FLite# - User Manual

CS4215 Programming Language Implementation

Simone Mezzaro
Lyu Xiaoteng

AY 2021-2022

Contents

1 Introduction **2**

2 Basic features **2**

3 Type system **3**

4 Syntax **3**

5 Units of measure **4**

6 Pattern matching **5**
6.1 Constant patterns 5
6.2 Variable patterns 5

1 Introduction

FLite# is a functional language representing a subset of the programming language F#. FLite# includes the basic features of F#, such as arithmetic and boolean operations, lists, tuples, functions and lambda expressions. Moreover it implements two peculiar F# features: units of measure and pattern matching.

As the original F#, FLite# is a statically typed language. However, differently from its parent, FLite# does not infer types and requires the programmer to explicitly specify the type of each name in the code.

FLite# comes with a Java interpreter which can be run using the following command

```
java -jar FLiteSharp.jar myCode.fs
```

where `myCode.fs` is the name of the file containing the source code. The result of this execution is the value of the last expression in the source code.

2 Basic features

The basic programming features provided by FLite# are:

- Arithmetic operations between integers or doubles: `+`, `-`, `*`, `/`, `**`. `**` is the exponentiation operator and works only with double values.
- Comparison operations between integers or doubles: `=`, `<>`, `<`, `>`, `<=`, `>=`.
- Boolean operations: `&&`, `||`, `not`.
- Lists and tuples. Lists are ordered sequences of elements with the same type. A list can be created by writing its elements between square brackets. It is possible to append an element at the beginning of the list using the attach operator `::`. It is also possible to concatenate two lists using the operator `@`.

  Tuples are ordered collections of elements which can have different types. A tuple can be created by writing its elements between round brackets.

Example 1 Lists and tuples

```
1 [];               // empty list
2 [1; 2; 3];        // list of integers
3 1 :: [2; 3];      // attach operator. The result is [1; 2; 3]
4 [1; 2] @ [3; 4]; // concatenate operator.
5                   // The result is [1; 2; 3; 4]
6 (1, true, 4.0);  // tuple
```

- Conditional expressions. In FLite# conditionals are always expressions and evaluates to the same value of the last expression of the taken branch. A conditional expression is shown in example 9.
- Bindings. In FLite# it is possible to use the keyword `let` to associate a name with a value. The value associated to a name is immutable and can never be altered.

Example 2 Binding

```
1 let a : int = 3 in
```

- Functions. The `let` keyword is used also to declare a function with a given name. Example 8 shows the declaration of a function associated with the name `f`, with parameters `x` and `y` and returning `x+y`. Line 6 shows how the function is applied.
- Recursive functions. Functions using recursion must be specified explicitly in FLite# using the keywords `let rec` instead of just `let`.
- Lambda expressions. The keyword `fun` allows to define anonymous functions.

Example 3 Lambda expression

```
1 fun (x : int)(y : int) -> x + y;
```

3 Type system

FLite# requires the programmer to explicitly specify the type of each name in the code. This is done by using the operator `:` after the name.

The primitive types available are `int`, `double` and `bool`.

The type of a list is defined using the keyword `list` preceded by the type of the list's elements.

Example 4 List type

```
let l : int list = [1; 2; 3] in
```

The type of a tuple is specified by listing the type of all its elements separeted by `*`.

Example 5 Tuple type

```
let t : (int * bool * double) = (1, true, 4.0) in
```

The type of a function is defined by listing the type of all its parameters followed by its return type. The listed types are separated by `->`.

Example 6 Function type

```
let f : (int -> int -> bool) = fun (x : int)(y : int) -> x < y in
```

Notice that when a function is declared using the `let` keyword no explicit type should be specified for the function name. Only the type of the parameters and the return type of the function (written after the parameters) must be specified (see example 8).

Finally, a special type named `unit` is available. This type is used to indicate the absence of a specific value. The only value of type unit is `()`. The `unit` type is useful, for example, to declare functions with no parameters.

Example 7 Unit type

```
let trueFunc : (unit -> bool) = fun () -> true in
```

4 Syntax

FLite# programs are typed using the verbose syntax of F#. In particular a FLite# program must comply with the following rules:

- Each sequence of expressions, named code block, must be preceded by the keyword `begin` and followed by the keyword `end`.
- Each expression in a code block must end with a semicolon.
- Each let binding must end with the keyword `in`. Notice that let bindings are not expressions and therefore do not end with a semicolon.
- Unit of measure definitions are not expressions and therefore do not end with a semicolon.

Example 8 Function declaration syntax

```
let f (x : int)(y : int) : int =
    begin      // start of function's body block
        x + y;
    end        // end of function's body block
    in
f 1 3;
```

Example 8 shows the syntax of a function declaration. The body of a function is always a code block and is therefore enclosed between `begin` and `end`. Moreover, each line in the body ends with a semicolon. Finally, since functions are declared with a let binding, a function declaration is terminated by the `in` keyword.

Example 9 Conditional expression syntax

```
let a : bool = true in
let x : int = 3 in
```

```
3  if a || false
4     then
5         begin      // code block of then branch
6             x * 2;
7         end
8     else
9         begin      // code block of else branch
10            x * 4;
11        end;
```

Example 9 shows the syntax of a conditional expression. `then` and `else` branches are body blocks and must comply with the body block syntax. Also notice that a conditional expression is indeed an expression and ends with a semicolon.

5 Units of measure

Units of measure can be associated to `int` and `double` types to enforce stricter rules on the type checking.

A unit of measure must be defined in the top-level of the program before it is used. Example 10 shows how to declare a new unit of measure `kg`.

Example 10 Unit of measure declaration

```
1  [<Measure>] type kg
```

A unit of measure can also be defined as the alias for a units of measure formula.

Example 11 Unit of measure declaration with formula

```
1  [<Measure>] type N = kg m / s^2
2  [<Measure>] type N = /s^2 * kg m 1
```

A formula is a fraction of previously declared units of measure. Measures at the beginning of a formula or following a `*` and separated by white spaces are considered part of the numerator. Measures following a `/` and separated by white spaces are considered part of the denominator. The units of measure in a formula can also have a positive or negative integer exponent and can be surrounded by parentheses. The special value `1` can be used in a formula to indicate a dimensionless value. The two formulas in example 11 are equivalent.

Single units of measure and formulas can associated both to a numerical value and to the type of a name using angular brackets.

Example 12 Associating units of measure

```
1  let a : int<kg> = 3<kg> in
2  let b : int<N> = 3<kg m / s^2> in
```

When two values or names are summed, subtracted or compared (using one of the comparison operators) the compiler checks not only that the type of the two operands is the same, but also that their associated units of measure are equivalent (i.e. are represented by equivalent formulas). If this condition is not satisfied a type error occurs.

Example 13 Units of measure with operators

```
1  4<N> + 3<kg m / s^2>;   // correct
2  4<m> >= 3<kg>;          // type error
```

The same check is applied when a value with a unit of measure is assigned to a name.

Example 14 Units of measure with bindings

```
1  let s : int<m> = 4<m> + 3<m> in   // correct
2  let p : int<m/s> = 4<m>/2<s> in   // correct
3  let w : int<m> = 3<kg> in         // type error
```

Notice that the unit of measure associated with the result of an addition or subtraction is the same measure of the operands. The unit of measure associated with the result of a multiplication or division is respectively the product or the division between the units of measure formulas of the operands.

6 Pattern matching

Pattern matching allows you to write a pattern matching expression that matches a value against a pattern.

The pattern can be seen as characteristics of the data. And they act as rules to transform values in some way. In FLite#, they are used with `match` expressions in this way:

Example 15 Pattern matching usage

```
1 match expression with
2 | pattern [ when condition ] -> result-expression
3 ...
```

In the `match` expressions, each patterns will be examined and each one will be evaluated from left to right. The first pattern that matches the value is used to evaluate the expression. If no pattern matches the value, the pattern matching expression fails and throws an exception at the runtime.

For the type system of pattern matching:

- All the pattern types need to be compatible with the input value type.
- All the branch return results needs to have the same type, which will be the type of pattern matching expression.
- The type of the condition in the `when` clause must be boolean. This one is important when using variable patterns.

The pattern matching in FLite# supports the following patterns:

- `Constant patterns`: they are numerical values, unit type value or list. For example, `1`, `true`, `[1; 2; 3]`.
- `Variable patterns`: they are names of variables. For example, `x`, `y`.

6.1 Constant patterns

The input is compared with the literal value and the pattern matches if the values are equal. The type of the literal must be able to be compatible with the type of the input value.

Example 16 Constant patterns example 1

```
1 let f (x : int) : int = begin
2 match x with
3 | 4 when false -> 1
4 | 1 -> -1
5 | 4 -> 0;
6 end in
7 f 4;
```

This example creates a function that will match the argument, a integer value, to 3 pattern branches:

The first branch matches the numeric value `4` and returns `1`. However, since the condition that pattern branch will always be evaluated to `false`, it will never be matched.

The second branch matches the numeric value `1` and returns `-1`.

The third branch matches the numeric value `4` and returns `0`.

So when we do `f 4`, the function will return `0`. And the example above will be evaluated to `0`.

6.2 Variable patterns

The variable pattern assigns the value being matched to a name, which is then available to use in the `when` clause and result expressions of that pattern branch.
The variable name can be matched to any input value. It can be used with constant patterns and list to decomposed them into variables.

Example 17 Variable patterns example 1

```
1  let max (x : int list) : int = begin
2  match x with
3  | [var1; var2] when var1 > var2 -> var1
4  | [var1; var2] when var1 < var2 -> var2
5  | [var1; var2] -> var1;
6  end in
7  max [123; 456];
```

The example above creates a function that will match the argument which is a integer list to 3 pattern branches:

The first branch matches a list with 2 elements: `var1` and `var2`. when `var1` is greater than `var2`. And it will return the value of `var1`.

The second branch matches a list with 2 elements: `var1` and `var2`. when `var2` is greater than `var1`. And it will return the value of `var2`.

The third branch matches a list with 2 elements: `var1` and `var2`.

So when we do `max [123; 456]`, the function will return `456`. And the example above will be evaluated to `456`.