THEATR

An actor based language

Language Reference Manual

Group members:
Beatrix Carroll (bac2108)
Suraj Keshri (skk2142)
Michael Lin (mbl2109)
Linda Ortega Cordoves (lo2258)

Contents

Preface

Goal

Create an actor-based language with fault-tolerance that simulates the logic of Erlang and the syntax of Scala in the Akka framework.

Motivation

The actor model provides a good framework for reasoning about concurrency and distributed systems. The model elegantly handles concurrency issues which can be complex when implemented in a thread-based model, such as locking and mutual access to resources. Instead of multiple threads accessing a shared resource, the actor model relies on autonomous actors performing asynchronous and independent computations.

Distributed systems requires comprehensive and secure communication capabilities with its various remote processors. To achieve this, distributed systems require that processors communicate via messages and that these messages be asynchronous. In the actor model, actors communicate with each other exclusively through messages, which are stored in mailboxes and processed asynchronously.

Description

We wish to create a programming language that implements the Actor Model with fault-tolerance.

As the Actor Model specifies, Actors will be the basic unit of computation in our language. Upon receiving a message, any actor will be able to only perform the following three actions:

1. Create another actor
2. Send message to another actor
3. Update the internal state (specify the state in which it will be when it next receives a message)

Each actor possesses an internal state. Actors will only be able to communicate with each other actors via passing messages. Each message an actor receives will be stored in its mailbox and processed asynchronously.

Programs

Our language is designed to write programs that take advantage of parallelism. Some examples of these types of programs are: chat servers, phone switches, web servers, message queues, and web crawlers. Our language is also designed for programs requiring multiple I/O computations, programs requiring strict fault tolerance, and programs requiring strict avoidance of deadlocking.

Lexical Elements

Identifiers

Identifiers can include letters, decimal digits, and the underscore character. The first character of an identifier cannot be a digit. Identifiers are case sensitive.

Keywords

These keywords are reserved for use in our language: if, else, for, while, break, continue, actor, int, float, string, char, boolean, receive, drop, after, new, none, return, match, case, func, main, timeout.

Constants

A constant is a literal numeric or character value, such as 5 or 'm'. All constants are of a particular data type.

An **integer** constant is a sequence of digits, with an optional prefix to denote a number base. An integer constant is a sequence of digits.

A **float constant** is a value that represents a fractional (floating point) number. It consists of a sequence of digits which represents the integer (or "whole") part of the number, a decimal point, and a sequence of digits which represents the fractional part. Either the integer part or the fractional part may be omitted, but not both.

A **character** constant is usually a single character enclosed within single quotation marks, such as 'Q'. A character constant is of type int by default.

A **string** constant is a sequence of zero or more characters, digits, and escape sequences enclosed within double quotation marks.

Separators

A separator separates tokens. White space (see next section) and tabs are separators, but they are not a token. The other separators are all single-character tokens themselves:

Separator	**Usage**
( )	Orders operations within an expression, groups expressions

[ ]	Accesses the element of a container: [n] accesses the nth element
{ }	Denotes scope of code blocks, or a definition of a struct
, (comma)	Separator for elements in a container, or function arguments list
; (semicolon)	Separator for key-value pairs in a Dict

White Space

Like Python, our language is whitespace sensitive: it defines the hierarchy of the code. The space character is also important in string and character constants and when separating tokens.

Comments

The /* */ symbols comments out entire sections of code, while // comments out a single line.

```
/*
comment
block
 */

// commented line
```

Data Types

Primitive data types:

Data type keyword	**Meaning**
int	Integers, 4 bytes in size
float	Real numbers in floating point notation, 8 bytes in size
boolean	True or false in value
char	Represent the ASCII character set, 1 byte in size
string	Collections of characters

Non-primitive types:

Containers:

Data type keyword (note capitalization)	**Meaning and usage**
List	Ordered list of items of a certain type, backed by a linked list A list can be defined using List<[data type]> [list name] For example: `List<int> int_list = new List[1, 2, 3]`
Array	A contiguous block of memory containing a collection of a certain data type. For example: `Array<int> int_array = new Array[1,2,3]`
Tuple	Ordered group of items of different types. For example: `Tuple my_tuple = new Tuple[1, “hello”, ‘h’]`
Set	Unordered collection of unique objects of a certain data type. For example: `Set<int> int_set = new Set[1, 2, 3]`
Dict	A key-value pair store. For example: `Dict<string, int> my_dict = [“bob”, 3; “jim”, 4]`

Functions:

Functions are treated as first class objects. Please see the details in the later section on syntax and usage.

Structs:

A struct is a custom data type which could contain other types as its members. They are defined using the following syntax:

```
struct my_struct {
       string skill;
       actor my_actor;
}
```

To use a struct, the `new` keyword is used:

```
my_struct matt = new my_struct();
matt.skill = "high";
```

Actors:

As an actor-based language, actors are a central data type! They are considered a non-primitive data type, with the following rules enforced by the compiler:

- All actors must implement a *receive* block (more details below)
- By default, an actor that receives a message not accounted for in the *receive* block does nothing. The programmer can override this behavior by providing a *drop* block
- The programmer can provide an optional *after([time in milliseconds])* block to provide behavior if it waits too long on the response from another actor
- Like all other variables, instances of actors are immutable. To achieve changes in state, an actor recursively calls itself with new arguments
- An actor can recreate itself recursively, but *cannot* recreate itself as a different type of actor
- At least one of the actor's actions in the *receive* block must result in the termination of the actor (i.e. by not calling itself recursively again)
- Actors have a built-in flush() function that outputs all the messages an actor has received
- Actors always have access to a *sender* identifier (reserved keyword) which denotes the handle to the actor that sent a message

The syntax of defining an actor is:

```
[actor name] ([actor state variables, as parameters]):
       receive:
              [message name]([message paramenters]):
                    [behavior upon receiving message]
        [drop:]
                [behavior upon receiving a message not accounted for in the receive
block]
        [after([time in milliseconds]):]
                [behavior upon waiting too long for a response]
```

For example, to define an actor dolphin that keeps track of a name and weight, with the ability to receive eat(int number) messages:

```
dolphin(string name, int weight):
        receive:
                eat(int numFoodPellets):
                        dolphin(name, weight + numFoodPellets)
                runAway():
                        print "So long, and thanks for all the fish!"
```

Expressions, Assignment, and Operators

Expressions and Assignment

An **expression** is any legal combination of symbols that represents a value. Assignments are expressions that set variables equal to the value of another expression. Note that "variables" in our language are **immutable**, so reassigning different values to the same variable results in the new value shadowing the old one, as in OCAML. The syntax for assignments is:

```
[type of variable] [variable name] = [expression]
```

For example:

```
int x = 5
int dist = sqrt(x*x, y*y)
```

Assignment, Access, Function, and Actor-specific Operators

Operator	Meaning	Associativity
=	Assignment	Left
. (period)	Method accessor for non-primitive types	Left
->	Denotes the return value for a lambda function, or the return type for a named function	N/A
| (pipe)	Send message. Usage: [message] | [target actor]	N/A

Arithmetic Operators

Operator	Meaning	Associativity
+	Addition	Left
-	Subtraction	Left
*	Multiplication	Left
/	Division	Left

%	Modulus	Left
- (unary minus)	Value of the expression multiplied by -1	Right

Comparison Operators

Operator	**Meaning**	**Associativity**
==	Equal to	N/A
!=	Not equal to	N/A
>	Greater than	N/A
<	Less than	N/A
<=	Less than or equal to	N/A
>=	Greater than or equal to	N/A

Logical Operators

Operator	**Meaning**	**Associativity**
!	Not, negation	Right
||	Or	Left
&&	And	Left

Type Casts

Our language will support casting between the float and int types, to facilitate arithmetic operations between the two types. The syntax for this is:

```
([new casted type]) [expression]
```

For example:

```
float a = 5.0 + (float) myInteger
```

Operator Precedence

The rules of precedence used when trying to solve an expression containing multiple operators. Below is a list of types of expressions, presented in order of highest precedence first.

1. Sending messages to actors and waiting for a response.
   1.1. Messages' arguments are evaluated before being sent, if the message's arguments contain a compound expression
   1.1.1. For example:
   1.1.1.1. make_deposit(x+5) | bank_account
2. Function calls, and membership access operator expressions.
3. Unary operators, including logical negation, increment, decrement, unary positive, unary negative, indirection operator, address operator, type casting, and sizeof expressions. When several unary operators are consecutive, the later ones are nested within the earlier ones: !-x means !(-x).
4. Multiplication, division, and modular division expressions.
5. Addition and subtraction expressions.
   5.1.1. For example, in the expression `a+b * f()` the order of precedence is to call the function `f` with no arguments, multiply the result by `b`, then add that result to `a`.
6. Greater-than, less-than, greater-than-or-equal-to, and less-than-or-equal-to expressions.
7. Equal-to and not-equal-to expressions.
8. Logical AND expressions.
9. Logical OR expressions.
10. All assignment expressions, including compound assignment.
    10.1. When multiple assignment statements appear as subexpressions in a single larger expression, they are evaluated right to left.
    10.2. Compound assignments using tuples are also evaluated left to right (on each respective side of the assignment operator
    10.2.1. For example:
    10.2.1.1. (a, b) = ("foo", "bar")
    10.2.1.2. // this executes as:
    10.2.1.2.1. a = "foo"
    10.2.1.2.2. b = "bar"
    10.2.1.2.3. (a, b) = (a, b)
    10.2.1.2.3.1. // => which = ("foo", "bar")
11. Comma operator expressions.

Statements

The *if* Statement

You can use the *if* statement to conditionally execute part of your program, based on the truth value of a given expression. Indentation determines the attachments of any `else` or `else if` statements to the parent `if`. The syntax is as follows (note the indentation):

```
if (test)
   then-statement
else
   else-statement
```

The *match* Statement

You can use the *match* statement to compare one expression with others, and then execute a series of sub-statements based on the result of the comparisons. The indentation of the `case` statements denotes the attachment to the parent `match` statement:

```
x match

   case 1: "one"

   case "two": 2

   case (int y): "scala.Int"
```

The *while* Statement

The while statement is a loop statement with an exit test at the beginning of the loop. The indentation of the statement denotes the scope of the while loop:

```
while (test):
   statement
```

The *for* Statement

The for statement is a loop statement whose structure allows easy variable initialization, expression testing, and variable modification.

```
for (initialize; test; step):
  Statement
```

The variable used to track the number of loops can be initialized in the for-loop's initialize statement.

The *break* Statement

You can use the *break* statement to leave the scope of a *while*, *for*, or *match* statement.

The *continue* Statement

You can use the *continue* statement in loops to terminate an iteration of the loop and begin the next iteration.

The *return* Statement

The *return* statement comes at the end of a function to denote the value that the function evaluates to.

Functions

Function Definitions

Named functions:

Named functions are declared using the func keyword, followed by the name identifier of the function, its argument list, and the colon character, as per the following:

```
func function_name (data_type x, data_type y) -> return_data_type:
        /* function_body */
        return value
```

For example:

```
func sqrt(int x) -> int:
        return x * x
```

Lambda functions:

Lambda functions can be defined in place and can take any number of arguments but return just one value in the form of an expression, using the following syntax:

```
([lambda function arguments]) -> [return value]
```

For example:

```
(int x, int y) -> x + y
```

Calling Functions

The syntax for calling named and lambda functions is as follows:

Named Function

```
int y = function_name (int x, int y)
```

Lambda Function:

```
List<(string, int)> list_housing_cost = [("Houston", 20), ("Miami", 30)]
calculate_total_cost(list_housing_cost.map((_,x)->x)))
```

Function Parameters

Within the function body, the parameter is a local copy of the value passed into the function; you cannot change the value passed in by changing the local copy. If you wish to use the function to change the original value, then you would have to incorporate the function call into an assignment statement:

```
int x = foo(x)
```

Passing functions as parameters

You can also call a function identified by a function identifier:

```
func function_name ( func f(int)->int, int times):
        for(int i = 0; i < times; i++):
                f(i)
```

The *main* Function

Every program requires at least one function, called *'main'*. This is where the program begins executing, and serves as its function definition.

To accept command line parameters, the main function must accept a parameter of type List<string> to accept the space-separated command line parameters:

```
main (List<string> args)
```

The main function may or may not return a value - execution of the program ends once the end of the main function is reached.

Recursive Functions

You can write a function that is recursive—a function that calls itself.

```
func factorial (int x):
        if (x < 1):
                return 1
        else:
                return (x * factorial (x-1))
```

Nested Functions

Our language allows programmers to define functions within other functions. Hence, why the *main()* function can call other functions.

```
main():
        other_function():
                return 5
```

Program Structure and Scope

Program Structure

The only requirement to run your program is the *main* function. That will be the entry point for every program. Other than that, files can be linked to manually or by putting them in a central directory that will be specified when compiling.

Scope

Scope refers to what sections of the code the program can “see.” All identifiers need to be defined before their usage.

Example 1:

```
// This is correct
actor_name():
        receive:
                do_anything():
                        “Hi” | Sender
main():
        actor new_actor = new actor_name()
        string response = do_anything | new_actor
```

Example 2 (usage before definition):

```
// Error: actor_name is not defined at the time of usage
main():
        actor new_actor = new actor_name()
        string response = do_anything | new_actor

        actor_name():
                receive:
                        do_anything():
                                “Hi” | Sender
```

Scope in Actors

Declarations made within actors are visible only within those actors. The initialization values of an actor is available across different sections (receive(), drop(), after()) and message types of the actor. These are called actor-level variables. Inputs of messages are known as message-level variables. These message-level parameters are only available within the message body and go out of scope when the message body ends.

Actors have access to the pid or handle of the actor whose message they are currently executing. They can access this pid by using the *sender* keyword while inside the actor definition.

To reiterate, because actors are also immutable like all other objects, after an actor completes the tasks of a message, an actor dies. Programmers can get around this by recursively calling the same actor after the end of each message. In this recursive call, the actor can be created with different actor-level variables.

Scope in Functions

Declarations made within functions are visible only within those functions.

A Sample Program

The Dolphin Example:

```
dolphin(int weight, string name):
        receive:
                eat(int num):
                        dolphin(weight+num, name)
                follow_me():
                        be_followed | sender
                        dolphin(weight, name)
                catch(actor target):
                        be_caught_and_eaten | target
                        dolphin(weight, name)
                twin_yourself(int weight, string name):
                        twin = new dolphin(weight, name)
                        twin | sender
                        dolphin(weight, name)
                be_free():
                        // at least one message case MUST not recursively call the actor
                        print "So long, and thanks for all the fish!"
        drop:
                print "This is not allowed for a dolphin"
                dolphin(weight, name)

        after 3000:
                timeout

trainer(string name, List<actor> followed_by):
        receive:
                be_followed():
                        trainer(name, [sender|followed_by] )
                throw_fish_to_dolphin(actor target, actor food):
                        catch(food) | target
                        trainer(name, followed_by)
        drop:
                print "This is not allowed for a trainer"
                trainer(name, followed_by)

fish(int weight):
        receive:
                Be_caught_and_eaten:
                        eat(weight) | sender
                        // do not recursively call self, so the fish has died
        drop:
                print "I'm only a fish, I can't do this"
                fish()
```

```
/* This is the entry point for the program */
func main() -> int:
      bottlenose = new dolphin(3, “bottlenose”)
      trainer = new trainer(“Bob”, new List<actor>())
      salmon = new fish(15)
      yellowtail = new fish(12)
      bluefin_tuna = new fish(100)
      throw_fish_to_dolphin(bottlenose, salmon)
      throw_fish_to_dolphin(bottlenose, yellowtail)
      throw_fish_to_dolphin(bottlenose, bluefin_tuna)
      be_free() | bottlenose
      return 0
```

Our CFG

Below is a rudimentary CFG that represents the type of CFG we would use. This is not fully fleshed out yet.

The Notation of this CFG:

- the “|” symbol will be replaced by the “#” symbol. “e” represents the empty string.
- [ ] will represent groupings within the cfg grammar, because [] are not used in our alphabet but () are.
    - For instance:
    - “[arg,]*arg” means:
        - Arg,arg,arg,arg,arg
        - Or:
        - arg
        - Or:
        - arg,arg
        - etc.
- [tab] represents the tab character
- [\n] represents the new line character

Our Sample CFG

```
type ->
                primitive_type
        #       nonprim_type

primitive_type ->
                int
        #       float
        #       bool
        #       char
        #       string

nonprim_type ->
                container
        #       function
        #       actor
        #       struct
```

```
container ->
                List
        #       Array
        #       Tuple
        #       Set
        #       Dictionary

expression ->
                literal
        #       id
        #       ( expression )
        #       expression unary_operator
        #       expression binary_operator expression

actor_declaration ->
                actor_name (params):
                [\n]+[tab]receive:
                [[\n]+[tab][tab]msg:[tab][tab][tab]actor_instruction]*
                [\n][tab]drop:
                [[\n]+[tab][tab]msg:[tab][tab][tab] actor_instruction]*

declaration ->
                actor_declaration
        #       other_declation

other_declaration ->
                type id
        #       type id = literal

params ->
                e
        #       [expression,]* expression

actor_instruction ->
                actor_instruction(params)
        #       [| msg_target]*

actor_instantiation ->
                [\n]actor_id = new actor_name(actor_params)

compile_this ->
                func main(params):
                *[actor_instantiation]*[\n actor_instruction]*[expression]*]*
```

References

We borrowed the structure of our LRM heavily from the C Language Reference Manual:
https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html