程序设计语言原理

Fldw：支持模式匹配的流式编程语言

学院：计算机学院

姓名：李梦凡

学号：ZY2006317

2020 年 12 月

一、语言设计的背景

1.1 背景与动机

Fldw 语言的命名取自 The flow of data，寓意为数据的流动，被设计为是一个支持模式匹配的流式编程语言。Fldw 语言的设计背景来自于 Elixir 语言的快速排序程序，其示例如下所示。Elixir 语言借助模式匹配和 For 推导的特性，仅使用三行代码就完成了快速排序的核心算法部分的编写。在 Fldw 语言的设计中，使用了同样的模式匹配逻辑，并将模式匹配的语法应用到了流式编程中，实现了类似 For 推导的流式编程的语法。最终将模式匹配和流式编程相结合，使用 Fldw 语言可以编写类似的快速排序代码（见语言示例 D）。

```
defmodule QuickSort do
  def sort([]), do: []
  def sort([head|tail]), do:
    sort(for(x<-tail, x<=head, do: x)) ++
    [head] ++
    sort(for(x<-tail, x>head, do: x))
  end
end

IO.inspect QuickSort.sort([5, 6, 3, 2, 7, 8])
```

1.2 语言特性

Fldw 语言的语言特性为，动态类型，支持语句块和符号定义域，支持指针和值传递，支持函数的递归调用。在语法设计时，Fldw 的语法被设计为 LL(3) 语法，且基于 Java15 进行设计和实现。

在设计 Fldw 语言的语言特性时，其语言特性的设计思想源于 Streem，Elixir 和 Lua 等高级编程语言。

Streem

Streem 语言创立自 Ruby 之父松本行弘，其在《日经Linux》杂志上的连载，介绍了新语言Streem 的设计与实现过程，并将连载整个成书《松本行弘：编程语言的设计与实现》。Streem 的源代码已开源在 GitHub

Streem 是基于流的并发脚本语言。它基于类似于shell的编程模型，并受Ruby，Erlang和其他功能编程语言的影响。Streem 类似shell管道的编程模型深受笔者的喜爱，例如，使用 Streem 实现的 cat 程序为：

```
stdin | stdout
```

在 Streem 中，管道操作符 | 为主要的操作符，Fldw 语言借鉴了其编程模型的思想，使用了基于流的编程语言，并使用 | 为主要的操作符。使用 Fldw 实现的 cat 程序为：

```
import std.std
stdin | stdout
```

使用 Fldw 实现的 `hello world` 程序为：

```
import std.std
["Hello World!"] | stdout
```

调用 Fldw 自带的 cat 示例程序 `./examples/cat_example.sh`

```
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw % ./examples/cat_example.sh
Cat Example:

import std.Std
stdin | stdout
```

```
========================================
Input Flow:
1 1.23 true "hello" 1+2*3.3
1
1.23
true
"hello"
7.6
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw %
```

Elixir

Elixir 被设计为应对并发编程的高级编程语言，但是其拥有众多良好的编程特性值得借鉴。Elixir 可以看做 Ruby 基本块，Lisp 宏和 Erlang actor 并发模型的结合。Elixir 语言支持模式匹配和 for 推导，通过这两个特性，可以使得使用 Elixir 实现的快速排序符合直觉且易于理解。例如，Elixir 实现的快速排序如下：

```
defmodule QuickSort do
  def sort([]), do: []
  def sort([head|tail]), do:
    sort(for(x<-tail, x<=head, do: x)) ++
    [head] ++
    sort(for(x<-tail, x>head, do: x))
  end
end

IO.inspect QuickSort.sort([5, 6, 3, 2, 7, 8])
```

Fldw 语言借鉴了其模式匹配的实现和 for 推导的思想，实现了类似的语法。调用 Fldw 自带的快速排序示例程序 `./examples/quicksort_example.sh` 如下：

```
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw % ./examples/quicksort_example.sh
QuickSort Example:

import std.Std
function sort() {
    in | [!head;!tail]
    if ( head != null ) {
        [] | !leftHead
        [] | !rightHead
        for ( tail -> !tmp) {
            if ( tmp < head ) {
                #[tmp] | leftHead
            }
            else {
                #[tmp] | rightHead
            }
        }
        leftHead | sort() | out
        [head] | out
        rightHead | sort() | out
    } | out
}
stdin | sort() | stdout

========================================
Input Number(Int or Double) Flow:
5 6 3 2 7 8
2
3
5
6
7
8
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw %
```

Lua

Lua 被设计易于内嵌和迁移的配置语言。其灵活的函数参数和返回值使得 Lua 脚本作为配置脚本十分易用。

- **灵活的参数**：传参太少，未传入的参数作为 nil；传参太多，多余的参数会被忽略；也可以设置可变参数的函数。
- **灵活的返回值**：函数的返回值可以有多个，可以选择使用所有的返回值，或者忽略部分。

Fldw 借鉴了 Lua 灵活的传参思想。调用 Fldw 自带的灵活的函数传参示例程序 `func_dynamic_param.sh` 如下：

```
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw % ./examples/func_dynamic_param.sh
FuncDynamicParam Example:

import std.Std
function func([a, b, c]) {
    [a, b, c] | stdout
}
func([1, 2])
["-----"] | stdout
func([1, 2, 3, 4])
["-----"] | stdout
func()

========================================
Expect Output:
1
2
null
-----
1
2
3
-----
null
null
null
========================================
Actual Output:
1
2
null
"-----"
1
2
3
"-----"
null
null
null
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw %
```

二、语言的词法、语法、语义说明

2.1 词法说明

数据

```
INT_VALUE = ["1"-"9"] (["0"-"9"])*
          |   "0" ["x", "X"] (["0"-"9", "a"-"f", "A"-"F"])+
          |   "0" (["0"-"7"])*
DOUBLE_VALUE = ["1"-"9"] (["0"-"9"])*("."(["0"-"9"])*)?
BOOL_VALUE = "true" | "false"
STRING_VALUE = < "\"" >
                       ( < (~["\"", "\\", "\n", "\r"])+ >
                           |   < "\\"(["0"-"7"]){3} >
                           |   < "\\"~[] > )
               < "\"" >
NULL_VALUE = "null"
```

符号

```
SEMIC = ";"
COMMA = ","
LBR = "("
RBR = ")"
RCBR = "}"
LCBR = "{"
RSBR = "]"
LSBR = "["
DOT = "."
```

运算符

```
PLUS = "+"
MINUS = "-"
MULT = "*"
DIV = "/"
MOD = "%"
LOGIC_EQUAL = "=="
LOGIC_NOT = "!="
LOGIC_AND = "&&"
LOGIC_OR = "||"
LEFT = "<"
RIGHT = ">"
LEFT_EQUAL = "<="
RIGHT_EQUAL = ">="
```

关键字

```
IF = "if"
ELSE = "else"
WHILE = "while"
FOR = "for"
FUNC = "function"
IMPORT = "import"
```

其他符号

```
FLOWING = "|"         // 数据管道操作
MATCHING = "->"       // 模式匹配操作
HASHTAG = "#"         // 变量取值操作
EXLM = "!"            // 显示定义为临时变量
```

2.2 语法说明

基础数据类型

```
terminal_data ::= < INT_VALUE > | < DOUBLE_VALUE > | < BOOL_VALUE > |  < STRING_VALUE >
symbol_data ::= [ "!" ] < SYMBOL>

data ::= < expr_data > | < symbol_data > | < terminal_data >

expr_data ::= < expr1_data > ( "||" < expr1_data > )*
expr1_data ::= < expr2_data > ( "&&" < expr2_data > )*
expr2_data ::= < expr3_data > ( ( "<" | ">" | "<="
             | ">=" | "==" | "!=" ) < expr3_data > )*
expr3_data ::= < expr4_data > ( ( "+" | "-" )  < expr4_data > )*
expr4_data ::= < term > ( ( "*"  | "/" | "%" ) < term > )*
term ::= < terminal_data > | < symbol_data > | "(" < expr_data > ")"
```

流数据类型

列表流

```
list_flow ::= [ "#" ] “[" < data > ( [ "," ] < data > )* "]"
```

控制流

```
if_else_flow ::= "if" "(" < data > ")" < block > [ "else" < block > ]
while_flow ::= "while" "(" < data > ")" < block >
for_flow ::= "for" "(" < flow > "->" < symbol_data > ")" < block_flow >
```

语句块流

```
block_flow ::= "{" ( < flowing > )* "}"
```

模式匹配流

```
head_tail_flow ::= "[" < symbol_data > ";" [ "!" ] < SYMBOL > "]"
```

可执行语句

流执行语句

```
flow ::= < func_flow > | < head_tail_flow > | ([ "!" ] < SYMBOL >) | < list_flow >
       | < if_else_flow > | < while_flow > | < block_flow > | < for_flow >
flowing ::= < flow > ( ( "->" | "|" ) < flow > )*
```

import语句

```
import_stmt ::= "import" < SYMBOL > "." < SYMBOL >
```

函数定义语句

```
def_func_stmt ::= "function" < SYMBOL > "(" [ < list_flow > ] ")" < block_flow >
```

其他

```
stmt ::= < flowing > | < def_func_stmt > | < import_stmt >
stmts ::= ( < stmt > )*
program ::= < stmts > < EOF >
```

2.3 语义说明

抽象语义

```
Command ::= Skip
    | [ More_Data ]                          // PointerListFlow
    | #[ More_Data ]                         // ValueListFlow
    | SYMBOL                                 // SymbolFlow
    | !SYMBOL                                // TmpSymbolFlow
    | SYMBOL ( Actual_Parameter_Sequence )   // FuncFlow
    | If_Command                             // IfElseFlow
    | While_Command                          // WhileFlow
    | For_Command                            // ForFlow
```

```
    | HeadTail_Command                          // HeadTailFlow
    | Command -> Command                        // MatchFlowing
    | Command | Command                        // PushFlowing
    | Command Command                          // Flowings
    | { Command }                               // BlockFlow

More_Data ::= Skip
    | , More_Data
    | More_Data
    | Data

If_Command = if ( Data ) { Command }
    | if ( Data ) { Command } else { Command }

While_Command ::= while ( Data ) { Command }

For_Command ::=  for ( Command -> Data ) { Command }

HeadTail_Command ::= [ SYMBOL ; SYMBOL ]

Data ::= INT_VALUE
    | DOUBLE_VALUE
    | BOOL_VALUE
    | NULL_VALUE
    | STRING_VALUE
    | SYMBOL
    | ! SYMBOL
    | ( Data )
    | Data Operater Data

Operater ::= +
    | -
    | *
    | /
    | %
    | <
    | >
    | <=
    | >=
    | %%
    | ||
    | ==
    | !=

Declaration ::= import SYMBOL . SYMBOL
          | function SYMBOL ( Formal_Parameter_Sequence ) { Command }

Formal_Parameter_Sequence ::= Formal_Parameter
                            | Formal_Parameter , Formal_Parameter_Sequence

Actual_Parameter_Sequence ::= Actual_Parameter
                            | Actual_Parameter , Actual_Parameter_Sequence

Formal_Parameter ::= [ More_Data ]
                   | SYMBOL

Actual_Parameter ::= [ More_Data ]
                   | SYMBOL
```

语义函数

数据管道操作的指称语义为：

```
execute [ F1 | F2 ] env sto =
    let val = evaluate car(F1) env sto in
    cons(val, F2)
    if !empty(cdr[F1]) = boolean true
    then execute [ cdr[F1] | F2 ]
```

数据匹配的指称语义为：

```
execute [ F1 -> F2 ] env sto =
    let val = evaluate car(F1) env sto in
    let variable loc = find(env, car(F2)) in
    update(sto, loc, val)
    if !empty(cdr[F1]) && !empty(cdr[F2]) = boolean true
    then execute [ cdr[F1] -> cdr[F2] ]
```

模式匹配操作的指称语义为：

```
execute [ C | [ HEAD ; TAIL ] ] env sto =
    let head = evaluate car(C) env sto in
    let tail = evaluate cdr(C) env sto in
    let variable loc_head = find(env, HEAD) in
    let variable loc_tail = find(env, TAIL) in
    update(sto, loc_tail, tail)
    update(sto, loc_head, head)
```

其他语义：

```
execute [ if D1 { C1 } else { C2 } ] env sto =
if evaluate D1 env sto = boolean true
{ execute C1 }
else { execute C2 }

execute [ while D { C } ]
let execute_while env sto =
    if evaluate D env sto = boolean true
    { execute_while env (execute C env sto ) }
    else sto
in
execute_while

execute [ for ( C1 -> D ) { C2 } ] env sto =
let execute_for encv sto =
    if execute C1->D env (execute C1 env sto) = boolean true
    { execute_for env (execute C2 env sto) }
    else sto
in
execute_for

execute [ { C } ] env sto =
{ execute C }

execute [ Func(APS) ] env sto =
let function func = find(env S) in
let arg = give_argument APS env in
func arg

evaluate : Data → ( Environ → Store → Value )

evaluate [ S ] env sto =
    coerce( sto , identify S env sto )


evaluate [ D ] env sto =
    get_value( D )

evaluate [ Op D] env sto =
let operator op = find ( env Op ) in
let val = evaluate E env sto in
op val

evaluate [ D1 Op D2 ] env sto =
let val1 = evaluate D1 env sto in
let val2 = evaluate D2 env sto in
cal( D1,D2 )

evaluate [ ( D ) ] env sto =
    evaluate D
```

```
identify : SYMBOL → ( Environ → Store → Value)

identify [ S ] env sto = find( env , S )
```

辅助函数

```
empty: list → boolean                          // 判断列表是否为空
update: store × location × value → store       // 赋值操作
cons: value × list → list                      // 合并元素和列表
car: list → value                              // 获取列表的第一个元素
cdr: list → list                               // 获取列表除去第一个元素的列表
```

三、安装和部署

Fldw 语言支持 Windows 安装，MacOS 安装和 Linux 安装三种方式。同时还是支持 Docker 环境下的快速部署。

3.1 Windows 安装

Windows 下运行 Fldw 需要 Java 15 以上的环境，下载发行版本，调用启动脚本 `.\fldw.bat` ，输出 Hello World 如下所示：

```
C:\学习\projects\Fldw>.\fldw.bat

C:\学习\projects\Fldw>chdir C:\学习\projects\Fldw\

C:\学习\projects\Fldw>java -jar .\jar\Fldw-1.0.0.jar
Welcome to FLDW, version 1.0.0
fldw > import std.Std
fldw > ["Hello World!"] | stdout
"Hello World!"
fldw > exit
Bye!

C:\学习\projects\Fldw>
```

3.2 MacOS 或 Linux 安装

MacOS 或 Linux 下运行 Fldw 需要 Java 15 以上的环境，下载发行版本，调用启动脚本 `./fldw.sh` ，输出 Hello World 如下所示：

```
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw % ./fldw.sh
Welcome to FLDW, version 1.0.0
fldw > import std.Std
fldw > ["Hello World"] | stdout
"Hello World"
fldw > exit
Bye!
(base) limengfan@limengfandeMacBook-Pro 201122_Fldw %
```

3.3 Docker 部署

如果没有 Java15 环境，在安装了 Docker 的环境下也可以通过 Docker 镜像来运行，下载 Docker 镜像并输出 Hello World 如下所示：

```
(base) limengfan@limengfandeMacBook-Pro ~ % docker pull imortal/fldw:v1.0.0

v1.0.0: Pulling from imortal/fldw
Digest: sha256:152063ad4e6a8da966bfe40190cc4ab6b961148a62b71f64227457f5446ba9c0
Status: Image is up to date for imortal/fldw:v1.0.0
```

```
docker.io/imortal/fldw:v1.0.0
(base) limengfan@limengfandeMacBook-Pro ~ % docker run -it --rm imortal/fldw:v1.0.0 /bin/bash
root@6022d66718fe:/# ./Fldw/fldw.sh
Welcome to FLDW, version 1.0.0
fldw > import std.Std
fldw > ["Hello World!"] | stdout
"Hello World!"
fldw > exit
Bye!
root@6022d66718fe:/#
```

四、语言示例

A Cat 示例程序

```
import std.Std

stdin | stdout
```

B 动态传参和返回值示例程序

```
import std.Std

function func([a, b, c]) {
    [a, b, c] | stdout
}

func([1, 2])
["-----"] | stdout
func([1, 2, 3, 4])
["-----"] | stdout
func()
```

C 斐波那契数列示例程序

```
import std.Std

function func([n]) {
    if ( n == 0 || n==1 ) {
        [n] | out
    } else {
        func([n-1]) -> [n1]
        func([n-2]) -> [n2]
        [n1 + n2] | out
    } | out
}

func([0]) | stdout
func([1]) | stdout
func([2]) | stdout
func([3]) | stdout
func([4]) | stdout
func([5]) | stdout
func([6]) | stdout
func([7]) | stdout
func([8]) | stdout
func([9]) | stdout
```

D 快速排序示例程序

```
import std.Std

function sort() {
    in | [!head;!tail]
    if ( head != null ) {
        [] | !leftHead
        [] | !rightHead
        for ( tail -> !tmp) {
            if ( tmp < head ) {
                #[tmp] | leftHead
            }
            else {
                #[tmp] | rightHead
            }
        }
        leftHead | sort() | out
        [head] | out
        rightHead | sort() | out
    } | out
}

stdin | sort() | stdout
```