How can a time function exist in functional programming?

I've to admit that I don't know much about functional programming. I read about it from here and there, and so came to know that in functional programming, a function returns the same output, for same input, no matter how many times the function is called. It's exactly like mathematical function which evaluates to same output for same value of input parameter which involves in the function expression.

For example, consider this:

```
f(x,y) = x*x + y; //it is a mathematical function
```

No matter how many times you use `f(10,4)` , its value will always be `104` . As such, wherever you've written `f(10,4)` , you can replace it with `104` , without altering the value of the whole expression. This property is referred to as referential transparency of an expression.

As Wikipedia says (link),

> Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result f(x) both times.

So my question is: can a time function (which returns the *current* time) exist in functional programming?

- If yes, then how can it exist? Does it not violate the principle of functional programming? It particularly violates referential transparency which is one of the property of functional programming (if I correctly understand it).
- Or if no, then how can one know the current time in functional programming?

scala haskell f# functional-programming clean-language

edited Aug 2 '13 at 16:00
Ryan ♦
141k 26 277 315

asked Sep 1 '11 at 8:26
Nawaz
221k 70 490 695

protected by Ashwini Chaudhary Sep 19 '13 at 18:03

This question is protected to prevent "thanks!", "me too!", or spam answers by new users. To answer it, you must have earned at least 10 reputation on this site (the association bonus does not count).

8 I think most (or all) functional languages are not so strict and combine functional and imperative programming. At least, this is my impression from F#. – Alex F Sep 1 '11 at 8:33

12 @Adam: How would the caller know the current time in the first place? – Nawaz Sep 1 '11 at 8:35

24 @Adam: Actually it is illegal (as in: impossible) in purely functional languages. – sepp2k Sep 1 '11 at 8:49

42 @Adam: Pretty much. A general purpose language which is pure usually offers some facility to get at the "world state" (i.e. things like the current time, files in a directory etc.) without breaking referential transparency. In Haskell that's the IO monad and in Clean it's the world type. So in those languages a function which needs the current time would either take it as an argument or it would need to return an IO action instead of its actual result (Haskell) or take the world state as its argument (Clean). – sepp2k Sep 1 '11 at 9:00

9 When thinking about FP it's easy to forget: a computer is a big chunk of mutable state. FP doesn't change that, it merely conceals it. – Daniel Sep 1 '11 at 14:56

|

11 Answers

Another way to explain it is this: no *function* can get the current time (since it keeps changing), but an *action* can get the current time. Let's say that `getClockTime` is a constant (or a nullary function, if you like) which represents the *action* of getting the current time. This *action* is the same every time no matter when it is used so it is a real constant.

Likewise, let's say `print` is a function which takes some time representation and prints it to the console. Since function calls cannot have side effects in pure functional language, we instead imagine that it is a function which takes a timestamp and returns the *action* of printing it to the console. Again, this is a real function, because if you give it the same timestamp, it will return the same *action* of printing it every time.

Now, how can you print the current time to the console? Well, you have to combine the two actions. So how can we do that? We cannot just pass `getClockTime` to `print` , since print expects a timestamp, not an action. But we can imagine that there is an operator, `>>=` , which *combines* two actions, one which gets a timestamp, and one which takes one as argument and prints it. Applying this to the actions previously mentioned, the result is... tadaaa... a new action which gets the current time and prints it. And this is incidently exactly how it is done in Haskell.

```
Prelude> System.Time.getClockTime >>= print
Fri Sep  2 01:13:23 東京 (標準時) 2011
```

So, conceptually, you can view it in this way: A pure functional program does not perform any IO, it defines an *action*, which the runtime system then executes. The *action* is the same every time, but the result of executing it depends on the circumstances of when it is executed.

I don't know if this was any clearer than the other explanations, but it sometimes helps me to think of it this way.

edited Nov 16 '12 at 14:57
zoul
70.9k 30 194 291

answered Sep 1 '11 at 16:28
dainichi
1,110 7 9

10 It's not convincing to me. You conveniently called `getClockTime` an action instead of a function. Well, if you call so, then call every function *action*, then even imperative programming would become functional programmming. Or maybe, you would like to call it *actional* programmming. – Nawaz Sep 1 '11 at 16:54

51 @Nawaz: The key thing to note here is that you cannot execute an action from within a function. You can only combine actions and functions together to make new actions. The only way of executing an action is to compose it into your `main` action. This allows pure functional code to be separated from imperative code, and this separation is enforced by the type system. Treating actions as first class objects also allow you to pass them around and build your own "control structures". – hammar Sep 1 '11 at 18:25

5 @trinithis Well, maybe I should have made it clearer that actions are also functions (just like all values in e.g. Haskell are functions, at least if you call constants nullary functions), but *not all functions are actions*. But I don't think the term *action* is misleading or imprecise. An action is a type of value which you can distinguish by its type, just like you can distinguish `Int` s by their type. I didn't drill too much into the type argument, partly because it's covered in other replies, partly because I wanted to focus on my point. – dainichi Sep 2 '11 at 0:17

28 Not everything in Haskell is a function - that's utter nonsense. A function is something whose type contains a `->` - that's how the standard defines the term and that's really the only sensible definition in the context of Haskell. So something whose type is `IO Whatever` is **not** a function. – sepp2k Sep 2 '11 at 10:08

6 @sepp2k So, myList :: [a -> b] is a function? ;) – fuz Sep 10 '11 at 15:33

|

Yes and no.

Different FP languages solve them differently.

In Haskell (a very pure one) all this stuff has to happen in something called the IO Monad - see here. You can think of it as getting another input (and output) into your function (the world-state) or easier as a place where "impureness" like getting the changing time happens.

Other languages like F# just have some impureness built in and so you can have a function that returns different values for the same input - just like *normal* imperative languages.

As Jeffrey Burka mentioned in his comment: Here is the nice intro to the IO Monad straight from the HaskellWiki.

edited Jun 13 '12 at 21:18
Simon Marlow
11.6k 3 35 32

answered Sep 1 '11 at 8:31
Carsten
41.6k 6 53 93

197 The crucial thing to realise about the IO monad in Haskell is that it is not just a hack to get around this problem; monads are a general solution to the problem of defining a sequence of actions in some context. One possible context is the real world, for which we have the IO monad. Another context is within an atomic transaction, for which we have the STM monad. Yet another context is in the implementation of a procedural algorithm (e.g. Knuth shuffle) as a pure function, for which we have the ST monad. And you can define your own monads too. Monads are a kind of overloadable semicolon. – Paul Johnson Sep 1 '11 at 16:31

2 I find it useful to not call things like getting the current time "functions" but something like "procedures" (though arguable the Haskell solution is an exception to this). – singpolyma Nov 16 '12 at 18:28

from the Haskell perspective classical "procedures" (things that have types like '... -> ()') are somewhat trivial as a pure function with ... -> () cannot do anything at all. – Carsten Nov 16 '12 at 22:45

but in this case this would be something like '() -> ...' (a constant function) and **I** would not call this a procedure - but call it as you wish :D – Carsten Nov 16 '12 at 22:46

3 The typical Haskell term is "action". – Sebastian Redl May 31 '15 at 10:40

In Haskell one uses a construct called *monad* to handle side effects. A monad basically means that you encapsulate values into a container and have some functions to chain functions from values to values inside a container. If our container has the type:

```
data IO a = IO (RealWorld -> (a,RealWorld))
```

we can safely implement IO actions. This type means: An action of type `IO` is a function, that takes a token of type `RealWorld` and returns a new token, together with a result.

The idea behind this is that each IO action mutates the outside state, represented by the magical token `RealWorld` . Using monads, one can chain multiple functions that mutate the real world together. The most important function of a monad is `>>=` , pronounced *bind*:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

`>>=` takes one action and a function that takes the result of this action and creates a new action out of this. The return type is the new action. For instance, let's pretend there is a function `now :: IO String` , which returns a String representing the current time. We can chain it with the function `putStrLn` to print it out:

```
now >>= putStrLn
```

Or written in `do` -Notation, which is more familiar to an imperative programmer:

```
do currTime <- now
   putStrLn currTime
```

All this is pure, as we map the mutation and information about the world outside to the `RealWorld` token. So each time, you run this action, you get of course a different output, but the input is not the same: the `RealWorld` token is different.

edited Aug 12 '15 at 2:38
chamini2
1,464 1 12 34

answered Sep 1 '11 at 9:18
fuz
46.5k 12 101 217

2 -1: I'm unhappy with the `RealWorld` smoke screen. Yet, the most important thing is how this purported object is passed on in a chain. The missing piece is where it starts, where the source or connection to the real world is -- it starts with the main function which runs in the IO monad. – u0b34a0f6ae Sep 6 '11 at 14:12

1 @kaizer.se You can think of a global `RealWorld` object that is passed into the program when it starts. – fuz Sep 6 '11 at 14:21

4 Basically, your `main` function takes a `RealWorld` argument. Only upon execution is it passed in. – Louis Wasserman Jun 13 '12 at 16:26

9 You see, the reason why they hide the `RealWorld` and only provide puny functions to change it like `putStrLn` , is so some Haskell programmer doesn't change `RealWorld` with one of their programs such that Haskell Curry's address and birth-date is such they become next-door neighbors growing up (this could damage the time-space continuum in such a way to hurt the Haskell programming language.) – PyRulez Jul 22 '14 at 21:58

I think the `RealWorld` metaphor doesn't work very well when starting to think about concurrency. When forking an IO action, does that mean that the entire world is copied? If so, how is it joined again? An analogy to an abstract version of Java's `Runnable` might be better (i.e. `Runnable` s which you can never run yourself but only bind to functions to produce new runnables). – Frerich Raabe Jan 12 '16 at 8:12

|

Most functional programming languages are not pure, i.e. they allow functions to not only depend on their values. In those languages it is perfectly possible to have a function returning the current time. From the languages you tagged this question with this applies to scala and f# (as well as most other variants of ML).

In languages like Haskell and Clean, which are pure, the situation is different. In Haskell the current time would not be available through a function, but a so-called IO action, which is Haskell's way of encapsulating side effects.

In Clean it would be a function, but the function would take a world value as its argument and return a fresh world value (in addition to the current time) as its result. The type system would make sure that each world value can be used only once (and each function which consumes a world value would produces a new one). This way the time function would have to be called with a different argument each time and thus would be allowed to return a different time each time.

edited Nov 9 '11 at 12:45

answered Sep 1 '11 at 8:36
sepp2k
251k 31 544 563

2 This makes it sound as if Haskell and Clean do different things. From what I understand, they do the same, just that Haskell offers a nicer syntax (?) to accomplish this. – Konrad Rudolph Sep 1 '11 at 13:56

26 @Konrad: They do the same thing in the sense that both use type system features to abstract side effects, but that's about it. Note that it's very well to explain the IO monad in terms of a world type, but the Haskell standard doesn't actually define a world type and it's not actually possible to get a value of type World in Haskell (while it's very possible and indeed necessary in clean). Further Haskell does not have uniqueness typing as a type system feature, so if it did give you access to a World, it could not ensure that you use it in a pure way the way Clean does. – sepp2k Sep 1 '11 at 14:22

Clean calls this uniqueness typing. – Thom Wiggers Dec 12 '13 at 8:39

"Current time" is not a function. It is a parameter. If your code depends on current time, it means your code is parameterized by time.

answered Sep 1 '11 at 17:46

Vlad Patryshev
979 8 13

It can absolutely be done in a purely functional way. There are several ways to do it, but the simplest is to have the time function return not just the time but also *the function you must call to get the next time measurement*.

In C# you could implement it like this:

```
// Exposes mutable time as immutable time (poorly, to illustrate by example)
// Although the insides are mutable, the exposed surface is immutable.
public class ClockStamp {
    public static readonly ClockStamp ProgramStartTime = new ClockStamp();
    public readonly DateTime Time;
    private ClockStamp _next;

    private ClockStamp() {
        this.Time = DateTime.Now;
    }
    public ClockStamp NextMeasurement() {
        if (this._next == null) this._next = new ClockStamp();
        return this._next;
    }
}
```

(Keep in mind that this is an example meant to be simple, not practical. In particular, the list nodes can't be garbage collected because they are rooted by ProgramStartTime.)

This 'ClockStamp' class acts like an immutable linked list, but really the nodes are generated on demand so they can contain the 'current' time. Any function that wants to measure the time should have a 'clockStamp' parameter and must also return its last time measurement in its result (so the caller doesn't see old measurements), like this:

```
// Immutable. A result accompanied by a clockstamp
public struct TimeStampedValue<T> {
    public readonly ClockStamp Time;
    public readonly T Value;
    public TimeStampedValue(ClockStamp time, T value) {
        this.Time = time;
        this.Value = value;
    }
}

// Times an empty loop.
public static TimeStampedValue<TimeSpan> TimeALoop(ClockStamp lastMeasurement) {
    var start = lastMeasurement.NextMeasurement();
    for (var i = 0; i < 10000000; i++) {
    }
    var end = start.NextMeasurement();
    var duration = end.Time - start.Time;
    return new TimeStampedValue<TimeSpan>(end, duration);
}

public static void Main(String[] args) {
    var clock = ClockStamp.ProgramStartTime;
    var r = TimeALoop(clock);
    var duration = r.Value; //the result
    clock = r.Time; //must now use returned clock, to avoid seeing old measurements
}
```

Of course, it's a bit inconvenient to have to pass that last measurement in and out, in and out, in and out. There are many ways to hide the boilerplate, especially at the language design level. I think Haskell uses this sort of trick and then hides the ugly parts by using monads.

edited Jun 13 '12 at 13:39

answered Jun 13 '12 at 13:33

Craig Gidney
11.3k 2 39 101

Interesting, but that `i++` in the for loop isn't referentially transparent ;) – snim2 Jun 13 '12 at 13:35

@snim2 I'm not perfect. :P Take solace in the fact that the dirty mutableness doesn't affect the referential transparency of the result. If you pass the same 'lastMeasurement' in twice, you get a stale next measurement and return the same result. – Craig Gidney Jun 13 '12 at 13:41

@Strilanc Thanks for this. I think in imperative code, so it's interesting to see functional concepts explained this way. I can then imagine a language where this natural and syntactically cleaner. – WW. Jul 10 '13 at 6:20

You could in fact go the monad way in C# as well, thus avoiding the explicit passing of time stamps. You need something like `struct TimeKleisli<Arg, Res> { private delegate Res(TimeStampedValue<Arg>); }`. But code with this still wouldn't look as nice as Haskell with `do` syntax. – leftaroundabout Aug 3 '13 at 18:11

@leftaroundabout you can sort of pretend that you have a monad in C# by implementing the bind function as a method called `SelectMany`, which enables the query comprehension syntax. You still can't program polymorphically over monads though, so it's all an uphill battle against the weak type system :(– kai Apr 17 '16 at 11:58

Yes, getting time function can exist in FP using a slightly modified version on FP known as impure FP (the default or the main one is pure FP).

In case of getting the time (or reading file, or launching missile) the code needs to interact with the outer world to get the job done and this outer world is not based on pure foundations of FP. To allow a pure FP world to interact with this impure outside world people have introduced impure FP. After all a software which doesn't interact with the outside world isn't any useful other than doing some mathematical computations.

Few FP programming languages have this impurity feature inbuilt in them such that it is not easy to separate out which code is impure and which is pure (like F# etc) and some FP languages make sure that when you do some impure stuff that code is clearly stand out as compared to pure code, like Haskell.

Another interesting way to see this would be that your get time function in FP would take a "world" object which has the current state of the world like time, number of people living in the world etc. Then getting time from which world object would be always pure i.e you pass in the same world state you will always get the same time.

answered Sep 1 '11 at 8:57

Ankur
27.6k 2 23 53

1 "After all a software which doesn't interact with the outside world isn't any useful other than doing some mathematical computations." As far as I understand, even in this case the input to the computations would be hard-coded in the program, also not very useful. As soon as you want to read input data to your mathematical computation from file or terminal, you need impure code. – Giorgio Sep 1 '11 at 11:49

1 What about input data as command line arguments :) – Ankur Sep 1 '11 at 12:04

1 @Ankur: That is the same exact thing. If the program is interacting with something else than just itself(e.g. the world through they keyboard, so to speak) it's still impure. – identity Sep 1 '11 at 12:11

2 Having the "world object" including the number of people living in the world raises the executing computer to a near omniscient level. I think the normal case is that it includes things like how many files are on your HD and what's the home directory of the current user. – ziggystar Sep 1 '11 at 12:42

3 @ziggystar - the "world object" doesn't actually include anything - it is simply a proxy for the changing state of the world outside of the program. Its only purpose is to explicitly mark mutable state in a way that the type system can identify it. – Kris Nuttycombe Sep 1 '11 at 15:29

|

I am surprised that none of the answers or comments mention coalgebras or coinduction. Usually, coinduction is mentioned when reasoning about infinite data structures, but it is also applicable to an endless stream of observations, such as a time register on a CPU. A coalgebra models hidden state; and coinduction models *observing* that state. (Normal induction models *constructing* state.)

This is a hot topic in Reactive Functional Programming. If you're interested in this sort of stuff, read this: http://digitalcommons.ohsu.edu/csetech/91/ (28 pp.)

answered Sep 19 '14 at 0:11

Jeffrey Aguilera
591 5 7

2 And how is that related to this question? – Nawaz Sep 19 '14 at 2:06

2 Your question was about modeling time-dependent behavior in a purely functional way, e.g., a function that returns the current system clock. You can either thread something equivalent to an IO monad through all the functions and their dependency tree to get access to that state; or you can model the state by defining the observation rules rather than the constructive rules. This is why modeling complex state *inductively* in functional programming seems so unnatural, because the hidden state is really a *coinductive* property. – Jeffrey Aguilera Sep 30 '14 at 21:12

Yes! You are correct! Now() or CurrentTime() or any method signature of such flavour is not exhibiting referential transparency in one way. But by instruction to the compiler it is parameterized by a system clock input.

By output, Now() might look like not following referential transparency. But actual behaviour of the system clock and the function on top of it is adheres to referential transparency.

answered Sep 1 '11 at 18:02

MduSenthil
926 1 12 38

Yes, it's possible for a pure function to return the time, if it's given that time as a parameter. Different time argument, different time result. Then form other functions of time as well and combine them with a simple vocabulary of function(-of-time)-transforming (higher-order) functions. Since the approach is stateless, time here can be continuous (resolutionindependent) rather than discrete, greatly boosting modularity. This intuition is the basis of Functional Reactive Programming (FRP).

edited Oct 14 '15 at 23:22

answered Feb 25 '14 at 17:20
Conal
16k 2 29 37

Your question conflates two related measures of a computer language: functional/imperative and pure/impure.

A functional language defines relationships between inputs and outputs of functions, and an imperative language describes specific operations in a specific order to perform.

A pure language does not create or depend on side effects, and an impure language uses them throughout.

One-hundred percent pure programs are basically useless. They may perform an interesting calculation, but because they cannot have side effects they have no input or output so you would never know what they calculated.

To be useful at all, a program has to be at least a smidge impure. One way to make a pure program useful is to put it inside a thin impure wrapper. Like this untested Haskell program:

```
-- this is a pure function, written in functional style.
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)

-- This is an impure wrapper around the pure function, written in imperative style
-- It depends on inputs and produces outputs.
main = do
    putStrLn "Please enter the input parameter"
    inputStr <- readLine
    putStrLn "Starting time:"
    getCurrentTime >>= print
    let inputInt = read inputStr    -- this line is pure
    let result = fib inputInt       -- this is also pure
    putStrLn "Result:"
    print result
    putStrLn "Ending time:"
    getCurrentTime >>= print
```

answered Sep 2 '12 at 5:26
NovaDenizen
3,030 7 19

2 It would be helpful if you could address the specific issue of getting the time, and explained a little about to what extent we consider `IO` values and results pure. – AndrewC Sep 28 '12 at 10:31

In fact, even 100% pure programs heat up the CPU, which is a side-effect. – Jörg W Mittag Nov 26 '14 at 17:13