Lightweight computation for networks at the edge

Why time is evil in distributed systems and what to do about it

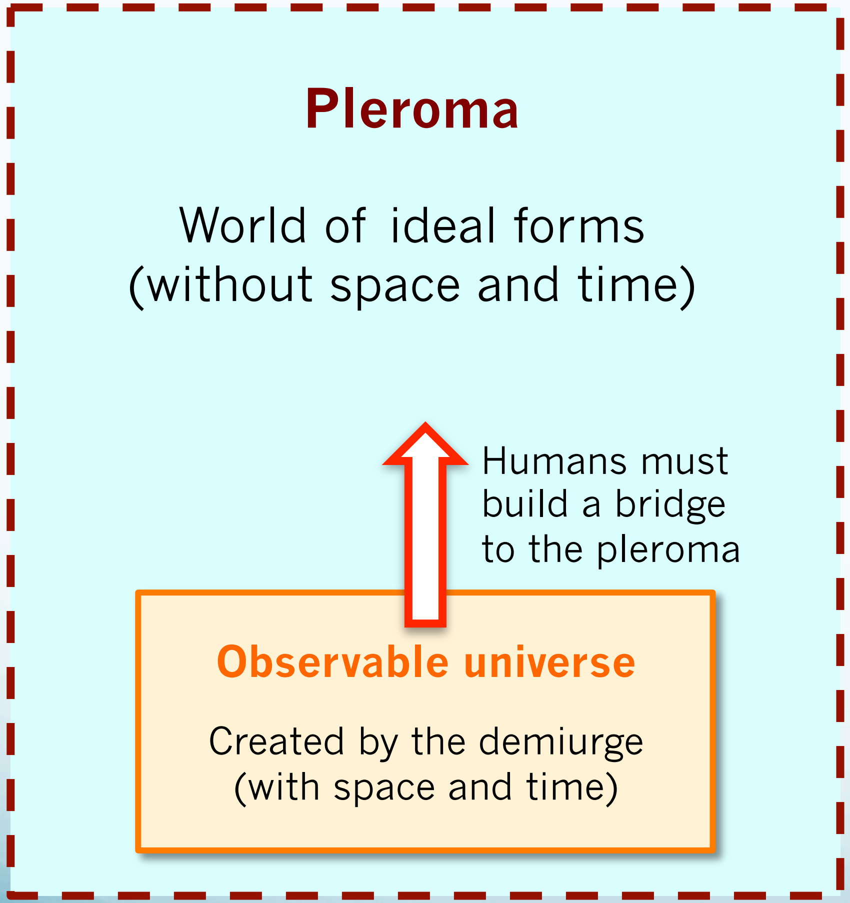
CodeBEAM 2019 keynote
May 16, 2019

Peter Van Roy
Université catholique de Louvain

Overview

- Motivation: understand the essential effect of time
  - Time is subtle: sometimes it makes a difference, or not
- Examples of distributed programs
  - Client/server, other distributed algorithms
- How to design with interaction points
  - Distributed functional concurrency
  - Multi-agent concurrency
  - Client/server redux
- Case study: eventual consistency
- Conclusions and advice for system designers

Gnosticism



"Time is the breath of the demiurge. And all his creation, the expansion of the universe, the evolution of species, the gradual development of his plan, could not occur without time. According to the Gnostics, the time-breath of the demiurge is as satanic as matter and as satanic as the demiurge himself."

– "The Forbidden Religion" by José M. Herrou Aragón

LIGHTKONE
Lightweight computation for networks at the edge

Motivation

LIGHTKONE

Lightweight computation for networks at the edge

Poker and chess

- Programming is superficially like mathematics, but there is a fundamental difference between the two
  - Programming is only interesting because computers run in the real world, whereas mathematics is a purely formal game of symbol manipulation
- "Programming is to mathematics as poker is to chess"
  - Poker is only interesting when real money is involved
- What real-world property is vital for programming?
  - It is time

We will build a bridge…

- …between two research communities
  - Distributed systems
  - Programming languages
- There is not much interaction between the two
  - Work on distributed abstractions, not much more
  - The communities do not understand each other
- In this talk we will make a deep connection
  - We will apply language theory in a fundamental way to build better distributed systems
  - It is surprising what a difference it can make

LIGHTKONE
Lightweight computation for networks at the edge

Functional programming

- **Confluent reduction of an initial expression to a final result**

✓ This has very strong mathematical properties that we can use

- For reasoning, debugging, testing, optimization, and maintenance
- For concurrency, parallelism, and distribution
- And there is no efficiency penalty compared to other paradigms

✗ But it can't interact with the real world! Let's see why:

- During the execution, we would like to accept inputs coming from the real world and outputs going back to it
- Functional programming can't do this because the execution of a functional program is a step-by-step reduction of an initial expression to a final result. Reduction steps take time, and the inputs will arrive during this time. The reduction can't use them unless we could put them in the initial expression. But we can't do this, because the inputs are not known in advance.

LIGHTKONE
Lightweight computation for networks at the edge

Imperative programming

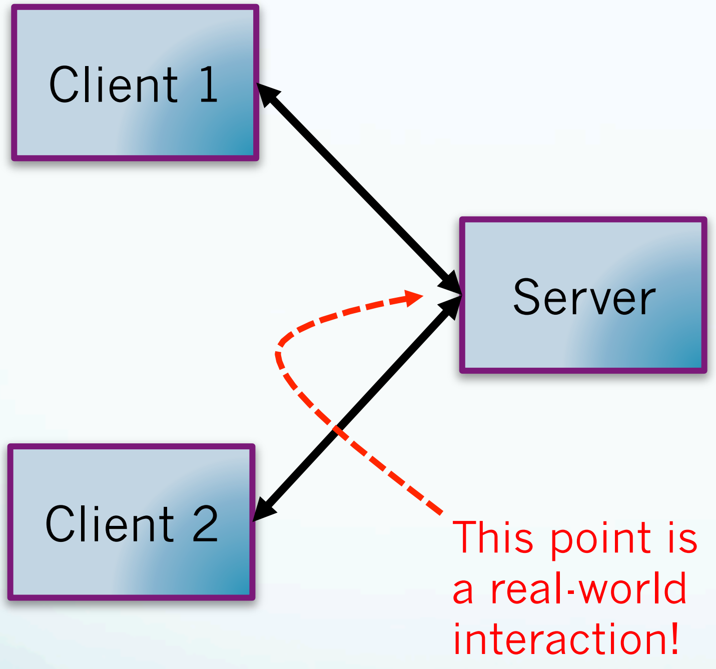
- To interact with the real world, we need to add something to functional programming
  - A way to receive an input during execution
- This lets us interact with the real world, but in the same breath we give up the good properties of functional programming
- Can we have our cake and eat it too? Both the good properties of functional programming and interaction with the real world?
  - No we can't! So what can we do…?

The solution

- Write most of the program with functional programming
  - And add small pieces of imperative programming only in those places that interact with the real world
  - Usually there are only a very few such places, so we keep most of the advantages of functional programming
- We can use this to improve existing systems too…
  - Existing systems are often not designed like this! They do way too much imperative programming. (Older systems like Java are especially bad.)
  - This gives us a measure to judge how well existing systems are designed (and a way to improve them)

Some examples

Client/server



- A client/server cannot be written in pure functional programming
- It is because to satisfy client liveness, the server must accept each incoming request in reasonable time
- Therefore the order of the requests cannot be determined in advance because it depends on client timing
- So the program is nondeterministic
  - There is one interaction point, where the program's result is affected by the real world: where the server receives messages

Interaction point = part of system where the real world affects the program's result

LIGHTKONE
Lightweight computation for networks at the edge

Interaction points are everywhere

- **Reliable broadcast** (i.e., all or none broadcast): Has no interaction point
- **Shared registers**: Linearizability has no interaction point. However, sequential consistency and regular registers can introduce interaction points.
  - Even for quorums, because order of updates is nondeterministic
- **Consensus** (e.g., Paxos or Raft): Does have an interaction point
  - Consensus is interesting, because agreement is a form of determinism
  - However, there is still nondeterminism in the choice of accepted proposal
  - Nondeterminism seems to be inherent when the consensus algorithm is running with partial synchrony (like the Internet). I have no proof, though.
- **Causality**: Concurrent events are often interaction points; ordered events not
- **Synchrony model**:
  - Partial synchrony or asynchrony: algorithms may have interaction points
  - Full synchrony: interaction points can easily be avoided

How to design with interaction points

Approach

- We want to design distributed systems with interaction points, so that we can add them only where needed and nowhere else
  - No existing languages let you do this (as far as I know)
  - Let us define a simple "design language" that does exactly this
  - This is the right way, because (1) it will be easy to think about designs in this language, and (2) the designs map easily to your favorite real language
    - So we're golden: we have the right tool and it's future-proof
- But what if you're stuck with **really bad legacy languages**?
  - Like Java and similar crud
  - Use the design language and translate to the legacy language
  - Add layers to clean up the legacy language

Functions and interaction points

- We define the design language in two steps:
  1. Distributed functional concurrency (pure)
  2. Multi-agent concurrency (adds interaction points)
- Everything we learn from the design language maps directly to real languages
  - Don't be fooled by complicated real languages: they add lots of bells & whistles to make coding easier, but they still do basically the same things as our design language

First step: functional concurrency

- In the first step we define a simple language to write purely functional distributed programs
- We do it from the ground up, based on λ calculus
  - We then give it a more convenient syntax
  - You can translate this into your favorite language
- In the second step, we add interaction points
  - This will be our design language
  - Write mostly functional, add a few interaction points

Lambda (λ) calculus

- Lambda calculus is the core of functional programming
  - We define it first and then we show functional concurrency
- Syntax
  - $x ::=$ (variables)
  - $t ::= x \ | \ (\lambda x.\ t) \ | \ (t_1\ t_2)$
- Semantics (using substitution operation $t[x]$)
  - $(\lambda x.\ t[x]) \rightarrow (\lambda y.\ t[y])$ α-conversion
  - $((\lambda x.\ t_1)\ t_2) \rightarrow t_1[x{:=}t_2]$ β-reduction
  - $((\lambda x.\ (t\ x)) \rightarrow t$ (if x not free in t) η-conversion

LIGHTKONE
Lightweight computation for networks at the edge

Properties of λ calculus

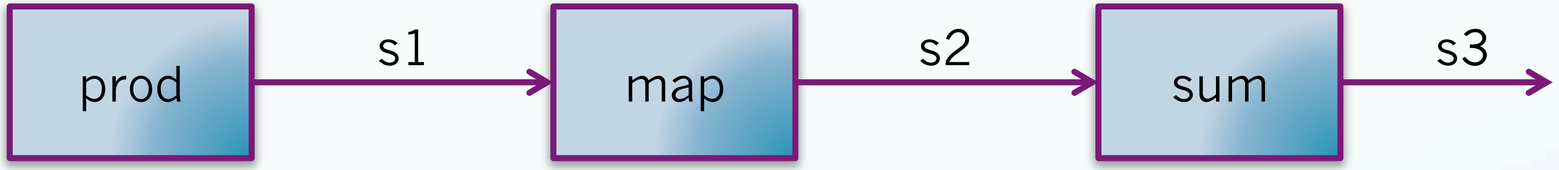
- Data types and control structures
  - Data types (lists, numbers, etc.) and control structures (if, case, etc.) can be added to the λ calculus without changing anything essential
- Confluence
  - **Church-Rosser theorem: Final result of a reduction is the same for all reduction orders (up to variable renaming)**
  - This holds for many variants of the λ calculus
- Functional concurrency: a more convenient syntax
  - λ calculus can express networks of concurrent agents. Each agent has its own state and sends and receives messages from neighboring agents.
  - It's only syntax; it keeps all the good properties of the λ calculus

LIGHTKONE
Lightweight computation for networks at the edge

Functional concurrency

- Define agents, streams, and threads:
  - Agent = tail-recursive function executing in its own thread
  - Stream = list read by one agent and created by another agent
  - Thread = a restriction on which reductions we are interested in

```
local s1 s2 s3 in
   thread s1=prod(1) end
   thread s2=map(s1,fun (x) x*x end) end
   thread s3=sum(s2,0) end
end
```



```
fun prod(n)
     delay(1000)
     n|prod(n-1)
end
```

```
fun map(s, f)
     case s of h|t then
          f(h)|map(t, f)
     [] nil then
          nil
     end
end
```

```
fun sum(s, a)
     case s of h|t then
          h+a|sum(t,h+a)
     [] nil then
          nil
     end
end
```

Distributed λ calculus

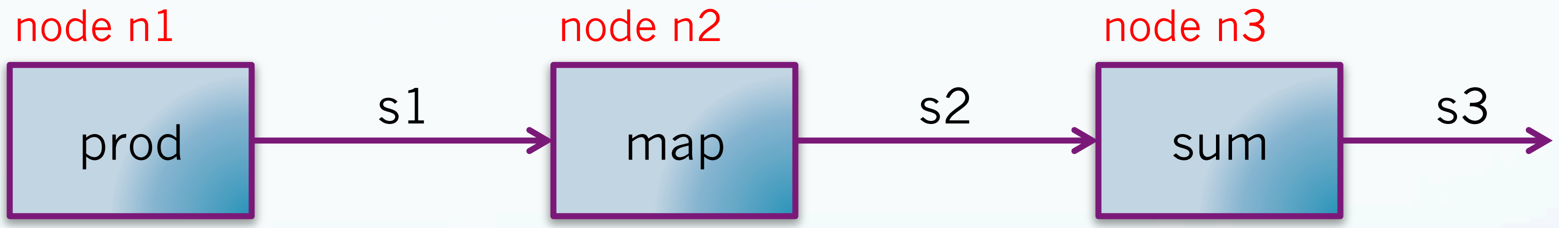
- We can easily make functional concurrency distributed
  - Consider a set of nodes N with $a, b, c, \ldots \in N$
- Localize each term on a node
  - x ::= (variables)
  - $t^a ::= x^a \mid (\lambda x.\ t^b)^a \mid (t^b_1\ t^c_2)^a$
  - Terms can reference subterms on other nodes
- Extend the reduction rules to execute on single nodes
  - $(\lambda x.\ t^a[x])^a \rightarrow (\lambda y.\ t^a[y])^a$ α-conversion
  - $((\lambda x. t^a_1)^a\ t^a_2)^a \rightarrow t^a_1[x := t^a_2]$ β-reduction
  - $((\lambda x.\ (t^a\ x^a)^a)^a \rightarrow t^a$ (if x not free in t^a) η-conversion
  - $t^a \rightarrow t^b$ μ-conversion (mobility)

LIGHTKONE
Lightweight computation for networks at the edge

Distributed functional concurrency

- We put each agent on a node
- This gives a distributed concurrent program that is purely functional
- An agent always knows from where the next input will come

```
local s1 s2 s3 in
   node s1=prod(1) end
   node s2=map(s1,fun (x) x*x end) end
   node s3=sum(s2,0) end
end
```



```
fun prod(n)
   delay(1000)
   n|prod(n-1)
end
```

```
fun map(s, f)
   case s of h|t then
      f(h)|map(t, f)
   [] nil then
      nil
   end
end
```

```
fun sum(s, a)
   case s of h|t then
      h+a|sum(t,h+a)
   [] nil then
      nil
   end
end
```

LIGHTKONE
Lightweight computation for networks at the edge

Second step: interaction points

- We add interaction points to our design language
- We again start with the λ calculus
  - We add new terms and rules for interaction points
- We again define a more convenient syntax
  - We extend distributed functional concurrency with interaction points
  - This gives multi-agent concurrency
- We show how to solve the client/server example
  - We need only one interaction point in the whole system

LIGHTKONE
Lightweight computation for networks at the edge

Read-write distributed λ calculus

- We add read and write operations to the distributed λ calculus
  - Result depends on reduction order and timing, so they are interaction points
  - If the read returns the result of the most recent write, then it's **mutable state**
  - But write and read can also behave like **send and receive**
- Add read and write terms
  - x ::= (variables)
  - $t^a ::= x^a \mid (\lambda x.\ t^b)^a \mid (t^b_1\ t^c_2)^a \mid (\sigma.t^b)^a \mid (\rho x.\ t^b)^a$
- Add two reduction rules

$(\lambda x.\ t^a[x]) \rightarrow (\lambda y.\ t^a[y])$	α-conversion
$((\lambda x.t^a_1)\ t^a_2)^a \rightarrow t^a_1[x{:=}t^a_2]$	β-reduction
$((\lambda x.\ (t^a\ x))^a \rightarrow t^a$ (if x not free in t^a)	η-conversion
$t^a \rightarrow t^b$	μ-conversion (mobility)
$(\sigma.t^a)^a \rightarrow t^a$	σ-reduction (write, i.e., send)
$(\rho x.t^a_1)^a \rightarrow t^a_1[x{:=}t^a_2]$	ρ-reduction (read, i.e., receive)

LIGHTKONE
Lightweight computation for networks at the edge

Multi-agent concurrency

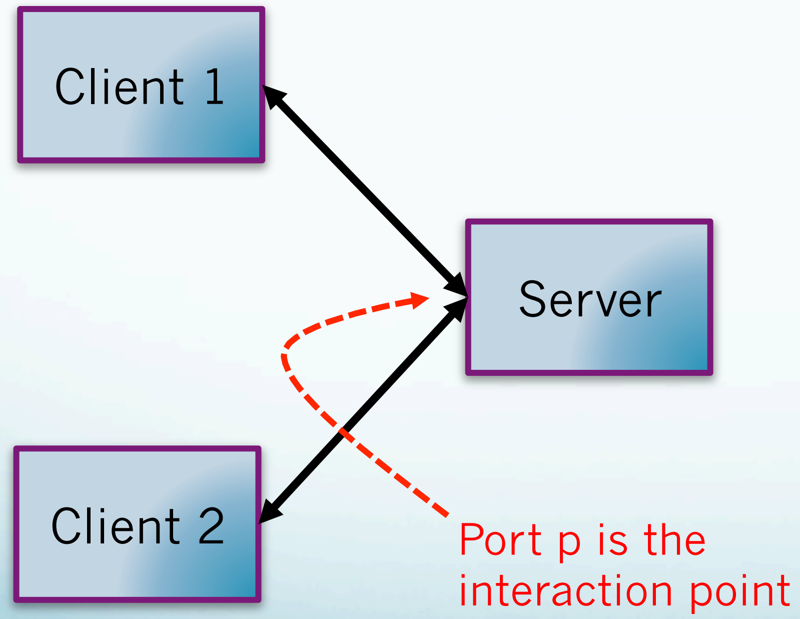
- We invent a convenient syntax for the read-write λ calculus
  - We start with functional concurrency (agents, streams, threads)
  - We add interaction points in a nice way
- Inspired by the client/server example, we add named streams
  - Sending a value to the name will add it to the stream; reading the stream will read the value

  -
    ```
    p=newport(s)     /* Create a name p for stream s */
    send(p, x)       /* Add x to the end of the stream named by p */
    ```
- This gives multi-agent concurrency
  - Most of the program is functional, with a few interaction points
  - The best paradigm for writing concurrent and distributed programs!
  - Client/server is easy to write with multi-agent concurrency…

LIGHTKONE
Lightweight computation for networks at the edge

Client/server redux

- Now we can define a client/server
- f_c and f_s are pure functions
- There is just one interaction point



```
local s p in
   node p=newport(s) server(state,s) end
   node client(state1,p) end
   node client(state2,p) end
   … /* as many clients as we need */
end

fun client(state,p)
   send(query(state),p)
   client(fc(state),p)
end

fun server(state,s)
    case s of q|t then
          server(fs(q,state),t)
     [] nil then
          nil
     end
end
```

One interaction point

LIGHTKONE
Lightweight computation for networks at the edge

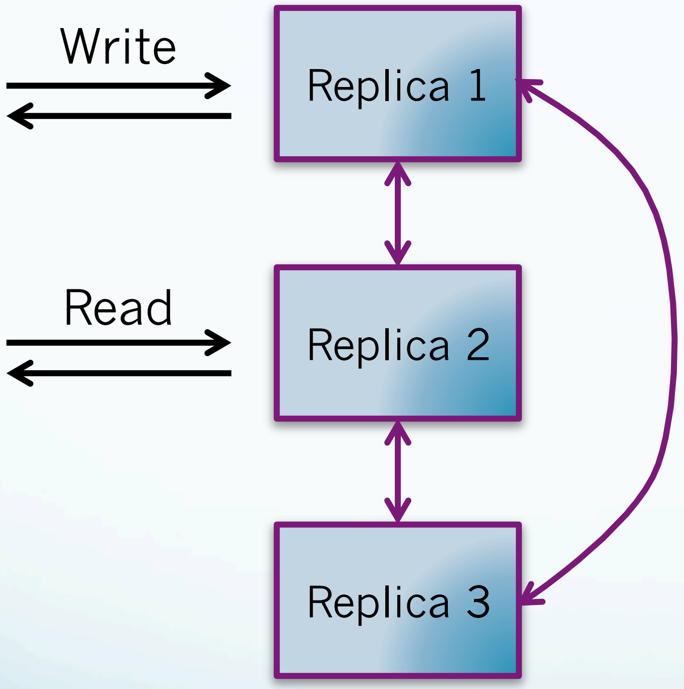
Now we can design!

- Now we can design distributed systems that manage all their interactions with the real world
- We write most of the program in functional concurrency
  - By design, this has zero interaction points (it is pure)
- We then add interaction points as needed
  - Only where we need to interact with the real world
  - The full program uses multi-agent concurrency
- As few as possible
  - Interaction points add messiness!

Case study:
Eventual consistency

LIGHTKONE
Lightweight computation for networks at the edge

Eventual consistency



- Commonly done for performance
  - Requests can be initiated concurrently; multiple requests can be "in flight" simultaneously; replies are returned as quickly as possible
  - Writes are eventually propagated to all replicas; reads are eventually handled by at least one replica
- Consider a replicated database
  - A write is done and immediately followed by a read (without waiting for the write to finish)
  - Does the read see the write?
    - Sometimes yes, sometimes no!
- How should we think about this?
  - Focus on the interaction points! Can we get rid of them?

LIGHTKONE
Lightweight computation for networks at the edge

Removing interaction points

- Improve the design by removing interaction points
- For eventual consistency there are several ways
  - Use strong consistency (quorums). This fixes part of the problem, but successive operations are still nondeterministic. We can improve it by adding causal order to the system, it's not that simple.
  - Use convergent consistency (Conflict-free Replicated Data Types – CRDTs). Make sure updates never lose information (use CRDT merge instead of arbitrary writes and each client keeps local CRDT replica that is also merged).
- Convergent consistency (e.g., Antidote or SwiftCloud):
  - Reads observe previous writes
  - Successive reads observe increasing set of writes
  - Writes applied after observed reads

LIGHTKONE
Lightweight computation for networks at the edge

Convergent consistency

- We should make this precise (ambiguity is the bane of distribution!)
  - We use events e on objects k with visibility between events $e_1 \prec_{vis} e_2$ (see "Principles of Eventual Consistency" by Sebastian Burckhardt)

Highly recommended! Free pdf!

- Eventual consistency
  - An operation is intermittent before becoming permanent
  - "All an object's events are seen by all other events on that object, except for a finite number"
  - For all objects k: $\forall e \in E_k .\ \{e' \in E_k \mid e \nprec_{vis} e'\}$ is finite where E_k is the set of k's events

- Convergent consistency
  - An operation once done is seen forever
  - "Event e of object k_1 once visible to k_2 is always visible to k_2"
  - $\forall e \in E_{k1}, \forall e', e'' \in E_{k2} : e \prec_{vis} e' \wedge e' \prec_{vis} e'' \Rightarrow e \prec_{vis} e''$

LIGHTKONE
Lightweight computation for networks at the edge

Conclusion

Conclusion

- Programming requires real-world interaction
  - This is why programming is like poker, not like chess
  - For distributed programming especially, time and order of events are crucial
- Functional programming is the best paradigm for writing programs
  - But it does not support real-world interaction
  - Imperative programming does, but drops most of the advantages
- The solution is to use both in the right way
  - Use functional concurrency by default because it is pure
  - Add interaction points for real-world interaction
  - Use a design language that lets you identify the interaction points
  - This is work in progress: we are still formalizing and elaborating it
- Exercise for you: define the design language as an Erlang variant!
  - Right now, each Erlang process is an interaction point, which is the wrong default; the variant needs to switch this default

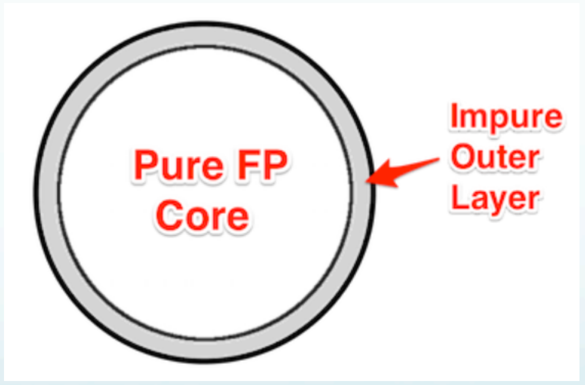
Extra information

A functional programming view

Excerpt from *Benefits of Functional Programming*, chapter of *Functional Programming, Simplified*, by Alvin Alexander, 2019

The rationale is good, but unfortunately Haskell is not a good starting point, for four reasons: (1) eager should be the default instead of lazy (nonstrict), (2) it does not have functional concurrency, (3) it is not distributed, and (4) you often want to hide an impure implementation so it looks pure from the outside

- Another way that pure functions make code easier to reason about won't be apparent when you're first getting started. It turns out that what *really* happens in FP applications is that (a) you write as much of the code as you can in a functional style, and then (b) you have other functions that reach out and interact with files, databases, web services, UIs, and so on — everything in the outside world.
- The concept is that you have a "Pure Function" core, surrounded by impure functions that interact with the outside world:



- Given this design, a great thing about Haskell in particular is that it provides a clean separation between pure and impure functions — so clean that you can tell by looking at a function's signature whether it is pure or impure.

Foundation of Gnosticism

Pleroma ("fullness")
The world of ideal forms or concepts, it is the only reality

The pleroma is not a space in the usual sense and does not have time; it is not based on metric time or space as we know them. It is the space of all possible concepts, infinitely denser and more intricate than the observable universe we know. Concepts are connected in infinite ways to an infinity of other concepts and are nested both towards the small and the large. Consciousness in the pleroma surveys concepts like scintillating rays of sunlight touching objects in a big dark room full of stuff. What is in the mind is what the light illuminates. The roaming of a person's thought processes is a faint echo of this.

Functional programming is analogous: an expression is reduced step by step. The expression being reduced is the illuminated part. All earlier and later expressions in the reduction also exist and are true (because expressions remain true forever), but they are not seen in the current reduction step. Time is defined as the order of the reduction sequence, but this order is just one among many possible orders of expressions. The Church-Rosser theorem states that no matter what choices are made to move the illuminated part, it ends up in the same place. Church-Rosser states that there is no free will in the execution of a functional program.

Kenoma ("emptiness")
The observable universe, it may be the limit of our consciousness

The kenoma is a small piece of the pleroma, with concepts connected according to rules set up by a minor divinity called the demiurge, and that we call the "laws of physics". The rays of light are forced to stay on a line, the line of time. This gives us the illusion of time and change, but in fact it is just a play between the concepts of time and change, acting on the concepts in the observable universe. Time is a concept of sequencing, that connects related objects using a relationship called the change concept. This is all that the demiurge can do, letting us participate in this game of an evolving world. The demiurge is like a child playing with blocks in the corner and we are part of this world of blocks. But our thought may sometimes touch the world of ideal forms, so maybe we have the potential to leave the kenoma.