Functional Programming

Loic, Niragire

NorthCentral University

TIM-8110: Programming Languages & Algorithms

Dr. Phillip Davis

Table of Contents

1. Introduction

This article explores the declarative computational model with an emphasis on functional programming. The single-assignment store, namely *value-store,* is the declarative model's main distinguishing attribute. One implication of this is that once a variable is bound to a value, it remains bound throughout the computation. On the contrary, computational models that support *cell-store* allow bound variables reassignment. Single-assignment store facilitates declarative concurrency and is needed for relational programming and constraint programming (Roy & Haridi, 2004). Proceeding sections dive into functional programming features and techniques. We provide an implementation comparison between Haskell and Scheme.

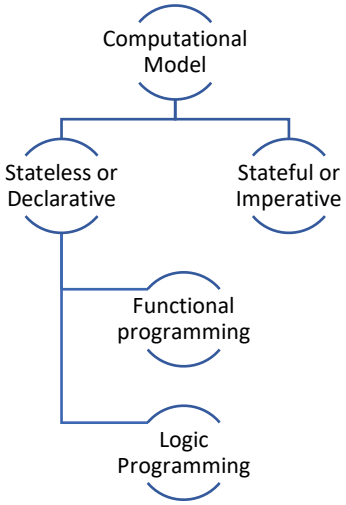

Figure 1 Computational models

We need to discuss three housekeeping items before diving further into features and techniques: the concept of "partial value," lambda calculus, and higher-order functions. A data structure containing unbound variables is referred to as a partial value from a definition standpoint. For instance, schedule record: *schedule (time, location: 'office')* is a partial value as it contains the unbound variable' *time.'* We can create a *complete value* by bounding all contained variables, e.g., *schedule (time:' 2:00', location: 'office').* Functional languages compute functions on values, partial or complete.

Alonzo Church introduced lambda calculus in the 1930s as a formal system of expressing computation based on functions. Suppose we have the following mathematical function $f(x) = 2 + x$. Its lambda calculus expression can be obtained by grouping all variables to the equation's left side $\lambda x.\ 2 + x$. The x before the dot is the function's argument and everything to the right is the function's expression. Notice also the usage of λ to denote our function. All lambda calculus functions have exactly one argument. To express multi-argument functions, we apply the principle of β-reduction to reduce terms (Stanford University - Center for the Study of Language and Information , 2018).

A higher-order function is a function that either accepts functions as input arguments or returns a function. Therefore, a function of multiple arguments is a higher-order function in λ-calculus. As an example, $f(x, y) = x + y$ can be expressed as a function that takes x as its input argument and returns a function of y. Thus, $f(x, y) = x + y$ is expressed as $\lambda x.\lambda y\ x + y$. Computing this function for $x = 10$, *i.e* $f(10, y) = 10 + y$, results in $(\lambda x.\lambda y\ x + y)\ 10$, which yields $\lambda y.\ y + 10$. By substituting x for 10 in the expression body. Functional programming consists of applying functions to expressions.

2. Problem domain

In this section, we examine key features that make the declarative computational model compelling. Parallel programming is a particular domain where declarative models provide significant leverage over imperative models. The current approach to parallel programming in imperative languages relies on *locks* and *condition variables*, resulting in programs prone to deadlocks (Microsoft Research, 2007). Declarative programs benefit from implicit parallelism, and the more declarative the language is, the more implicit parallelism there is to gain. Therefore, the system alone determines the most efficient way to run the program (Lloyd, 1994).

3. Haskell

Haskell is a statically typed functional programming language designed to handle symbolic expressions and list processing applications. Also, it is considered purely functional and supports lazy evaluation. By purely functional we mean, there are no statements or instructions, only expressions that cannot mutate variables – local or global. Its compiler, GHC, supports type inference and comes with a high-performance parallel garbage collector and a light-weight concurrency library (Haskell Organization, 2014-2021).

Figure 2 below contains a list of key features we will be exploring further with examples.

Features

1	User-defined types	Using *data* declaration
2	Polymorphic types	Functions that can be applied to arbitrary types.
3	Recursive types	Value types can be recursively defined
4	Type synonyms	Names for commonly used types. Using *type* declaration
5	List comprehensions	A concise syntax for iterations of lists
6	Lazy evaluation	Functions support undefined parameters
7	Higher-order functions	Functions are first-class. They can be passes are parameters and/or returned as results.
8	Pattern matching	The primary way that elements of a datatype are distinguished
9	Data abstraction	Hiding implementations from the rest of the program
10	Modules	Programs consist of a collection of modules

Figure 2 Highlighted Haskell features

3.1. User-defined types

```
-- A user-defined type representing a bank account.
-- BankAccount is assigned a value constructor, Account, which accepts
-- two parameters of string and double type.
-- Here we are simply representing an account owner and account balance.
data BankAccount = BankAccount {owner :: String, balance::Double}

-- updateBalance takes three parameters and returns a BankAccount value type
-- String: owner
-- Double: current balance
-- Double: value to update the account by
updateBalance :: String -> Double -> Double -> BankAccount
updateBalance owner currentBalance value =  BankAccount owner (currentBalance + value)

-- Let's create a bank account with balance of 10.0
-- Note that this creates a value type on the stack
loicAccount = BankAccount "Loic" 10.0

-- we can qury the balance on the account above:
currentBalance = (balance loicAccount) -- this should return 10.0

-- Now let’s update the account balance by adding $20 to the account.
-- For this we call updateBalance function, which should return a new BankAccount
updatedAccount = updateBalance
            (owner loicAccount)       -- owner of the BankAccount to be updated
            (balance loicAccount)     -- current balance on the account to be updated
             20                       -- amount to update account by


updatedBalance = (balance updatedAccount)   -- this should return 30.0
```

3.2. Polymorphic types

Haskell supports two types of polymorphism: parametric polymorphism and bounded polymorphism (also known as Ad-hoc).

```
-- Parametric polymorphism:
-- A value is polymorphic if there is more than one type it can have
-- Here we create a polymorphic BankAccount where accountBalance can be of any type.
-- Our earlier BankAccount type had a balance of type Double
data BankAccountOf a = BankAccountOf {ownedBy :: String, accountBalance:: a}

-- We now can create a BankAccountOf Double, Float, Integer...
bankAccount = BankAccountOf "Loic" 10
anotherBankAccount = BankAccountOf "Loic" 20.0
```

3.3. Recursive types

```
-- Here we are defining a polymorphic value type called 'BankAccounts'
-- It can either be an enpty tree or a Node of 'BankAccounts'
data BankAccounts a = EmptyTree | Node a (BankAccounts a) (BankAccounts a)
```

3.4. Type synonyms

```
-- Type synonyms improves readability by creating synonyms of existing types
-- to best match our application
type CurrentBalance = Double
type ValueToUpdateBy = Double
type AccountOwner = String

-- Now we can implement UpdataBalance function using created synonyms above
cleanerUpdateBalance :: AccountOwner -> CurrentBalance -> ValueToUpdateBy -> BankAccount
cleanerUpdateBalance owner currentBalance value = BankAccount owner (currentBalance + value)
```

3.5. List comprehensions

```
-- We can return a list of elements created by evaluation of some generator
-- Here we are saying, give me a list of elements such that each element is less than three,
-- where x is an element of [1,2,3,4,5,6]
vt = [x < 3 | x <- [1,2,3,4,5,6]]
-- this should return [True,True,False,False,False,False]
```

3.6. Lazy evaluation

```
-- Parameters are not evaluated unless their results are needed
-- In this example we create a function 'add' that takes three integers and returns an integer
-- Note that the last parameter 'z' is never used in the function body
add :: Double -> Double -> Double -> Double
add x y z = x + y

-- As a consequence, we can call 'add' as:
sum = add 1 2 (1/0)
-- Since parameters are never evaluated unless needed,
-- we do not get an infinity or divide by zero exception
-- sum returns 3
```

3.7. Higher-order functions

```
-- Functions in Haskell can take other functions as parameter or return functions.
-- Here we are declaring a 'withdraw' function that takes the following parameters:
-- A function that takes a 'BankAccount' and returns a Double. Responsible for returning
account balance
-- A BankAccount to withdraw from
-- A Double representing the amount to withdraw
withdraw::(BankAccount -> Double) -> BankAccount -> Double -> BankAccount
withdraw f account amount = BankAccount (owner account) ((f account) - amount)


-- Here we declare a simple function that takes a BankAccount and returns its balance
getBalance :: BankAccount -> Double
getBalance account = balance account

-- Now we can call 'withdraw' function by passing it 'getBalance' as:
accountWithdraw = withdraw getBalance loicAccount 4.0
--print (balance accountWithdraw) --should return 6.0
```

3.8. Pattern matching

```
-- Suppose we want to log different messages based on deposited amount in our bank model.
-- We can apply pattern matching and guards to match against transaction amount.
-- This is a top-down matching pattern where the first matching condition is executed.
logMessage :: Double -> String
logMessage i
        | i > -500    = "Small withdraw"
        | i > -1000   = "Medium withdraw"
        | i > -10000  = "Large withdraw"
        | i < 0       = "Withdraw request"
        | i <= 100    = "Small Deposit"
        | i <= 5000   = "Medium Deposit"
        | otherwise   = "Large Deposit"

-- usage
-- print (logMessage 100.0)
-- print (logMessage (-3000))
```

3.9. Data abstraction

```
-- We can define reusable data types that leave some implemetation details undefined
data Tree a = Nil
        | Node { left  :: Tree a,
                 value :: a,
                 right :: Tree a
               }

-- With the above definition, we can various bank accounts
checkingAccount = BankAccount "Loic" 100
savingsAccount = BankAccount "Loic" 200
retirementAccount = BankAccount "Loic" 30

bankAccountAccounts :: Tree BankAccount
bankAccountAccounts = Node (Node Nil savingsAccount Nil) checkingAccount (Node Nil retirementAccount Nil)
```

3.10. Haskell sample code

```
-- Author: Loic Niragire
-- Banking Module
-- TIM8110 - Programming Languages and Algorithms


module Banking where

  -- Defines a few synonyms used in this module
  type ValueToUpdateBy = Double
  type AccountOwner = String
  type CurrentBalance = Double

  -- Abstract Data Type
  data Tree a = Nil
              | Node { left  :: Tree a,
                       value :: a,
                       right :: Tree a
                     }

  -- Declares a BankAccount type
  data BankAccount = BankAccount {  owner :: String,
                                    balance::Double
                                 }

  -- Declares a polymorphic bank account
  data BankAccountOf a = BankAccountOf {  ownedBy :: String,
                                          accountBalance:: a
                                       }

  -- Declares a function that updates account balance
  updateBalance :: String -> Double -> Double -> BankAccount

  -- Declares a function that updates account balance using synonyms
  cleanerUpdateBalance :: AccountOwner -> CurrentBalance -> ValueToUpdateBy -> BankAccount

  -- Declares a function that performs a withdraw transaction
  withdraw::(BankAccount -> Double) -> BankAccount -> Double -> BankAccount

  -- Declares a function that retrieves an account balance
  getBalance :: BankAccount -> Double

  -- Declares a function that generates logging messages based on transaction amount
  logMessage :: Double -> String

  -- Implementations
  updateBalance owner currentBalance value =  BankAccount owner (currentBalance + value)

  cleanerUpdateBalance owner currentBalance value = BankAccount owner (currentBalance + value)

  withdraw f account amount = BankAccount (owner account) ((f account) - amount)

  getBalance account = balance account

  logMessage i
           | i > -500    = "Small withdraw"
           | i > -1000   = "Medium withdraw"
           | i > -10000  = "Large withdraw"
           | i < 0       = "Withdraw request"
           | i <= 100    = "Small Deposit"
           | i <= 5000   = "Medium Deposit"
           | otherwise   = "Large Deposit"
```

4. Scheme

Scheme is a strict dynamically typed functional programming language. Variable types are only known at runtime, and function arguments are evaluated before executing the function body. We explore *Chez Scheme*, which is a dialect and implementation of Scheme. Compilation of source files, *".ss"* extension, helps speed up program execution. We do this by issuing a *(compile-file "file path")* command at the command prompt, which generates an object file with a *".so"* extension. Additional compiling options provided include compile-program, compile-script, and compile-library. Issuing a *(load "path_to_object_file")* command at the prompt is essentially the same as loading the source file, except it's much faster since it's been pre-compiled. Users can generate one concatenated object file using the *"concatenate-object-file"* procedure if the runtime needs multiple object files.

Chez Scheme provides a fully compiled runtime environment, *Petite Chez*, for application distribution. Petite Chez is itself a Scheme compiled application that uses a preprocessor to convert the code into a form that can efficiently be interpreted (Cisco Systems, Inc, 2020). To distribute an application, we ship the runtime environment and application-specific libraries.

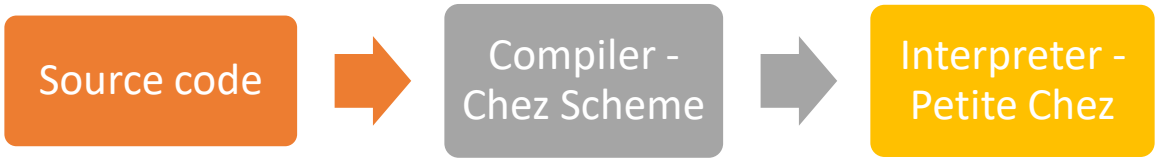

Figure 3 Chez Scheme execution

The command prompt's interaction with Chez Scheme is done through a Scheme program called a *"Waiter,"* which runs in an environment called a *"Café."* Users can request a new environment by issuing a *(new-cafe)* command at the command prompt.

4.1. Scheme sample code

```
;; Loic Niragire
;; TIM8110 - Programming languages and algorithms

;; ********* User-defined types ********************************************************************************
;; User-defined types are created using 'define-record-type'
;; We can declare a bankAccount type with two fields; owner and balance as:
;; By default fields are immutable, by we can specify 'mutable' to change default behavior.
;; Note also that we can explicitly mark fields are 'immutable'
(define-record-type bankAccount (fields owner (mutable balance)))
;; define-record-type implicitly defines a constructor, predicate, and accessor procedures for bankAccount
;; we can generate a bankAccount type by issuing
;; (define myAccount (make-bankAccount "Loic" 10.0))
;; which declares a variable, myAccount, and initializes it with the output of (make-bankAccount "Loic" 10.0)
;; make-bankAccount is an auto-generated constructor for the bankAccount type.
;; For each field marked as 'mutable', the compiler generates a corresponding procedure to alter them.
;; By convention, the generated procedure is of the form
;; <typename>-<fieldname>-set! <variable> <value>

;; ********* Field accessor procedures ********************************************************************************
;; Field accessor procedures are also auto-generated by define-record-type
;; To retrieve the 'owner' field
;; Assuming we defined our type as: (define myAccount (make-bankAccount "Loic" 10.0))
;; We issue:
;; (bankAccount-owner myAccount)

;; To assert whether 'myAccount' is of type 'bankAccount'
;; We call the auto generated predicate as
;; (bankAccount? myAccount)
;; Note that true and false are represented as #t and #f respectively.


;; Deposits amount into the given account.
(define deposit
  ;; chez scheme is dynamically typed, so actual types for 'account' and 'amount' is determined at runtime
  (lambda (account amount)
            ;; balance is mutable so we can change its value
            (bankAccount-balance-set! account (+ (bankAccount-balance account) amount))))


;; Withdraws amount from an account
(define withdraw
  (lambda (account amount)
            ;; make sure current balance is greater than amount to withdraw
            (if (< (bankAccount-balance account) amount)
              (display "Insufficient balance\n")
              (bankAccount-balance-set! account (- (bankAccount-balance account) amount)))))


;; ********* Inheritance support ********************************************************************************
;; Record types can inherit fields from other defined types.
;; To prevent inheritance, record type has to be marked as 'sealed'
;; Here we are defining a retirementAccount as a bankAccount which has a retirementBalance
(define-record-type retirementAccount
  (parent bankAccount)
  (fields (mutable retirementBalance)))

;; we can create a retirementAccount as:
;; (define retirement (make-retirementAccount "Loic" 120.0 400.0))
;; Note that generated default constructor expects all fields from parent type, plus additional fields
;; defined by the child type.
;; Note that to retrieve a field value inherited from a parent type, we have to reference the parent type.
;; (define retirement (make-retirementAccount "Loic" 10.0 48.0))
;; (retirementAccount-balance retirement)          -> fails.
;; (bankAccount-balance retirement)                -> Correct way to retrieve fields from parent type
```

5. Reference List

Cisco Systems, Inc. (2020, 8). Retrieved from Chez Scheme version 9 - User's Guid: https://cisco.github.io/ChezScheme/csug9.5/csug9_5.pdf

Haskell Organization. (2014-2021). Haskell. Retrieved from Haskell.org: https://www.haskell.org

Lloyd, J. (1994). Practical Advantages of Declarative Programming. Joint Conference on Declarative Programming (pp. 3-17). Peñiscola: ResearchGate.

Microsoft Research. (2007, May 1). Beautiful concurrency. Retrieved from https://homepages.cwi.nl/~storm/teaching/reader/PeytonJones07.pdf

Roy , P. V., & Haridi, S. (2004). Concepts, Techniques, and Models of Computer Programming. In P. V. Roy, & S. Haridi, Concepts, Techniques, and Models of Computer Programming (pp. 42-54). MIT Press.

Stanford University - Center for the Study of Language and Information . (2018). Stanford Encyclopedia of Philosophy. Retrieved from The Lambda Calculus: https://plato.stanford.edu/entries/lambda-calculus/