Functional Programming Invades Architecture

George Fairbanks
SATURN 2017
3 May 2017

Programming in the Large

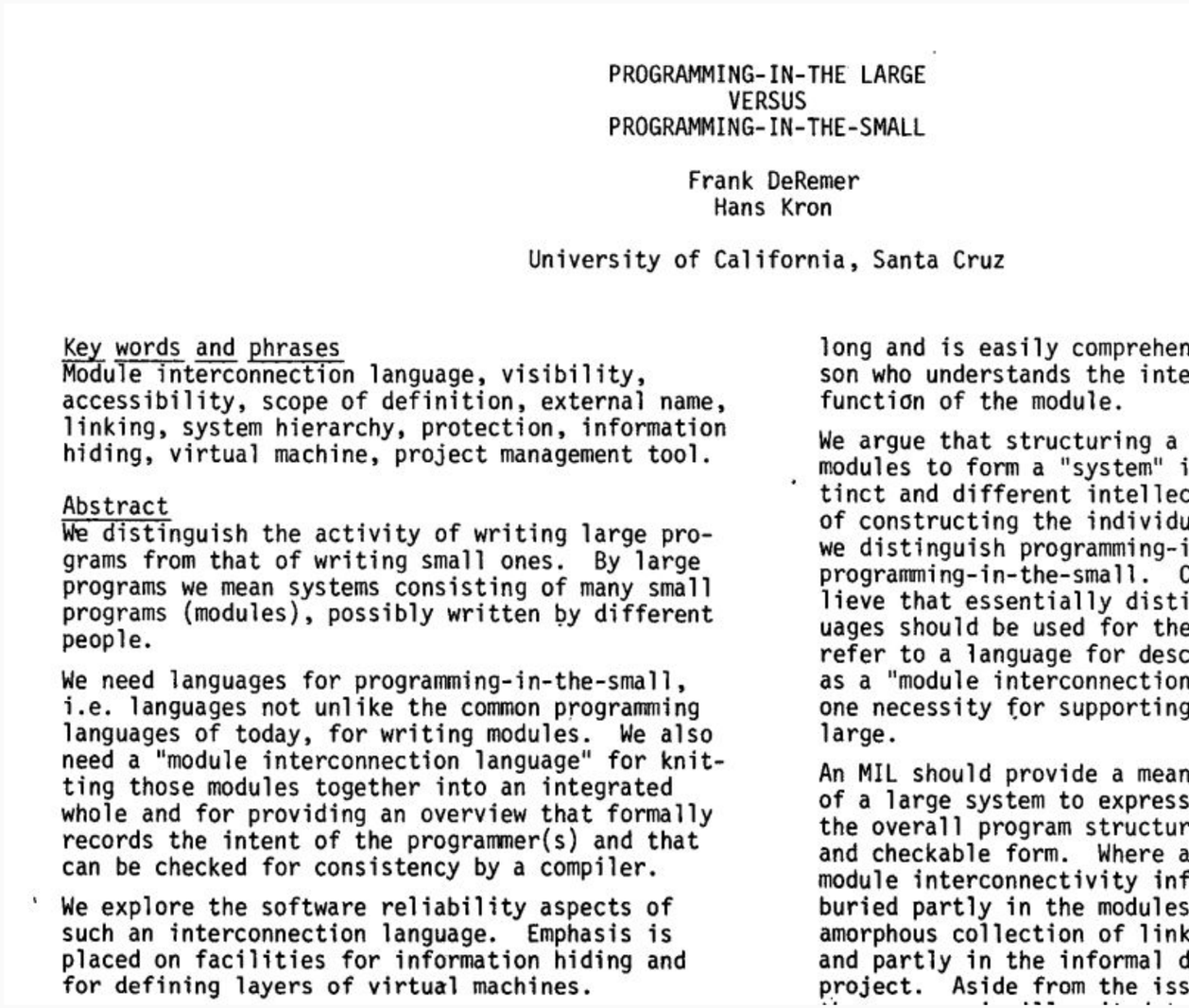
Yesterday:

Functional Programming is PITS,
i.e., “just inside modules”

Today:

FP is also PITL

DeRemer and Kron, Programming-in-the large versus programming-in-the-small, ACM SIGPLAN Notices, Volume 10 Issue 6, June 1975.



PROGRAMMING-IN-THE LARGE
VERSUS
PROGRAMMING-IN-THE-SMALL

Frank DeRemer
Hans Kron

University of California, Santa Cruz

Key words and phrases
Module interconnection language, visibility, accessibility, scope of definition, external name, linking, system hierarchy, protection, information hiding, virtual machine, project management tool.

Abstract
We distinguish the activity of writing large programs from that of writing small ones. By large programs we mean systems consisting of many small programs (modules), possibly written by different people.

We need languages for programming-in-the-small, i.e. languages not unlike the common programming languages of today, for writing modules. We also need a "module interconnection language" for knitting those modules together into an integrated whole and for providing an overview that formally records the intent of the programmer(s) and that can be checked for consistency by a compiler.

We explore the software reliability aspects of such an interconnection language. Emphasis is placed on facilities for information hiding and for defining layers of virtual machines.

long and is easily comprehen
son who understands the inte
function of the module.

We argue that structuring a
modules to form a "system" i
tinct and different intellec
of constructing the individu
we distinguish programming-i
programming-in-the-small.
lieve that essentially disti
uages should be used for the
refer to a language for desc
as a "module interconnection
one necessity for supporting
large.

An MIL should provide a mean
of a large system to express
the overall program structur
and checkable form. Where a
module interconnectivity inf
buried partly in the modules
amorphous collection of link
and partly in the informal d
project. Aside from the iss

Why FP now?

Problems grew bigger

- FP good for
  parallelism and concurrency
- Systems run on many computers

Coping strategy for complexity

- As systems get bigger
  but our brains stay the same size
  we invent tech to cope
- Statelessness, immutability, and pure functions make it easier to reason about how a system behaves

We have more money than sense

- 100 cloud machines
  is cheaper than
  one junior developer
- So why not:
  - Continuous Integration with every source code commit?
  - Server farm running different versions of your code?
  - Burn RAM with immutable data structures?

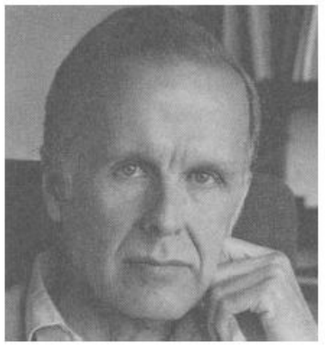
Current perspective on old ideas

- FP increasingly popular in PITS
  - Lambdas and streams in Java, JS, Python, Ruby, …
  - Renewed interest in pure FP languages

- **We can intersect two domains**
  - Architecture and FP
  - Perspective on existing patterns

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

John Backus, Can Programming Be Liberated from the von Neumann Style?, Turing Award Lecture, ACM 1977.

Group exercise

Cube Composer

Get ready: RxMarbles

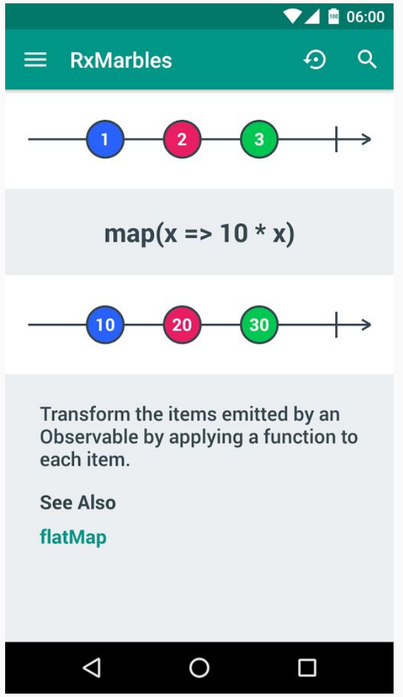
We won’t use this until later, but let’s get it now.

Laptop: www.rxmarbles.com

Android: https://play.google.com/store/apps/details?id=com.moonfleet.rxmarbles

iOS: https://itunes.apple.com/us/app/rxmarbles/id1087272442

Ooh, shiny toy! But please wait for it...

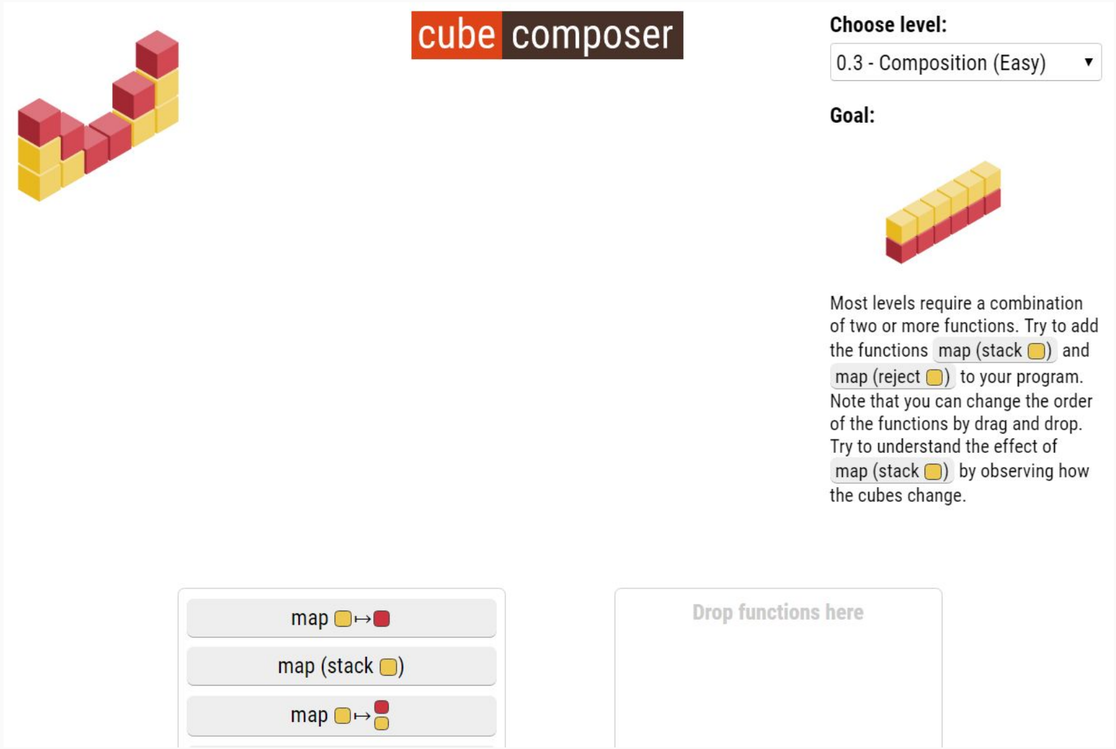


Cube Composer game

Goals

- Get a feel for functional transformation
- Feel awkward, like CS 101
- Sense that you will get better at this
- Sense of “rightness” from a different reasoning style

http://david-peter.de/cube-composer/



Boring slides about an interesting topic

Big Ideas in FP

Function composition

- Build programs by combining small functions

  `g(f(x))` or `f(x) |> g(x)`

- Seen in pipelines, event-based systems, machine learning systems, reactive

  `ls | grep “foo” | uniq | sort`

Note: We’re just covering the FP ideas that seem relevant to architecture

Pure functions, no side effects

- Calling function with same params always yields same answer
- So: Reasoning about the outcome is easier

```
curl http://localhost/numberAfter/5 → [always 6]
curl http://localhost/customer/5/v1 → [always v1 of customer]
```

vs

```
curl http://localhost/customer/5 → [may be different]
```

Statelessness and Immutability

Statelessness

- If there's no state:

- Easy to reason about
- All instances are equivalent

Immutability

- If you have state, but it never changes:

- Easy to reason about
- Concurrent access is safe

Idempotence

- Idempotence: get the same answer regardless of how many times you do it

  `resizeTo100px(image)` vs `shrinkByHalf(image)`

- Often hard to guarantee something is done exactly once

- Eg: Is the task stalled or failed?
  - Go ahead and schedule it again without fear that you'll get a duplicate result

Declarative, not imperative

- Functional programming
  - Define what something ***is***
    … or how it relates to other things
- Versus
  - Series of operations that yield desired state

- Definition, not procedure

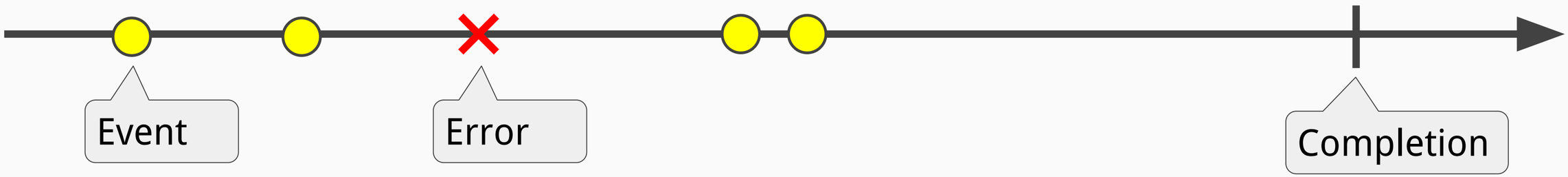
- Example: how much paint do I need?
  - `while(!done) { fence.paint(); }`
  - Vs function parameterized by length and width

More boring slides about an interesting topic

Reactive Programming

Reactive programming

- Operations on time-series event streams
- Reactive = Observer Pattern + onError and onCompletion
- Inband vs out of band signaling



Andre Staltz, The introduction to Reactive Programming you've been missing, https://gist.github.com/staltz/868e7e9bc2a7b8c1f754, Aug 2015.

Streams and operators

- Event streams
- Rich set of stream operators
  - Transform streams into streams
- Example: Detect double clicks
  - Input: user click stream
  - Transform: group nearby clicks
  - Transform: count group size
  - Transform: drop size-1 groups
  - Output: double click stream

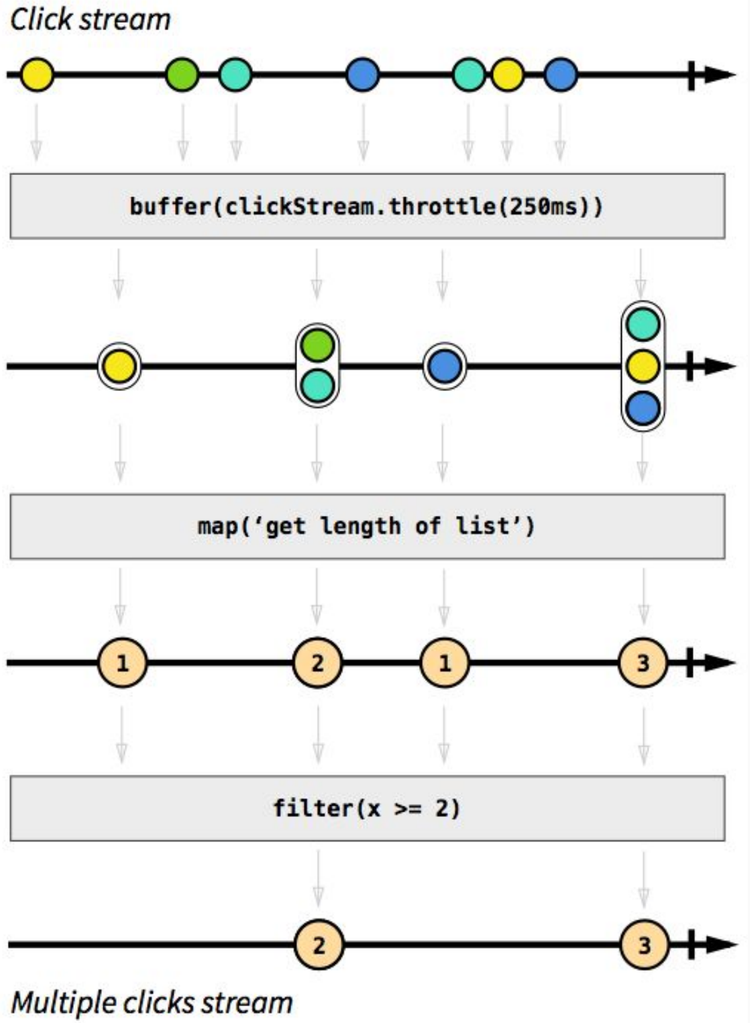


Diagram source: Andre Staltz, The introduction to Reactive Programming you've been missing, https://gist.github.com/staltz/868e7e9bc2a7b8c1f754, Aug 2015.

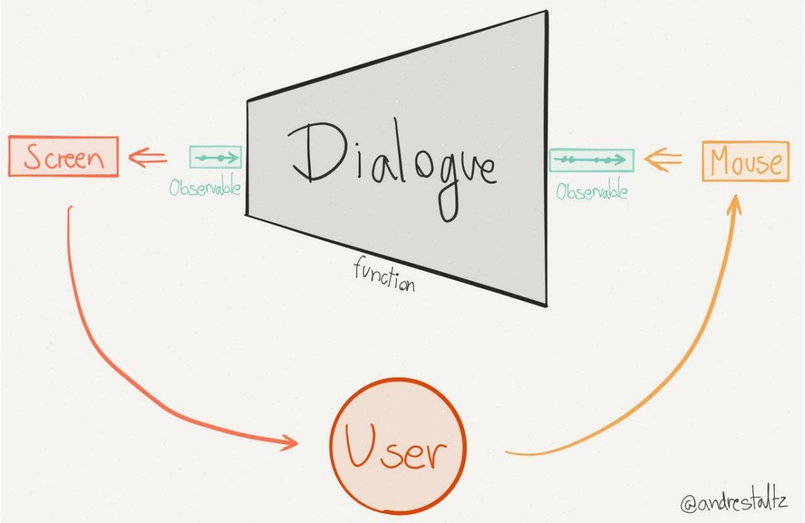
Uni-directional User Interfaces

Model-View-Controller and siblings

- Mutable state
- Problems with async calls to server

Reactive, Uni-directional UI pattern

- One-way stream transformation
- Human action → Transform → HTML



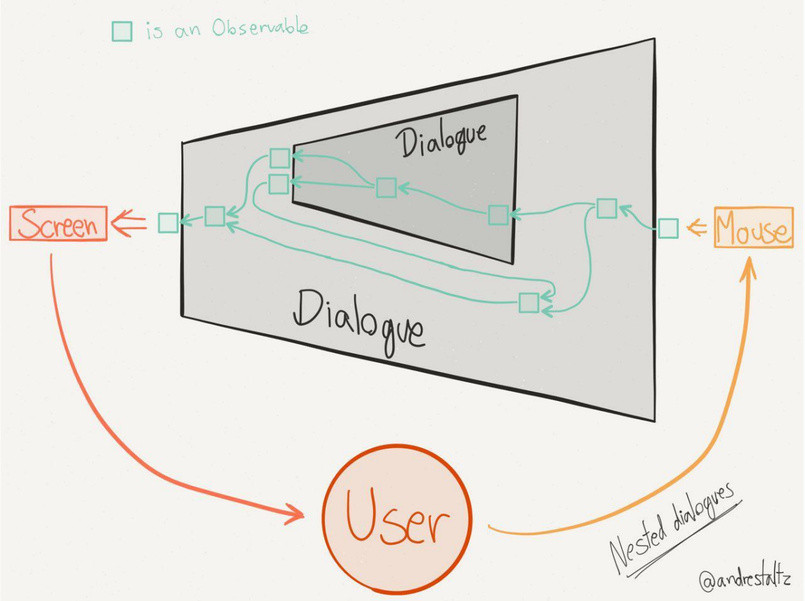


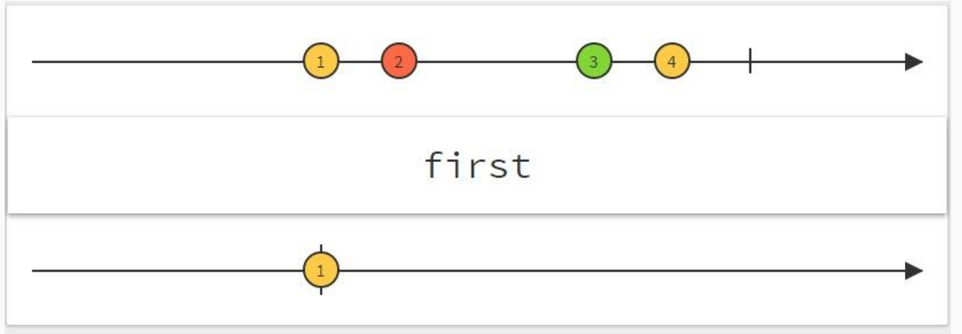
Diagram source: Andre Staltz, Unidirectional User Interface Architectures, https://staltz.com/unidirectional-user-interface-architectures.html, Aug 2015.

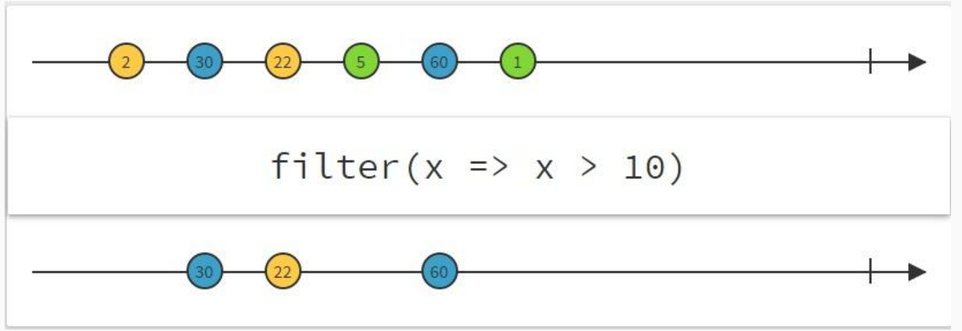
Group exercise

RxMarbles

Filtering

- first, last
- filter
- take, takeUntil





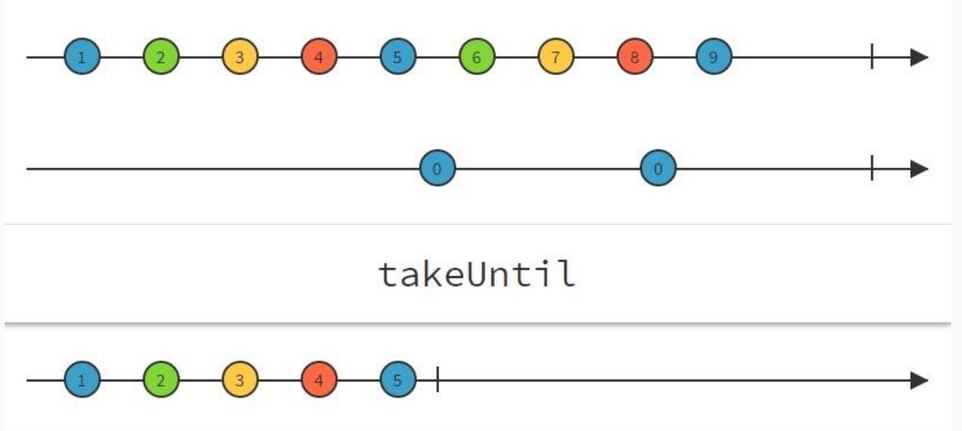
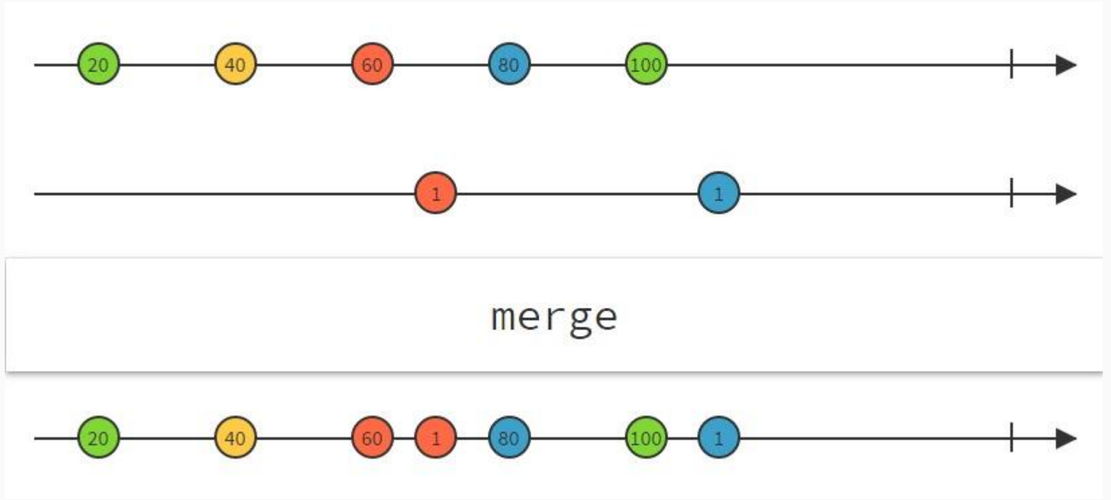


Diagram source: Andre Staltz, rxmarbles.com

Transforming

- merge



- map

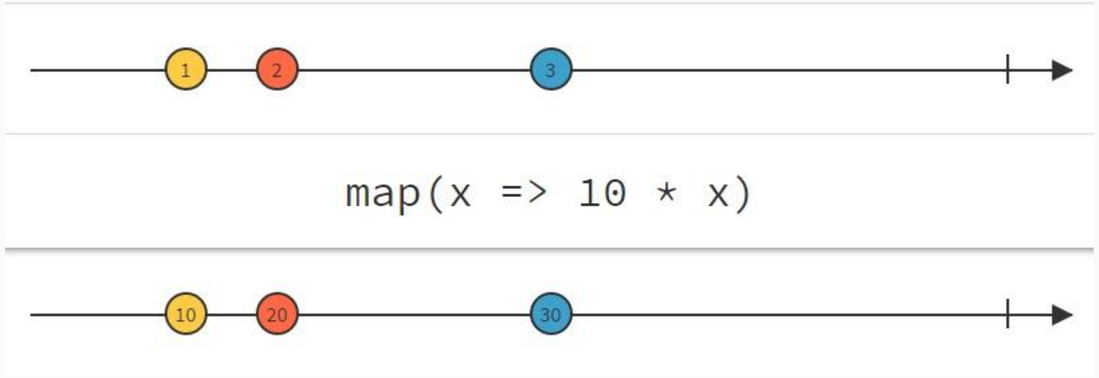
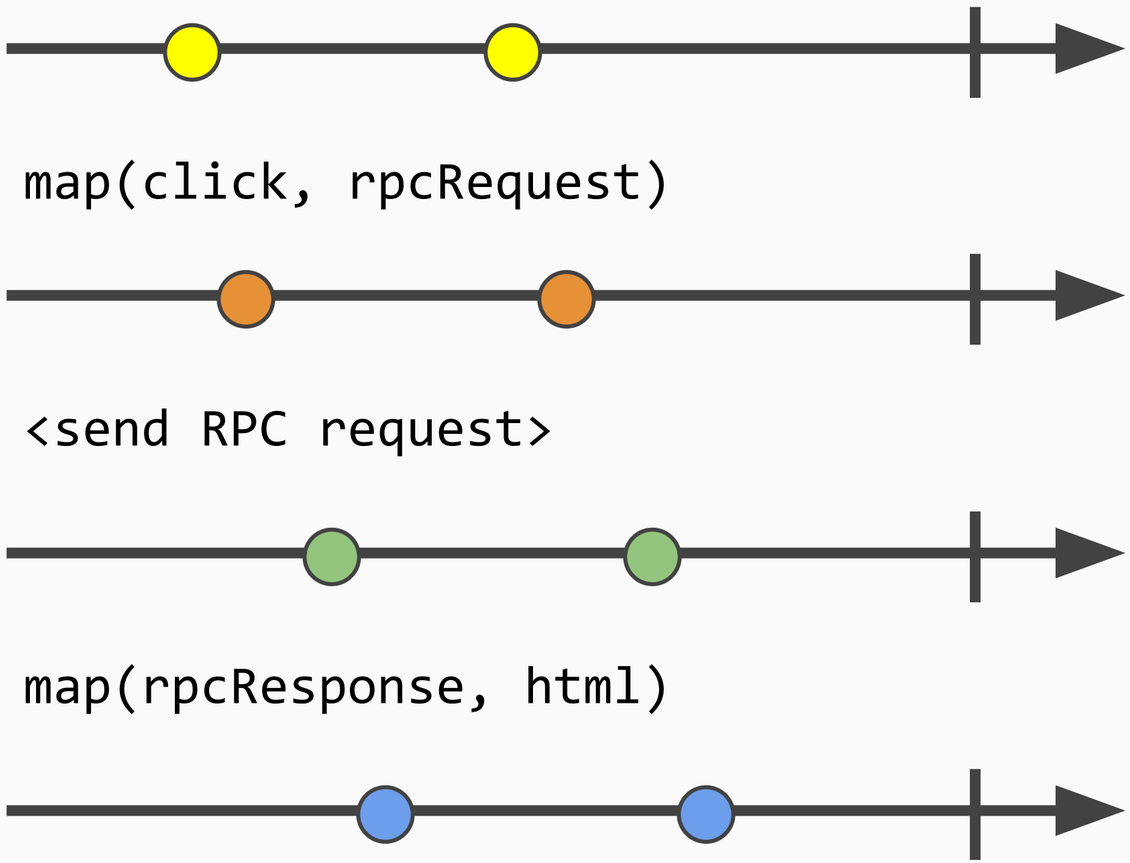


Diagram source: Andre Staltz, rxmarbles.com

Calling a server

- Stream 1: User clicks
- Stream 2: RPC requests
- Stream 3: RPC responses
- Stream 4: UI data



Group exercise

- Situation: Your UI needs an authentication token

- Challenge 1:
  - Build a stateless UI (ie no field holding the token)
  - Can you use a (transformed) stream of events?

- Challenge 2:
  - Tokens that expire and must be refreshed

Even more boring slides about an interesting topic

Functional ideas in architecture

Client-side

- Reactive patterns and frameworks in UI
  - React, Elm (and The Elm Architecture), CycleJS, Flux, Redux

- (So-called) “serverless”
  - App composes domain-neutral remote infra services

Andre Staltz, Unidirectional User Interface Architectures, https://staltz.com/unidirectional-user-interface-architectures.html, Aug 2015.

Mike Roberts, Serverless Architectures, https://martinfowler.com/articles/serverless.html, August 2016.

Server-side

- Stateless middle tier servers, pure functions (AWS lambda)
- Immutable state
  - Resource versioning (not mutation), eg with REST
- (So-called) “serverless” pure functions
  - On-demand deployment, no stable-identity “business logic” servers
- Reactive services, event queues
  - Services: Transform input stream to output stream
  - Events/Messages: Routed to services

Persistence

- Append-only datastores
- Event sourcing
- Command Query Response Segregation (CQRS)

Batch sequential & pipeline

- Big data processing (Hadoop, Spark)
- Map-Reduce
- Tensorflow and other machine learning graph-based functional transformation

DevOps infrastructure

- Version control everything (append-only)
  - Including the scripts that test and deploy code

- So-called “immutable” and “idempotent” infrastructure
  - Never modify a running service
  - Redeploy to change config (no mutation)

Group exercise

Architecture using FP

Design a library: non-FP

- Problem: Library (browse, checkout, checkin)

- Design 1 (do this together):
  - Styles used: Client-Server, 3-tier, Repository (Relational DB)

- Outputs:
  - Runtime view
  - Message sequence diagram
  - DB tables

Design a Library: Functional Ideas

- Design this in groups of 3-5

- Try using some of:
  - Reactive client
  - Reactive server
  - Stateless server functions
  - Event sourcing or append-only database

- Hints
  - Can you define what is on the screen or DB as “All the <things> such that…”
  - Are there places (eg tiers) you can eliminate state?

The end of the boring slides

Conclusion

Conclusion

- FP not just PITS
- Many FP ideas in PITL
  - Function composition
  - Pure functions
  - Statelessness / Immutability
  - Idempotence
  - Declarativeness
- Architecture advantages, especially for large distributed systems
- Big changes in UI / client
  - (mutable) MVC → Uni-directional
  - Object graphs → stream transforms
- Wanted: FP tactics catalog
  - Why use Event Queues?
  - Or Append-only datastores?
  - Or stateless servers?

Interesting links

http://www.eugenkiss.com/b/overview-of-reactive-gui-programming/

DeRemer and Kron, Programming-in-the large versus programming-in-the-small, ACM SIGPLAN Notices, Volume 10 Issue 6, June 1975.

Andre Staltz, The introduction to Reactive Programming you've been missing, https://gist.github.com/staltz/868e7e9bc2a7b8c1f754, Aug 2015.

Andre Staltz, Unidirectional User Interface Architectures, https://staltz.com/unidirectional-user-interface-architectures.html, Aug 2015.

Mike Roberts, Serverless Architectures, https://martinfowler.com/articles/serverless.html, August 2016.