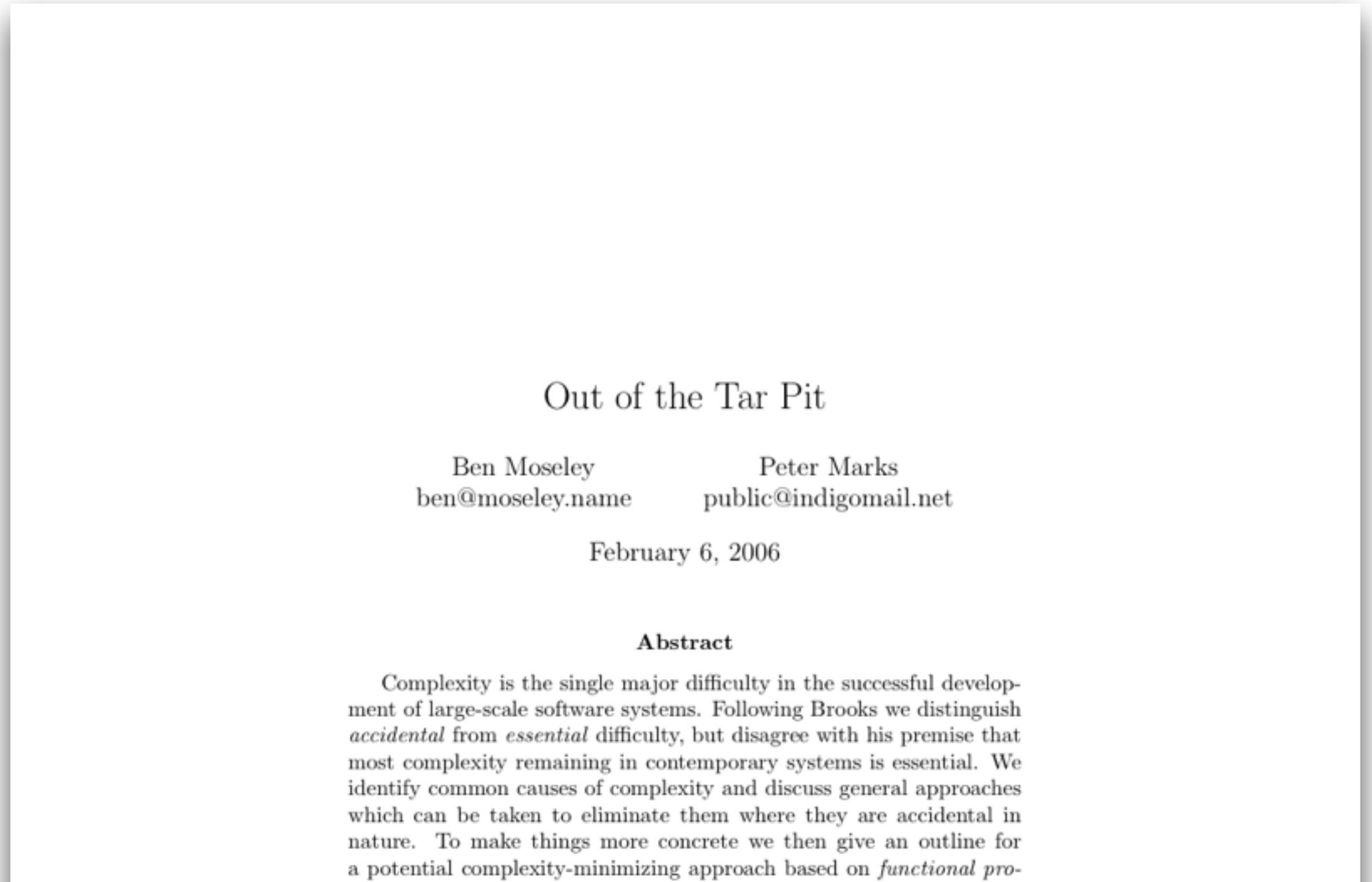
State, Immutability, and Persistent Data Structures

Brian Maddy
@bmaddy

Functional Relational Programming

(not reactive programming - see Colin Lee's talk for that)



Out of the Tar Pit

Ben Moseley
ben@moseley.name

Peter Marks
public@indigomail.net

February 6, 2006

Abstract

Complexity is the single major difficulty in the successful development of large-scale software systems. Following Brooks we distinguish *accidental* from *essential* difficulty, but disagree with his premise that most complexity remaining in contemporary systems is essential. We identify common causes of complexity and discuss general approaches which can be taken to eliminate them where they are accidental in nature. To make things more concrete we then give an outline for a potential complexity-minimizing approach based on *functional pro-*

The Plan

- Amount and impact of mutable state
- Identity vs. Value
- Benefits of less mutable state
- What can we do now?

Sound familiar?

“Try it again”

“Reboot”

“Restart the program”

“It must have been a fluke”

“Try refreshing the page”

“Reinstall the program”

“Restart the server”

“Well, it seems to work now”

“Reload the file”

“Reinstall the operating system and then the program”

http://shaffner.us/cs/papers/tarpit.pdf

How many possible states?

1 bit: 2 possible states

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

How many possible states?

1 bit: 2 possible states

# of atoms on Earth: 2^{167} atoms

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

How many possible states?

1 bit: 2 possible states

# of atoms on Earth: 2^{167} atoms

6 32-bit long ints: 2^{192} possible states

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

How many possible states?

1 bit: 2 possible states

# of atoms on Earth: 2^{167} atoms

6 32-bit long ints: 2^{192} possible states

# of atoms in the Universe: 2^{266} atoms

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

How many possible states?

1 bit: 2 possible states

# of atoms on Earth: 2^{167} atoms

6 32-bit long ints: 2^{192} possible states

# of atoms in the Universe: 2^{266} atoms

5 JavaScript numbers: 2^{320} possible states

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

How many possible states?

1 bit: 2 possible states

# of atoms on Earth: 2^{167} atoms

6 32-bit long ints: 2^{192} possible states

# of atoms in the Universe: 2^{266} atoms

5 JavaScript numbers: 2^{320} possible states

8GB of RAM: $2^{64000000000}$ possible states

Inspired by: https://youtu.be/lKXe3HUG2l4?t=783

Program Correctness

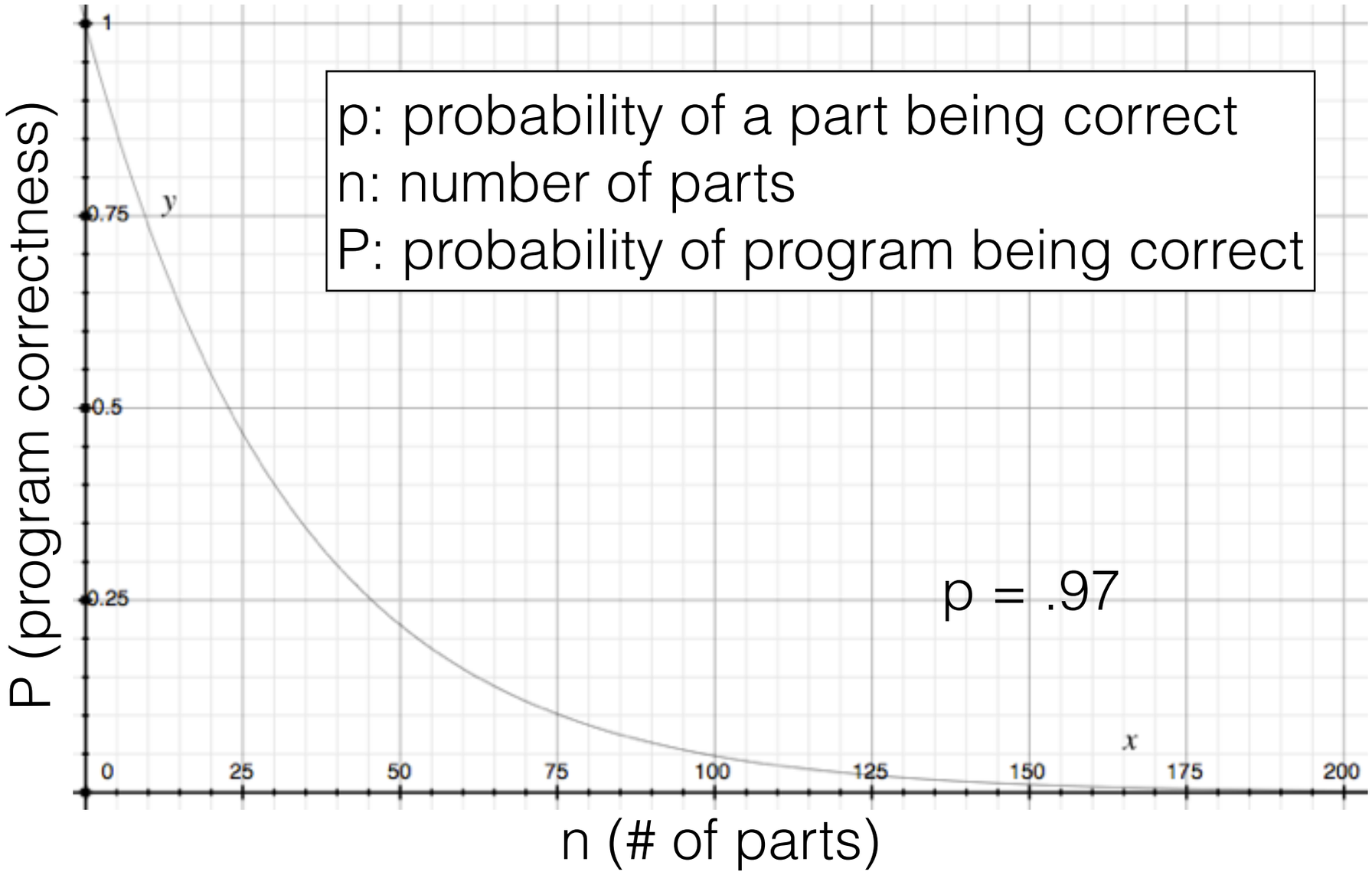
$$P = p^n$$

p: probability of a part being correct
n: number of parts
P: probability of program being correct

Notes on Structured Programming by Edgar Dijkstra, 1969

Program Correctness

$$P = p^n$$



Notes on Structured Programming by Edgar Dijkstra, 1969

Pure Functions

- always returns the same result for a given input (uses no mutable state)

- no side effects

Mutable State Pollutes

```
(defn stateless-func []
  ...do stuff...
  ...do stuff...
  ...do stuff...)
```

http://shaffner.us/cs/papers/tarpit.pdf

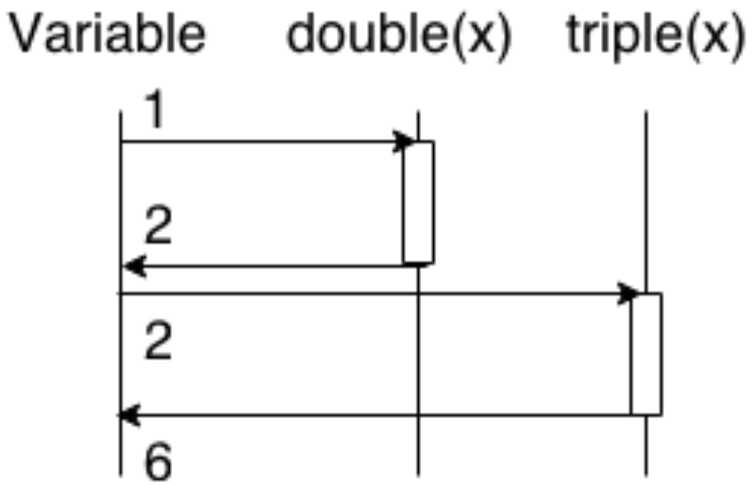
Mutable State Pollutes

Not anymore!

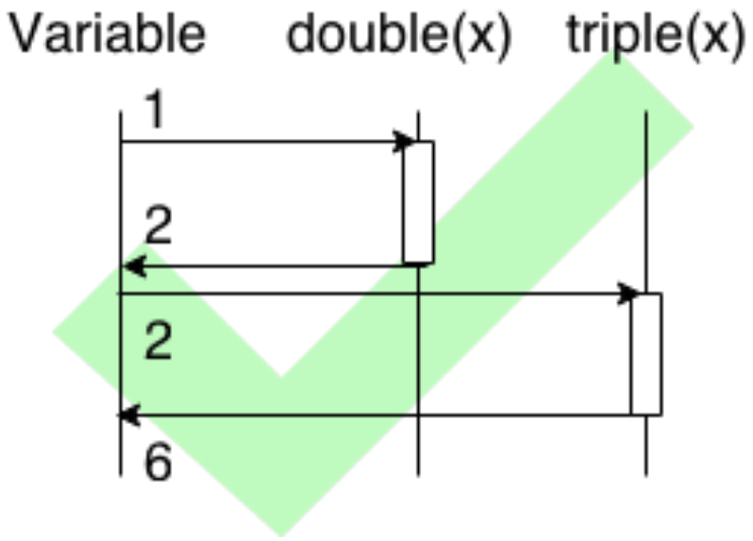
```
(defn stateless-func []
  ...do stuff...
  (stateful-func)
  ...do stuff...)
```

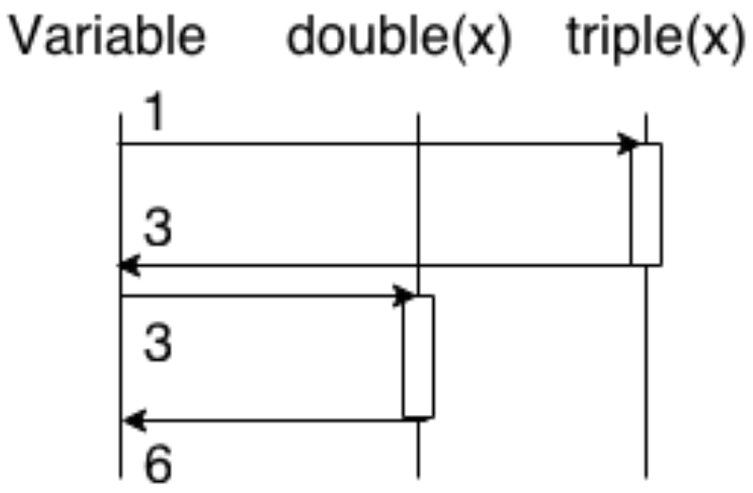
http://shaffner.us/cs/papers/tarpit.pdf

Concurrency

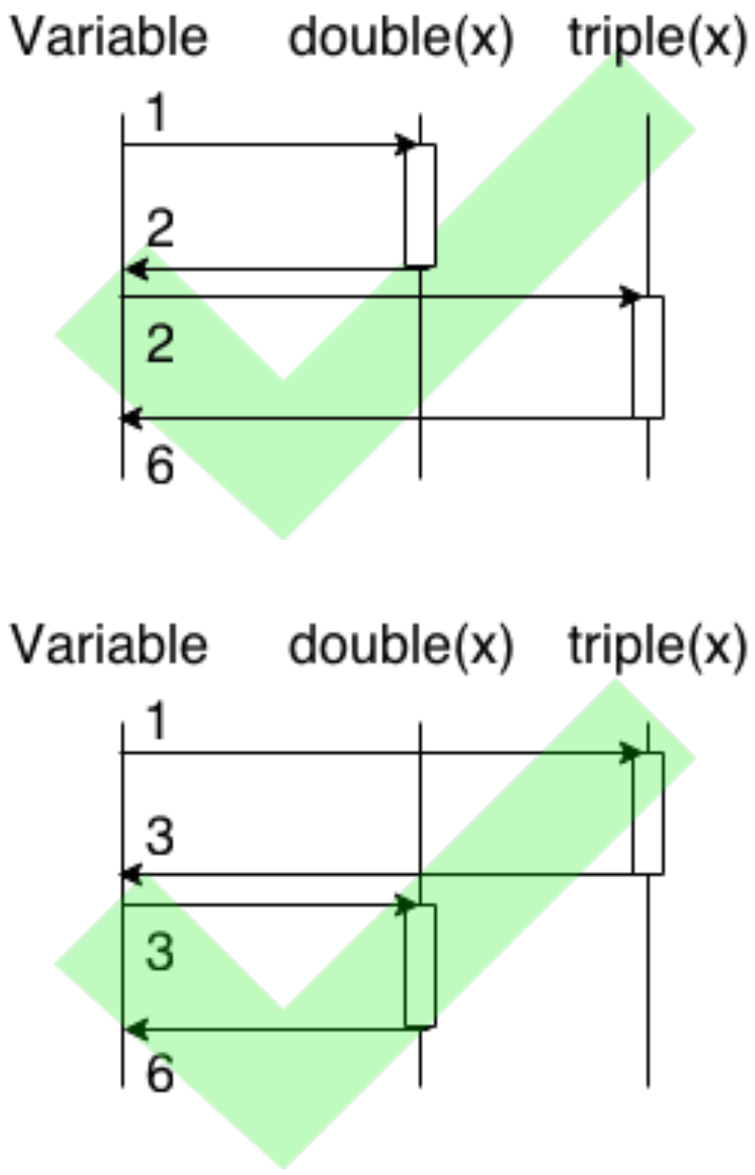


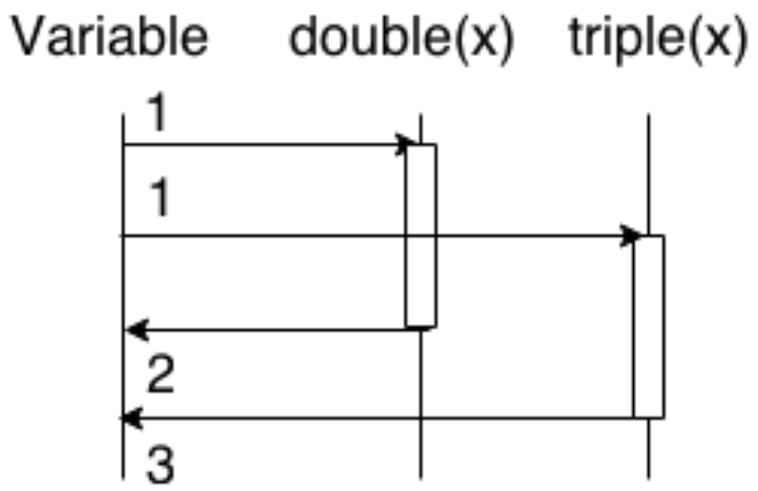
Concurrency



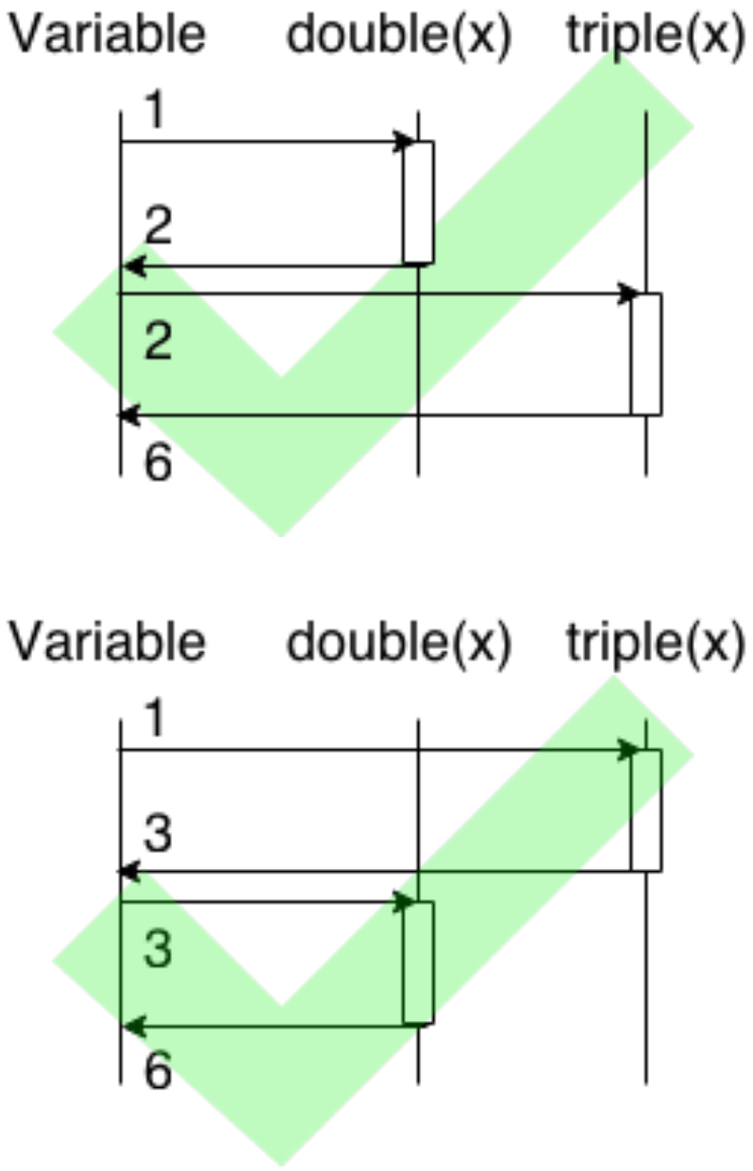


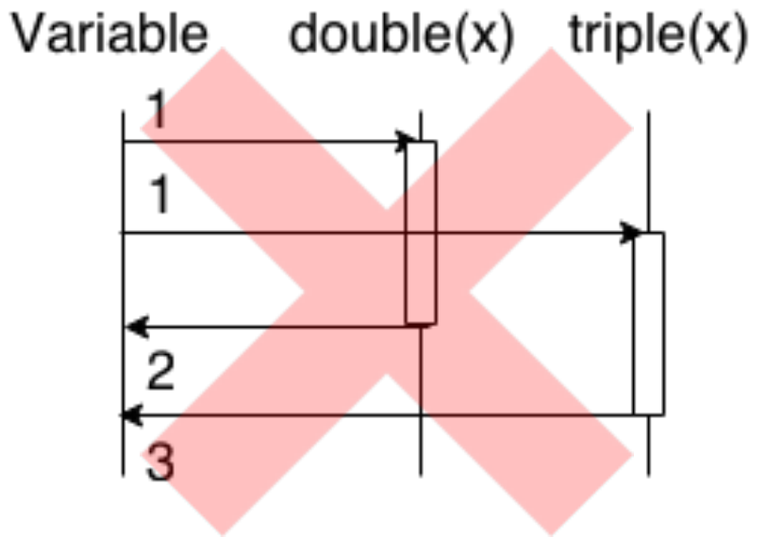
Concurrency

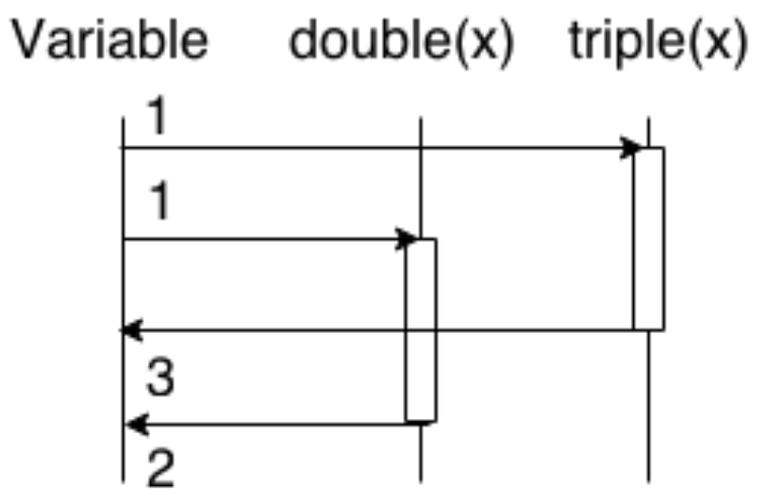




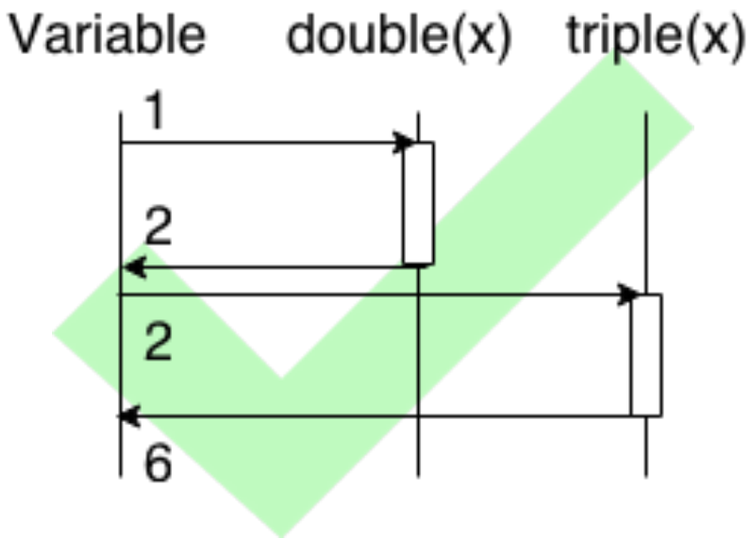
Concurrency

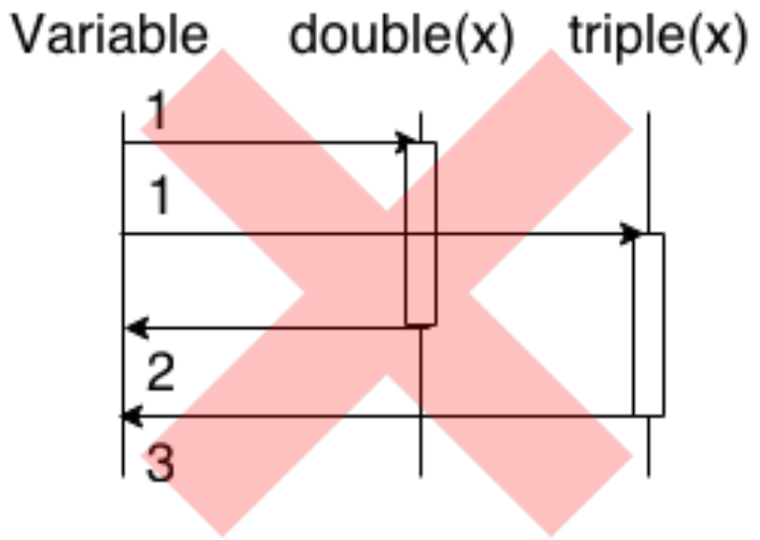


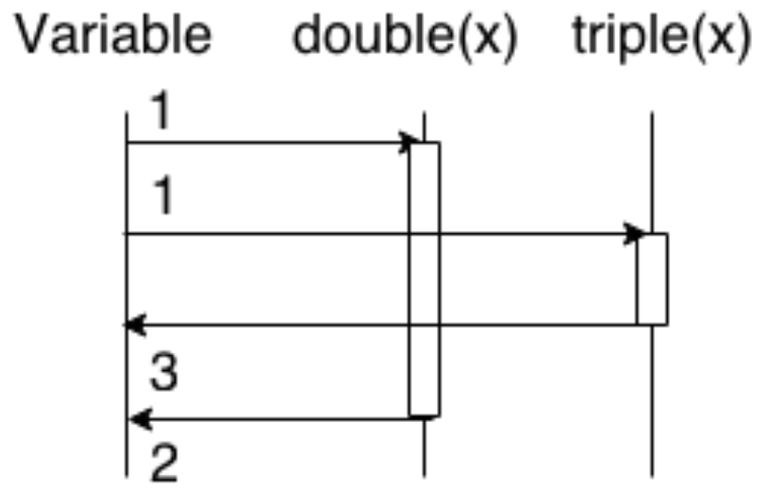


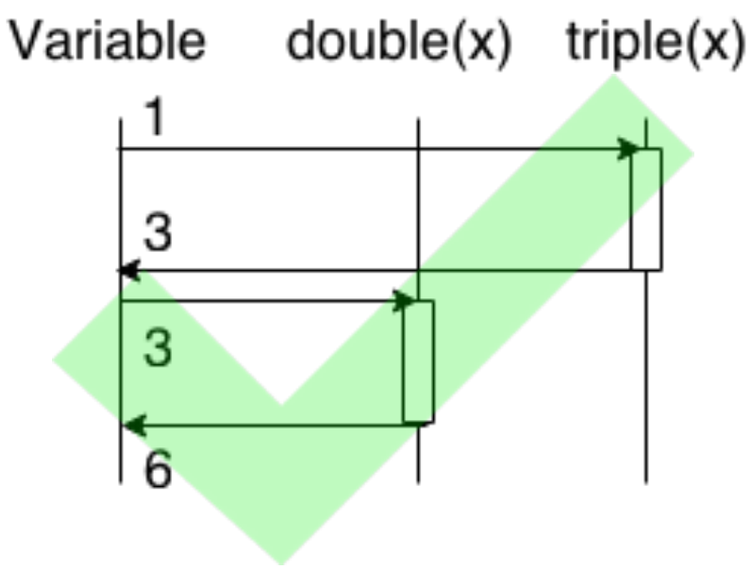


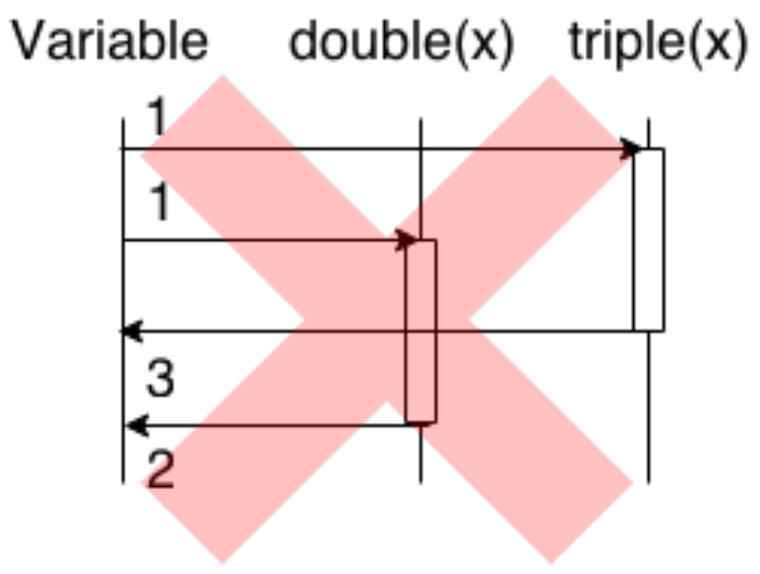
Concurrency



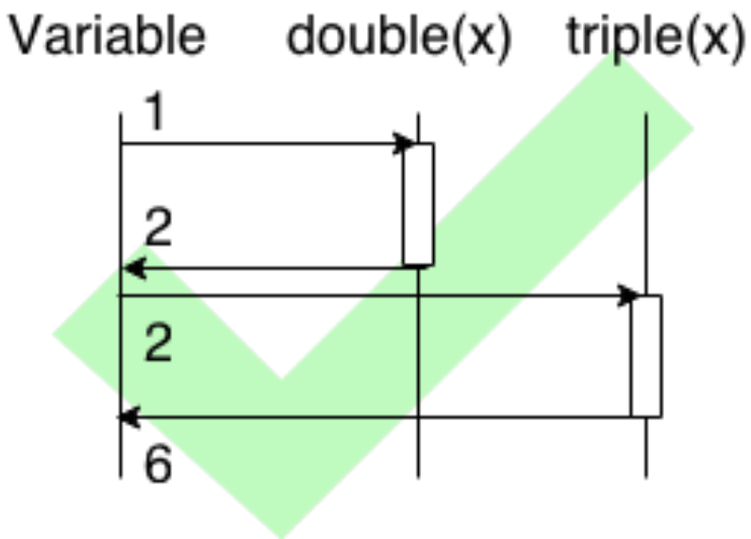


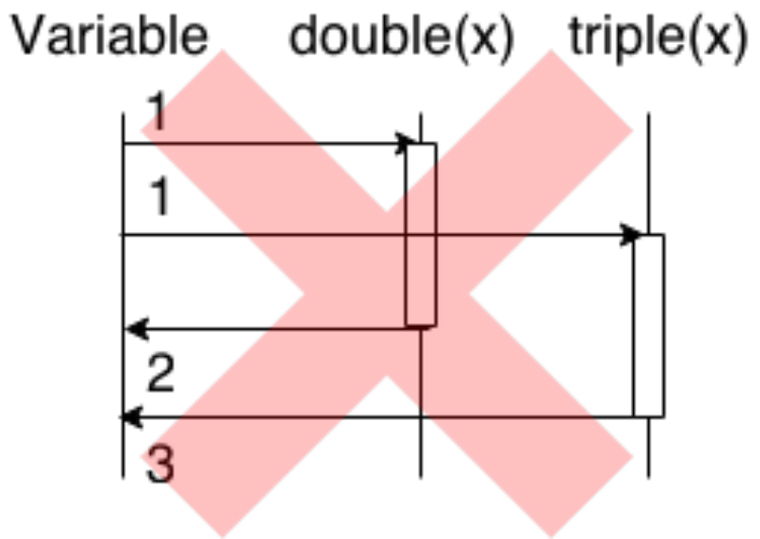


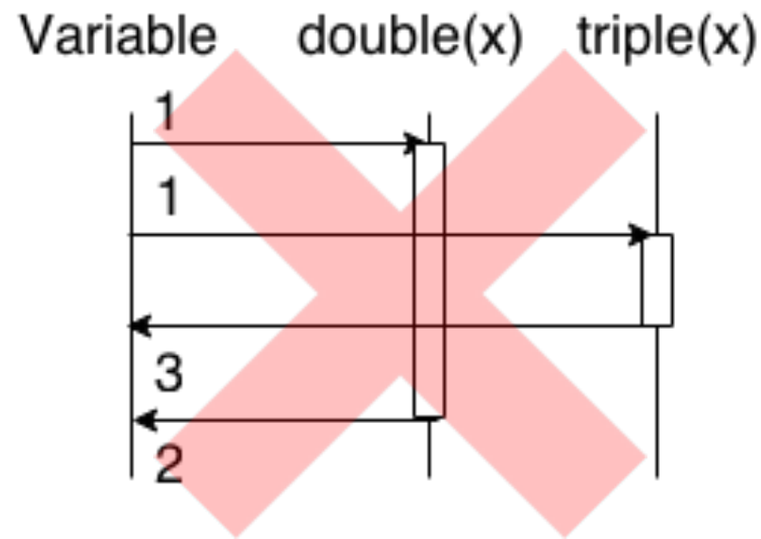


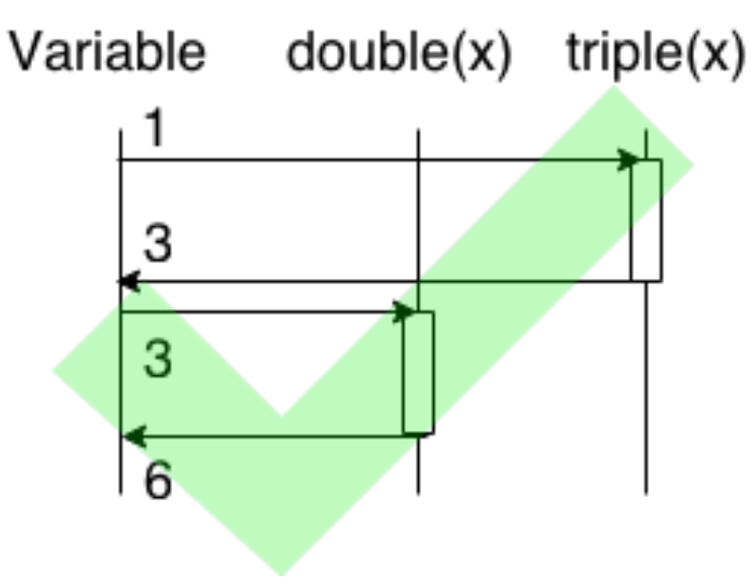


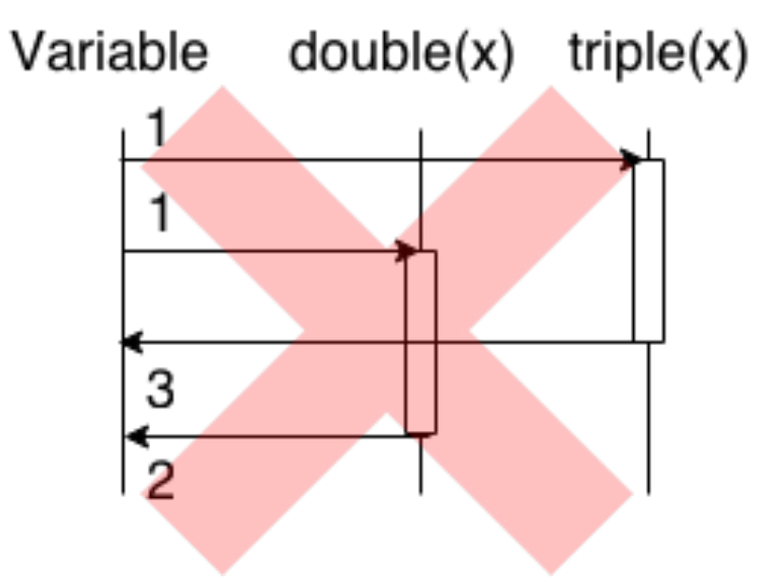
Concurrency

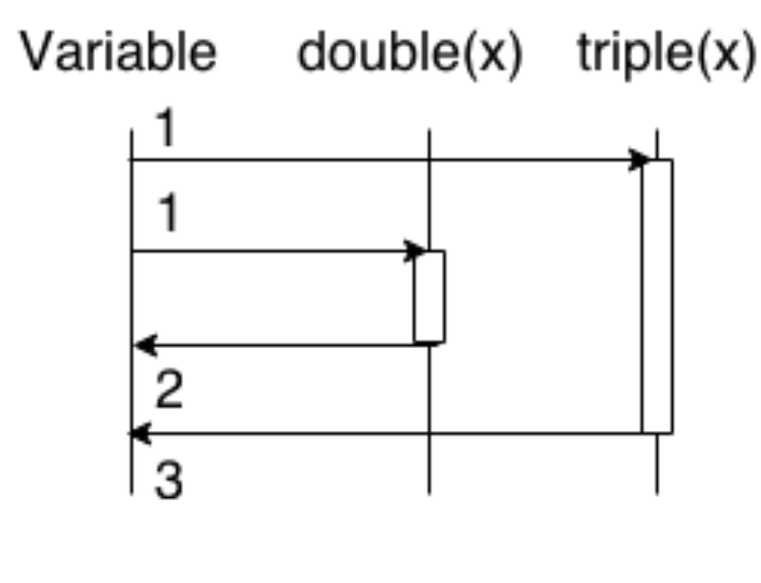




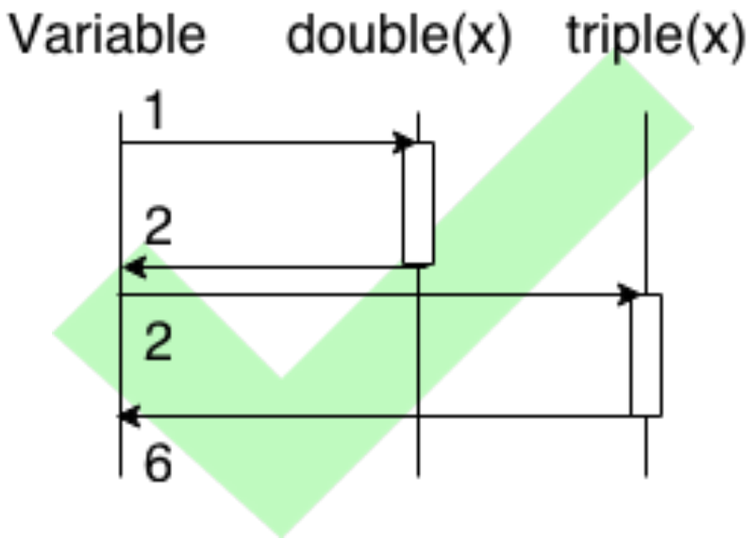


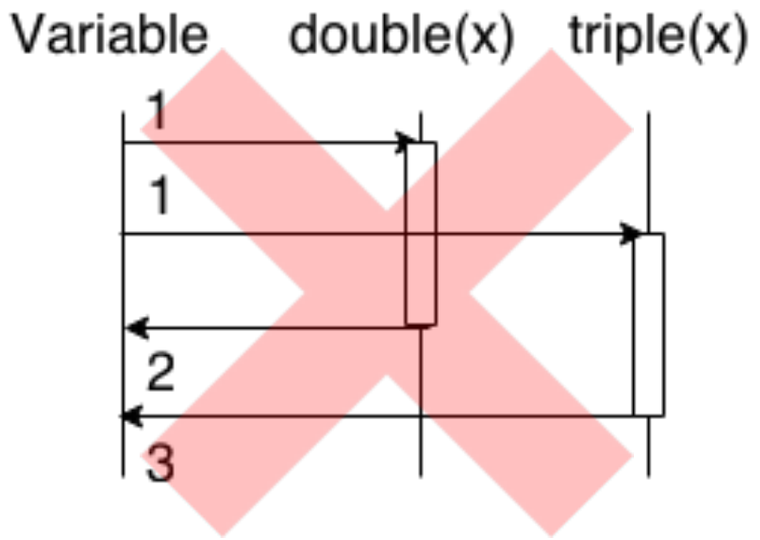


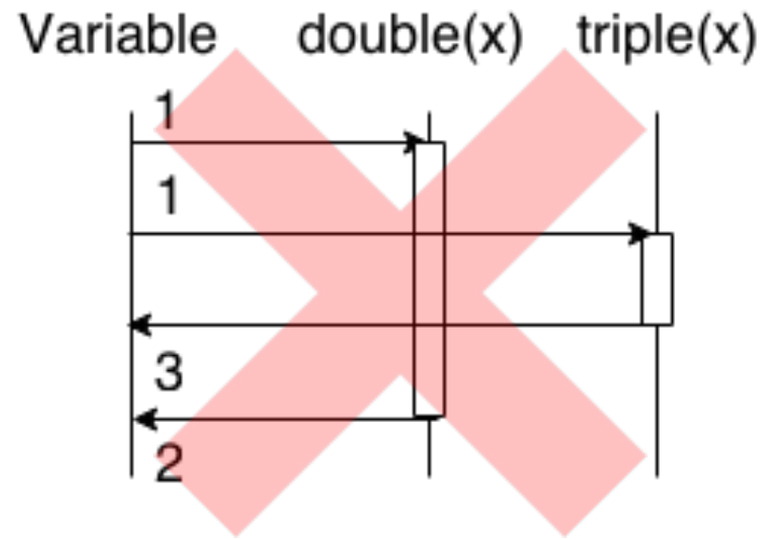


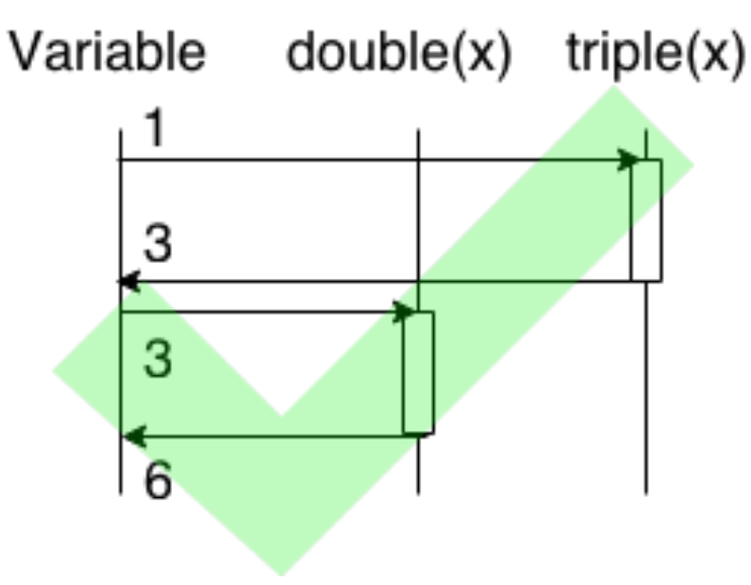


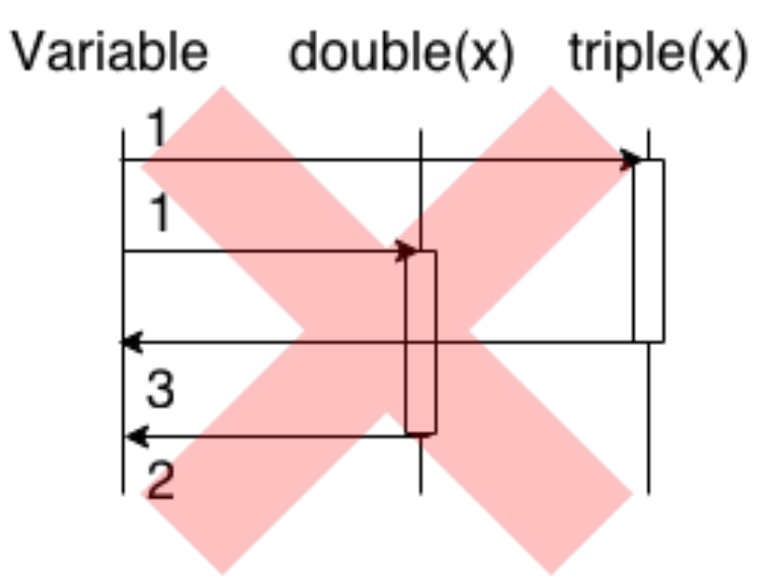
Concurrency

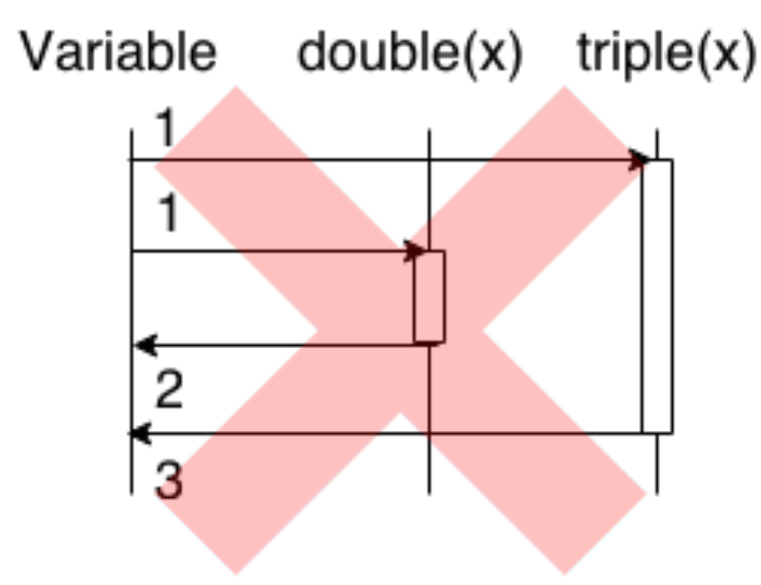












“testing is hopelessly inadequate”

– Edgar Dijkstra, *Notes on Structured Programming*

Get rid of all state?

Mutable Counter

```
procedure int getNextCounter()
   // counter is initialized elsewhere
   count = count + 1
   return count

// Usage:
getNextCounter() // => 1
getNextCounter() // => 2
getNextCounter() // => 3
```

http://shaffner.us/cs/papers/tarpit.pdf

Immutable Counter

```
function (int,int) getNextCounter(int oldCount)
  int result = oldCount + 1
  int newCount = oldCount + 1
  return (newCount, result)

// Usage:
countA, resultA = getNextCounter(0)
countB, resultB = getNextCounter(countA)
countC, resultC = getNextCounter(countB)
resultC // => 3
```

http://shaffner.us/cs/papers/tarpit.pdf

Sometime mutable state is nice…

How can we minimize mutable state?

Essential State

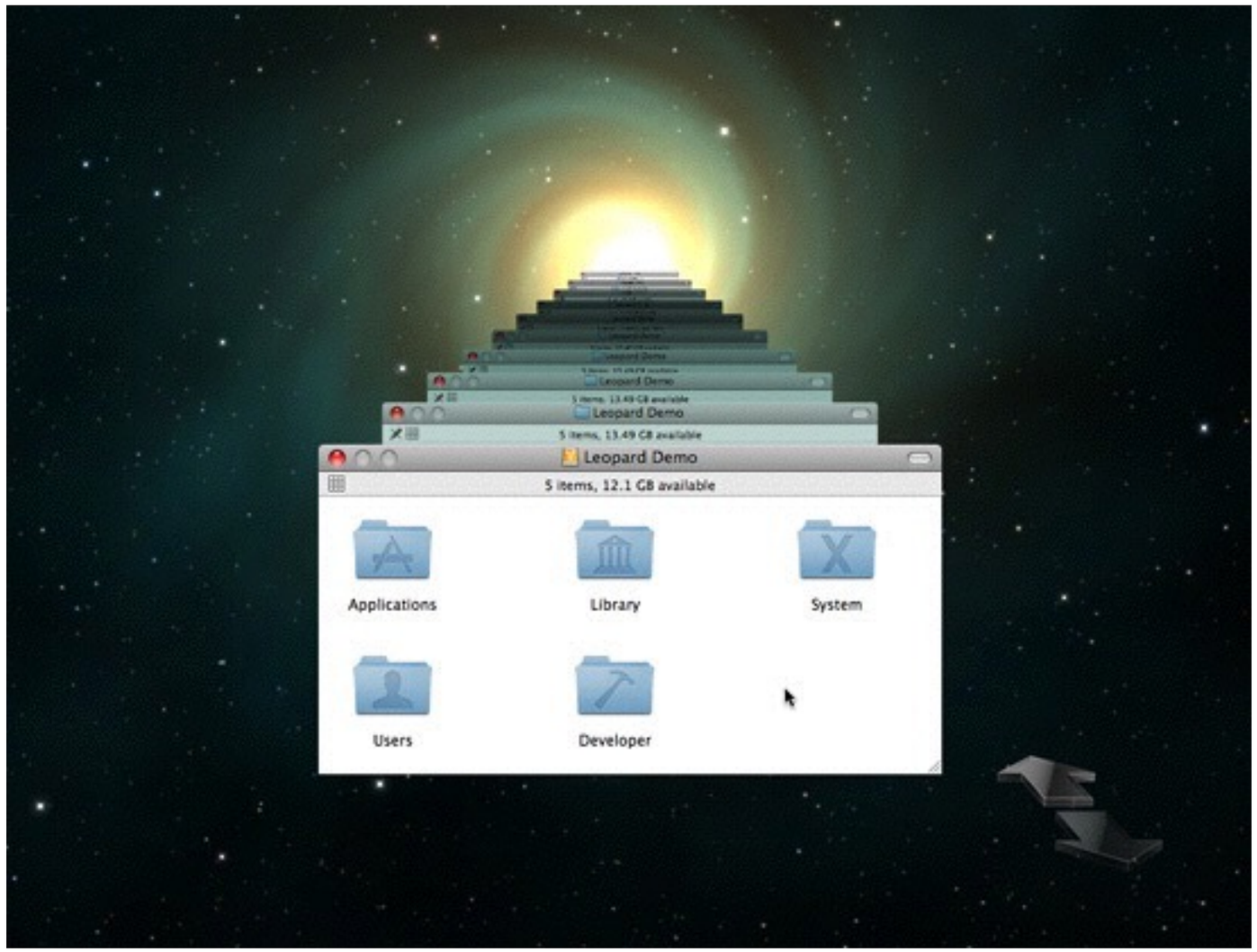
vs.

Derived State

http://shaffner.us/cs/papers/tarpit.pdf

Identity vs. Value

Identity is a succession of values



Versioned Backups

image: http://arstechnica.com/apple/2007/10/mac-os-x-10-5/14/

(code)

“This, milord, is my family's axe. We have owned it for almost nine hundred years, see. Of course, sometimes it needed a new blade. And sometimes it has required a new handle, new designs on the metalwork, a little refreshing of the ornamentation ... but is this not the nine hundred-year-old axe of my family? And because it has changed gently over time, it is still a pretty good axe, y'know. Pretty good.”

– Terry Pratchett, *The Fifth Elephant*

https://github.com/Day8/re-frame#on-derived-values

“No man can cross the same river twice, because neither the man nor the river are the same.”

–Heraclitus

More detail: http://blog.glenjamin.co.uk/names-values-identities-states-and-time

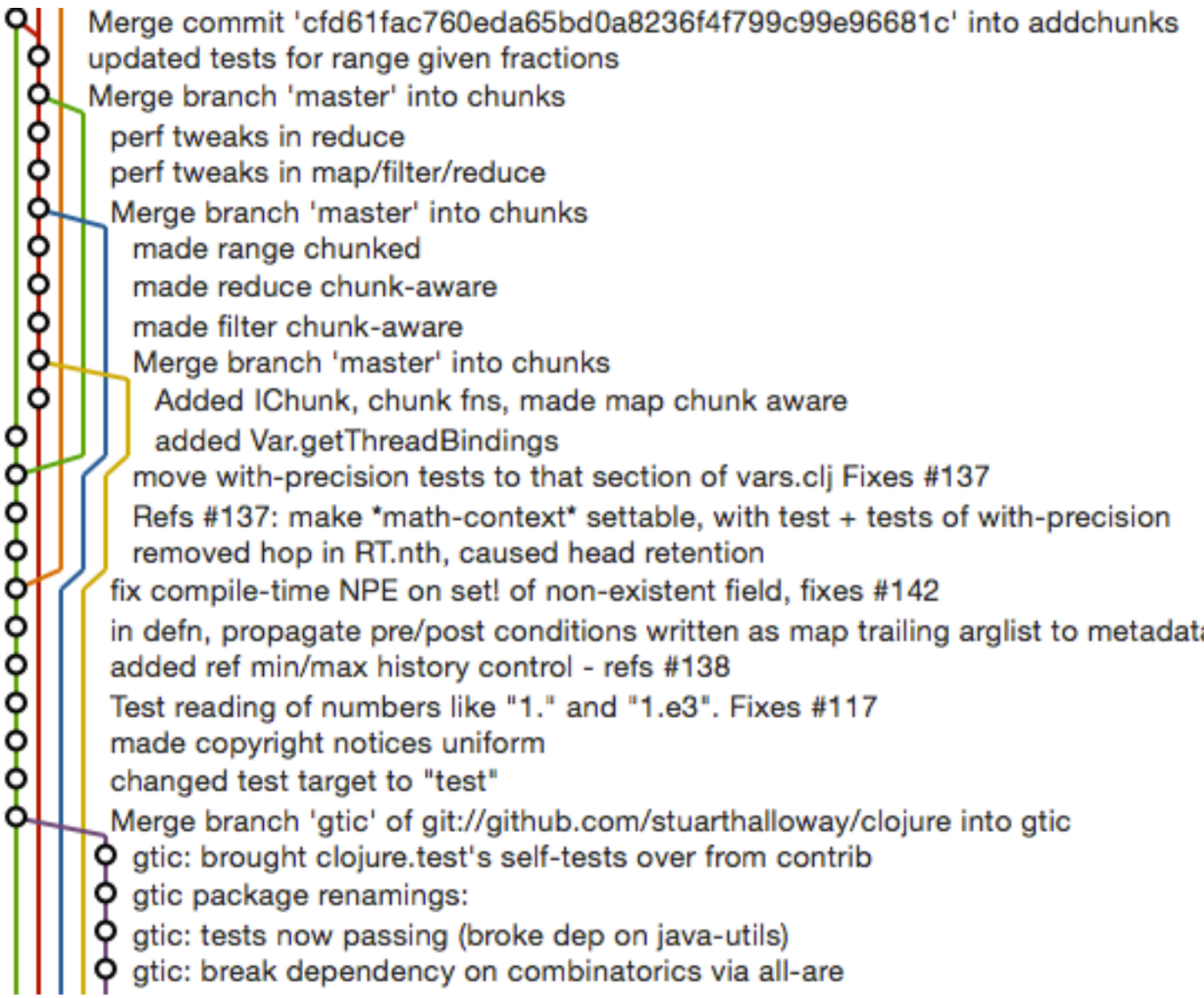
(code)

“[Think] of an experience from your childhood. Something you remember clearly, something you can see, feel, maybe even smell, as if you were really there. After all you really were there at the time, weren't you? How else could you remember it? But here is the bombshell: you weren't there. Not a single atom that is in your body today was there when that event took place. Every bit of you has been replaced many times over (which is why you eat, of course). You are not even the same shape as you were then. The point is that you are like a cloud: something that persists over long periods, while simultaneously being in flux. Matter flows from place to place and momentarily comes together to be you. Whatever you are, therefore, you are not the stuff of which you are made.”

–Steve Grand, *Creation: Life and How to Make It*

More detail: https://stevegrand.wordpress.com/2009/01/12/where-do-those-damn-atoms-go/

Git


Merge commit 'cfd61fac760eda65bd0a8236f4f799c99e96681c' into addchunks
updated tests for range given fractions
Merge branch 'master' into chunks
perf tweaks in reduce
perf tweaks in map/filter/reduce
Merge branch 'master' into chunks
made range chunked
made reduce chunk-aware
made filter chunk-aware
Merge branch 'master' into chunks
Added IChunk, chunk fns, made map chunk aware
added Var.getThreadBindings
move with-precision tests to that section of vars.clj Fixes #137
Refs #137: make *math-context* settable, with test + tests of with-precision
removed hop in RT.nth, caused head retention
fix compile-time NPE on set! of non-existent field, fixes #142
in defn, propagate pre/post conditions written as map trailing arglist to metadata on arglist
added ref min/max history control - refs #138
Test reading of numbers like "1." and "1.e3". Fixes #117
made copyright notices uniform
changed test target to "test"
Merge branch 'gtic' of git://github.com/stuarthalloway/clojure into gtic
gtic: brought clojure.test's self-tests over from contrib
gtic package renamings:
gtic: tests now passing (broke dep on java-utils)
gtic: break dependency on combinatorics via all-are

Let’s make identities that point to successive values!

But deep copying is slow and takes up a lot of space…

Persistent Data Structures

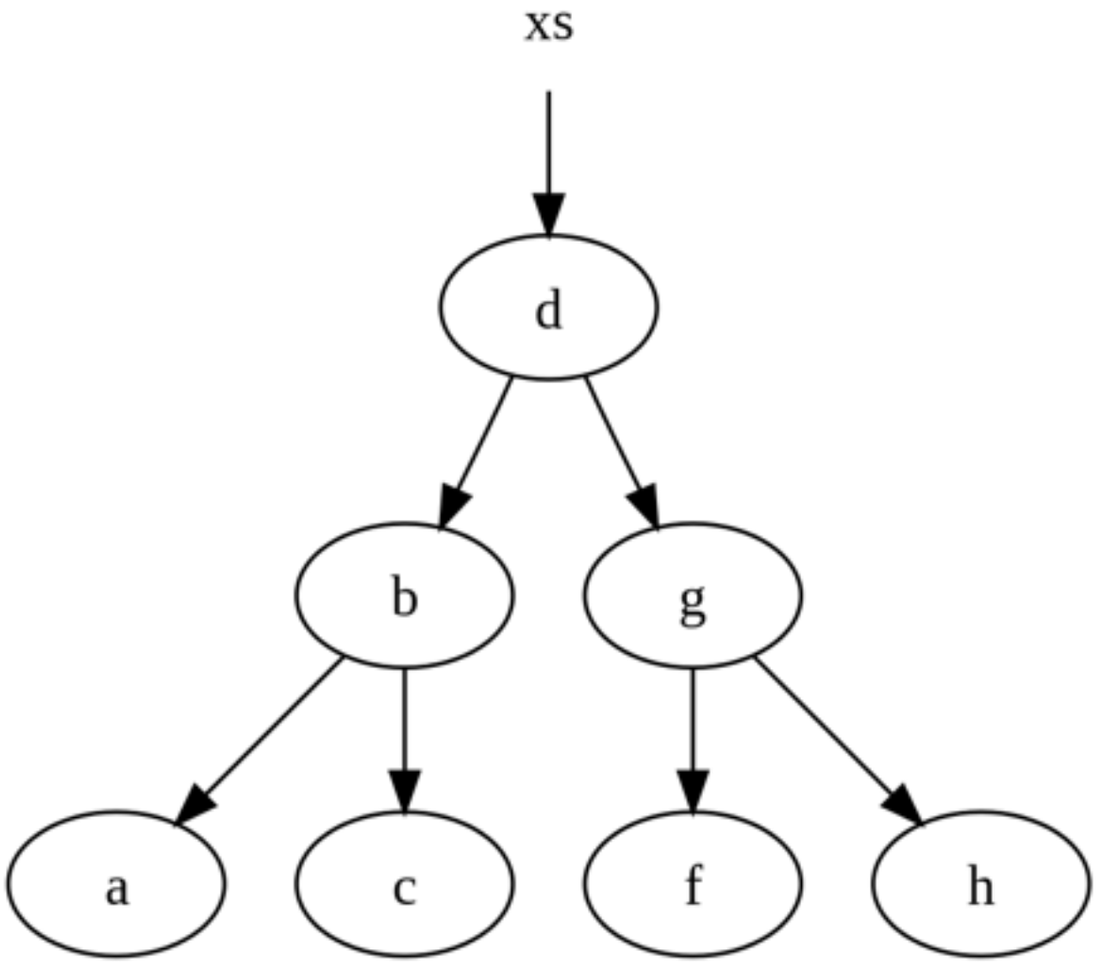


image: https://en.wikipedia.org/wiki/Persistent_data_structure

Persistent Data Structures

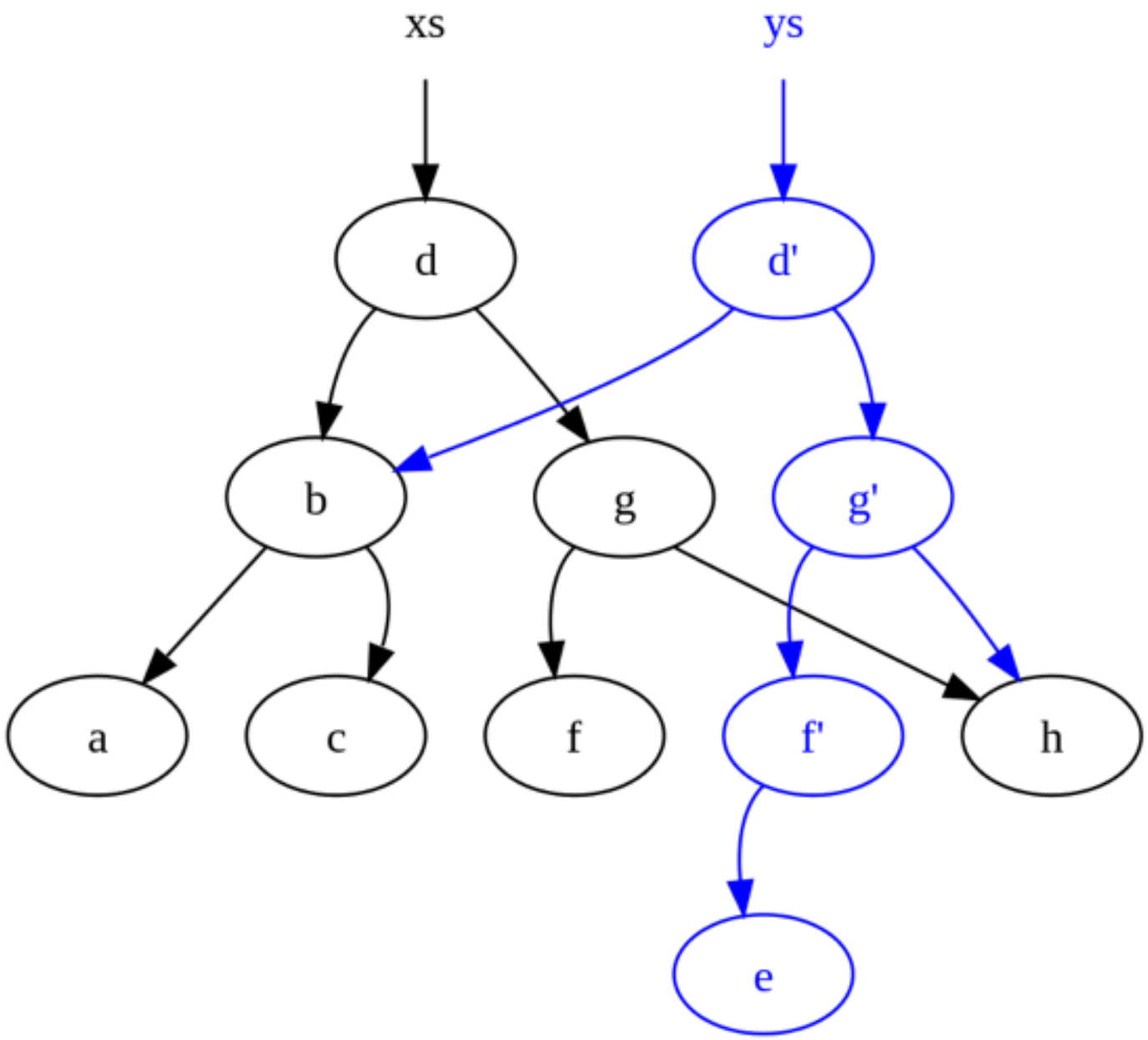


image: https://en.wikipedia.org/wiki/Persistent_data_structure

Benefits

Benefits

- Concurrency: can share state

Benefits

- Concurrency: can share state
- Instant history

Benefits

- Concurrency: can share state
- Instant history
- No defensive copying

Benefits

- Concurrency: can share state
- Instant history
- No defensive copying
- `==` vs. `.equals()`

What can be done now?

1. Write pure functions when possible (don't modify state, only return new versions)

2. Use `map`, `reduce`, `filter`, etc. instead of `for` loops

3. Use persistent data structures when possible

4. Differentiate between inherent state and derived state (make derived state immutable)

5. Highly performance sensitive code? Resort to mutable state

(demo)

Thanks!

@bmaddy

Resources

http://clojure.mn - local Clojure user group

http://4clojure.org - Koan style Clojure puzzles

http://vidku.com - We're hiring - Front end, Back end, iOS, and Android - postings on tech.mn