor 1;;
let lxor_int = x_int lxor 8;;
let asr_int = x_int asr 1;;
let lsl_int = x_int lsl 1;;
let lsr_int = x_int lsr 1;;
```

will produce the output:

```
val mod_int: Int63 = x_int % 3
val land_int: Int63 = x_int & 8
val lor_int: Int63 = x_int | 1
val lxor_int: Int63 = x_int ^ 8
val asr_int: Int63 = x_int >>> 1
val lsl_int: Int63 = x_int << 1
val lsr_int: Int63 = x_int >> 1
```

3.2.5 Tuples

Tuples in Scala are almost identical to OCaml tuples, except that they may contain maximum 22 elements. If we detect a tuple in the OCaml program that has more that 22 elements, we raise a failure.

3.2.6 Lists

For lists, instead of creating them with brackets ([ ]), we use the constructor **List**. For the empty list ('[]'), we use the constructor **Nil()**. The '::' operator is either translated by directly creating the list using the **List()** constructor if we have all the elements, or by using '::'. For the '@' operator (concatenation), we translate this with '++'. Here is an example of input:

```
let list = ['a'];;
let list_of_list = [['a']; ['b']];;
let list_cons = 'a' :: list;;
let list_cons_2 = 1 :: 2 :: [];;
let list_cons_3 = 'a' :: 'b' :: ['c';
    'd'];;
let list_concat_1 = ['a'; 'b'] @ ['c';
    'd'];;
let list_concat_2 = 'a' ::
    list_concat_1 @ [];;
```

with output:

```
val list: List[Char] = List('a')
val list_of_list: List[List[Char]] =
    List(List('a'), List('b'))
val list_cons: List[Char] = 'a' :: list
val list_cons_2: List[Int63] = List(1,
    2)
val list_cons_3: List[Char] =
    List('a', 'b', 'c', 'd')
val list_concat_1: List[Char] =
    List('a', 'b') ++ List('c', 'd')
val list_concat_2: List[Char] = 'a' ::
    list_concat_1 ++ Nil()
```

Since we are using **Int63** for ints, some problems may appear when creating lists of integers because the implicit conversion from integer literals to **Int63** might not work.

As explained in section 3.2.13, some functions are also available for lists.

3.2.7 Arrays

We translate OCaml arrays into **Array**. Since an array is a mutable object, when an array is assigned to a variable, we need to use the keyword **var** instead of **val**. This might cause problems when we use Stainless to verify the program since for Stainless, variables are only allowed within functions and as constructor parameters. We tried to wrap the entirety of the code in a function but this created problems with some programs. If the user wants to use arrays, they might want to wrap the part of the code that use the array in a function themselves. We also tried to use the keyword **def** but this made the array to not update correctly.

To access an element of the array, we use parenthesis. To update an element of the array, we use the Scala method **update**. For example, with input:

```
let array = [|0; 2|];;
let array_assignment = array.(0) <- 1;;
let array_access = array.(1);;
let tuple_with_array = (1, [|1; 2|]);;
let tuple_without_array = (1, 2);;
```

the output is:

```
var array: Array[Int63] = Array(0, 2)
val array_assignment: Unit =
    array.update(0, 1)
val array_access: Int63 = array(1)
var tuple_with_array: (Int63,
    Array[Int63]) = (1, Array(1, 2))
val tuple_without_array: (Int63,
    Int63) = (1, 2)
```

3.2.8 Records

As described in section 3.1.1, records definitions are translated into case classes. Thus, for creating a record value, we encountered a problem. In OCaml, to create a value, the user needs to assign to each parameter a value but they do not need to precise which record type is used. Since the parameters are labeled, the user can assign a value to them in any order. In Scala, when we use case classes, the user needs to know which case class is used. Then, the parameters need to be given in the same order as the case class definition. To solve this clash between the two languages, each time we detect a record definition, we store in an associative list a pair '((number_of_parameters, list_of_parameters_in_order), (record_name, is_mutable))'. Then, when we detect the creation of a record value, we look into this list to retrieve the record name and the order of the parameters. If we have this input in OCaml:

```
type record_type =
    { field1: char;
      field2: string
};;
let record_value =
    { field1 = 'a';
      field2 = "abc"
};;
let record_value_with_mixed_parameters
    =
    { field2 = "def";
      field1 = 'b'
};;
```

we will retrieve this output:

```
case class Record_type(field1: Char,
    field2: String) {}
val record_value: Record_type =
    Record_type('a', "abc")
val
    record_value_with_mixed_parameters:
    Record_type = Record_type('b',
    "def")
```

To copy records, OCaml uses the keyword **with**. We translated this using the **copy** method in Scala.

```
let record_copie =
  { record_value with
        field1 = 'c'};;
```

becomes:

```
val record_copie: Record_type =
    record_value.copy(field1 = 'c')
```

When a record has mutable fields, we uses the keyword **var** instead of **val**, as for mutable arrays in section 3.2.7. We translated assignments to mutable fields like this:

```
type record_type =
  { normal_field: char;
    mutable mutable_field: string
  };;
let record_assign_to_mutable =
  record_value.mutable_field <-
    record_value.mutable_field ^
       "def";;
```

becomes:

```
case class Record_type(
  normal_field: Char,
  var mutable_field: String
)  {}
val record_assign_to_mutable: Unit =
    record_value.mutable_field =
        record_value.mutable_field +
        "def"
```

The last example also shows how one can access parameters of records.

3.2.9 Sequences

In OCaml, sequences of statements are written with semi-colons to separate statements and can be written between parenthesis, between the keywords **begin** and **end** and or just as is. To translate sequences, we translate each statement separately and then put everything inside brackets. For example, if we have one of those three OCaml inputs:

```
let sequence =
  begin
    print_endline "First";
    print_endline "Second"
  end;;
let sequence =
  (
    print_endline "First";
    print_endline "Second"
  );;
let sequence = print_endline "First";
   print_endline "Second";;
```

we will retrieve this output in Scala:

```
val sequence: Unit = {
    println("First")
    println("Second")
}
```

When there are nested sequences, we flatten those sequences.

```
let nested_sequences =
  begin
    print_endline "First";
    begin
      print_endline "Second";
      print_endline "Third"
    end
  end;;
```

becomes:

```
val nested_sequences: Unit = {
    println("First")
    println("Second")
    println("Third")
}
```

3.2.10 If statements

In OCaml, if statements have a condition, a **then** clause and optionally an **else** clause. When the then or else clauses are on multiple lines, we wrap them into brackets to ensure the correctness of the execution. With the input:

```
let if_then_one_line = if 1 > 0 then
    ();;
let if_then_else_one_line = if 1 < 0
    then 1 else 0;;
let if_then_else_multi_lines = if
    false then
     print_endline "then␣clause␣that␣is␣
        too␣long␣for␣one␣line"
     else print_endline "else␣clause";;
```

we obtain the output:

```
val if_then_one_line: Unit = if (1 >
    0) ()
val if_then_else_one_line: Int63 = if
    (1 < 0) 1 else 0
val if_then_else_multi_lines: Unit =
     if (
       false
     ) {
       println("then␣clause␣that␣is␣too␣
          long␣for␣one␣line")
     } else {
       println("else␣clause")
     }
```

3.2.11 Pattern matching

Pattern matching is quite similar in OCaml and Scala. Instead of using the keyword **when** for pattern guards, Scala uses the keyword **if**. To ensure the correctness of the execution, we wrap the matching expression in brackets. For pattern matching on lists, we use the case class **Cons** instead of the '::' symbol for the program to work better with Stainless. Here are some examples with pattern matching on case classes, on lists and on options. With the input:

```
type case_class =
    | A
    | B of int
    | C of char * int;;
let rec match_with_case_classes e =
  match e with
    | A -> 0
    | B (e0) -> e0
    | C (e0, e1) -> e1;;
let rec match_with_lists e =
  match e with
     [] -> 1
  | hd::tl when hd > 10 ->
      match_with_lists tl
  | _ -> 0;;
let match_with_options e = match e with
    Some x -> true
  | None -> false;;
```

we get the output:

```
sealed abstract class Case_class {}
case object A extends Case_class {}
case class B(param0: Int63) extends
    Case_class {}
case class C(param0: Char,  param1:
    Int63) extends Case_class {}
def match_with_case_classes(e:
    Case_class): Int63 = {
     e match {
        case A => { 0 }
        case B(e0) => { e0 }
        case C(e0, e1) => { e1 }
     }
}
def match_with_lists(e: List[Int63]):
    Int63 = {
     e match {
        case Nil() => { 1 }
        case Cons(hd, tl) if hd > 10 =>
            { match_with_lists(tl) }
        case _ => { 0 }
     }
}
def match_with_options[A](e:
    Option[A]): Boolean = {
     e match {
        case Some(x) => { true }
        case None() => { false }
     }
}
```

3.2.12 Functions

OCaml offers the possibility to create anonymous functions. The keyword **function** is similar to pattern matching and the keyword **fun** works similarly as lambdas in Scala. When there are multiple parameters, we create a matching using the keyword **case**. Here is some examples of inputs:

```
let function_def = function
      | h::t when h > 10 -> t
      | [] -> []
      | _ -> [];;
let fun_def = fun x -> if x > 10 then
    false else true;;
let fun_def2 = fun x y -> if x > 10 &&
    y then false else true;;
```

with output:

```
val function_def: List[Int63] =>
    List[Int63] = (
     x =>
      x match {
        case Cons(h, t) if h > 10 => {
            t }
        case Nil() => { Nil() }
        case _ => { Nil() }
      }
    )
val fun_def: Int63 => Boolean = ( (x)
    => { if (x > 10) false else true } )
val fun_def2: (Int63, Boolean) =>
    Boolean = {
     case (x, y) => { if (x > 10 && y)
         false else true }
}
```

3.2.13 Function applications

In OCaml, to apply functions, one usually put the function name followed by the arguments separated by spaces. In Scala, one usually puts the function name and then the arguments, separated by commas, between parenthesis. With input:

```
let fun1 a = 10;;
let fun2 a b = a + b;;
let apply1 = fun1 (1 + 2);;
let apply2 = fun2 1 2;;
```

we will get:

```
def fun1[A](a: A): Int63 = { 10 }
def fun2(a: Int63, b: Int63): Int63 =
    { a + b }
val apply1: Int63 = fun1(1 + 2)
val apply2: Int63 = fun2(1, 2)
```

It is also possible to do partial application of functions in OCaml. Unfortunately, it is complicated to translate in Scala as is since it sometimes requires creating a new variable. If the user wants to use partial application, they might want to create an intermediate variable that they will translate themselves.

Since list functions(7) are often used, we translated directly some of them (the ones that had a correspondence in Stainless). We support '*append*', '*concat*', '*cons*', '*exists*', '*filter*', '*find_opt*', '*flatten*', '*forall*', '*hd*', '*length*', '*map*', '*mem*', '*nth*', '*tl*' and '*rev*'. Most of list functions are directly applied to the list using '.' in Scala. It results in this kind of translation:

```
let list_head = List.hd list;;
let list_map = List.map (fun x -> 'b')
    list;;
let list_mem = List.mem 'a' list;;
```

From the input above, we get:

```
val list_head: Char = list.head
val list_map: List[Char] = list.map((
    (x) => { 'b' } ))
val list_mem: Boolean =
    list.contains('a')
```

It is possible to encounter some problems with '*length*' since we are using **Int63** and it returns **BigInt**. It can also happen with '*nth*' whose translation requires a **BigInt** as argument.

We provide supports for some functions of the module Option(8) too. The supported functions are: '*some*', '*get*', '*is_none*', '*is_some*' and '*map*'. For example we can have:

```
let some = Option.some 1;;
let is_none = Option.is_none some;;
let map = Option.map (fun x -> 2)
    some;;
```

that outputs:

```
val some: Option[Int63] = Some(1)
val is_none: Boolean = some.isEmpty
val map: Option[Int63] = some.map((
    (x) => { 2 } ))
```

To raise exceptions in OCaml, one needs to call the function '*raise*' on an exception (see 3.1.2 for exception definition). To translate this phenomenon in Stainless, we use assertions. For example, this input:

```
exception Failure_simple;;
exception Failure_string of string;;
let simple_failure a = if a then raise
    (Failure_simple);;
let string_failure b = if b then raise
    (Failure_string "abc");;
```

outputs:

```
sealed case class Failure_simple()
    extends Exception {}
sealed case class
    Failure_string(param0: String)
    extends Exception {}
def simple_failure(a: Boolean): Unit =
    {
     if (a) assert(false,
         "Failure_simple")
}
def string_failure(b: Boolean): Unit =
    {
     if (b) assert(false,
         "Failure_string␣with␣abc")
}
```

3.2.14 Local variables

Local variables are defined with a '**let** id optional_arguments = id_expr **in** expr' statement. These statements can also be recursive if they use the keyword **rec**. To translate these statements in Scala, we first determine if we will define a **val**, a **var** or a **def**. If the variables is recursive or if is has arguments, we use **def**. If it is mutable (like with an array or a mutable record), we use **var**. Otherwise, we use **val**. To simulate the local environment, we create a new **val** that will contain the value definition and the expression. We need to create this **val** since Stainless does not allow expressions that starts with brackets as is. We number these new **val** to ensure there is no duplicates. Here is an example of input:

```
let x_in = 10 and y_in = 11 in y_in +
    x_in;;
let x_in = 10 in
     let x_in_in = 11 in
     x_in_in + x_in;;
let fun_in (e : int) = e + 10 in
     let fun_in_in (e : int) = e + 11 in
     fun_in 10 + fun_in_in 11 ;;
```

with output:

```
val _0 = {
     val x_in = 10
     val y_in = 11
     y_in + x_in
}
val _1 = {
     val x_in = 10
     val _2 = {
       val x_in_in = 11
       x_in_in + x_in
     }
}
val _3 = {
     def fun_in(e: Int63) = { e + 10 }
     val _4 = {
       def fun_in_in(e: Int63) = { e +
           11 }
       fun_in(10) + fun_in_in(11)
     }
}
```

3.2.15 While loops

For while loops, we translate separately the condition and the statements to execute at each iteration. The condition is then put between parenthesis and the statements are printed vertically and put between brackets. For example,

```
let while_simple = while true do
   print_endline "First";
   print_endline "Second";
done;;
```

is translated:

```
val while_simple: Unit = while ( true
    ) {
     println("First")
     println("Second")
}
```

3.2.16 For loops

For loops in OCaml are loops through a range of integers with a incrementing or decrementing step of one. For incrementing loops, the OCaml keyword is **to**. For decrementing, it is **downto**. To translate these loops in Scala, we use ranges with the keyword **to**. If it is a decreasing loop, we use 'by −1'. The statements that are executed each iteration are translated separately and wrapped in brackets. With the input below:

```
let for1 = for i = 1 to 10 do
  print_endline "For loop to 10";
done;;
let for2 = for i = 10 downto 1 do
  print_endline "For loop downto 1";
done;;
let for3 = for i = 2 + 3 to 4 * 5 do
  print_endline "For loop with weird 
      boundaries";
done;;
```

we will retrieve this output:

```
val for1: Unit = for ( i <- 1 to 10 )
    { println("For loop to 10") }
val for2: Unit = for ( i <- 10 to 1 by
    -1 ) { println("For loop downto 1")
    }
val for3: Unit = for ( i <- 2 + 3 to 4
    * 5 ) {
     println("For loop with weird 
         boundaries")
}
```

These for loops create an error when they are run with Stainless but are compilable with Scala.

3.2.17 Assertions

Assertions are quite similar in OCaml and Scala apart from the fact that Scala offers the possibility to add a message when performing an assertion. Here is an input as example:

```
let assert1 = assert(if 0 < 1 then
    true else false);;
```

The output will be:

```
val assert1: Unit = assert(if (0 < 1)
    true else false)
```

3.3 Patterns

Patterns are translated almost like a subset of expressions. To translate patterns, we overwrite *pr_patt*, the printer for patterns. If the program contains a pattern that was not implemented, a failure is raised.

3.3.1 Similar to expressions

Integers, floats, characters, strings, identifiers and tuples are translated similarly to their expression equivalent. List concatenation (' :: ') is translated using the constructor **Cons**.

3.3.2 Pattern with type

When the user defines a variable, global or local, using a **let** expression, the identifier and type of the variable is treated as a pattern. For example, in the statement

```
let variable_identifier :
    variable_type = ...
```

the pattern is ' variable_identifier : variable_type'. We translate this pattern using *pr_patt* for ' variable_identifier ' and *pr_ctyp* for 'variable_type'.

3.3.3 Wildcard pattern

The wildcard pattern is similar in OCaml and Scala. In both languages, it matches all possible values and is represented by the symbol '_'.

3.3.4 Multiple patterns matching

In OCaml, it is possible to do pattern matching on multiple patterns at the same time. It is also possible in Scala but not in Stainless. Here is the version in OCaml:

```
let is_a_vowel c = match c with
  'a' | 'e' | 'i' | 'o' | 'u' | 'y' ->
     true
  | _ -> false ;;
```

and the output in Scala:

```
def is_a_vowel(c: Char): Boolean = {
    c match {
      case 'a' | 'e' | 'i' | 'o' | 'u'
          | 'y' => { true }
      case _ => { false }
    }
}
```

3.4 Types

3.4.1 Retrieving the types

To provide as much information as possible to the Stainless verifier, we add the types of variables and functions to their definition. We retrieve the types using the **ocamlc -i** command on the input. This provides us with the interface of the OCaml program. We then parse that interface with a Scala program. We then go through the given OCaml program and look for variable and function definitions using regular expressions. For example, if we have 'val val_def : int' in the interface, we know that we are looking for a variable with no parameters that is called *val_def*. We give the type **int** to this variable. If we have 'val fun_def : int -> char -> bool' in the interface, we are either looking for a variable *fun_def* with two arguments or for an anonymous function definition, thus a variable *fun_def* with no parameters. In the first case, we will type each parameter, the first one as **int** and the second one as **char** and give to *fun_def* the return type **bool**. In the OCaml program, it will look like

```
let fun_def ( x : int) ( y : char) :
    bool = ...
```

In the second case, it will look like

```
let fun_def : int -> char -> bool = ...
```

The output of this operation is then written in a new OCaml file.

To insert types, we request that the user writes the **let** and the = on the same line.

Unfortunately, with **ocamlc -i**, we cannot retrieve the types of local variables (let ... in) thus is it better if the user types local variables before translating the OCaml program. The same problem arises when there are two global variables with the same name since **ocamlc -i** will only return the type of the last global variable with that name. There are also ambiguities when local variables have the same name as global variables and they have the same number of arguments. In the ambiguous cases, we do not type the variables. This last problem might be solved by renaming the variables.

3.4.2 Translating types

After parsing the OCaml program with its variables typed, the AST will contain nodes of type *ctyp*. To print these nodes, we overwrite *pr_ctyp*, the printer for types.

The basic OCaml types are translated as follows:

- **int** is translated to **Int63**. Integers in OCaml(9) depend on the system it is run on. The **max_int** is equal to $2^{Sys.int\_size-1} - 1$. Since currently most systems are 64 bits, we need to be able to store $2^{63-1} - 1$ integers. The type that corresponds the better to OCaml integers is **Int63**(10) of the BitVectors library. Unfortunately, this library is not yet available for execution. We discussed the use of **Long**s(11) since they are compilable and executable but their semantic is not exactly the same.
- **float** becomes **Double**. Floats in OCaml(12) are 64-bit IEEE-754 floating point numbers. The closest type in Scala is **Double**(13). Double expressions are not accepted in Stainless but this translation allows the program to be run with Scala.
- **bool** is translated **Boolean**.
- **char** becomes **Char**.
- **string** is translated **String**.
- **unit** becomes **Unit**.

For other type identifiers, we capitalize the first letter of the type. This works in most of the cases since OCaml and Stainless have a lot of similarities. For example, **list** becomes **List**. This is also used for abstract and case classes.

For applied types like 'string list list ', which is the type of a list of list of strings, we wrap each type with brackets. This gives us the type 'List [List [String]] '.

For tuples type, like 'bool * char', which could be the type of '(true, 'a') ', we collect every type and then put them in parenthesis, separated by commas. Here, the output would be '(Boolean, Char)'.

For functions types, the translation depends on the number of arguments of the function. If there is only one argument, we just replace the '−>' symbol by '=>'. For example, 'char −> bool' becomes 'Char =>Boolean'. With functions that have multiple arguments, we collect all the types that are before, after and in between the '−>' symbol. The last type collected is the return type of the function. The others types are the types of the arguments. For example, 'char −> string −> bool' becomes '(Char, String) =>Boolean'.

3.5 Comments

We translate comments that are outside structure items to Scala multiline comments. The comments that are inside a structure item or an expression are not included in the output. For example, this input:

```
(* Comment 1 *)
(** Comment 2 *)
(* Outer comment 3
     (* Inner
     comment 3 *) *)
let expression_with_comment = (*
    Comment 4 *) "No_comment";;
```

will give this output:

```
/* Comment 1 */
/* Comment 2 */
/* Outer comment 3
     /* Inner
     comment 3 */ */
val expression_with_comment: String =
    "No_comment"
```

4 Evaluation

To evaluate the front end, we had access to a repository (`https://github.com/drganam/TestML`) that contained 16 different exercises that were assignments for students. For each of these exercises, there was a solution file and between 100 to 500 files that students had submitted.

Unfortunately, the translation often results in Scala programs that have some errors when compiled. Most of the time, the error comes from these causes:

- Lack of types for local function definitions.
- Too many parenthesis were inserted during the translation.
- A function whose translation is unknown.
- Partial function application.
- Comparison of Int63 with Int or BigInt.
- Error while inserting the type inside the OCaml source file.

Most of these errors can be quickly solved manually, for example by just removing the parenthesis in excess or by adding the types. For unknown functions, users could define them themselves either in the OCaml source file or in the resulting Scala file. Partial function application is harder to solve since it is really different in OCaml and in Scala. The user might need to rewrite part of the program and introduce new variables.

Some errors are only happening within Stainless:

- The fact that value definition of arrays and records with mutable fields are assigned to a **var** can create an error since in Stainless, variables are only allowed within functions and as constructor parameters. To solve this, the user might wrap the part of the code that uses these variables with a function. We did not do this by default since it might create other errors with Stainless.
- Operations on Strings will likely create an error since there are less operations available on strings in Stainless.

For example of numbers, for the '*crazy2add*' assignment, 43/240 assignments compiled directly after the translation. For most of the other ones, the problem came from the lack of types for the parameters of a local function. Due to this problem, some unnecessary parenthesis were added around those parameters.

5 Conclusion

Lessons that can be learned from this project is that, even though OCaml and Scala have a lot of similarities, a perfect equivalence is hard to realize, even more when we try to translate OCaml in such a way that it can be used with the Stainless verifier. Stainless currently only supports a subset of Scala and even if the translation is compilable, it might be impossible to perform verification on it. An example of this problem is OCaml floats. With new libraries made available for Stainless, we might expect the translation to work better.

Another lesson that can be drawn comes from our usage of Camlp5. For sure, reinventing the wheel and creating our own parser and grammar would have been complicated and we might have missed some cases and created some unwanted errors, but by only performing the translation with our own printer, some information that could have been useful for the translation was really difficult to obtain or even unavailable.

This project was challenging due to the fact I did not know OCaml at the beginning of the semester thus, it was quite difficult to determine which expressions and structures are used and how they are used by OCaml programmers. Fortunately, Dragana Milovancevic, my supervisor, provided a set of OCaml files that were assignments of students in a functional programming course. These files as well as her experience with OCaml gave me an helpful insight on the usage of OCaml.

Another difficulty was that Camlp5 does not provide a lot of documentation and most of their code is written in OCaml revised syntax, which is quite complex to apprehend.

Despite these complications, the project was a benefical experience since it made me discover OCaml and Camlp5 and helped me improve my knowledge of Scala, Stainless and shell scripts.

References

[1] INRIA (Institut National de Recherches en Informatique et Automatique). (2021). Batch compilation (ocamlc). Available at: `https://ocaml.org/manual/comp.html` (Accessed: 16 June 2021)

[2] INRIA (Institut National de Recherches en Informatique et Automatique). (2007). Camlp5. Available at: `https://github.com/camlp5/camlp5` (Accessed: 16 June 2021)

[3] INRIA (Institut National de Recherches en Informatique et Automatique). (2007). The Pcaml module. Available at: `https://camlp5.readthedocs.io/en/latest/pcaml.html` (Accessed: 16 June 2021)

[4] INRIA (Institut National de Recherches en Informatique et Automatique). (2007). Syntax tree - strict mode. Available at: `https://camlp5.readthedocs.io/en/latest/ast_strict.html` (Accessed: 16 June 2021)

[5] OCaml. Custom Data Types. Available at: `https://ocaml.org/learn/tutorials/data_types_and_matching.html` (Accessed: 16 June 2021)

[6] Cornell. (2019). Physical equality. Available at: `https://www.cs.cornell.edu/courses/cs3110/2019sp/textbook/ads/physical_equality.html` (Accessed: 16 June 2021)

[7] OCaml. List. Available at: `https://ocaml.org/api/List.html` (Accessed: 20 June 2021)

[8] OCaml. Option. Available at: `https://ocaml.org/api/Option.html` (Accessed: 20 June 2021)

[9] OCaml. Int. Available at: `https://ocaml.org/api/Int.html` (Accessed: 16 June 2021)

[10] LARA. BitVectors. Available at: `https://github.com/epfl-lara/stainless/blob/master/frontends/library/stainless/math/BitVectors.scala` (Accessed: 16 June 2021)

[11] EPFL. (2003). Long. Available at: `https://www.scala-lang.org/api/2.12.4/scala/Long.html` (Accessed: 16 June 2021)

[12] OCaml. Float. Available at: `https://ocaml.org/api/Float.html` (Accessed: 16 June 2021)

[13] EPFL. (2002). Double. Available at: `https://www.scala-lang.org/api/2.12.9/scala/Double.html` (Accessed: 16 June 2021)