Arza Documentation

Release 0.1

gloryofrobots

May 04, 2018

Contents

Source code on GitHub

Contents:

CHAPTER 1

Overview

Arza is an experimental eager functional programming language with dynamic type system supporting multiple dispatch and immutable data out of the box.

It can be described like Erlang with a lot of syntactic sugar and some new ideas.

Currently it exists only as a prototype written in Python.

I am considering either to add RPython jit generation or to rewrite full interpreter in C.

1.1 Most prominent features of arza

- Laconic whitespace aware syntax inspired by F#, Python, Lisp
- Immutable data structures (lists, tuples, maps)
- Pattern matching
- Lexical clojures and lambdas
- Usual number of primitives (if-else, let-in, try-catch)
- User defined operators
- User defined abstract and struct types
- Nominal subtyping
- Multiple dispatch generic functions
- Interfaces supporting multiple dispatch paradigm
- Traits as functions operating on types
- Special syntax for partial application
- Stackless virtual machine
- Asymmetric coroutines

- Symmetric coroutines aka processes
- Decorators for functions, types and generic specialisations
- Special syntax for modifiing deeply nested immutable structures
- Special support for creating links to parts of immutable structures aka lenses

1.2 What is missing for now

- Standart library, currently Arza have only some functions for sequence manipulation
- Tail call optimisation. Interpreter is stackless and it is currently very slow to work with, so I decided against compiler complication for now
- No REPL. Interpreter works only with files
- No macroses. I have not decided yet if I want them in the language at all

1.3 What are the biggest problems

- Interpreter is so slow it is painfull to work with.
- Error reporting in some places is very bad
- Language design is still unstable

1.4 How to run

```
python targetarza.py program.arza
```

However it is much better to use PyPy

1.5 Examples

Examples and tests are in `ROOTDIR/test/arza` folder.

Entry point for test suite is `ROOTDIR/test/arza/main.arza`

CHAPTER 2

Quick tour

This is simple but absolutely useless program that help to illustrate some of the most important Arza features

```
import io (print)
import affirm
include list (range_to as range)
import seq

interface GetSet(I) =
    get(I, key)
    set(I, key, value)

interface Storage(I) =
    contains(I, key)
    use get(I, key)
    use set(I, key, value)
    size(I)

interface StorageKey(I) =
    use get(storage, I)
    use set(storage, I, value)

interface Serializable(I) =
    serialize(storage of I, serializer)

interface Serializer(I) =
    use serialize(storage, serializer of I)

//abstract type


type Nothing
def str(s of Nothing) = "(novalue)"

type First
```

(continues on next page)

(continued from previous page)

```
type Second
type Third

// parens here mean that this is record type
// it will inherit standart behavior like at, put, has, ==, != ...etc
type Collection()

type Single is Collection
    (first)
    init(s) = s.{first = Nothing}

def set(s of Single, k of First, val) =
    s.{first = val}

def get({first} of Single, k of First) = first

def size(s of Single) = 1

//////////////////////////////////////////////////////////////

type Pair is Single
    (second)
    init(p) =
        super(Single, p).{second = Nothing}

// define trait and apply it immidiately to Pair
trait TPair(T) for Pair =
    def set(s of T, k of Second, val) =
        s.{second = val}
    def get(s of T, k of Second) = s.second


// cast Pair to its supertype Single
def size(s of Pair) = size(s as Single) + 1

//////////////////////////////////////////////////////////////

type Triple is Pair
    (third)
    init(t) =
        super(Pair, t).{third = Nothing}

trait TTriple(T) for Triple =
    def set(s of T, k of Third, val) =
        s.{third = val}
    def get(s of T, k of Third) = s.third


def size(s of Triple) = size(s as Pair) + 1

//////////////////////////////////////////////////////////////

// lets create completely unrelated type to Single :> Pair :> Triple
// but use traits for pair and triple to avoid code dublication

type SecondAndThird is Collection (second, third)
```

(continues on next page)

(continued from previous page)

```
instance TPair(SecondAndThird)

instance TTriple(SecondAndThird)

def size(s of SecondAndThird) = 2

//////////////////////////////////////////////////////

type Dictionary is Collection (items)
    init(d) =
        d.{ items = {} }

// do not subtype from Dictionary but use its structure
type Array is Collection
    (size, ...Dictionary)
    init(s, size) =
        //silly idea of arrays implemented as lists
        s.{items = seq:consmany([], Nothing, size), size=size}

// create anonymous trait and apply it serially to list of types
trait (T) for [Dictionary, Array] =
    def size({items} of T) = len(items)

trait TStorageWithItems(T, KeyType) =
    def set(s of T, k of KeyType, val) =
        s.{
            items = @.{ (k) = val }
        }

    def get(s of T, k of KeyType) = s.items.[k]

instance TStorageWithItems(Dictionary, Symbol)
instance TStorageWithItems(Dictionary, StorageKey)
instance TStorageWithItems(Array, Int)

//redefine array size to avoid list
override (prev) size(a of Array) =
    a.size

type InvalidKeyError is Error

// redefine set function for Array
// to avoid index out of range problems
// prev is previous method
// override expression do not adds this method to specific signature set(Array, Int)
// but replaces it completely
// so only indexes > 0 and < size will be accepted
override (prev) set(arr of Array, i of Int, val)
    | ({size}, i, val) when i >= 0 and i < size = prev(arr, i, val)
    | (_, i, _) = throw InvalidKeyError(i)


def ==(d of Dictionary, m of Map) = d.items == m

def ==(m of Map, d of Dictionary) = d.items == m

//////////////////////////////////////////////////////
```

(continues on next page)

(continued from previous page)

```
// define method for parent subtype
def contains(s of Collection, k) =
    let val =
        // if method is not implemented for specific key it will throw␣
↪NotImplementedError exception
        // we catch it and tell user key not exists
        try
            get(s, k)
        catch
            | e of NotImplementedError = Nothing
            | e = throw e

    match val
        | type Nothing = False
        | _ = True

/// there are other more preferable way to implement such behavior
//// this method will be called if specific get(Storage, Key) was undefined
//// for example get(Single, Second) will otherwise crash with not implemented error
def get(s of Collection, k of Any) = Nothing
// after this declaration NotImplementedError will never be thrown in get generic



//////////////////////////////////////////////////////

//ensure that all types are satisfiing interface
describe (Dictionary, Array, Pair, Triple, Single, SecondAndThird) as (Storage,␣
↪GetSet)

def serialize({first, second} of Pair, serializer of Dictionary) =
    serializer
        |> set(_, First, first)
        |> set(_, Second, second)

def serialize(s of Triple, serializer of Dictionary) =
    serializer
        |> serialize(s as Pair, _)
        |> set(_, Third, s.third)

def serialize(s of Array, serializer of List) =
    seq:concat(s.items, serializer)

describe (Triple, Pair) as Serializable
describe Dictionary as Serializer
describe Array  as Serializable

fun array_map({items} as arr, fn) =
    // lets pretend this Array implementation is not based on lists
    // and write some ridiculously slow map implementation
    // there are zip in seq module
    // but lets define our own here
    fun zip(seq1, seq2)
        | (x::xs, y::ys) = (x, y) :: zip(xs, ys)
        | (_, _) = []
```

(continues on next page)

(continued from previous page)

```
    // same with foldl but here we call set directly
    fun fold
        | ([], acc) = acc
        | (hd::tl, acc) =
            let
                (index, item) = hd
                new_acc = set(acc, index, fn(item))
            in
                fold(tl, new_acc)

    let
        arrsize = size(arr)
        indexes = range(arrsize)
    in
        fold(
            seq:zip(indexes, items),
            Array(arrsize)
        )


fun test() =
    let
        single = Single()
            |> set(_, First, #one)

        pair = Pair()
            |> set(_, First, #one)
            |> set(_, Second, #two)

        triple = Triple()
            |> set(_, First, #one)
            |> set(_, Second, #nottwo)
            |> set(_, Third, #three)

        arr = Array(10)
            |> set(_, 0, #zero)
            |> set(_, 5, #five)
            |> set(_, 8, #eight)

        dict =
            Dictionary()
            |> set(_, #one, 0)
            // update
            |> set(_, #one, 1)
            |> set(_, #two, 2)
    in
        affirm:is_equal_all(
            get(single, First),
            get(pair, First),
            get(triple, First),
            #one
        )

        affirm:is_not_equal(get(triple,  Second), get(pair, Second))

        let
            dict1 = dict.{ items = @.{three = [1,2,3]} }
```

(continues on next page)

(continued from previous page)

```
            //deeply nested update
            dict = dict1.{items.three = 0::@}
        in
            affirm:is_true(dict == {one=1, two=2, three=[0,1,2,3]})

        // this is old dict value
        affirm:is_true(dict == {one=1, two=2})
        let
            // lets try some function composition
            fn = (`++` .. "Val->") << str
            // this is equivalent to
            fn2 = (x) -> "Val->" ++ str(x)
            //where (args) -> expression is lambda expression
            arr_str = array_map(arr, fn)
            arr_str2 = array_map(arr, fn2)
        in
            affirm:is_equal(arr_str.items, arr_str2.items)

        let
            dict_ser = serialize(triple, dict)
        in
            affirm:is_true(dict_ser == {(First) = #one, (Second) = #nottwo, (Third) =
↪#three,  one=1, two=2})

            // using func like infix operator
            affirm:is_true(dict_ser `contains` First)
            affirm:is_true(dict_ser `contains` #two)

        affirm:is_true(single `contains` First)
        affirm:is_false(single `contains` Second)
        affirm:is_true(pair `contains` Second)
        affirm:is_true(triple `contains` Third)

        let arr2 =
            try
                set(arr, 10, 10)
            catch e of InvalidKeyError = Nothing
            finally
                set(arr, 9, 42)

        affirm:is_true(get(arr2, 9) == 42)
```

CHAPTER 3

Indentantions and layouts

Arza is indentation “aware” language

If you are familiar with a language like Python that also is whitespace sensitive, be aware that that the rules for indentation in Arza are subtly different.

Arza syntax very similar to F# light syntax where indentation used as statement and expression delimiter but instead of using simple dedents and indents like in Python, Arza uses code layouts to determine block borders

```
// layout is similar to pythons four space indent
fun f(x) =
    1
    2

// layout begins straight after = token
fun f(x) = 1
           2

// this would be a syntax error
fun f(x) = 1
    2

if x == 1 then
     2
else 3

fun add
| (1, 2) = 3
| (x, y) = x + y

match x
| True = False
| False =
     True
```

There are special rules for operators to continue expressions from line above, which differs from F# syntax and more similar to Ruby syntax

```
fun f() =
    1 +
    2 +
    3
    // returns 6
```

However this technique creates problem with ambidextra operators (operators having have both prefix and infix binding powers) Examples of such operators are - and (

To resolve parse conflicts Arza uses new lines as terminators

```
fun f() =
    //lambda expression
    ((x, y) -> x + y)
    // parser treats `(` as prefix expression because of new line
    (1, 41)

    f() == (1, 41)

fun f2() =
    // parser treats `(` as infix expression and interprets
    // this expression as call to lambda with arguments (1, 41)
    ((x, y) -> x + y)(1, 41)

f2() == 42
```

If you do not like to use indentation aware syntax at all, you can enclose any block in (and)

You can enclose in (and) almost any syntax construct and use free code layout without worrying about whitespaces.

```
(fun f() =
        1
+
2 + 3)

(interface Map
    put
        (key, value, Map)
at
    (key,
    Map)
)
```

If you need to use nested statements inside such free layout you must enclose each of them in ()

```
// Nine billion names of God the Integer
fun nbn () =
    string:join(
        seq:map(
            fun(n) =
                string:join_cast(
                seq:map(
                        (fun (g) =
                            //let in block enclosed in ()
                            (let
                                (fun _loop (n, g) =
                                    // if block enclosed in ()
```

(continues on next page)

(continued from previous page)

```
                                    (if g == 1 or n < g then 1
                                    else
                                        seq:foldl(
                                            // fun block enclosed in ()
                                            (fun (q, res) =
                                                // if block enclosed in ()
                                                (if q > n - g  then
                                                    res
                                                else
                                                    res + _loop(n-g, q)
                                                )
                                            ),
                                            1,
                                            list:range(2, g)
                                        )
                                    )
                                )
                            in _loop(n, g)
                            )
                        ),
                        list:range(1, n)
                ),
                " "
                ),
        list:range(1, 25)
        ),
        "\n"
    )
```

However because it is common pattern to use if or match expression inside function call there are special support for such syntax

```
add(match x
     | #one = 1
     | #two = 2
   //here comma terminates free layout and there are no need to enclose match in ()
   , 2)

add(match x
     | #one = 1
     | val of Float =
        // but nested expressions must be enclosed
        (if val < 0.0 then abs(val)
        else val)
   //here comma terminates free layout and there are no need to enclose match in ()
   , 2)
```

CHAPTER 4

Operators

You can define custom operators in prelude global to all modules, or define them locally in you module for your module only.

Some of the operators like `::` `.` `:` `and` `or` `->` `#` are declared internally and have special meaning for compiler

All defined operators in prelude

```
// first name is operator, second is function which be used by compiler and third is␣
↪precedence
// right binding
infixr (:=, :=, 10)

// internal infix as of precedence 15

infixr (<-, <-, 15)
infixr (!, __send__, 15) // sending message
infixl (<|, <|, 15) // pipe left
infixl (|>, |>, 20) // pipe right

// internal infix or precedence 25

infixl (<<, <<, 25) // func composition left
infixl (>>, >>, 25) // func composition right

// internal infix and precendece 30

infixl (<, <, 35)
infixl (>, >, 35)
infixl (>=, >=, 35)
infixl (<=, <=, 35)
infixl (==, ==, 35)
infixl (!=, !=, 35)
infixl (++, ++, 40)
infixl (+, +, 40)
```

(continues on next page)

(continued from previous page)

```
infixl (-, -, 40)
infixl (*, *, 50)
infixl (/, /, 50)

// prefix operator
// cannot use - it is set for infix operator
// use qualified name to prevent infinite loops in cases of declaring local negate␣
↪function using prefix -
prefix (-, arza:negate, 55)

// internal infix :: precedence 60

infixl (**, **, 60) // pow

// internal prefix # precedence 70

prefix (&, &, 70) // deref
prefix (&&, &&, 70) //deref deref
infixl (.., .., 90) // carrying

// internal infix (  .{ .[ precedence 95

prefix (~, ~, 96) // carried function

// internal infix . : precedence 100
```

Later you must create functions for declared operators like

```
fun |>(x, f) = f(x)
fun <|(f, x) = f(x)
fun >>(f, g) = x -> g(f(x))
fun <<(f, g) = x -> f(g(x))
// ... and others
```

When Arza parses expression `1 + 2` it compiles it to `+(1, 2).`

The same with prefix operator. Expression `-1` will be transformed into `arza:negate(1)`

4.1 Special operators

- infix operator `:` like in `module:function()` treats by compiler as exported name and as path separator in `import include` expressions

  ```
  import my:modules:module1
  let three = module1:add(1, 2)
  ```

- infix operators `and or` are compiled into jump instructions
- infix operator `->` creates lambda functions like `(x, y) -> x + y`
- infix operator `::` compiles into call `cons(left, right)` in expressions and receives special treatment in pattern matching
- infix operator `of` compiles into call `kindof(left, right)` in expressions and receives special treatment in pattern matching

- infix operator `as` compiles into call `cast(left, right)` in expressions and receives special treatment in pattern matching
- infix operator `.` like in `left.right` compiles into `at(left, #right)` where `#right` is symbol
- infix operator `.[` like in `left.[right]` compiles into `at(left, right)` where `right` is any expression
- infix operator `.{` like in `left.{key=value}` compiles into `put(left, #key, value)`. If `key` is enclosed in parens like `left.{(key) = value}` it compiles to `put(left, key, value)`.

```
let x = {x=1, (1+2)=2}
let x1 = x.{x=2, (True)=2, (4-1)=2, "key"="value"}
```

- infix operator `(` like in `myfunc(ar1, arg2)` compiles into special bytecode instruction and receives special treatment in pattern matching
- infix operator `{` like in `MyType{key1=value1, key2}` receives special treatment in pattern matching
- infix operator `|` delimits clauses in pattern matching
- prefix operator `not` compiles into special instruction
- prefix operator `#` like in `#I_AM_SYMBOL` constructs symbols in expressions and in match clauses
- prefix operator `...` like in `[x, x1, ...xs]` and `myfunc(...varargs)` receives special treatment in pattern mathing and in function calls

```
match [1, 2, 3]
| [head, ...tail]

fun f(...args) =
    //calling other func
    // ...args flush sequence into call arguments
    f2(1, 2, ...args, 3, 4)
```

4.2 Functions as infix operators

To call function as infix operator enclose it in ‘‘.

Such function operator will have precedence 35

```
mymap `has` #key
i `kindof` Int
1 `add` 2
```

CHAPTER 5

Builtin types

5.1 Booleans

```
True False
```

5.2 Integers

```
1 -42 100012020202
```

5.3 Floats

```
2.02020330 -0.0000000001
```

5.4 Strings

```
"I am string"

"""
I a
             m
       multiline string
"""
```

5.5 Symbols

```
#atomic_string_created_onLyOnce
```

5.6 Tuples

```
// Tuples are always created inside parens, so you can't write just 1,2,3
() (1,) (1,2,3)
```

5.7 Lists

```
[] [1] [1,2,3] 1::2::3::[]
```

5.8 Maps

```
{} {x=1} {x=1, y=(2,3), z=f(), (1+1)=(fun()=1)}
```

5.9 Functions

```
fun add3 (x, y, z) = x + y + z
fun (a, b) = a not b
// lambdas
(x, y, z) -> x + y + z
// equivalent to single element tuple (x,) -> x
x -> x
// tuple from tuple
((x,y,z),) -> x + y + z
```

5.10 Intefaces and generic functions

```
// following code  will create interface Num
// and generic functions -, +, *, /, mod, negate
interface Num(I) =
    -(I, I)
    +(I, I)
    *(I, I)
    /(I, I)
    mod(I, I)
    // unary -
    negate(I)
```

5.11 Types

```
// Abstract singleton type without ability to create instances
type None
type Bool
type True is Bool
type False is Bool
//record type
type Vec2(x,y)
type Vec3 is Vec2(...Vec2, z)
```

5.12 Records

```
// Records are instances of record types
// To create instance use type like a function
let v2 = Vec2(1, 2)
let v3 = Vec3(1, 2, 4)
```

CHAPTER 6

User types and subtyping

In Arza programmer can define abstract and concrete types

6.1 Abstract types

```
type None
type Bool
type One
type LessThan
type GreaterThan
```

Such types has an instances of themselves and can be used as singleton values For example in ML family languages one could write

```
data Maybe a = Just a | Nothing
```

in Arza it would be just

```
type Maybe
// this is subtyping
type Nothing is Maybe
type Just(val)  is Maybe

// now use this type as pattern
// all clauses will be successful here
match Just(1)
| maybe of Maybe
| maybe of Just
| {val} of Just
| {val=1 as value} of Just
// treating types as Tuples
| Just(val)
| Just(1)
```

(continues on next page)

(continued from previous page)

```
// Treating types as maps
| Just{val=1 as value}

match Nothing
| maybe of Maybe
| maybe of Nothing
| type Nothing //need to distinguish between name and empy typeS
// TODO make simpler type literal
| x when x == Nothing

// now  lets write some function
fun operation(data) =
    if can_do_something(data) then
        let val = do_something(data)
        Just(val)
    else
        Nothing
```

6.2 Concrete types

Type `Just` from example above is a concrete type. Such types when called like functions create records. Records in Arza are something like structs in C or named tuples. Internally they differ from tuples because they provide more efficient data sharing between mutated copies.

Records support both access by name and by field index.

It is forbidden to add new fields to records. You only create copy of existing ones with different values

```
let t = (1, 2, 3)
//this is highly undesirable
let t1 = put(t, 0, 42)
// instead
type Three(x, y, z)
let t2 = Three(1, 2, 3)
// much more efficient
let t3 = put(t2, 0, 42)
```

By default concrete types initalize fields in order of declaration in constructor, but programmer can create custom initalizer. Such initializer is function defined with `init` keyword.

Initializer receives uninitialized record as first argument and must set all of it's declared fields. If any of the fields will not be set then exception will be thrown

```
type Library(available_books, loaned_books)
     //initializer
     init(lib, books of List) =
          // here lib variable is an empty record with uninitialized fields
          // returning modified copy of lib
          lib.{available_books = books, loaned_books}

 // lets write function for borrowing book from library
 fun loan_book(library, book_index) =
     let book = library.available_books.[book_index]
     new_lib = library.{available_books = seq:delete(@, book), loaned_books = book::@}
     //return tuple with book and modified library
```

(continues on next page)

(continued from previous page)

```
     (new_lib, book)

 // reverse process
 fun return_book(library, book) =
     library.{
           available_books = book::@,
           loaned_books = seq:selete(@, book)
     }
```

6.3 Subtyping

Arza supports nominal subtyping for abstract and concrete types. Type can have only one supertype and supertype can have multiple subtypes.

Concrete types can not be used as supetypes for abstract types.

Subtypes inherit behavior from supertypes and can be used in multiple dispatch in same roles.

When defining subtype from concrete supertype fields of supertype will be added to fields of would be subtype

```
type Vec2(x, y)
type Vec3 is Vec2 (z)
// Vec3 will have fields(x, y, z)
// defining generic method
def sum(v of Vec2) = v.x + v.y
let v2 = Vec2(1, 2)
let v3 = Vec2(1, 2, 3)
// because sum not defined for Vec3
sum(v2) == sum(v3)
//but after
def sum(v of Vec3) = v.x + v.y + v.z
sum(v3) == 6 != sum(v2)
```

If you don't need behavioral subtyping but want to reuse fields from other types you can paste type in field declaration

```
type Vec2 (x, y)
// paste fields from Vec2
type Vec3 (...Vec2, z)
// Vec2 and Vec3 are unrelated

// More complex example
 type AB(a, b)
 type C(c)
 type DE(d, e)
 type FGH(f, g, h)

 // paste multiple types in multiple position
 type Alphabet (...AB, ...C, ...DE, ...FGH, i, j, k)
```

CHAPTER 7

Conditions and pattern matching

7.1 If-elif-else condition

If condition must have else branch and might have zero or many elif branches if one of the branches succeeds result of it's last expression will be result of entire if expression

```
//if as expression inside function call
affirm:is_true(if 5 > 4 then True else False)
fun f() =
    if something() then
        anything()
    elif something_else() == True then
        // series of expressions inside ()
        // equivalent to {} in C or Java
        io:print("I am here")
        nothing()
    else
        42

// if-elif-else always evaluates to value
let I1 = if 2 == 2 then 2 else 4
let I2 =
    if 2 == 1 then 2
    elif 3 == 4 then 3
    elif {x=1, y=2} == (1,2,3) then 4
    else 5
```

7.2 Pattern matching

Pattern matching is key concept of Arza. It allows to write short and expressive programs.

Also using pattern matching is the only way to bind value to a name.

There are no assignment in Arza.

Pattern matching used in function clauses, generic function specializations, let bindings before = token, lambda functions before -> token, `catch` and `match` expressions.

Arza doesn't have loops so pattern matching and recursion are used to create iterative and recursive processes.

PM expressions can have one or more clauses delimited by | token

```
match [1,2,3,4]
    | 1::2::3::4::[] = #ok
    | x::xs = (x, xs)
```

The expression after `match` is evaluated and the patterns are sequentially matched against the result If a match succeeds and the optional guard is true, the corresponding body is evaluated. If there is no matching pattern with a true guard sequence, runtime error occurs.

Example with guard

```
match (1,2,3)
    | (x, y, z) when z == 2 = #first
    | (x, y, z) when z == 3 and y == 3 = #second
    | (x, y, z) when z == 3 and y == 2 and x == 3 = #third
    | (x, y, z) when z == 3 and y == 2 and x == 1 and A == 2 = #fourth
    | (x, y, z) when z == 3 and y == 2 and x == 1 and not (A `is` True) and greater_
↪then_ten(9) = #fifth
    | (x, y, z) when z == 3 and y == 2 and x == 1 and A `is` True or greater_then_
↪ten(11) = #sixth
    | _ = 12
```

Lets describe all possible patterns for pattern matching in arza (Right sides ommited below, for clarity)

```
match some_expression
    // underscore binds to anything
    | _

    // integers
    | 1

    // floats
    | 2.32323

    // strings
    | "Hello"

    // symbols
    | #World

    // Booleans
    | False
    | True

    // name binds value to variable and succeeds matching of this subbranch
    | x
    | SomeLONG_NAME


    // Tuples
    | ()
```

(continues on next page)

(continued from previous page)

```
    | (1)
    | (1,2,3)
    | (a, b, 42, ...rest)
    // ...rest will take rest of the tuple and put it into new tuple

    // [] destructs all types implementing Seq interface including List
    // ... destructs rest of the data structure
    // :: is cons operator
    | []
    | [1, 2, 3]
    | [1,2,3, x, (a,b,...rest_in_tuple), ...rest_in_list]
    | x::[]
    | 1::2::3::x::rest

    // {} destructs all types implementing Dict interface including Maps and Records
    | {}
    | {x}
    | {x="some value", y, z=42}


    // operator `of` restricts value to type or interface
    | x of Int
    | _ of List
    | {field1, field2=value2} of MyType

    // operator as binds value or expression to variable

    // expression will succeeds if map has key a=True and then it will bind it not to␣
↪a name but to b
    | {a=True as b}

    | {genre, "actress"="Lily" as LilyName, age=13} as Result
    | 42 as i

    // when guard can be used to specify conditions for identical patterns
    | (a, (x, y, z)) when z == 3 and y == 2 and x == 1 and not (a == True)
    | (a, (x, y, z) when z == 4
    | (a, (x, y, z))

    // match types
    | type None
    // if type here is omitted like
    | None it will bind everything to name None
    // interface
    | interface Seq
    // in case of concrete types
    //treating custom types as tuples
    | Vector3(x, y, z)
    //treating custom types as maps
    | Vector3{x, y, z}
```

All data structure pattern except tuples `(n1, n2, ...n)` are accepting user defined data types that implement specific protocols.

- To support patterns `x::x1::xs` and `[x, x1, ...xs]` type must implement `Seq` interface
- To support `{key1=value, key2=value}` type must implement `Dict` interface

Some examples

```
match {name="Bob", surname=("Alice", "Dou"), age=42}
    | {age=41, names} =  (name, age, 0)
    | {name, age=42} =  (name, age, 1)
    | {age=42} =  (age, 2)
    | _ =  42

match (1, 2, 1)
    | (A, x, A)  = (#first, A)
    | (A, x, B)  = (#second, A, B)
    | (3, A) = #third

match {x=1, y="YYYY"}
    | {x of String, y of Int} = #first
    | {x of Int, y="YY" of String} = #second
    | {x of Int, y="YYYY" of String} = #third

match [1,2,3]
    | [a, b, c as B2] as B1 = (B1, B2, a, b, c)
    | _ = 42
// result will be ([1, 2, 3], 3, 1, 2, 3)
```

7.3 let, let-in

Let, Fun, Let-in and match expressions are only ways to bind value to name.

Let expression binds names to values. All patterns, but without guards can be placed by the left hand side of = operator.

```
let a = 1
// checks if true
let 1 = a

// let creates layout and we can write multiple bindings at once
let
    x::xs = [1,2,3]
    1 = x
    [2, 3] = xs

// this expression will fail with MatchError
let {x, y=2} = {x=1, y=3}
```

To avoid conflicts between names one can use let-in expression

Let-in creates nested, lexically-scoped, list of declarations The scope of the declarations is the expressions after *let* and before *in* and the result is the expression after *in*, evaluated in this scope

```
let
    x = 1
in
   let
       x = 2
   in
       x + 2
   x - 2
```

Also let in can be used as expression

```
sum =
    let
        x = 1
        y = 2
    in
        x + y
```

7.4 try-catch-finally

Overall exception handling is very similar to imperative languages with difference that exceptions are matched to catch clauses and if there are no successful branch ExceptionMatchError will be thrown

Any value can be used as exception in throw expression

```
try
    try
        1/0
    catch e1 = e1
catch e2 =
    try
        something()
        something_else()
    catch e3 =
        e3

try
    try
        1/0
    catch e1 = e1
    finally
        something()
        something_else()
catch e2 =
    try
        error(#Catch)
    catch e3 = 42
    finally
        (e2, e3)

// With pattern matching in catch
try
    throw (1,2,"ERROR")
catch
    | err @ (1, y, 3) = #first
    | (1,2, "ERROR@") = #second
    | err @ (1, 2, x) = #third
finally =
    (#fourth, err, x)
```

CHAPTER 8

Functions and partial application

8.1 Functions

Functions are the most important part of any functional language.

In Arza function syntax somewhat similar to ML but has distinctly Erlangish attributes. Main difference from Erlang is that in Arza arity is not part of the function definition. So you can't create functions with same name and different arity. This is conscious choice in language design. For example instead of defining three functions for ranges

```
fun range(to)
fun range(from, to)
fun range(from, to, by)
```

Better to name different processes differently

```
fun range(to)
fun range_from(from, to)
fun range_by(from, to, by)
```

If functions with variadic arity are wanted one can use variadic arguments

```
fun range(...args) =
    match args
    | (to) = // code here
    | (from, to) = // code here
    | (from, to, by) = // code here
```

Function in Arza can be viewed as `match` operator applied to tuple of arguments

The same as with `match` for `fun` expression in clauses, arguments are sequentially matched against patterns. If a match succeeds and the optional guard is true, the corresponding body is evaluated. If there is no matching pattern with a true guard sequence, runtime error occurs.

There are three different types of `fun` expression

8.1.1 Simple function

This is function with only one clause and optional guard

```
fun any(p, l) = disjunction(map(p, l))

fun all(p, l) =
     conjunction(map(p, l))

fun print_2_if_greater(val1, val2) when val1 > val2 =
    io:print("first", val1)
    io:print("second", val2)
```

8.1.2 Case function

This is function with multiple clauses

```
fun foldl
    | (f, acc, []) = acc
    | (f, acc, hd::tl) = foldl(f, f(hd, acc), tl)

// if after | token there are only one argument and it is not tuple enclosing␣
↪parentheses might be omitted
fun to_str
    | 0 = "zero"
    | 1 = "one"
    | 2 = "two"
    // tuples must be enclosed anyway
    | (()) = "empty tuple"
```

8.1.3 Two level function

This is function that combines syntax of previous two. It is a syntactic sugar for common problem of saving first state in deeply recursive processes and also for performing some checks only once

Consider, for example this problem

```
// this function creates inner function and applies it to all it's arguments
// because it does not want to check all types every iteration and also
// it saves coll from first call
fun scanl(func of Function, accumulator of Seq, coll of Seq) =
    fun _scanl
        | (f, acc, []) = acc :: empty(coll) // coll contains initial value from first␣
↪call
        | (f, acc, hd::tl) = acc :: scanl(f, f(hd, acc), tl)
    in _scanl(func, accumulator, coll)

//In Arza there is special syntax for such operation

fun scanl(func, accumulator, coll)
    | (f, acc, []) = acc :: empty(coll)
    | (f, acc, hd::tl) = acc :: scanl(f, f(hd, acc), tl)

// it is compiled to
fun scanl(func, accumulator, coll) =
```

(continues on next page)

(continued from previous page)

```
    let
        fun scanl
            | (f, acc, []) = acc :: empty(coll) // coll contains initial value from␣
↪first call
            | (f, acc, hd::tl) = acc :: scanl(f, f(hd, acc), tl)
    in scanl(func, accumulator, coll)
// so when recursion calls scanl it will calls inner function not outer
```

Some function examples

```
fun count
    | 1 = #one
    | 2 = #two
    | 3 = #three
    | 4 = #four

fun f_c2
    | (a of Bool, b of String, c) = #first
    | (a of Bool, b, c) = #second
    | (a, "value", #value) = #third

fun f_c3
    | (0, 1, c) when c < 0 =  #first
    | (a of Bool, b of String, c) = #second
    | (a of Bool, b, c) when b + c == 40 = #third

fun map(f, coll)
    | (f, []) = empty(coll)
    | (f, hd::tl) = f(hd) :: map(f, tl)
```

8.2 Partial application

Arza has special syntax for partial application

```
// underscores here called holes
let add_2 = add(_, 2)
5 = add_2(3)
let sub_from_10 = sub(10, _)
5 = sub_from_10(5)

// you can use more than one hole
let foldempty = foldl(_, [], _)
```

Also there is builtin function curry which receives normal function and returns carried version

```
carried_add = curry(add)
3 = carried_add(1)(2)

// in prelude there are two operators
//prefix
fun ~ (func) = curry(func)
3 = ~add(1)(2)
//infix
```

(continues on next page)

(continued from previous page)

```
fun .. (f, g) = curry(f)(g)
3 = add .. 1 .. 2
```

Because all data immutable in Arza, partial application and currying combined with pipe and composition operators is often the best way to initialize complex data structures or perform chain of operations.

```
//from prelude
infixl (<|, <|, 15)
infixl (|>, |>, 20)
infixl (<<, <<, 25)
infixl (>>, >>, 25)

fun |>(x, f) = f(x)
fun <|(f, x) = f(x)
fun >>(f, g) = x -> g(f(x))
fun <<(f, g) = x -> f(g(x))


fun twice(f) = f >> f
fun flip(f) = (x, y) -> f(y, x)


//now we can do
let
    l = list:range(0, 10)
in
    affirm:is_equal (
        l |> seq:filter(_, even),
        [0, 2, 4, 6, 8]
    )

    affirm:is_equal(
        l |> flip(seq:filter) .. even
          |> flip(seq:map) .. (`+` .. 1),
         [1, 3, 5, 7, 9]
    )

    affirm:is_equal (
        l |> seq:filter(_, even)
          |> seq:map(_, `+` .. 1)
          |> seq:map(_, flip(`-`) .. 2),
        [-1, 1, 3, 5, 7]
    )

    affirm:is_equal(
        l |> flip(seq:filter) .. (even)
          |> flip(seq:map) .. (`+` .. 1)
          |> flip(seq:map) .. (flip(`-`) .. 2),
        [-1, 1, 3, 5, 7]
    )

    affirm:is_equal(
        l |> seq:filter(_, even)
          |> seq:map(_, `+`(1, _))
          |> seq:map(_, ~(flip(`-`))(2)(_)),
        [-1, 1, 3, 5, 7]
```

(continues on next page)

(continued from previous page)

```
    )

    let
        square = (x -> x * x)
        triple = `*` .. 3
    in
        affirm:is_equal (
            l |> seq:filter(_, even)
              |> seq:map(_, `+` .. 1)
              |> seq:map(_, flip .. `-` .. 2)
              |> seq:map(_, triple >> square),
            [9, 9, 81, 225, 441]
        )

        affirm:is_equal (
             (seq:filter(_, even)
                 >> seq:map(_, `+`(1, _))
                 >> seq:map(_, flip(`-`)(2, _))
                 >> seq:map(_, triple >> square))(l),
             [9, 9, 81, 225, 441]
        )

        affirm:is_equal (
            l |> seq:filter(_, even)
              >> ~(flip(seq:map))(`+` .. 1)
              >> seq:map(_, flip(`-`)(2, _))
              >> ~(flip(seq:map))(triple >> square),
            [9, 9, 81, 225, 441]
        )
```

CHAPTER 9

Interfaces and multimethods

Arza's approach to polymorphism is probably the most original part of the language

If you want to learn in details about multiple dispatch you can read excelent Eli Bendersky's articles https://eli.thegreenplace.net/2016/a-polyglots-guide-to-multiple-dispatch/

Arza uses generic functions with multiple dispatch and intefaces that restricts and formalises relations between types and generics.

9.1 Interfaces

Interface is a term that most often is an attribute of object oriented system. It describes set of operations that specific class can do.

But how can concept of interfaces can be applied to multimethods? In Arza interface can be described as a set of generic functions and specific argument positions in this functions.

For example

```
interface Storage =
   get(storage of Storage, key)

interface Key =
   use get(storage, key of Key)
```

In example above we declare two interfaces that share the same generic function `get`.

Expression `storage of Storage` bind first argument of function `get` to interface `Storage`.

Interface `Storage` will be implemented by all types that act like first arguments in `get`. Interface `Key` will be implemented by all types that act like second arguments in `get`.

The same code can be written in shorter way

```
// all declarations are equivalent

interface Storage =
   get(Storage, key)

//here I is interface alias
interface Storage(I) =
   get(storage of I, key)

// or even shorter
interface Storage(I) =
   get(I, key)
```

Generic function can be declared only inside interface. They can not be declared twice

```
interface Storage =
   get(storage of Storage, key)

interface Key =
   // this is error, function get already declared above
   get(storage, key of Key)
   // instead define reference with `use` expression
   use get(storage, key of Key)
```

Interfaces do not create namespaces, our function `get` will be available as `get`, not as `Storage:get`

Generic function will dispatch only on arguments for whom interfaces are declared.

Interfaces in Arza perform two important functions

- Formalisation of type behavior. Consider Arza's pattern matching. If custom type used as first argument in `first` and `rest` generics, it can be destructured by `x::xs and [x, x1, ...xs]` patterns.

  Because in prelude there is interface

  ```
  interface Seq =
      first(Seq)
      rest(Seq)
  ```

  Compiler just perform this check `arza:is_implemented(customtype, Seq) == True`

  Also consider complex program with a lot of multimethods. In some point you may want to ensure that specific generics implemented for specifice types

- Limit number of arguments to perform multiple dispatch. Multiple dispatch is a costly procedure, especially for dynamic languages. Even with cache strategies substantial amount of methods can degrade performance. By limiting number of dispatch arguments compiler can more easily fall back to single dispatch.

  function `put` has one of the biggest call rate in Arza and because this function defined only with one interface it's call time is more optimised

  ```
  interface Put(I) =
      put(I, key, value)
  ```

One interface can have multiple roles in one generic function

```
interface Eq(I) =
   ==(I, I)
   !=(I, I)
```

Only types that have both first and second roles in **==** and **!=** generics will implement **Eq** interface

Interfaces can be concatenated

```
interface Put(I) =
    put(I, key, value)

interface At(I) =
    at(I, key)
    has(I, key)

// you can combine interfaces
interface Coll is (Put, At)
```

In some specific case there is a need to dispatch not on type of the argument but on argument value. Example is the `cast` function.

```
interface Cast(I) =
    cast(I, valueof to_what)

interface Castable(I) =
    use cast(what, I)
```

To specify that we need to dispatch on value keyword `valueof` is used.

Afterwards we can use it like

```
import affirm

type Robot(battery_power)

def cast(r of Robot, type Int) = r.battery_power

def cast(r of Robot, interface Seq) = [r as Int]

def cast(r of Robot, interface Str) = "Robot"

def at(s of Robot, el) when el == #battery_power  =
    // casting back to Record type to avoid infinite loop
    (s as Record).battery_power + 1


fun test() =
    let
        r = Robot(42)
    in
        affirm:is_equal(r.battery_power, 43)
        affirm:is_equal(r as Int, 43)
        affirm:is_equal(r as Seq, [43])
        affirm:is_equal(r as Str, "Robot")
```

If concrete type defines custom method for `at` generic then to access it's internal structure you must cast it to parent Record type. Like in example above

```
def at(s of Robot, el) when el == #battery_power  =
    // casting back to Record type to avoid infinite loop
    (s as Record).battery_power + 1
```

Most of the times our programs can be easily implemented with single dispatch. In some cases especially for mathematical operations double dispatch is very usefull. But sometimes there is a need for even bigger arity of dispatch function.

I never actually encounter them in my own work, but here I found this example of *Triple Dispatch* on the internet

9.2 Defining methods

To define new method for generic function use `def` expression

```
interface Num =
    //interface must be in both roles
    add(Num, Num)
    // only first argument
    sub(Num, other)


type Vec2(x, y)

def add(v1 of Vec2, v2 of Vec2) = Vec2(v1.x + v2.x, v1.y + v2.y)

//However this would be an error
// because we define second argument to have specific type
def sub(v1 of Vec2, v2 of Vec2) = Vec2(v1.x - v2.x, v1.y - v2.y)

// This is correct
def sub(v1 of Vec2, v2) =
    match v2
    | Vec2(x, y) = Vec2(v1.x - x, v1.y - y)
```

Method definition can be simple function and two level functions but not case function.

Also method definition can have guards

```
interface Racer(R) =
    race_winner(v1 of R, v2 of R)

type Car (speed)
type Plane (speed)

fun faster(v1, v2) = v1.speed > v2.speed

def race_winner(c1 of Car, c2 of Car)
    | (c1, c2) when faster(c1, c2)   = c1
    | (c1, c2) when arza:at(c1, #speed) < c2.speed = c2
    | (c1, c2) when c1.speed == c2.speed = c1

// plane always wins
// Double dispatch
def race_winner(c of Car, p of Plane) = p

def race_winner(p of Plane, c of Car) = p
```

There is a possibility to declare method not as function but as expression

```
def somegeneric(t of Sometype) as someexpression()
```

(continues on next page)

(continued from previous page)

```
// often it's used with functions defined in native modules

// native module
import arza:_string

// assign native functions as methods
def slice(s of String, _from, _to) as _string:slice
def drop(s of String, x) as _string:drop
def take(s of String, x) as _string:take
```

Sometimes there is a need to override existing method

To do so use `override` expression

```
interface F =
    f1(F)

def f1(i of Int)
    | 1 = #one
    | i = i

// overriding
// expression (_) after override means that we do not need previous method
override(_) f1(i of Int) = 21

// here we bind previous method to name super and call it in our new method
override(super) f1(i of Int) = super(i) + 21

// this can be done indefinitely
override(super) f1(i of Int) = super(i) + 42


type Val(val)

// specifying builtin operator +
def + (v1 of Val, v2 of Val) = v1.val + v2.val

//overriding
override (super) + (v1 of Val, v2 of Val) = super(v1, v2) * 2

fun test() =
    affirm:is_equal(signatures(f1), [[Int]])
    affirm:is_equal_all(f1(1), f1(0), f1(10000), f1(-1223), 84)

    let
        v1 = Val(1)
        v2 = Val(2)
    in
        affirm:is_equal((v1 + v2), 6)
```

9.3 Ensuring interface implementation

After implementing all interface roles type will put reference to interface in it's list of implemented interfaces.

But if there is a need to ensuring that this type(types) implements one or more interfaces you can assert this with `describe` expression.

```
describe Symbol as  Concat

describe String as (Eq, Repr,
     Len, Coll,
     Prepend, Append, Concat, Cons,
     ToSeq, Slice, Empty)

describe (Triple, Pair) as Serializable

describe (Dictionary, Array, Pair, Triple, Single, SecondAndThird) as (Storage,␣
↪GetSet)
```

If some of the types does not implement even one of the interfaces then exception will be thrown.

9.4 Traits

Trait in Arza is a function that can work on types. This function consist of one or more `def instance override` expressions. `instance` expression is a trait application to specific number of types.

Traits are tools for code reuse and expressiveness. If subtype-supertype relationship between types is unwanted traits can help to share behavior between them.

```
// creating trait
// trait excepts two types and defines for them two methods
trait TEq(T1, T2) =
    def equal (first of T1, second of T2) = first == second
    def notequal (first of T1, second of T2) = first != second


// applying previously defined trait to couple of types
instance TEq(Int, Int)
instance TEq(Float, Float)
instance TEq(Int, Float)
instance TEq(Float, Int)
```

Arza has special syntax for applying trait immidiatelly after it's declaration

```
trait TValue(T) for MyType =
    def val(v of T) = v.value

// to apply this to trait to more than one type

trait TValue(T) for [MyType1, MyType1, MyType3] =
    def val(v of T) = v.value

// in case of more arguments
trait TEq(T1, T2) for (Int, Float) =
    def equal (first of T1, second of T2) = first == second
    def notequal (first of T1, second of T2) = first != second

 // or to cover all relations
trait TEq(T1, T2)
        for [(Int, Float), (Int, Int), (Float, Float), (Float, Int)] =
```

(continues on next page)

(continued from previous page)

```
    def equal (first of T1, second of T2) = first == second
    def notequal (first of T1, second of T2) = first != second
```

9.4.1 Anonymous traits

If we do not care about reusing trait after declaration we can ommit trait name

```
trait (T1, T2)
        for [(Int, Float), (Int, Int), (Float, Float), (Float, Int)] =
    def equal (first of T1, second of T2) = first == second
    def notequal (first of T1, second of T2) = first != second

// applying anonymous trait to multiple types in serial order
trait (T) for [Float, Int] =
    // applying trait inside trait
    instance TEq(T, T)
    def - (x of T, y) as _number:sub
    def + (x of T, y) as _number:add
```

CHAPTER 10

Working with immutable state

10.1 Modifications

Because all data in Arza immutable there is special need for support of deeply nested modifications of data structures

Consider Map

```
let person = {
   name = "Bob",
   addresses = {
     work="Vatutina st. 24/15",
     homes=["Gagarina st. 78", "Gorodotskogo st. 15"]
   }
}
```

If we need to create new copy of this map with new home address and if we have only standart function `put` to work with, code might be very verbose

```
let new_adress = "Zelena st. 20"
let new_person = put(person,
                     #adresses,
                     put(person.adresses,
                         #homes,
                          cons(new_adress, person.adresses.homes)))
```

This is hard to read and very error prone. Instead in Arza you can just write

```
let new_adress = "Zelena st. 20"
let new_person = person.{adresses.homes = cons(new_adress, @)}
// Here @ placeholder means current path inside data structure
// in case of this example it will be person.addresses.homes
```

Syntax like `object.property = value` impossible in Arza.

Instead you can use more powerfull modification syntax where you can add more than one change at once. With this syntax you can also emulate += operator from imperative languages

More complex examples

```
fun test_map() =
    let
        d = {
            y = 2,
            s1 = {
                (True) = False,
                s2 = {
                    x = 1,
                    s3 = {
                        a = 10
                    }
                }
            }
        }
        d1 = d.{
            s1.True = not @,
            s1.s2.x = @ + 1,
            s1.s2 = @.{
                x=42,
                z=24
            },
            s1.s2 = @.{
                s3 = @.{
                    a = @ - 30,
                    b = 20
                }
            },
            s1.s2.x = @ - 66,
            y = (@ +
                @/2.0*@ *
                seq:reduce([@, @, @], `+`)
                ) + `*`(@, @)
        }
    in
        affirm:is_equal(d1, {y=18.0, s1={s2={z=24, x=-24, s3={b=20, a=-20}},
↪(True)=True}})

fun test_list() =
    let
        d =[
            [0,1,2],
            3,
            4,
            [5,6,7, [8, 9, [10]]]]
        d1 = d.{
            0 = seq:map(@, (x) -> x * x),
            1 = @ * @,
            2 = @,
            (3).(3) = @.{
                0 = @ * 8,
                1 = @ * 9
            },
            (3).(3).((fun () = 2)()).0 = ((x) -> @ * x)(4.2)
```

(continues on next page)

(continued from previous page)

```
        }
    in
        affirm:is_equal(d1, [[0, 1, 4], 9, 4, [5, 6, 7, [64, 81, [42.0]]]])
```

10.2 Default values

Arza does not support keyword arguments in functions, if you want to receive some kind of arbitrary options you can use maps. However often in such option maps some keys must be set to default values.

Arza support special syntax for updating data structure value if it is absent

```
let
    v = {x=1, y=2}
    // right side of or operator will be assigned to x
    // only if there are no previous value
    v1 = v.{x or 42, z or 42, y = 42}
    // the same works with lists, tuples and other data structs
    l = [0, 1, 2, 3]
    l1 = l.{0 or 5}
in
    affirm:is_equal(v1, {y = 42, x = 1, z = 42})
    affirm:is_equal(l1, l)
```

CHAPTER 11

Import and export

Arza uses files as modules. Modules in Arza could be used as top level objects with methods but most of the time there are no need for this. Instead arza treats modules as namespaces or mixins. You include names from one module to another and give to this names specific prefix to avoid conflicts

```
// import qualified names (prefixed by module name)
import seq
import io

// Afterwards, all exported names from this modules available as qualified names
let _ = io:print(seq:reverse([1,2,4,5]))

// import other module
import my:modules:module1

// only last part of module identifier used as qualifier
let three = module1:add(1, 2)

// use aliases to resolve name conflicts
import my:modules:module1 as mod1
import my:module1 as mod1_1

let x = mod1:add(mod1_1:add(1, 2), 3)

// import only required qualified names
import my:module1 (f1 as f1_1, f2 as f2_1)
let _ = module1:f1_1()
let _ = module1:f2_1()

import my:modules:module1 as mod1 (f1, f2)
let _ = mod1:f1()
let _ = mod1:f2()

import my:module1 as mod1 (f1 as f1_1, f2 as f2_1)
let _ = mod1:f1_1()
```

(continues on next page)

(continued from previous page)

```
let _ = mod1:f2_1()

// binding funcs from two modules to same name if there are no conflicts between them
import tests:lib_az:abc:ts:module_s as s
import tests:lib_az:abc:ts:module_t as s

// hiding names
import my:modules:module1  hiding (CONST)
let _ = module1:f1()
let _ = module1:f2()

import my:modules:module1 as mod1 hiding (f1)
let _ = mod1:f2()
let _ = mod1:CONST

import tests:lib_az:abc:module_ab as ab5 hiding (f_ab, CONST)

/// UNQUALIFIED IMPORT
// import specified unqualified names
include my:modules:module1 (f1, f2, CONST as const)
let _ = f1()
let x = f2() + const

// import all unqualified names from module

include my:modules:module1

// hiding specific names
include my:modules:module1 hiding (CONST)
let x = f1() * f2()
```

Also module can specify export list to forbid acces to private values

```
// By default all names except operators can be imported outside
// You can limit it with export expression
let CONST = 41
fun f_ab () = CONST + 1
fun f_ab_2 = f_ab()

export (f_ab, f_ab_2, CONST)
```

11.1 Loading Order

Lets consider what happens when running such arza like

```
python arza.py /root/home/coder/dev/main.arza
```

- compiler imports module prelude.arza If prelude is absent execution will be terminated. All names from prelude will be available as builtins for other modules
- compiler imports rest of standart library (list, tuple, map, . . . etc)
- interpeter compiles file `/root/home/coder/dev/main.arza`, finds in this script function `fun main()` and executes it

How do Arza resolve module imports? When compiler parses expression `import(include) module1.module2.mymodule` it tries to find this module in global cache. If module is absent compiler transforms it's name to path "module1/module2/mymodule.arza". Then it will look for this path in specific order

- Main program folder. If you ran arza as `python arza.py /root/home/coder/dev/main.arza` this directory would be `/root/home/coder/dev/`
- Lib folder. This is directory __lib__ inside script folder `/root/home/coder/dev/__lib__`
- Arza standart library. This is directory from environment variable ARZASTD. If this var is empty all required modules must be located in __lib__ directory

If file is found Arza will load it, compile it and store it in global state. Modules always have unique names throughout all program. Relative imports are not possible. Modules are loaded only once.

CHAPTER 12

Metaprogramming

Arza metaprogramming facilities are limited.

I have not decided yet if I want macroses in language or not.

Instead I borrowed concept of decorators from Python to generate functions and types at compile time.

12.1 Decorators

Decorators are syntactic sugar borrowed from Python for function composition.

Decorators can be applied to functions, types, and methods

In case of decorating functions decorator is a function which recieves other function and optional list of arguments and must return different function.

Example

```
fun add1(fn, v) =
    fun (x, ...args) =
        // ...args will flush contents of a sequence into arguments
        fn(x+v, ...args)

fun add2(fn, v1, v2) =
    fun (x, y) =
        fn(x+v1, y+v2)

// applying decorators
@add1(10)
@add2(0.1, 0.2)
fun f(x, y) = x + y

// now f is functions decorated by to functions add1 and add2

// decorators can be applied to specific methods
```

(continues on next page)

(continued from previous page)

```
interface I =
    add(I, I)

@add1(10)
@add2(0.1, 0.2)
def add(x of Int, y of Int) = x + y

@add1(10)
override (super) sub(x of Int, y of Int) =  super(x, y) + super(x, y)

// decorators can be used in traits also
trait Add(T) for Float =
    @add1(0.1)
    @add1(0.01)
    def add(x of T, y of T) = x + y

    @add2(0.001, 0.0001)
    override (super) add(x of T, y of T) = super(x, y) * -1

// lets test our new functions
affirm:is_equal_all(f(1,2), add(1,2), 13.3)
affirm:is_equal(add(1.0, 2.0), -3.1111)
```

When decorating types decorator will receive tuple of three elements as first argument.

This tuple will consist of supertype, fields as list of symbols and initialisation function.

```
// this decorator will add specific field to type fields if this field is not already␣
↪there
fun add_field ((supertype, fields, _init) as typedata, field) =
    if not has(fields, field) then
        (supertype, append(fields, field), _init)
    else
        typedata

// this decorator will add field #y
let add_y = add_field(_, #y)

// this decorator will init specific field with value after initialisation
fun init_field((supertype, fields, _init), field, value) =
    let
        fun _wrap(...args) =
            let
                data = _init(...args)
            in
                data.{(field) = value}
    in
        (supertype, fields, _wrap)

// this is almost the same like above but initialize field with special function
fun init_field_with((supertype, fields, _init), field, value, fn) =
    let
        fun _wrap(...args) =
            let
                data = _init(...args)
            in
                data.{(field) = fn(@, value)}
```

(continues on next page)

(continued from previous page)

```
    in
        (supertype, fields, _wrap)

// Lets apply them to some types
@add_field(#z)
@add_y
type XYZ(x)

@add_field(#c)
@add_field(#b)
@add_field(#a)
type ABC()


@init_field(#b, #b)
@init_field_with(0, #c, (x, y) -> x ++ y)
@add_field(#b)
type AB(a)
    init (ab, a) = ab.{a=a}

type Sum(v)
    init (sum, x, y) =
        sum.{v = x + y}

@extends(Sum)
type Sum2

// now we can test with

let
    xyz = XYZ(1, 2, 3)
    abc = ABC(1, 2, 3)
    ab = AB(#a)
    sum1 = Sum(1,2)
    sum2 = Sum(1, 2)
in
    affirm:is_equal_to_map(xyz, {x=1, y=2, z=3})
    affirm:is_equal_to_map(abc, {a=1, b=2, c=3})
    affirm:is_equal_to_map(ab, {a=#ac, b=#b})
    affirm:is_equal_all(sum1.v, sum2.v, 3)
```

CHAPTER 13

Processes

Arza is heavily inspired by Erlang and uses its idea of processes as a concurrency tool.

Processes or actors or symmetric coroutines are independent universal primitives of concurrent computation.

They can exchange messages but can not share any data.

Arza syntax for process creation and message handling very similar to Erlang

```
//to spawn process
let pid = spawn(somefunc, args)

// get pid of current process
let this_pid = self()

// to receive messages from other processes
receive
    | clause1 = branch1
    | clause2 = branch2

// to kill process
close(pid)
```

Ping-Pong example

```
// usefull functions
import process

type PingPongFinished(store)

fun ping(n, pong_pid, store) =
    if n == 0 then
        // This is message sending
        pong_pid ! #finished
        store
    else
```

(continues on next page)

(continued from previous page)

```
        // self() returns current pid
        pong_pid ! (#ping, self())
        receive #pong =
                ping(n-1, pong_pid, #pong :: store)

fun pong(store) =
    receive
        | #finished =
            // if excepion occures it became process result
            throw PingPongFinished(store)

        | (#ping, ping_pid) =
            ping_pid!#pong
            pong(#ping :: store)

    // this disables printing exceptions in processes to stderr
    process:set_error_print_enabled(pong_pid, False)
    let pong_pid = spawn(pong, ([],))
    // using currying just for clarity
    ping_pid = spawn .. ping .. (3, pong_pid, [])
    // waiting for all processes to end
    result = process:wait_for([pong_pid, ping_pid])
in
    affirm:is_equal(result.[ping_pid], [#pong, #pong, #pong])
    affirm:is_equal(result.[pong_pid], ValueError([#ping, #ping, #ping]))
    // closing
    close(pong_pid)
    close(ping_pid)
```

With symmetric coroutines it's easy to implement asymmetric coroutines

```
// function from std lib
fun coro(fn) =
    let
        proc1 = self()
        proc2 = process:create()

        fun make_chan(pid) =
            fun (...args) =
                if is_finished(pid) then
                    throw CoroutineEmpty(result(pid))
                else
                    match args
                        | () = pid ! ()
                        | (m) = pid ! m
                    // receiving
                    receive msg = msg
        chan1 = make_chan(proc2)
        chan2 = make_chan(proc1)

        fun wrapper(...args) =
            let
                (res =
                    (try
                        fn(...args)
                    catch e = e
                    )
```

(continues on next page)

(continued from previous page)

```
                 )
            in
                proc1 ! res
                res

        fun _coro(...args) =
            if is_idle(proc2) then
                start(proc2, wrapper, (chan2,) ++ args)
                receive msg = msg
            else
                chan1(...args)
    in
        _coro

fun test_coro() =
    let
        fun test1() =
            let
                c = process:coro(fun(yield, start) =
                    (let
                        x = yield(start)
                    in
                        yield(x)
                    )
                )
                c1 = process:coro((yield) -> #zero)
            in
                affirm:is_equal .. c1() .. #zero
                affirm:is_throw(c1, ())

                affirm:is_equal .. c(#first) .. #first
                affirm:is_equal .. c(#second) .. #second
                affirm:is_equal .. c(#last) .. #last
                affirm:is_throw(c, ())
                affirm:is_throw(c, ())

        fun test2() =
            let
                c = process:coro(fun(yield) = throw 1)
            in
                affirm:is_equal .. c() .. 1
                affirm:is_throw(c, ())
    in
        test1()
        test2()
```

With processes you can create emmulation of mutable state

Example: *Mutable State*

CHAPTER 14

Code examples

14.1 Simple game

```
// This is simple battleship game
// All ships have 1x1 size and they don't move throughout course of the game
// gameplay consists of random shooting by each of the ships

// module imports
// print functions
import io
// sequence combinators
import seq
import string
// test assertions
import affirm

// type declaration
type Ship(id, hp, pos)

// defining method for generic function str which will be called by print function

def str({id, hp, pos} of Ship) =
    // Arza lacks sprintf for now, instead this is simple concatenation
    string:all("<Ship id=", id, " hp=", hp, " pos=", pos, ">")

// Other type with initializer
// Product of this type will be value which initializer returns
type Game(cols, rows, ships, _id)
    init(game, cols, rows) =
        game.{
            // field attrs
            rows = rows,
            cols = cols,
            // list with ships, list is not a best type of data structure here but␣
↪the simplest one
```

(continues on next page)

(continued from previous page)

```
            ships=[],
            // special id increment for new ships
            _id=0
        }


def str({rows, cols, ships} of Game) =
    string:all(
        "<Game (",
        rows,
        ", ",
        cols,
        ") ",
        // if is an expression like almost everything
        if not is_empty(ships) then "ships: \n...." else "",
        string:join("\n....", ships),
        " >"
    )

// checking if position is on board
fun is_valid_pos({rows, cols}, (x, y) as pos) =
    x >= 0 and x < cols and y >= 0 and y < rows

// add ship and return new game record
// because values are immutable in arza
fun add_ship(game, pos) =
    // increment id counter
    let new_id = game._id + 1
    // create new ship with 2 hit points
    let ship = Ship(new_id, 2, pos)
    // .{ operator allows to create modified immutable structure
    // here we creating new instance of Game from old one with changed keys _id and␣
↪ships
    // @ placeholder means previous value and :: is cons operator
    game.{
        _id = new_id,
        ships = ship::@
        // can be written as
        // ships = cons(ship, @)
    }


// using seq module for finding ship at pos ship_pos
fun atpos({ships} of Game, ship_pos) =
    // function arguments are subjects of pattern matching
    // {ships} of Game means argument must be of type Game
    // and must implement Map interface and has attribute ships
    // binding ships will be created
    seq:find_with(
        ships,
        // default value
        None,
        //lambda expression
        ship -> ship_pos == ship.pos
    )
```

(continues on next page)

(continued from previous page)

```
fun update_ship(game, newship) =
    // modifing game.ships
    game.{
        ships = seq:map(
            // equivalent to game.ships
            @,
            // using parens to delimit multi expression function
            (fun(ship) =
                (if ship.id == newship.id then
                    newship
                else
                    ship))
        )
    }


// fire at random position
fun fire({rows, cols} as game, ship) =
    let
        x = randi(0, rows)
        y = randi(0, cols)
        fire_pos = (x, y)

    if fire_pos == ship.pos then
        //retry
        fire(game, ship)
    else
        fire_at(game, ship, fire_pos)


// as operator in pattern matching will bind left value to right name in case of␣
↪successful branch
fun fire_at({rows, cols, ships} as game, ship, fire_pos) =
    let enemy = atpos(game, fire_pos)
    // if we found enemy change its hp
    // this all immutable of course, so we return new game state
    match enemy
        | enemy of Ship =
            update_ship(game, enemy.{hp = @ - 1})
        | None =
            game


fun turn({rows, cols, ships} as game) =
    // this basically foreach through all ships
    // foldl is used because we can put current state as accumulator
    /*
        foldl is basically this function
        fun foldl
            | ([], acc, f) = acc
            | (hd::tl, acc, f) = foldl(tl, f(hd, acc), f)
    */
    seq:foldl(
        ships,
        game,
        fun (ship, new_game) =
            fire(new_game, ship)
```

(continues on next page)

(continued from previous page)

```
    )


// win conditions
// all ships are dead then draw
// if one ship alive she is the winner
// else continue playing
fun checkgame(game) =
    let (alive, dead) = seq:partition(game.ships, fun({hp}) = hp > 0 )
    match alive
        | [] = (game, (#DRAW, "All dead"))
        | x::[] = (game, (#WINNER, x))
        | _ = None


// This game main loop
// This type of function is called recursive wrappers in arza
// first branch will be executed only once
// and subsequent calls will not check when count > 0 guard
fun run(game, count) when count > 0
    | (game, 0) = (game, (#DRAW, "Time is out"))
    | (game, count_turns) =
        let game1 = turn(game)
        match checkgame(game1)
            | None = run(game1, count_turns - 1)
            | result = result


// just simple random game
fun random_game() =
    let
        size = 4
        pos = () -> randi(0, size)
        (game, result) = Game(size, size)
                |> add_ship(_, (pos(), pos()))
                |> add_ship(_, (pos(), pos()))
                |> run(_, 100)
    io:p(#GAME, game)
    io:p(#RESULT, result)


// and some testing
fun test() =
    fun test_game() =
        let game = Game(4, 4)
                |> add_ship(_, (3,1))
                |> add_ship(_, (0,0))
        let ship1 = atpos(game, (3, 1))
        let ship2 = atpos(game, (0, 0))
        (game, ship1, ship2)

    let
        (game, ship1, ship2) = test_game()
    in
        let
            (game1, result) = game
                |> fire_at(_, ship1, ship2.pos)
```

(continues on next page)

(continued from previous page)

```
                |> fire_at(_, ship2, ship1.pos)
                |> fire_at(_, ship1, ship2.pos)
                |> fire_at(_, ship2, ship1.pos)
                |> checkgame(_)
        in
            affirm:is_equal(result.0, #DRAW)

    let
        (game, ship1, ship2) = test_game()
    in
        let
            (game1, (label, winner)) = game
                |> fire_at(_, ship1, ship2.pos)
                |> fire_at(_, ship2, ship1.pos)
                |> fire_at(_, ship1, ship2.pos)
                |> checkgame(_)
        in
            affirm:is_equal(label, #WINNER)
            affirm:is_equal(winner.id, ship1.id)
```

14.2 Mutable State

```
// this program will implement mutable state via processes

import process
import decor

type State(pid)

// special error
type StateError is Error

// because State will implement at generic all calls like state.key or
// matches {key1, key2} will be infinitely recursive
// to avoid this we need to cast state to parent Record type
// asrecord defined in prelude like fun asrecord(r) = r as Record
fun pid(s) = asrecord(s).pid


fun is_valid(s) =
    not process:is_finished(pid(s))

fun __ensure_process(s) =
    if not is_valid(s) then
        throw StateError("Process inactive")
    else
        s

// creating assertion decorators as partially applied function decor:call_first
let ensure1 = decor:call_first(_, 1, __ensure_process)
let ensure2 = decor:call_first(_, 2, __ensure_process)
let ensure3 = decor:call_first(_, 3, __ensure_process)

// trait is function which can operate on types
```

(continues on next page)

(continued from previous page)

```
// traits have global side effects
// they used to specify behavior for one or more types
// and can be applied to different set of types with 'instance' expression
// this is anonymous trait. They are used just for convinience to avoid typing long␣
↪type names

// generic functions at, put has specific meaning in arza because expression
// x.y transforms by compiler into at(x, #y) and x.{y=1} into put(x, #y, 1)
trait (T) for State =
    // T means State
    def close(s of T) =
        process:kill(pid(s), 0)

    // all ensure decorators assert that state process is not dead
    @ensure3
    def put(s of T, key, value) =
        // sending tuple to process
        // #put is symbol specifiing type of action
        pid(s) ! (#put, key, value)
        // returning itself
        s

    @ensure2
    def at(s of T, key) =
        // sending request
        pid(s) ! (#at, self(), key)
        // and receiving reply
        receive (#at, val) = val

    @ensure1
    def &(s of T) =
        pid(s) ! (#get, self())
        receive (#get, val) = val

    @ensure2
    def := (s of T, val) =
        pid(s) ! (#set, val)
        s

    @ensure2
    def del(s of T, el) =
        pid(s) ! (#del, el)
        s

    @ensure2
    def has(s of T, el) =
        pid(s) ! (#has, self(), el)
        receive (#has, val) = val

    @ensure1
    def arza:len (s of T) =
        pid(s) ! (#len, self())
        receive (#len, val) = val

    @ensure2
    def ==(s of T, data) = &s == data
```

(continues on next page)

(continued from previous page)

```
    @ensure1
    def arza:is_empty(s of T) = len(s) > 0


// this is actual process
fun _loop(data) =
    // this block will receive messages from other processes
    receive
        | (#set, new_data) =
            // just replace data
            _loop(new_data)

        | (#get, pid) =
            // receiving action with receiver
            // replying to receiver
            pid ! (#get, data)
            // going to loop again because otherwise process will be finished
            _loop(data)

        | (#at, pid, key) =
            pid ! (#at, data.[key])
            _loop(data)

        | (#has, pid, key) =
            // calling has generic func  as has operator
            pid ! (#has, data `has` key)
            _loop(data)

        | (#len, pid) =
            pid ! (#len, len(data))
            _loop(data)

        | (#put, key, val) = _loop(data.{(key)=val})

        | (#del, key) = _loop(del(data, key))
        | msg = throw (#InvalidMessage, msg)

//constructor function
/*
    you can use this module like
    import state
    let s = state:new({x=1, y=2, z=3})
    updates state structure
    s.{x=2}
    replaces state value
    s:=1
*/
fun new(data) =
    let pid = spawn(_loop, data)
    State(pid)
```

14.3 Triple Dispatch

```
import seq

//case for triple dispatch described here https://softwareengineering.stackexchange.
↪com/questions/291525/a-real-world-use-case-for-triple-dispatch
//
//This program represents a repository of citation information, containing books,␣
↪articles and journals
//with action of formatting those books for consumption on demand.
//
//Let's take two approaches to formatting. National Library of Medicine (derived from␣
↪the Vancouver Project)
//specifies citations in a particular way, mostly affecting how author names are laid␣
↪out.
//NLM differs from American Psychological Association (APA) formatting.
//
//Also we have to publish these citations and choice of outputs are: plain text, PDF,␣
↪HTML.
//Some of these items require different layout strategies,
//depending on the type of the format (APA indents following lines, NLM doesn't).

// Declaring interfaces

interface Repo(I) =
    add(I, item)

interface Source(I) =
    format(item of I, format_standart, output_format)

interface Standart(I) =
    use format(item, format_standart of I, output_format)

interface Output(I) =
    use format(item, format_standart, output_format of I)

fun format_books(library) =
    let books = seq:map(library.books, format)

// Declaring types

type Item(author, name)
type Record(id, item)
type Book is Item
type Article is Item
type Journal is Item

type FormatStandart
type NLM is FormatStandart
type APA is FormatStandart

type OutputFormat
type PDF is OutputFormat
type HTML is OutputFormat
type TXT is OutputFormat

type Library(_id, items)
```

(continues on next page)

(continued from previous page)

```
    init(l) =
        l.{items = []}

// Defining generic functions

def add(l of Library, item) =
    let id = l._id + 1
    l.{
        id = _id,
        items = Record(id, item)::@
    }


def format(b of Book, c of NLM, f of TXT) = None // do something here
def format(b of Article, c of NLM, f of TXT) = None // do something here
def format(b of Journal, c of NLM, f of TXT) = None // do something here

def format(b of Book, c of APA, f of TXT) = None // do something here
def format(b of Article, c of APA, f of TXT) = None // do something here
def format(b of Journal, c of APA, f of TXT) = None // do something here

// and so on and so on
```

14.4 Some sequence functions

```
fun foldl
    | ([], acc, f) = acc
    | (hd::tl, acc, f) = foldl(tl, f(hd, acc), f)


fun foldr
    | ([], acc, f) = acc
    | (hd::tl, acc, f) = f(hd, foldr(tl, acc, f))


fun reduce(sq, f)
    | (x::xs, f) = foldl(xs, x, f)
    | ([], f) = throw EmptySeqError(sq)


fun map(sq, f)
    | ([], f) = empty(sq)
    | (hd::tl, f) = f(hd) :: map(tl, f)


fun filter(sq, predicate)
    | ([], p) = empty(sq)
    | (x::xs, p) =
        if p(x) then
            x :: filter(xs, p)
        else
            filter(xs, p)
```

(continues on next page)

(continued from previous page)

```
fun sort(s, f) =
    let
        fun _merge
            | ([], ys) = ys
            | (xs, []) = xs
            | (x::xs, y::ys) =
                if f(x, y) then x :: _merge(xs, y::ys)
                else y :: _merge(x::xs, ys)

        fun _sort
            | [] = []
            | [x] as s = s
            | xs =
                let (ys, zs) = split(xs)
                in _merge(_sort(ys), _sort(zs))

    in _sort(s)


fun zip(seq1, seq2)
    | (x::xs, y::ys) = (x, y) :: zip(xs, ys)
    | (_, _) = []


fun unzip(l) =
    let fun _unzip
        | ((x, y) :: ts, xs, ys) = _unzip(ts, x :: xs, y :: ys)
        | ([], xs, ys) = (reverse(xs), reverse(ys))

    in _unzip(l, [], [])


fun zipwith(seq1, seq2, f)
    | (x::xs, y::ys, f) = f(x, y) :: zipwith(xs, ys, f)
    | (_, _, _) = []


fun span(sq, predicate)
    | ([], p) =
        let c = empty(sq)
        in (c, c)
    | ([x, ...xs1] as xs, p) =
        if not(p(x)) then
            (empty(sq), xs)
        else
            let (ys, zs) = span(xs1, p)
            in (x::ys, zs)
```

CHAPTER 15

Indices and tables

- search