Assignment vs. Evaluation

nprenet@bsu.edu

CS431 - Spring 2022

Introduction

How do we structure programs so that they can be *modular*? A natural strategy is to model the structure of our program on the structure of the system being modeled: object → object, action → operation.

Two approaches:

- focusing on **objects**, a collection of distinct entities whose behavior may change over time
- focusing on **streams** of informations that flow into the system

These lecture notes are mostly based on Chapter 3 of Abelson & Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, 1996 (also known to the CS community as the *SICP* book).[1] Some examples have been transposed from Scheme to Javascript (Assignment section) or Haskell (Streams section) and/or been modified; other examples have been added.

Assignment and local state

Objects with local state

The world seen as populated by independent objects, each of which has a **state** that changes over time.

Ex. Bank account, where the answer to the question "Can I withdraw $100?" depends on its history.

State variables allow us to maintain information about history, and determine behavior.
To maintain state variables, we need an **assignment operator**. Haskell does not provide an assignment operator![2] For this reason, we use Javascript to show how assignment allows for functional programs to model change:

E.g. Observe the repeated calls on the `withdraw()` function:

```
withdraw = makeWithdraw()

console.log( withdraw(600) ) // 300
console.log( withdraw(300) ) // 100
console.log( withdraw(300) ) // "Insufficient fund"
```

Clearly, this is not a (mathematical) function: called twice with the same argument (300), it returns different values. The `withdraw` function maintains a hidden variable `balance`, whose value is modified with each call to the function:

A stateful function

```
function makeWithdraw (){
```

[1] I warmly recommend you read this book after taking this class: it will make an enlightened programmer of you.

[2] Monads allow for maintaining state in Haskell, but the topic is beyond the syllabus for this course.

```
   var balance = 1000;

   return function (amount){
      if (balance >= amount){
         balance = balance - amount;
         return balance;
      }
      else {
         console.log("Insufficient funds");
         return -1;
      }
   }
}
```

Note that:

- `balance` is a name defined in the local environment, but is not accessible by any other procedure;
- the example illustrates a common programming technique for constructing computational objects with local state.

A parameterized stateful function

We can improve and make our function more modular, by passing the balance as a parameter:

```
function makeWithdraw (balance){

   return function (amount){
      if (balance >= amount){
         balance = balance - amount;
         return balance;
      }
      else {
         console.log("Insufficient funds");
         return -1;
      }
   }
}
makeWithDraw(1000)(300)
700
```

In the function's body, the formal parameter's name (`balance`) is bound to the balance's value: the effect is the same as the one obtained through the explicit creation of a local variable.

A crude bank account object

The following program improves the modularity of our function by adding two functionalities:

- a `deposit` operation
- a front-end interface, that hides the two functions that actually modify the state

```
function makeAccount ( balance ){

   function withdraw( amount ){
      if (balance >= amount){
         balance = balance - amount;
         return balance;
      }
      else { console.log("Insufficient funds") }
```

```
            return balance;
    }

    function deposit( amount ){
        balance = balance + amount;
        return balance;
    }

    function dispatch( op ){
        if (op == "withdraw"){ return withdraw }
        else if (op == "deposit"){ return deposit }
    }
    return dispatch
}
```

Only the `dispatch` function is exposed: it accepts messages (`"deposit"` or `"widthdraw"`), and then delegates the actual operation to the proper internal subroutine.

```
> account = makeAccount(1000);
> account("deposit")(200)
1200
> account("withdraw")(700)
500
```

This **message-passing style** (that more commonly uses the dot syntax, as in `account.withdraw()`) is one of the few defining features of object-oriented languages. Note that the example above implements it with functional patterns—`makeAccount` is not a proper object, but a function.

Assignment: benefits and pitfalls

Benefits

Powerful technique to maintain modular design. From the point of view of one part of the system, other parts appear to change with time, with hidden variables: easier than by passing parameters explicitly.

- the patterns shown above are the building blocks for **abstract data structures (ADT)**, that encapsulates data behind a well-defined interface: we can use them to define mutable data structures such as lists, tables, . . .

Costs

The **substitution model for evaluation** (whose pure form is given by lambda-calculus) **does not work anymore**. Procedures are no longer mathematical functions, that given an input, always return the same value:

```
withdraw1 = makeWithdraw( 1000 )
withdraw2 = makeWithdraw( 1000 )

console.log( withdraw1(100) )
console.log( withdraw1(100) )
console.log( withdraw1(50) )
console.log( withdraw1(800) )

console.log( withdraw2(50))
```

Compare with the following function, that, given a specific input value, always return the same output:

```
> d = makeDecrementer( 25 )
15
> d(20)
5
> d(25)
0
```

Exercise: Try to apply the substitution model to the evaluation of expression `makeWithdraw(1000)(300)` [*Hint: replace every occurrence of the* ***`balance`*** *keyword with the value 1000, and every occurrence of the* ***`amount`*** *name with the value 300. Can you then evaluate?*]

- substitution model: names are just values
- assigment model: names are places for values

Assignment comes with different notions of sameness and change: a language that supports "equals can be substituted for equals" in an expression without changing its value is said to be *referentially transparent*. To determine sameness here, instead, we have to observe the effects of change. It makes it much harder to reason about programs. For example, some bugs are specific to imperative programming: we have to decide of the order of assignments (different from declarative style), because each statement has to use the correct version of variables that have changed over time. Can you spot the bug in this function?

```
function buggyFactorial( n ){
   var product = 1
   var counter = 1

   while( counter <= n){
      counter = counter + 1
      product = product * counter
   }
   return product
}
```

Compare with the following Haskell function by Alyssa, that computes an arithmetic value:

```
func x = foo i j k
  where
    foo i j k = i + j + k
    j = k^2
    i = j / 3
    k = x + 1
```

Ben believes there is bug: because the last line (`k = (x + 1)`) defines variable `k`, it should be written *before* the definition of `j`, that has `k` as an input value. Alyssa tells him that it does not matter. What do you think?

Answer: Alyssa is correct. The `where` clause is not made of assignments, but of *definitions*. Time is not involved in definitions: because of referential transparency, every definition holds at the same time, no matter its location in the `where` clause. Within a given scope, every occurrence of a given name evaluates to the same value.[3]

Streams

Streams offer an alternative way to model time in a program. If we concentrate on the entire time history of values, the function itself does not change! Stream processing lets us model systems that have state without ever using assignment or mutable data.

[3] If that helps, you may think of Haskell bindings the way of definitions in a dictionary: the fact that they are ordered is a convenience—it makes it easier to access a given entry—, but it is not a condition of the dictionary's consistency.

Implementing streams

Streams are delayed lists: although we write programs as if we were processing complete sequences, we implement streams by automatically and transparently interleaving the *construction* of a list with its *use*. Thus we can dispense with manipulating and copying around huge lists.

- In a language that relies on strict evaluation[4], such as Scheme, the trick is to delay the evaluation of the list constructor, such that the function

  ```
  (cons x xs)             ; Lisp-speak for (x:xs)
  ```

  only evaluates its second operand (`xs`) on a as-needed basis: then it forces the evaluation of the first element of the tail, and so on.[5]
- In Haskell, streams are built into the language, because the delayed evaluation of the arguments (aka. **normal order** evaluation) is the default. No specific step is necessary to work with streams in Haskell. For instance, the following function generates an infinite list of integers:

```
generate n = n : generate (n+1)
```

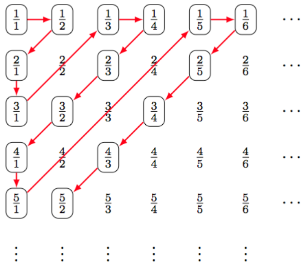
Haskell does not evaluate the tail of the list at list construction time: only the head of the list (`n`) is actually evaluated. Of course, one can always force the evaluation of any finite sublist:

```
Prelude> take 2 $ drop 1 $ generate 1
[2,3]
```

Examples

Example 1. Listing all pairs of integers

Enumerating rational numbers systematically is possible, as shown below:



The following program may be used for that purpose. It generates an infinite list of pairs of positive integers:

```
pairs :: (Fractional a, Eq a) => [(a, a)]$
pairs = generate 1 1 $
 where generate a b$
```

[4] aka. **applicative order** evaluation, where arguments of a function are evaluated before the function is applied to them.

[5] Implementing such a delay is relatively straightforward: SICP 3.5 shows how to do it at language-level. Building upon a handful of Lisp-style list procedures (`cons`, `car`, `cdr`), a Javascript emulation of this technique may look like this:

```
/*
 * Streams
 */
function force( proc ){ return proc(); }

// parameter s is a lambda expression, not a value --> delayed evaluation
function consStream(x, s){ return cons( x, s ) }
function streamCar( s ){ return car(s) }
function streamCdr( s ){ return force( cdr( s )) }

// Example:  infinite list
function integersFrom(n){ return st.consStream( n, () => integersFrom( n+1 )) } }
```

```
          | b == 1 = (a,b) : generate 1 (a+1)$
          | otherwise = (a,b) : generate (a+1) (b-1)$
```

Obtaining a unique instance of each rational value would require some filtering, of course.

Example 2. Approximating π

We can approximate π with the following series:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots$$

Here is how we generate a stream of values that converge (rather slowly) to π's value:

```
piSummands n = (1.0 / n) : (map negate (piSummands (n + 2)))

partialSums l@(x:xs) = x : sumLists xs (partialSums l)
 where sumLists (x:xs) (y:ys) = (x+y) : sumLists xs ys

piStream = map (*4) $ partialSums $ piSummands 1

Prelude> piStream
[4.0,2.666666666666667,3.466666666666667,2.8952380952380956,3.3396825396825403,...]
```

The examples above afford us to glimpse at the possibilities offered by streams: the ability to discretize time, and to build arbitrary sets of data based on it is precious for simulating and testing complex systems.

Conclusion: functional vs. imperative view of time

With assignment, we can model the temporal behavior of the objects in the world by the temporal behavior of stateful computational objects. Streams provide an alternative way to model objects with time: a changing quantity is modeled by a stream that represents the time history of successive states: we decouple time in our simulated world from the sequence of eventgs that take place during evaluation. Actually, due to delayed (lazy) evaluation, there is little relation between the simulated time series, and the order of computational events during the evaluation.

There is an important word of caution regarding the two approaches: **neither one is a magic bullet for solving the complex issues that come with concurrency**. This latter topic deserves its own lecture. If you are impatient, you can read more about the problem in SICP, 3.4 (`https://mitpress.mit.edu/sites/default/files/sicp/full-text/book/book-Z-H-23.html#%_sec_3.4`).