Caml tail

Caml Light, but smaller and less useful

Language Reference Manual

Jennifer Lam

Fiona Rowan

Sandra Shaefer

Serena Shah-Simpson

March 2, 2016

Table of Contents

Introduction

Caml tail is a functional programming language that is based on OCaml -- without the O. *Caml tail* will be strongly typed (with a smaller set of primitive types). It will support basic arithmetic, pattern matching, function definitions both imperative and recursive, anonymous functions, and nested functions. Our language will compile to LLVM.

1. Lexical Elements

This part of the manual describes the lexical elements of *tail*.

1.1 Separators

Horizontal tabs, newlines, carriage returns, and whitespace separate tokens. They are otherwise ignored. Double semicolons delineate phrases. A single semicolon is used as a sequencer of 1) multiple expressions, or 2) list elements.

1.2 Phrases

phrase:
    expr
    | *value-definition*
    | *type-definition*

A phrase is defined as an expression, a value definition, and a type definition. It is always terminated by double semicolons.

1.3 Comments

Comments are ignored, and are enclosed by (* and *).

1.4 Identifiers

ident:
    letter {letter | 0...9 | _}*

letter:
    A...Z | a...z

Identifiers are a sequence of characters beginning with a letter.

1.5 Integer literals

int_literal:
*[-] {*0...9*}*+

Integer literals are a sequence of digits in the decimal system, optionally beginning with a minus sign to signify negative numbers.

1.6 Floating point literals

float_literal:
*[-] {*0...9*}*+ . *{*0...9*}*+

Floating point literals are a sequence of digits with a decimal point at the beginning, end, or any other intermediary position within the sequence. An optional minus sign at the beginning designates a negative number.

1.7 Character literals

char_literal:
(a...zA...Z)
| ` \ *(*\ | ` | n | t | b | r*)* `
| ` \ *(*0...9*) (*0...9*) (*0...9*)* `

Character literals are single letters, digits, or symbols enclosed by two single quotes.

1.8 String literals

String literals are sequences of letters, digits, or symbols enclosed by two double quotes. Strings can be concatenated via a carat: *"O"* ^ *"Caml, guys!"* evaulates to *"OCaml, guys!"*

1.9 Keywords and special symbols

Keywords are special identifiers that are predefined by the language, and cannot clash with user-defined function or variable names. Below is the list of keywords in our language:

`int`	`char`	`string`	`float`	`unit`	`bool`
`if`	`then`	`else`	`let`	`match`	`with`

```
fun     func    rec     true    false   type
in      as      abst    open
```

The following sequence of characters are special symbols:

```
|     :     _     (     )     ->    ,     =     [
]     ==    !=    <     <=    >     >=    &&    ||
not   +     -     *     /     mod   +.    -.    *.
/.    ::    ^     ‘     “     ;     ;;    >.    >=.
<.    <=.
```

2. Values

2.1 Lists

typedecl list :
[item; item; item;…] | [item] | [ ]
item:
int
| *float*
| *char*
| *string*

A list is designated by opening and closing square brackets. All items in a list must be literals. The literals must be of the same type (no type mixing or tuples).

2.1.1 List Operators

Prepend to a list is denoted by double colons. For example, (new_item :: existing_list) prepends new item to an existing list.

2.2 Constants

The following table shows constant values.

`false`	the boolean false
`true`	the boolean true
`unit`	the void value
`[]`	the empty list

2.3 Functions

Functional values are mappings from values to values.

3. Type expressions

3.1 Type variables

These are variables that are bound to types, which may be user defined. They take the following format: *typedecl identifier* = $value_1$ | … | $value_n$

3.2 Parenthesized Types

A parenthesized type represents the same type as are inside the parentheses.

3.3 Function Types

The expression $type_1$ -> $type_2$ represents the type mapping of a function, which takes input of $type_1$ and outputs $type_2$.

4. Patterns

4.1 Variable patterns

These patterns consist of identifiers and match any value, thus binding that variable to said value. The wildcard symbol _ also matches any value, but there is no binding involved.

4.2 Alias patterns

These are patterns bound to identifiers, and are created in the following format: *pattern as identifier.* If a value matches this pattern successfully, the value is bound to the identifier.

4.3 Parenthesized patterns

A pattern enclosed in parentheses matches the same value as just that pattern would. Type constraints can be placed on patterns in this way as well, in the following manner: *(pattern : type).*

4.4 “Or” patterns

Or patterns are represented by two patterns separated by the symbol |. A value matches an “or” pattern if it either of these patterns.

4.5 Constant patterns

These patterns consist of constants, and only match values that are equal to those constants.

5. Expressions

5.1 Simple expressions

5.1.1 Variables

An expression consisting of a variable always evaluate to the value bounded to that variable.

5.1.2 Parenthesized expressions

An expressions inside parentheses evaluate to the value of that expression.

5.1.3 Function abstraction

Here, we use the keyword *fun* to match a set of values to a set of patterns. If the set of values matches the *i*th row of patterns, then the *i*th expression is evaluated -- that is, if the value v_j matches the $pattern^j$ in row *i*, $expr_i$ is evaluated. The first row of patterns matched is the one whose expression is evaluated. These statements occur in the following format:

fun abst
    $pattern^1$ … $pattern^m$ -> $expr_1$
    | $pattern^1$ … $pattern^m$ -> $expr_2$
…
    | $pattern^1$ … $pattern^m$ -> $expr_n$
    | _ -> $expr_{default}$

All rows must have the same number of patterns. If each row has only one pattern, use the keyword *match* instead of *fun* (see section 4.23). A default expression must exist in the case that the set of values does not match any of the given rows of patterns.

5.1.4 Function application

The expression $expr_1$ $expr_2$ … $expr_n$ evaluates the expression $expr_1$ to $expr_n$. $expr_1$ must evaluate to a functional value, which is applied to the values that follow.

5.1.5 Local definitions

We bind variables locally using the following formats:

let $pattern_1 = expr_1$ *and … and* $pattern_n = expr_n$ *in expr*

Each of the indexed expressions are evaluated, and if their values match their corresponding patterns, then *expr* is evaluated as the value of the entire let statement. If matchings succeed, *expr* is evaluated in environment enriched by bindings performed during matching(s), and the value of *expr* is returned as value of whole *let* expression. Local variables defined in the preceding pattern matches can be used in the evaluation of *expr*.

Alternatively, we may bind recursive definitions of variables locally with the following format:

let rec $pattern_1 = expr_1$ *and … and* $pattern_n = expr_n$ *in expr*

5.1.6 Function definitions

Anonymous functions have variable input specified within parentheses. The last expression evaluated is the functional return value of the function, and is immediately applied to the parameter values specified after the function body in parentheses. The type of $value_i$ must match the type of $identifier_i$. Anonymous functions are defined in the following way:

$type_{return}$ *fun (*$type_1$ $identifier_1$*, …,* $type_2$ $identifier_n$ *) ->*
    $expr_1$*;*
        …
    $expr_m$ *(* the last expr returns a value of* $type_{return}$ **) (*$value_1$*, … ,* $value_n$*)*

Declarative functions can be bound to an identifier. The last expression evaluates to the return type, and the functions have the following format:

let $type_{return}$ *function_name (*$type_1$ $formal\_arg_1$*, … ,* $type_n$ $formal\_arg_n$*) =*
        $expr_1$*;*
        …
        $expr_m$ *(* the last expr returns a value of* $type_{return}$ **)*

Recursive functions use the *rec* keyword, and have the following format:

let rec $type_{return}$ *function_name (*$type_1$ $identifier_1$*, … ,* $type_n$ $identifier_n$*) =*
    $expr_1$*;*
    …
    function_name($ident_1$, $ident_2$*, …,* $ident_n$*)*

5.2 Control constructs

5.2.1 Sequence

The expression $expr_1$ *;* $expr_2$ evaluates $expr_1$ first and then returns the value of $expr_2$.

5.2.2 Conditional

The expression *if* $expr_1$ *then* $expr_2$ *else* $expr_3$ evaluates the first function, and if that function evaluates to true, it evaluates the second function and returns its value. Otherwise, the third function is evaluated and its value is returned. The else part may be omitted.

5.2.3 Case Expression

The following expression matches *expr* with the following sequence of patterns. If it matches one of the patterns, its corresponding expression is evaluated, and the entire match expression evaluates to its value. If expr matches multiple patterns, the first pattern matched is considered the successful matching.

match expr
with $pattern_1$ *->* $expr_1$
  | $pattern_2$ *->* $expr_2$
…
  | $pattern_n$ *->* $expr_n$
  | _ *->* $expr_{default}$

5.3 Operators

The following are operators within boolean expressions:

==	infix	Equality test.
!=	infix	Inequality test.
<	infix	“Less than” integer test.
<=	infix	“Less than or equal to” integer test.
>	infix	“Greater than” integer test.

`>=`	infix	“Greater than or equal to” integer test.
`not`	prefix	Boolean negation.
`&&`	infix	Test if both expressions are true.
`||`	infix	Test if at least one expression is true.

The following are numerical operators:

`+`	infix	Integer addition.
`-`	infix	Integer subtraction.
`-`	prefix	Integer negation.
`*`	infix	Integer multiplication.
`/`	infix	Integer division.
`mod`	infix	Integer or float modulus.

The following are string and list operators:

^	prefix	String concatenation.

The prepend operator is a list operator, and consists of ::. The expression $expr_1$ :: $expr_2$ evaluates to the list with $expr_1$ as its head and $expr_2$ as its tail.

6. Standard library

The standard library can be included using the *open* keyword.

6.1 General

`print_char`	Prints a character to standard output.
`print_string`	Prints a string to standard output.
`print_int`	Prints an integer to standard output.
`print_float`	Prints a float to standard output.
`random()`	Generates a random float between 0 and 1.
`int_to_string`	Converts integers to strings.
`char_to_string`	Converts characters to strings.
`float_to_string`	Converts float to strings.

6.2 List operations

`len`	*List.len* returns the length of a list.
`rev`	*List.rev* returns a new list with the items of the original list reversed.
`hd`	*List.hd* returns the first element of the list.
`tl`	*List.tl* returns a list consisting of all elements of the original list except the head.
`nth`	*List.nth* returns the nth item.
`iter`	*List.iter* f [a1; ...; an] applies function f in turn to a1; ...; an.
`map`	*List.map* f [a1; ...; an] applies function f to a1, ..., an, and builds the list [f a1; ...; f an] with the results returned by f.
`fold_left`	*List.fold_left* function accumulator [item1; item2; …; itemn] is equivalent to function (...((accumulator item1) item2) … itemn)
`fold_right`	*List.fold_right* folds the opposite way as List.fold_left.
`concat`	*List.concat* [[a1;...; an];[z1;...; zn];[]] concatenates a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result.

7. Sample code

7.1 Rec function

```
let rec int fib (int n) =
    if n=1 then 1;
    else
      if n=2 then 1;
      else fib(n-1) + fib(n-2);;
```

7.2 Pattern matching

```
let rec int fib (int n) = match n with
    1 -> 1
    |2 -> 1
    |_ -> fib(n-1) + fib(n-2);;


let int gcd(int a, int b) =
     let rec int gcd_helper(int c, int d, int r) =
          match r with
               0 -> d
               | _ -> let c = c % d in let gcd_helper(d, c, d % c)
     in gcd_helper(a, b, a % b);;
```

7.3 Nested and anonymous functions

```
let int x = 5 in
     (int fun (int y) -> y + x) x;;

(* This line evaluates to ((function x -> x+x) 5), which returns 10
*)
```

7.4 Standard library use

```
(* Demonstrating a hello world program *)
let unit hello_world() =
     let string yo = “Hello “;
     let string dude = “World!”;
     print_string (yo ^ dude);;

hello_world();; (* Prints Hello World! to the screen *)


(* Demonstrating list operations *)
let unit hello() =
     let char gniteerg = [‘o’; ‘l’; ‘l’; ‘e’; ‘H’];
     let char greeting = List.rev gniteerg;
     List.iter print_char greeting;;
```

```
hello();; (* Prints Hello to the screen *)
```