FLite# - Developer Manual

CS4215 Programming Language Implementation

Simone Mezzaro
Lyu Xiaoteng

AY 2021-2022

Contents

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 defined 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 interpreter executes three steps:

- *Parsing*: it parses the source file and creates a tree structure representing the code and its instructions. The parser also detects possible syntax errors and signals them to the user.
- *Type-checking*: it analyses the tree structure to determine if the program is well-typed and associate each instruction in the tree with its return type. This step involves also few other compile-time checks: it controls for example that each name is defined before it is used. An error message is prompt to the user if any of the checks fail.
- *Evaluation*: it evaluates the tree structure and computes the result of each instruction in the code. The result of the last instruction is displayed to the user. There are a few run-time errors which can occur in this step such as a division by zero or a pattern matching instruction unable to match the given expression.

The interpreter's code together with test cases, documentation and FLite# examples can be found in this GitHub repository.

2 Syntax

The syntax of FLite# is defined by a custom-made grammar which embeds the verbose syntax of F#.

In FLite# every instruction and operation is an expression. The only two exceptions are let bindings and units of measure declarations. To comply with the verbose syntax, each expression must follow these 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.

2.1 Grammar

The following grammar shows exactly which expressions are available in FLite#. Notice that this grammar has been simplified by removing some implementation details and writing tokens values explicitly. A complete version is available in the directory `/FLiteSharp/tree/simone-master/app/src/main/antlr/FLiteSharp.g4`.

Example 1 Grammar

```
topLevelBlock
    : (sequenceLine | unitDeclaration)* EOF
;

expression
    : '(' expression ')' # Parentheses
    | blockExpression # Block
    | expression '**' expression # Power
    | '-' expression # Negative
    | expression ('*' | '/') expression # MultiplicationDivision
    | expression ('+' | '-') expression # AdditionSubtraction
    | expression '<' expression # LessThan
    | expression '<=' expression # LessThanOrEqual
    | expression '>' expression # GreaterThan
    | expression '>=' expression # GreaterThanOrEqual
    | expression '=' expression # Equal
    | expression '<>' expression # NotEqual
    | 'not' expression # Not
```

```
    | expression '&&' expression # And
    | expression '||' expression # Or
    | expression '::' listExpression # Attach
    | listExpression '@' listExpression # Concatenate
    | patternMatching # PatternMatchingExpression
    | conditionalExpr # ConditionalExpression
    | VARIABLE # Variable
    | funcApplication # FunctionApplication
    | INTEGER ('<' unitFormula '>')? # Integer
    | DOUBLE ('<' unitFormula '>')? # Double
    | BOOLEAN # Boolean
    | UNIT # Unit
    | tupleExpression # Tuple
    | listExpression # List
    | lambdaExpression # LambdaFunction
;

blockExpression
    : 'begin' sequentialExpression 'end'
;

sequentialExpression
    : ((sequenceLine)* expression ';')?
;

sequenceLine
    : bind 'in'
    | funcDeclaration 'in'
    | recFuncDeclaration 'in'
    | expression ';'
    ;

conditionalExpr
    : 'if' expression 'then' blockExpression ('else' blockExpression)?
;

funcApplication
    : VARIABLE expression+
;

tupleExpression
    : '(' expression (',' expression)+ ')'
;

listExpression
    : '[' (expression (';' expression)*)? ']'
;

lambdaExpression
    : 'fun' lambdaParameters '->' expression
;

funcDeclaration
    : 'let' VARIABLE lambdaParameters ':' typeDeclaration '=' blockExpression
;

recFuncDeclaration
    : 'let rec' VARIABLE lambdaParameters ':' typeDeclaration '=' blockExpression
;

lambdaParameters
    : UNIT | ('(' VARIABLE ':' typeDeclaration ')')+
;

bind
    : 'let' VARIABLE ':' typeDeclaration '=' expression
;
```

As can be seen in the grammar bindings are not expressions and must always be followed by an expression (see `blockExpression` and `sequentialExpression`) rules. However the top level block also accepts bindings on their own. Units of measure declarations instead can appear only in the top level block.

The grammar has been written using the language supported by Antlr, an automatic generator for language recognizer. This tool has been employed to create automatically the java classes which perform the parsing step of the interpreter.

3 Type system

FLite# is a statically-typed language. Every time a name is defined in the code it must be accompanied by its type. This types can never change during the execution of the program, but are used in the type-checking phase to ensure that the evaluation will not produce any type error.

The primitive types available in the language are `int`, `double`, `bool` and `unit`. `unit` is a special type named used to indicate the absence of a specific value. The only value of type unit is `()`. The `unit` value is useful to declare functions with no parameters, but can also be inserted at the end of block to make it type-check to `unit` type.

Example 2 Conditional expression

```
1  let a : bool = true in
2  let x : int = 3 in
3  if a || false then begin
4      x * 2;
5      ();
6  end;
```

The expression in the example above is well typed and has type `unit`.

FLite# provides also some compound types to represent types of lists, tuples and functions. The syntax to declare each type is described by the following simplified grammar, where the token `TYPE` is either `'int'`, `'double'`, `'bool'` or `'unit'`.

Example 3 Type declarations grammar

```
1  typeDeclaration
2      : '(' typeDeclaration ')' # ParenthesesType
3      | TYPE ('<' unitFormula '>')? # PrimitiveType
4      | typeDeclaration 'list' # ListType
5      | typeDeclaration '*' typeDeclaration # TupleType
6      | typeDeclaration '->' typeDeclaration # FunctionType
7  ;
```

See the grammar in syntax section to view where type declarations are required in the code.

3.1 Type rules

The following rules describe how the type-checker works. In these rules, unless differently stated, t or t_i are placeholders for any type (with or without unit of measure), u and w are placeholders for any units of measure formula, x and x_i are placeholders for names, E and E_i represents expressions and γ represents the environment.

3.1.1 Literals

$$\frac{}{\Gamma \vdash n : \text{int<1>}} \quad \frac{}{\Gamma \vdash q : \text{double<1>}} \quad \frac{}{\Gamma \vdash n\text{<u>} : \text{int <u>}} \quad \frac{}{\Gamma \vdash q\text{<u>} : \text{double<u>}}$$

$$\frac{}{\Gamma \vdash \mathit{true} : \text{bool}} \quad \frac{}{\Gamma \vdash \mathit{false} : \text{bool}} \quad \frac{}{\Gamma \vdash () : \text{unit}}$$

3.1.2 Compound data

- List

$$\frac{\Gamma \vdash E_1 : t \quad \Gamma \vdash E_2 : t \quad ... \quad \Gamma \vdash E_n : t}{\Gamma \vdash [E_1; E_2; ...; E_n] : t \text{ list}}$$

- Tuple

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2 \quad ... \quad \Gamma \vdash E_n : t_n}{\Gamma \vdash (E_1, E_2, ..., E_n) : (t_1 * t_2 * ... * t_n)}$$

3.1.3 Operations

- Unary operators

$$\frac{\Gamma \vdash E : \text{int <u>}}{\Gamma \vdash -E : \text{int <u>}} \quad \frac{\Gamma \vdash E : \text{double <u>}}{\Gamma \vdash -E : \text{double<u>}} \quad \frac{\Gamma \vdash E : \text{bool}}{\Gamma \vdash not\ E : \text{bool}}$$

- Binary operators

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma \vdash E_2 : t_2}{\Gamma \vdash p[E_1, E_2] : t}$$

p	t_1	t_2	t
$+$	int<u>	int<u>	int<u>
$+$	double<u>	double<u>	double<u>
$-$	int<u>	int<u>	int<u>
$-$	double<u>	double<u>	double<u>
$*$	int<u>	int<w>	int<u·w>
$*$	double<u>	double<w>	double<u·w>
$/$	int<u>	int<w>	int<u/w>
$/$	double<u>	double<u>	double <u/w>
$**$	double<1>	double<1>	double<1>

p	t_1	t_2	t
$=$	int<u>	int<u>	bool
$=$	double<u>	double<u>	bool
<>	int<u>	int<u>	bool
<>	double<u>	double<u>	bool
$<$	int<u>	int<u>	bool
$<$	double<u>	double<u>	bool
$>$	int<u>	int<u>	bool
$>$	double<u>	double<u>	bool
<=	int<u>	int<u>	bool
<=	double<u>	double<u>	bool
>=	int<u>	int<u>	bool
>=	double<u>	double<u>	bool
||	bool	bool	bool
&&	bool	bool	bool
::	t'	t' list	t' list
@	t' list	t' list	t' list

3.1.4 Binding

$$\frac{\Gamma[x \leftarrow t]\Gamma' \quad \Gamma \vdash E_1 : t \quad \Gamma' \vdash E_2 : t'}{\Gamma \vdash (\text{let } x : t = E_1).E_2 : t'}$$

3.1.5 Block

$$\frac{\Gamma \vdash E_2 : t}{\Gamma \vdash E_1.E_2 : t}$$

3.1.6 Control flow

- Conditional

$$\frac{\Gamma \vdash E_0 : \text{bool} \quad \Gamma \vdash E_1 : t \quad \Gamma \vdash E_2 : t}{\Gamma \vdash \text{if } E_0 \text{ then } E_1 \text{ else } E_2 : t} \quad \frac{\Gamma \vdash E_0 : \text{bool} \quad \Gamma \vdash E_1 : \text{unit}}{\Gamma \vdash \text{if } E_0 \text{ then } E_1 : \text{unit}}$$

3.1.7 Functions

- Function declaration

$$\frac{\Gamma[x_1 \leftarrow t_1, ..., x_n \leftarrow t_n]\Gamma' \quad \Gamma' \vdash E : t}{\Gamma' \vdash (x_1 : t_1)...(x_n : t_n) : t = E \quad : \quad (t_1 \rightarrow ... \rightarrow t_n \rightarrow t)}$$

- Application

$$\frac{\Gamma \vdash E_1 : t_1 \quad ... \quad \Gamma \vdash E_n : t_n \quad \Gamma \vdash E : (t_1 \rightarrow ... \rightarrow t_n \rightarrow t)}{\Gamma \vdash E\, E_1 \,...\, E_n : t}$$

- Lambda expressions

$$\frac{\Gamma[x_1 \leftarrow t_1, ..., x_n \leftarrow t_n]\Gamma' \quad \Gamma' \vdash E : t}{\Gamma' \vdash (x_1 : t_1)...(x_n : t_n) \rightarrow E \quad : \quad (t_1 \rightarrow ... \rightarrow t_n \rightarrow t)}$$

3.1.8 Units of measure

$$\frac{\Gamma \vdash E : t}{\Gamma \vdash ([\text{<Measure>}] \text{ type u}).E : t} \quad \frac{\Gamma \vdash E : t}{\Gamma \vdash ([\text{<Measure>}] \text{ type u} = U).E : t}$$

3.1.9 Pattern matching

$$\frac{\Gamma \vdash E : t \quad \Gamma_j \vdash B_j : t_r \ \forall j \wedge 1 \leq j \leq n}{\Gamma \vdash (\text{match } E \text{ with } |B_1 \quad |... \quad |B_i \quad |... \quad |B_n) : t_r}$$

where

- $\Gamma[var(B_j) \leftarrow t]\Gamma_j$ if B_j is a single variable pattern and $var(B_j)$ is the name of its variable
- $\Gamma[\ ]\Gamma_j$ and $\Gamma \vdash const(B_j) : t$ if B_j is a constant pattern and $const(B_j)$ is its value
-
- $\Gamma_j \vdash when(B_j) : bool$ if B_j has a when clause and $when(B_j)$ is the expression following the clause

4 Units of measure

Units of measure are labels which can be associated to `int` and `double` types and values 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 and can also be defined as the alias for a units of measure formula. A formula is a fraction of previously declared units of measure. The following grammar shows the syntax used to write units of measure declarations and formulas. Refer to the grammar in syntax section to see how these declarations and formulas can be inserted in the code.

Example 4 Units of measure grammar

```
1  unitDeclaration
2      : '[<Measure>] type' VARIABLE ('=' unitFormula)?
3  ;
4
5  unitFormula
6      : ('/')? (unitElement ' ')+ (unitProduct)*
7  ;
8
9  unitProduct
10     : (MUL | DIV) (unitElement ' ')+
11 ;
12
13 unitElement
14     : INTEGER # OneUnit
15     | VARIABLE # SingleUnit
16     | '(' unitFormula ')' # ParenthesisUnit
17     | unitElement '^' exponent # ExponentialUnit
18 ;
```

Notice that the above grammar accepts any integer value in a measures formula. However only the integer 1 can be present in such formulas and represents a dimensionless quantity. Since the grammar does not take this into account, an extra check has been added to the parses to ensure this rule is respected. `exponent` on the other hand can really be any integer (positive or negative) value.

Measures formulas are preprocessed during the parsing step in order to later simplify their type-checking. Every time a measure is defined it is added to a specific storage with the represented normalized formula if any. When a formula is encountered (either in a measure definition or associated to a type) it is normalized in the following way:

- Every unit of measure in the formula is searched in the storage and substituted with its own formula. In this way every measure which is an alias is resolved immediately when the formula is defined, avoiding the recursive process of resolving the tree of alias every time the formula must be later used.
- The formula is then transformed into a sequence of multiplications (a fraction with denominator equal to 1). This makes the comparison between formulas easier because equivalent formulas written in different ways are converted into the same representation.

Units of measure add new constraints to arithmetical operations and bindings. Refer to the type system section for the formal type rules involving measures.

5 Semantic

As already described in the syntax section, in FLite# every instruction and operation is an expression. The only two exceptions are let bindings and units of measure declarations.

Instructions and operations available in the language 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. 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.
- Conditional expressions. In FLite# conditionals are always expressions and evaluates to the same value of the last expression of the taken branch. As shown in example 2 a conditional expression must always have both branches unless the `then` branch will evaluate to `unit`.
- Lambda expressions representing anonymous functions.
- Pattern matching expression. See pattern matching section.

All these expressions behaves in the same way as in many popular languages. For this reasons evaluation rules are not provided for them.

The following bindings operations are also available

- Variable bindings. In FLite# it is possible to associate a name with an immutable value which can never be altered.
- Functions and recursive functions. Bindings are also used to associate functions to a name. These association is also immutable.

As seen in syntax section these instructions must always be followed by an expression. Their evaluation complies to the following semantic rules:

$$\frac{\Gamma[x \leftarrow v_1]\Gamma' \quad \Gamma \vdash E_1 : v_1 \quad \Gamma' \vdash E_2 : v_2}{\Gamma \vdash (\text{let } x = E_1).E_2 : v_2} \qquad \frac{\Gamma[x \leftarrow v_1]\Gamma' \quad \Gamma \vdash E_1 : v_1}{\Gamma \vdash \text{let } x = E_1 : undefined}$$

Notice that in the top-level block these instructions can be written on their own (i.e. they can be the last instruction in the program), in which case they evaluate to undefined.

Finally units of measure declaration instructions are available. However these instructions are removed during the parsing step after the normalization described in units of measure section has been performed. Therefore they are never evaluated.

5.1 Scoping

In FLite# each block introduce a new scope. This includes explicit blocks, if branches, functions body and pattern matching branches. The scope is represented by a special container called environment. The environment is composed of environment frames which store the value and type associated with each name. When the value of a name in a block is looked for in the environment the values corresponding to the innermost scope visible to that block.

Functions and lambda expressions are evaluated using the environment frame in which they have been declared extended with the arguments of the function application. For this reason that frame is stored within the functions.

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 5 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.

Refer to the type system section for pattern matching type rule.

The pattern matching in FLite# supports the following patterns: The pattern matching in FLite# supports the following patterns:

- `Constant patterns`: they are numerical values, unit type value or list.
  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.
- `Variable patterns`: they are names of variables.
  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.

6.1 Evaluation rules

Pattern matching expressions are evaluated following these evaluation rules

- Constant pattern branch

$$\frac{\Gamma \vdash E : v_2}{\Gamma \vdash (|v_1 \rightarrow E) : v_2} \quad \frac{\Gamma \vdash E_1 : true \quad \Gamma \vdash E_2 : v_2}{\Gamma \vdash (|v \text{ when } E_1 \rightarrow E_2) : v_2} \quad \frac{\Gamma \vdash E_1 : false \quad \Gamma \vdash E_2 : v_2}{\Gamma \vdash (|v \text{ when } E_1 \rightarrow E_2) : undefined}$$

- Variable pattern branch with single variable

$$\frac{\Gamma[x \leftarrow v]\Gamma' \quad \Gamma' \vdash E : v_2}{\Gamma \vdash (|x \rightarrow E) : v_2} \quad \frac{\Gamma[x \leftarrow v]\Gamma' \quad \Gamma' \vdash E_1 : true \quad \Gamma' \vdash E_2 : v_2}{\Gamma \vdash (|x \text{ when } E_1 \rightarrow E_2) : v_2}$$

$$\frac{\Gamma[x \leftarrow v]\Gamma' \quad \Gamma' \vdash E_1 : false \quad \Gamma' \vdash E_2 : v_2}{\Gamma \vdash (|x \text{ when } E_1 \rightarrow E_2) : undefined}$$

- Variable pattern branch with list

$$\frac{\Gamma[x_1 \leftarrow v_1]\Gamma_1 ... \Gamma_{m-1}[x_m \leftarrow v_m]\Gamma_m \quad \Gamma_m \vdash E : v}{\Gamma \vdash (|x \rightarrow E) : v}$$

$$\frac{\Gamma[x_1 \leftarrow v_1]\Gamma_1 ... \Gamma_{m-1}[x_m \leftarrow v_m]\Gamma_m \quad \Gamma_m \vdash E_1 : true \quad \Gamma_m \vdash E_2 : v}{\Gamma \vdash (|x \text{ when } E_1 \rightarrow E_2) : v}$$

$$\frac{\Gamma[x_1 \leftarrow v_1]\Gamma_1 ... \Gamma_{m-1}[x_m \leftarrow v_m]\Gamma_m \quad \Gamma_m \vdash E_1 : false \quad \Gamma_m \vdash E_2 : v}{\Gamma \vdash (|x \text{ when } E_1 \rightarrow E_2) : undefined}$$

  where $x_1, ..., x_n$ are names in the pattern list, $v_1, ..., v_n$ are corresponding values in the list to match and x is the pattern list.

- Pattern matching

$$\frac{\Gamma \vdash E : v \quad \Gamma_j \vdash B_j : undefined \; \forall j \wedge 1 \leq j \leq i-1 \quad \Gamma_i \vdash B_i : v_i}{\Gamma \vdash (\text{match } E \text{ with } |B_1 \quad |... \quad |B_i \quad |... \quad |B_n) : v_i}$$

  where $\Gamma[var(B_j) \leftarrow v]\Gamma_j$ if B_j is a single variable pattern and $var(B_j)$ is the name of its variable and $\Gamma[\ ]\Gamma_j$ if B_j is a constant pattern.

7 Java interpreter

The implemented Java interpreter is based on a tree structure returned by the parser and representing the program. Each element of the tree is a `Component`, which represents an instruction of the program. Each type of instruction has its own component class, child of `Component`, which stores all the information about it.

A program is therefore represented as a tree where each instruction's component contains references to its sub-instructions, which must be executed before the represented instruction. For example the expressions representing the operands of an addition must be executed before the addition itself and are therefore stored (as components) in the `AdditionComponent`.

Each `Component` has two main methods: `checkType` and `evaluate`.

- `TypeElement checkType(EnvFrame env)`: this method is used to check that the instruction represented by the component is well typed and assigns its own return type to the instruction. This operation is performed by a recursive depth-first exploration of the tree. If a compile-time error occurs during the type checking an exception is thrown.

- `DataComponent evaluate(EnvFrame env)`: this method is used to evaluate the instruction represented by the component and returns its result. This operation is also performed by a recursive depth-first exploration of the tree. If a runtime error occurs during the evaluation an exception is thrown.

Both the methods receive a parameter `env` which represents the environment in which the component should be evaluated or type-checked.

The return value of `checkType` is a `TypeElement`, which is a class used to represent types. `TypeElement` stores also an attribute of class `UnitOfMeasure` which contains the formula possibly associated with that type.

The return value of `evaluate` is a `DataComponent`. Data components are special components representing values that can be returned as the result of an evaluation and that can be associated with names. These values are integers, doubles, booleans, the unit element, functions and lambda expressions.