

Introduction to Timber

©Andrey Kruglyak, 2010

Wednesday, April 28, 2010

1

Timber

- ◁ Timber is a general-purpose object-oriented programming and modeling language
- ◁ It was developed primarily to target real-time systems
- ◁ It has a number of interesting features, such as:
 - ◁ reactivity (time-constrained reactions)
 - ◁ object-level concurrency
 - ◁ an advanced type system with a type inference mechanism
- ◁ <http://www.timber-lang.org>

Wednesday, April 28, 2010

2

Timber

- ◁ Current state:
 - ◁ stable syntax & semantics
 - ◁ compiler for Mac OS X & Linux released in fall 2008 (no optimizations)
 - ◁ limited run-time systems for ARM and for POSIX under Mac OS X & Linux (no GUI)
- ◁ Compilation: Timber code \rightarrow timberc C code \rightarrow gcc executable
 - ◁ C compiler is invoked automatically by timberc

Wednesday, April 28, 2010

3

WARNING!

- ◁ Timber syntax is layout-sensitive (like Haskell)
 - ◁ do not mix spaces and tabs for layout!
- ◁ Capitalization is important

Wednesday, April 28, 2010

4

Programming paradigm of Timber

- ◁ Timber is OO, imperative, functional, highly parallel language
- ◁ But not an ad-hoc mix of those features!
 - ◁ We would like to describe computations in a short and clear manner, so expression evaluation is functional, i.e. calculates a value without producing side-effects. This includes functions, which can only operate on their arguments (no global variables).
 - ◁ We would like to encode interactions, so we introduce state variables and encapsulate them into objects. These can be updated from the object's own methods (but within an expression, they can only be read, not written):
s := fcn (s+3)

Wednesday, April 28, 2010

5

Programming paradigm

- ◁ We would like to introduce concurrency into our programs, so we allow methods of different objects to execute concurrently. This is known as object-level concurrency.
- ◁ However, to ensure state consistency (to protect state variables from concurrent modification), we only allow one method of each object to be active at any particular time (and the variables cannot be accessed from outside the object).

Wednesday, April 28, 2010

6

Objects

- ◁ Objects contain the system's state, but not everything is objects (unlike Java): constant values, including functions, and procedures can be defined outside objects.
- ◁ Objects are created by using the keyword "new" followed by a class expression:

```
c = new myclass arg1 arg2
myclass p1 p2 = class
... /* class definition */
OR
c = new class
... /* class definition */
```

Wednesday, April 28, 2010

7

Class expressions

- ◁ "class" +
a sequence of commands +
"result"-command
- ◁ Commands:
 - ◁ Declaration & initialization of a state variable (name or pattern := expression)
s := 32
m := s^2
Note that the expression can include any names bound within the class (including those bound below in the code), but can only include state variables defined above in the code.

Wednesday, April 28, 2010

8

Class expressions

◀ Commands:

◀ Bindings (name or pattern = expression)

```
k = 14  
d = 7+k
```

A binding is not a variable assignment! The name cannot be re-bound to another value, and the expression on the right is evaluated functionally, i.e. with the only purpose to obtain a value (no side-effects, i.e. no state assignments and no execution of methods in another object). Note that while an object of the class is being created, state variables have not been initialized and hence cannot be read.

◀ All bindings within a class form a recursive group - more about it later

Class expressions

◀ Commands:

◀ Special bindings with "= new" (name or pattern = new expression)

```
k = 14  
d = s+k
```

The expression after "new" should evaluate to a class, and once it is evaluated, an object of the class is instantiated

```
c = myclass 1 3 /* name binding */  
o = new myclass 1 3 /* instantiation of an  
                        object of the class */
```

Creating one object may involve creating other objects (but not method invocation)!

◀ These bindings are part of the recursive group of bindings

Class expressions

◀ Commands:

◀ "result"-command: always last and always necessary in a class (result + expression). The expression is evaluated when an object of the class is created, and then it is bound to the name (or pattern) to the left of "= new":

```
c = new class  
  s := 12  
  result 45+s
```

We call this an interface, as the returned value (typically a struct containing methods, but can be any value) is the only way to communicate with the created object.

Class expressions

◀ A class expression can be used anonymously:

```
c = new class  
  s := 0  
  inc = action  
    s := s+1  
  result inc
```

◀ ... or it can be bound to a name and then used multiple times:

```
c1 = new mkC 4  
c2 = new mkC 8  
mkC p1 = class  
  s := p1  
  inc = action  
    s := s+1  
  result inc
```

Class expressions

- ◁ A class defined inside another class can access names (including parameters) of the enclosing class as these are constants, but it cannot access the state variables of the enclosing class:

```
mkC p = class
  n = p*2
  s := 0
  o = new class
    v1 = p      /* This is OK */
    v2 = n      /* This is OK */
    v3 = s      /* This is a compile-time error */
    result ()
  result ()
```

Method and procedure expressions

- ◁ A class expression may contain bindings, where a name or a pattern is bound to a method expression or a procedure expression. Methods can be asynchronous (action) or synchronous (request). Procedures (do) may also return a value.
- ◁ "action/request/do" + a sequence of commands + "result"-command
- ◁ If no "result"-command is specified, the compiler automatically adds "result ()" at the end. Note that a "result"-command in an asynchronous method (action) has no effect!

Method expressions

- ◁ Methods operate on state of a particular object. When executed, methods lock that object and operate on its state.
- ◁ Methods are defined on a particular object, i.e. the coupling to that object is performed when a method expression is evaluated.
- ◁ In today's syntax, this coupling is always implicit, as methods are assumed to operate on the state of the current object, "self". That is why method expression can only be written within a class or its methods/procedures.

Procedure expressions

- ◁ Unlike methods, procedures are not coupled to a particular object.
- ◁ Procedures that do not operate on any object's state should be defined outside classes. Multiple instance of such procedures can then execute in parallel.
- ◁ Procedures that operate on an object's state can only be invoked from a method of this object, so that the object remains locked. The coupling to the object is performed when the procedure is executed.
- ◁ In today's syntax, procedures defined within a class are always assumed to operate on the state of an object of that class. That is why they cannot be invoked from an object of another class.

Method and procedure expressions

- ◀ In addition to recursive groups of bindings also possible in classes (such groups will be interrupted by any of the following commands), methods and procedures may include:
 - ◀ Updates of state variables (name or pattern := expression)
`s := 3+s`
 - ◀ Invocations of methods
([name or pattern <-] method expression)
`c = new counter`
`v <- c.get`
`c.inc /* same as: _ <- c.inc */`
 - ◀ Invocations of procedures
([name or pattern <-] procedure expression)

Wednesday, April 28, 2010

17

Method and procedure expressions

- ◀ "if" commands:
`if expression then`
 a sequence of commands
`elsif expression then`
 a sequence of commands
`else`
 a sequence of commands
- ◀ When an "if" command is last:
 - the last command in a sequence must be a "result" command (but "result ()" can be added by the compiler);
 - if the "then"-branch returns something other than (), then the "else" branch can not be omitted (and should return a value of the same type)

Wednesday, April 28, 2010

18

Method and procedure expressions

- ◀ "case" commands:
`case expression of`
 pattern ->
 a sequence of commands
 pattern ->
 a sequence of commands
- ◀ When a "case" command is last:
 - the last command in a sequence must be a "result" command (but "result ()" can be added by the compiler);
 - if none of the patterns match, an exception is raised at run time

Wednesday, April 28, 2010

19

Method and procedure expressions

- ◀ An example of a "case" command with guards:

```
case mylist of
  x:xs | x > 0 ->
    s := x
  x:xs | x == 0 && length xs > 3 ->
    s := 0
  x:xs ->
    s := -1
[] ->
  s := -2
```

Wednesday, April 28, 2010

20

forall

- Can be used in classes, methods and procedures
- name = "forall" pattern <- expression1 "new" expression2
 - expression1 must be a list expression
 - expression2 must be a class expression
 - the whole construction means: for each element of the list, create an object and bind the name to a list of interfaces

Wednesday, April 28, 2010

21

forall example

```
actions = forall x <- [1..4] new myclass x
```

```
myclass x = class  
  s := x  
  inc = action  
  s := s+1  
  result inc
```

{- 'actions' becomes a list of four actions, each coupled to a separate object with state variable 's' initialized to 1, 2, 3, 4, respectively -}

Wednesday, April 28, 2010

22

forall

- Can be used in classes, methods and procedures
- name = "forall" pattern <- expression1 "class"
a sequence of commands
"result"-command
 - expression1 must be a list expression
 - the whole construction means: create a list of classes and bind it to the name

Wednesday, April 28, 2010

23

forall example

```
classes = forall x <- [1..4] class  
  s := x  
  inc = action  
  s := s+1  
  result inc
```

{- 'classes' becomes a list of four classes parametrized with "x" where x is 1 for the first class, 2 for the second class, etc. -}

Wednesday, April 28, 2010

24

forall

- Can be used in classes, methods and procedures
- "forall" pattern <- expression, "do" + a sequence of commands ["result" command]
 - expression must be a list expression
 - the whole construction denotes a procedure expression
 - executing the procedure involves executing the sequence of commands for each element of the list
 - if the sequence doesn't end in a "result" command, it is equivalent to writing "result ()"

forall example

```
proc = forall x <- [0..99] do
  temp = x*x
  env.stdout.write (show temp)
  result temp
values <- proc

{- 'proc' becomes a procedure expression; when it
is executed, 'values' becomes a list of 100 values -}
```

Timber syntax summary

```
exp ::= ...
      |'action' cmd
      |'request' cmd
      |'do' cmd
      |'class' cmd_r

cmd ::= cmd_r | cmd_e

cmd_r ::= {c_e}* c_r

cmd_e ::= {c_e}*
```

Timber syntax summary

```
c_r ::= 'result' exp
      |'case' exp 'of' {pat '->' cmd_r}+
      |'if' exp 'then' cmd_r {'elseif' cmd_r}* 'else' cmd_r

c_e ::= pat = exp
      |pat = 'new' exp
      |pat := exp
      |pat <- exp
      |exp
      |'case' exp 'of' {pat -> cmd_e}+
      |'if' exp 'then' cmd_e {'elseif' cmd_e}* ['else' cmd_e]
```

Execution of methods & procedures

- ◀ Method and procedure expressions can be evaluated (after "="), which is not the same as they are executed (after "<-" or as a standalone command within a method or a procedure).
- ◀ When they are executed, methods and procedures can invoke (execute/schedule execution of) other methods and procedures.

Execution of methods & procedures

- ◀ The question arises: how do we start execution in general? At the top level in each module we only have a number of name or pattern bindings and type declarations and signatures.
- ◀ We define a special procedure called "root" that is called when a program is started. It can use its parameter (of type `World`) to create one or several environment objects, and also to create other objects. An environment object would typically have methods for installing handlers (invoked by the runtime system when a particular external event occurs), which can be methods of created objects or top-level procedures.

A POSIX example

```
1 module Test where
2
3 import POSIX
4
5 struct Echo where
6   echoString :: String -> Action
7
8 echo env = class
9   echoString str = action
10   env.stdout.write str
11   result Echo{..}
12
13 root :: RootType
14 root w = do
15   env = new posix w
16   echoObj = new echo env
17   env.stdin.installR echoObj.echoString
```

Execution model of Timber

- ◀ Execution is event-driven: "Start" event is handled by the "root" procedure, other events by installed handlers (top-level procedures or methods of specific objects)
- ◀ Methods lock objects (to protect state consistency) and may invoke procedures defined in the same class or top-level procedures without releasing the lock. Procedures may be recursive:

```
myproc a = do
  if a==0 then
    result 0
  else
    temp <- myproc (a-1)
    result a+temp
```
- ◀ Methods can invoke other methods (=post a message to an object)

Execution model of Timber

- ◁ A request may return a value that is needed, so the caller object waits for it to return (and remains locked)
- ◁ Recall: at most one method of each object can be active at any given time, but two methods of different objects may execute concurrently
- ◁ Beware: for a cyclic sequence of requests, this leads to a deadlock!

Execution model of Timber

- ◁ An action does not return a value so we use it to introduce concurrent execution: the called method can execute concurrently with the caller

- ◁ An action can also be delayed by a certain amount of time

```
after (sec 3) obj.incCounter
```

- ◁ Invoking an action actually return a reference to the message that can be used to abort the action before it has been started

```
m <- after (sec 3) obj.incCounter
...
abort m
```

A POSIX example

```
1 module Test where
2
3 import POSIX
4
5 struct Echo where
6   echoString :: String -> Action
7
8 echo env = class
9   print = action
10  env.stdout.write str
11  echoString str = action
12    after (sec 3) print
13    result Echo{..}
14
15 root :: RootType
16 root w = do
17   env = new posix w
18   echoObj = new echo env
19   env.stdin.installLR echoObj.echoString
```

Pattern matching

- ◁ Patterns are used quite a lot in Timber. They can be used on the left-hand side of any binding, in function arguments, and in "case" commands and expressions. For example:

```
x = Just 3
{- binds the value "Just 3" to the name "x" -}
Just y = x
{- binds the value "Just 3" to the pattern "Just 3", binding the value "3" to the name "y" -}
```

- ◁ When a value, either that of an expression after "=", "new" or "case", or that of an argument to a function, is matched against a pattern, the matching may either succeed or fail:

```
v = [1,2]
x:xs = v -- succeeds
3:xs = v -- fails
```

Pattern matching

- In a binding (“=” and “= new”), a failed pattern matching results in a run-time exception. In a function, the next definition will be tried. In a “case” expression or command, the next branch will be tried.

```
v = [1,2]
3:xs = v    -- runtime exception
```

- If all patterns for which a function or a “case” expression or command is defined fail, this results in a runtime exception.

```
f 1 = 0    -- function definition
f 2 = 1
```

```
cmd = do
  result f 3 -- runtime exception
```

Pattern matching

- Pattern matching of:

- a variable x against a value v always succeeds, binding x to v
- a literal l against a value v succeeds only if v is the value denoted by l ; no variable is bound
- a (data type) constructor pattern $C p_1 \dots p_n$ against a value v succeeds only if $v = C v_1 \dots v_n$ and matching p_i against v_i succeeds for all i
- a struct pattern against a struct value succeeds only if matching succeeds for each selector

Functions and function bindings

- λ -expressions

```
f = \x y -> x+y    -- function binding
5 + (\x -> x^2) 4  -- an anonymous function
```

- Function bindings (incomplete patterns result in run-time exception)

```
zip (x:xs) (y:ys) = (x,y) : zip xs ys
zip _ _ = []
```

- Function bindings with guards (boolean expressions)

```
lookup x [] = Nothing
lookup x ((a,b) : xs)
  | x == a = Just b
  | True   = lookup x xs
```

Functions and function bindings

- Function bindings with additional bindings within guards

```
fcn 0 _ _ = 0
fcn 1 list1 list2
  | x:xs <- list1, y:ys <- list2 = x+y
  | otherwise = -1
```

- Function bindings with a where-clause

```
formatLine n xs = concatMap f xs
  where f x = rJust n (show x)
```

More about functions

- ◁ Higher-order functions can take other functions as arguments:
`applyTwice f a = f (f a)`
- ◁ Functions that take several arguments can be applied to some of them (partial application), producing closures:
`f a b = a+b`
`g = f 3`
is equivalent to
`g b = 3+b`
Note that it is the leftmost arguments that will be filled
- ◁ This is handy when for creating specialized versions of classes, methods, and functions

Recursive bindings

- ◁ A group of bindings is recursive, i.e. we can refer to name defined later on in the group:
`a = new mkA b c`
`b = f a c`
`c = new mkC a`
- ◁ All bindings in a class expression form a single recursive group. In a method or a procedure, these are interrupted by a state variable update, a method or a procedure invocation, "if" and "case" commands
- ◁ The programmer should ensure, however, that the inner structures of the "early" used names are not accessed until they are constructed; for example, trying to access a selector of the struct "c" inside "mkA" or "f" in the example above will result in a runtime exception!

Expressions

- ◁ Remember: Anything to the right of "=", ":", "=", "new", "<-" and "result" is an expression! (examples? exs - legal and illegal examples)
- ◁ Basic expressions:
 - ◁ names, bound at the top level and locally (including parameters and names bound in a where-clause)
`x y myFunction MyModule.f`
 - ◁ state variables
`s x`
 - ◁ literals, including data constructors
`4 3.5 True Nothing`

Expressions

- ◁ Basic expressions:
 - ◁ applications of data constructors
`Just x Celsius 4`
 - ◁ applications of functions and λ -expressions
`elem 3 xs`
`(\x -> x*x) 7`
 - ◁ applications of operators
`x + 3 7+12.3`
 - ◁ λ expressions (anonymous functions)
`\x y -> x+y`

Expressions

Basic expressions:

Tuples

```
(3, 'd', "start")
```

elements of a tuple are accessed using pattern matching:

```
fst (x,_) = x
```

```
snd (_,x) = x
```

Lists

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

Expressions

Basic expressions:

List constructors, e.g. a list of integers from 7 to 12 (not list compr.)

```
[7..12]
```

List comprehensions

- a list of elements x from $[1..100]$ such that $(x \bmod 3 == 0)$

```
[x | x <- [1..100], x `mod` 3 == 0]
```

- a list of pairs (a,b) in all possible combinations, where a is from $[a,b,c]$ and b is from $[1,2]$

```
[(a,b) | a <- [a,b,c], b <- [1,2]]
```

Expressions

Basic expressions:

Newly created arrays

```
array [1,2,3] -- creates an array from a list
```

```
uniarray 10 'a' -- creates an array of length 10 with each element equal to 'a'
```

Array element access (the k -th element of the array "arr")

```
arr!k
```

Note 1. Updating an element of an array uses the same syntax (but only arrays that are state variables are mutable)

```
arr!k := 4
```

Note 2. The current implementation of arrays is temporary. For example, no control is performed on out-of-bounds access.

Expressions

Basic expressions:

Structs. Assuming we have the following struct type definition

```
struct Point where
```

```
  x :: Int
```

```
  y :: Int
```

we can define a struct value in three ways:

```
- Point{x=3, y=4}
```

```
- {x=3, y=4}
```

```
- struct
```

```
  x = 3
```

```
  y = 4
```

The latter allows to define a function binding spanning several lines (with pattern matching) and the bindings can be recursive

Expressions

- Basic expressions:
 - Note also the notation that denotes automatic completion of a struct using names "in scope" (locally bound names):

```
x = 4
y = 5
Point{..}      -- stands for {x = x, y = y}
Point{x = 3,..} -- stands for {x = 3, y = y}
```
 - A struct member access ("." + selector name)

```
p.x
```

Expressions

- "let" expressions (may include recursive definitions)

```
let size = f 100
    f 0 = 0
    f x = g x (x+2)
in concat (map f xs)
```
- "if" expressions (the "else" branch must be present)

```
if x == 3 then "Yes" else "No"
```
- "case" expressions (incomplete patterns result in run-time exception)

```
case f a b of
[] -> 0
x:xs -> g x + h 3 xs
```

Expressions

- Class expressions
- Method (action and request) expressions
- Procedure expressions

Once again...

- Expressions are evaluated without any side-effects
- Class expressions after "= new" create new objects
- Method and procedure expressions after "<-" and in a standalone position are executed and may produce side-effects
 - "<-" and standalone method and procedure expressions are only allowed within other methods and procedures

Extract from POSIX.t

```
module POSIX where

type RootType = World -> Cmd () ()

struct Env where
  exit      :: Int -> Request ()
  {- command-line arguments -}
  argv     :: Array String
  stdin    :: RFile
  stdout   :: WFile
  {- opens a file for reading -}
  openR    :: String -> Request (Maybe RFile)
  {- opens a file for writing -}
  openW    :: String -> Request (Maybe WFile)
  ...
```

Extract from POSIX.t

```
struct RFile < File where
  {- reads all available data -}
  read      :: Request String
  {- installs a handler of type String -> Action -}
  installR  :: (String -> Action) -> Request ()

struct WFile < File where
  {- writes a String to the file or output stream -}
  write     :: String -> Request Int
  ...
```

A Timber example

```
1 struct S where
2   inc :: Int -> Action
3   get :: Request Int
4
5 makeS p = class
6   state := p
7   inc a = action
8   state := state + a
9   get = request
10  result state
11  result S{..}
12
13 root w = do
14   obj = new makeS 10
15   obj.inc 7
16   value <- obj.get
```