how a pure functional programming language manage without assignment statements?

When reading the famous SICP, I found the authors seem rather reluctant to introduce the assignment statement to Scheme in Chapter 3. I read the text and kind of understand why they feel so.

As Scheme is the first functional programming language I ever know something about, I am kind of surprised that there are some functional programming languages (not Scheme of course) can do without assignments.

Let use the example the book offers,the `bank account` example. If there is no assignment statement, how can this be done?How to change the `balance` variable? I ask so because I know there are some so-called pure functional languages out there and according to the Turing complete theory, this must can be done too.

I learned C, Java, Python and use assignments a lot in every program I wrote. So it's really an eye-opening experience. I really hope someone can briefly explain how assignments are avoided in those functional programming languages and what profound impact (if any) it has on these languages.

The example mentioned above is here:

```
(define (make-withdraw balance)
    (lambda (amount)
        (if (>= balance amount)
            (begin (set! balance (- balance amount))
                balance)
            "Insufficient funds")))
```

This changed the `balance` by `set!` . To me it looks a lot like a class method to change the class member `balance` .

As I said, I am not familiar with functional programming languages, so if I said something wrong about them, feel free to point out.

programming-languages functional-programming scheme

edited May 1 '15 at 13:03
Robert Harvey
142k 34 313 512

asked Apr 12 '12 at 4:57
Gnijuohz
1,035 2 14 17

1 Regarding learning a purely functional language: I wouldn't necessarily recommend doing that right away. If you learn Haskell then in addition to learning how to write programs without mutable variables, you'll also have to learn about laziness and Haskell's way to perform IO. That might be a bit much all at once. If you want to learn to write programs without mutable state, the easiest way would probably be to just write a bunch of scheme programs without using `set!` or other functions that end with a `!` . Once you're comfortable with that, the transition to pure FP should be easier. – sepp2k Apr 13 '12 at 13:48

6 Answers

> If there is no assignment statement,how can this be done?How to change the balance variable?

You can't change variables without some sort of assignment operator.

> I ask so because I know there are some so-called pure functional languages out there and according to the Turing complete theory,this must can be done too.

Not quite. If a language is Turing complete that means that it can calculate anything that the any other Turing complete language can calculate. It doesn't mean that it has to have every feature other languages have.

It's not a contradiction that a Turing complete programming language has no way of changing the value of a variable, as long as for every program that has mutable variables, you can write an equivalent program that does not have mutable variables (where "equivalent" means that it calculates the same thing). And in fact every program can be written that way.

Regarding your example: In a purely functional language you simply wouldn't be able to write a function that returns a different account balance each time it's called. But you'd still be able to rewrite every program, that uses such a function, in a different way.

Since you asked for an example, let's consider an imperative program that uses your make-withdraw function (in pseudo-code). This program allows the user to withdraw from an account, deposit to it or query the amount of money in the account:

```
account = make-withdraw(0)
ask for input until the user enters "quit"
    if the user entered "withdraw $x"
        account(x)
    if the user entered "deposit $x"
        account(-x)
    if the user entered "query"
        print("The balance of the account is " + account(0))
```

Here's a way to write the same program without using mutable-variables (I won't bother with referentially transparent IO because the question wasn't about that):

```
function IO_loop(balance):
    ask for input
    if the user entered "withdraw $x"
        IO_loop(balance - x)
    if the user entered "deposit $x"
        IO_loop(balance + x)
    if the user entered "query"
        print("The balance of the account is " + balance)
        IO_loop(balance)
    if the user entered "quit"
        do nothing

 IO_loop(0)
```

The same function could also be written without using recursion by using a fold over the user input (which would be more idiomatic than explicit recursion), but I don't know whether you're familiar with folds yet, so I wrote it in a way that doesn't use anything you don't know yet.

edited Apr 12 '12 at 15:14

answered Apr 12 '12 at 5:58
sepp2k
3,889 1 15 24

I can see your point but let see I want to a program which also simulate the bank account and can also do these things(withdraw and deposit),then is there any easy way to do this? – Gnijuohz Apr 12 '12 at 7:08

@Gnijuohz It always depends on what problem exactly you're trying to solve. For example if you have a starting balance and a list of withdrawals and deposits and you want to know the balance after those withdrawals and deposits, you can simply calculate the sum of the deposits minus the sum of the withdrawals and add that to the starting balance. So in code that would be `newBalance = startingBalance + sum(deposits) - sum(withdrawals)` . – sepp2k Apr 12 '12 at 7:22

1 @Gnijuohz I've added an example program to my answer. – sepp2k Apr 12 '12 at 15:14

Thanks for the time and efforts you put into writing and rewriting the answer! :) – Gnijuohz Apr 13 '12 at 6:37

I would add that using continuation could also be a mean to achieve that in scheme (as long as you can pass an argument to the continuation ?) – dader51 May 19 '13 at 17:18

You're right that it looks a lot like a method on an object. That's because that's essentially what it is. The `lambda` function is a closure that pulls the external variable `balance` into its scope. Having multiple closures that close over the same external variable(s) and having multiple methods on the same object are two different abstractions for doing the exact same thing, and either one can be implemented in terms of the other if you understand both paradigms.

The way pure functional languages handle state is by cheating. For example, in Haskell if you want to read input from an external source, (which is nondeterministic, of course, and will not necessarily give the same result twice if you repeat it,) it uses a monad trick to say "we've got this other pretend variable that represents *the state of the entire rest of the world*, and we can't examine it directly, but reading input is a pure function that takes the state of the outside world and returns the deterministic input that that exact state will always render, plus the new state of the outside world." (That's a simplified explanation, of course. Reading up on the way it actually works will seriously break your brain.)

Or in the case of your bank account problem, instead of assigning a new value to the variable, it can return the new value as the function result, and then the caller has to deal with it in a functional style, generally by recreating any data that references that value with a new version containing the updated value. (This is not as bulky an operation as it might sound if your data is set up with the right sort of tree structure.)

answered Apr 12 '12 at 6:03
Mason Wheeler
69.1k 16 198 280

I am really interested in our answer and the example of Haskell but due to lack of knowledge about it I can't fully understand the last part of your answer(well,also the second part :() – Gnijuohz Apr 12 '12 at 7:12

3 @Gnijuohz The last paragraph is saying that instead of `b = makeWithdraw(42); b(1); b(2); b(3); print(b(4))` you can just do `b = 42; b1 = withdraw(b1, 1); b2 = withdraw(b1, 2); b3 = withdraw(b2, 3); print(withdraw(b3, 4));` where `withdraw` is simply defined as `withdraw(balance, amount) = balance - amount` . – sepp2k Apr 12 '12 at 7:19

"Multiple-assignment operators" is one example of a language feature which, generally speaking, has side effects, and is incompatible with some useful properties of functional languages (such as lazy-evaluation).

That, however, doesn't mean that assignment in general is incompatible with a pure functional programming style (see this discussion for example), nor does it mean that you can't construct syntax which allows actions that look like assignments in general, but are implemented without side effects. Creating that kind of syntax, and writing efficient programs in it, is time consuming and hard, though.

In your specific example, you're right - the set! operator *is* an assignment. It's *not* a side-effect free operator, and it's a place where Scheme breaks with a purely functional approach to programming.

Ultimately, any purely functional language is going to have to break with the purely functional approach sometime - the vast majority of useful programs *do* have side effects. The decision of where to do it is usually a matter of convenience, and language designers will try to give the programmer the highest flexibility in deciding where to break with a purely functional approach, as appropriate for their program and problem domain.

answered Apr 12 '12 at 5:48
blueberryfields
9,448 6 40 76

"Ultimately, any purely functional language is going to have to break with the purely functional approach sometime - the vast majority of useful programs do have side effects" True, but then you're talking about doing IO and such. Plenty of useful programs can be written without mutable variables. – sepp2k Apr 12 '12 at 5:50

1 ...and by "the vast majority" of useful programs, you mean "all", right? I'm having difficulty even imagining the possibility of the existence of any program that could be reasonably called "useful" that does not perform I/O, an act which requires side effects in both directions. – Mason Wheeler Apr 12 '12 at 6:05

@MasonWheeler SQL programs don't do IO as such. It's also not uncommon to write a bunch of functions that don't do IO in a language that has a REPL and then simply calling those from a REPL. This can be perfectly useful if your target audience is capable of using the REPL (especially if your target audience is you). – sepp2k Apr 12 '12 at 6:10

1 @MasonWheeler: just a single, simple counter-example: conceptually calculating *n* digits of pi doesn't require any I/O. It's "only" math and variables. The only required input is *n* and the return value is Pi (to *n* digits). – Joachim Sauer Apr 12 '12 at 8:28

1 @Joachim Sauer eventually you'll want to print the result to the screen, or otherwise report on it to the user. And initially you'll want to load some constants into the program from somewhere. So, if you want to be pedantic, *all* useful programs have to do IO at some point, even if it's trivial cases that are implicit and always hidden from the programmer by the environment – blueberryfields Apr 14 '12 at 15:32

@blueberryfields: but that's the point: the IO does not need to be part of the functional program: If the IO is externalized to the runtime system (which need not be a purely functional system), then the actual user-written program needs no IO at all. – Joachim Sauer Apr 15 '12 at 8:39

As far as I know "purity" does not refer to the absence of side-effects but to referential transparency. For example, in Haskell IO functions produce values that are actions. In this sense, IO functions are referentially transparent because they always produce the same actions given the same inputs. On the other hand, side effects are the result of executing those actions, and an action can have different side effects depending on the environment. But actions are executed outside the realm of functions and function application, therefore functions are still pure. – Giorgio Jun 9 '13 at 21:07

In a purely functional language, one would program a bank account object as a stream transformer function. The object is regarded as a function from an infinite stream of requests from the account owners (or whoever) to a potentially infinite stream of responses. The function starts off with an initial balance, and processes each request in the input stream to calculate a new balance, which is then fed back to the recursive call to process the remainder of the stream. (I recall that SICP discusses stream-transformer paradigm in another part of the book.)

A more elaborate version of this paradigm is called "functional reactive programming" discussed here on StackOverflow.

The naive way of doing stream transformers has some problems. It is possible (in fact, pretty easy) to write buggy programs that keep all the old requests around, wasting space. More seriously, it is possible to make the response to the current request depend on future requests. Solutions to these problems are currently being worked on. Neel Krishnaswami is the force behind them.

Disclaimer: I do not belong to the church of pure functional programming. In fact, I do not belong to any church :-)

edited May 23 at 12:40
Community ♦
1

answered Apr 15 '12 at 20:16
Uday Reddy
297 1 9

I guess you belong to some temple? :-P – Gnijuohz Apr 16 '12 at 9:10

1 The temple of free thinking. No preachers there. – Uday Reddy Apr 16 '12 at 10:57

It's not possible making a program 100% functional if it is supposed to do anything useful. (If side effects are not needed then the whole think could have been reduced to a constant compile time) Like the withdraw-example you can make most of the procedures functional but eventually you'll need procedures that has side effects (input from user, output to console). That said you can make most of your code functional and that part will be easy to test, even automatically. Then you make some imperative code to do the input/output/database/... which would need debugging, but keeping most of the code clean it won't be too much work. I'll use your withdraw-example:

```
(define +no-founds+ "Insufficient funds")

;; functional withdraw
(define (make-withdraw balance amount)
    (if (>= balance amount)
        (- balance amount)
        +no-founds+))

;; functional atm loop
(define (atm balance thunk)
  (let* ((amount (thunk balance))
         (new-balance (make-withdraw balance amount)))
    (if (eqv? new-balance +no-founds+)
        (cons +no-founds+ '())
        (cons (list 'withdraw amount 'balance new-balance) (atm new-balance thunk)))))

;; functional balance-line -> string
(define (balance->string x)
  (if (eqv? x +no-founds+)
      (string-append +no-founds+ "\n")
      (if (null? x)
          "\n"
          (let ((first-token (car x)))
            (string-append
             (cond ((symbol? first-token) (symbol->string first-token))
                   (else (number->string first-token)))
             " "
             (balance->string (cdr x)))))))

;; functional thunk to test
(define (input-10 x) 10) ;; define a purly functional input-method

;; since all procedures involved are functional
;; we expect the same result every time.
;; we use this to test atm and make-withdraw
(apply string-append (map balance->string (atm 100 input-10)))

;; no program can be purly functional in any language.
;; From here on there are imperative dirty procedures!

;; A procedure to get input from user is needed.
;; Side effects makes it imperative
(define (user-input balance)
  (display "You have $")
  (display balance)
  (display " founds. How much to withdraw? ")
  (read))

;; We need a procedure to print stuff to the console
;; as well. Side effects makes it imperative
(define (pretty-print-result x)
  (for-each (lambda (x) (display (balance->string x))) x))

;; use imperative procedure with atm.
(pretty-print-result (atm 100 user-input))
```

It is possible to do the same in almost any language and produce the same results (less bugs), though you might have to set temporary variables within a procedure and even mutate stuff, but that doesn't matter to much as long as the procedure actually acts functional (the parameters alone determines the result). I belive you become a better programmer at any language after you have programmed a little LISP :)

answered Aug 6 '12 at 22:13

Sylwester
509 2 6

+1 for the extensive example and realistic explanations about functional parts and non pure functional parts of the program and mentioning why FP matters nevertheless. – Zelphir Sep 23 '16 at 9:18

Assignment is bad operation because it divides the state-space to two parts, before-assignment, and after-assignment. This causes difficulties tracking how the variables are being changed during program execution. The following things in functional languages are replacing assignments:

1. Function parameters linked directly to return values
2. choosing different objects to be returned instead of modifying existing objects.
3. creating new lazy-evaluated values
4. listing all *possible* objects, not just the ones that need to be in memory
5. no side effects

answered Apr 12 '12 at 15:25

tp1
1,638 7 6

This doesn't seem to address the question posed. How do you program a bank account object in a pure functional language? – Uday Reddy Apr 15 '12 at 19:42

it's just functions transforming from one bank account record to another. The key is that when such transformations happen, new objects are chosen instead of modifying existing ones. – tp1 Apr 17 '12 at 14:30

When you transform one bank account record to another, you want the customer to do the next transaction on the new record, not the old one. The "contact point" for the customer must constantly be updated to point to the current record. That is a fundamental idea of "modification". Bank account "objects" are not bank account records. – Uday Reddy Apr 17 '12 at 19:16