facebook

# Haxl:
# a Big Hammer for Concurrency

**Simon Marlow**

Facebook

# I/O

It’s slow

It’s hard to test

It’s hard to debug

- Slow
- Hard to test
- Hard to debug

“But I know how to solve these problems!”

# Every language can do this

Python 3

```
r1,r2 = await asyncio.wait([thing1(),thing2()])
```

JavaScript

```
var [r1,r2] = await Promise.all([thing1,thing2])
```

Haskell

```
(r1,r2) <- concurrently thing1 thing2
```

C++

```
std::future<T> f1 = …
std::future<T> f2 = …
r1 = f1.wait()
r2 = f2.wait()
```

# What’s wrong with these?

1. I have to remember to do it
2. I might wait too early
3. Have to fix the awaits when refactoring
4. Concurrency clutters the code
5. Refactoring is harder due to the extra structure

“But.. but.. what about …
*side effects*…”

# What if there are no side effects?

gather data, make decisions, take action

No side-effects in this bit

# What if there are no side effects?

gather data, make decisions, take action

No side-effects in this bit

- How often do we do this?
- e.g. rendering a web page:
  - fetch data, generate HTML = no side effects

When there are no side-effects,
*concurrency is a better default*

# What about the other problems

- How do we test our I/O code?

# What about the other problems

- How do we test our I/O code?

Image: “hope,” pol sifter, https://creativecommons.org/licenses/by/2.0/

## Test Plan

Worked once on my machine

# I/O interacts with the world

So how can we make reliable tests?

# Reserve part of the world for testing

For testing only!

Image: microbiologybytes, Flickr, https://creativecommons.org/licenses/by/2.0/

# Fake the world

Image: microbiologybytes, Flickr, https://creativecommons.org/licenses/by/2.0/

# Mock = substitute API for testing

- But where do we get the test data?
- Writing it manually is hard & error-prone
- Building mocks that record and replay is hard
- Different for each kind of I/O

I want a mock API
with record/replay
for no extra effort.

# What about debugging?

- What if something goes wrong in production?
- How can you reproduce it?

# Logging

Image: “logs”, Anna L Martin, Flickr, CC BY 2.0

# Logging… meh

- I have to remember to do it
- I might not log enough stuff
- round trip: add more logging, wait for another repro
- It clutters the code

# Logging… meh

- I have to remember to do it
- I might not log enough stuff
- round trip: add more logging, wait for another repro
- It clutters the code

Just let me reproduce exactly what happened.

# BIG HAMMERS

# A little digression: Big Hammers

- A technology, library, or abstraction
  - … that solves one or more problems
  - … and solves them for good

# A little digression: Big Hammers

- A technology, library, or abstraction
  - … that solves one or more problems
  - … and solves them for good
- Might be non-trivial to adopt
  - … often requires some effort
  - … but benefits are worth it

# Big Hammers that you might use

- Distributed source control

# Big Hammers that you might use

- Distributed source control
- Garbage collection

# Big Hammers that you might use

- Distributed source control
- Garbage collection
- Language-independent RPC (Thrift etc.)

# Haxl

# Haxl

- **Haxl** is a Haskell library
  - Provides an abstraction over concurrent I/O
- Works best with **ApplicativeDo,** a compiler extension
  - Added to GHC 8.0 (released 2016)

# Example: update script

# Example: update script

```
getLatestVersion    :: Haxl Version
```

query the latest version of the software

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
```

query the set of hosts

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
```

get the version installed on a host

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()
```

update software on a host to this version

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()

do
  latest <- getLatestVersion
```

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()

do
  latest <- getLatestVersion
  hosts <- getHosts
```

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()

do
  latest <- getLatestVersion
  hosts <- getHosts
  installed <- mapM getInstalledVersion hosts
```

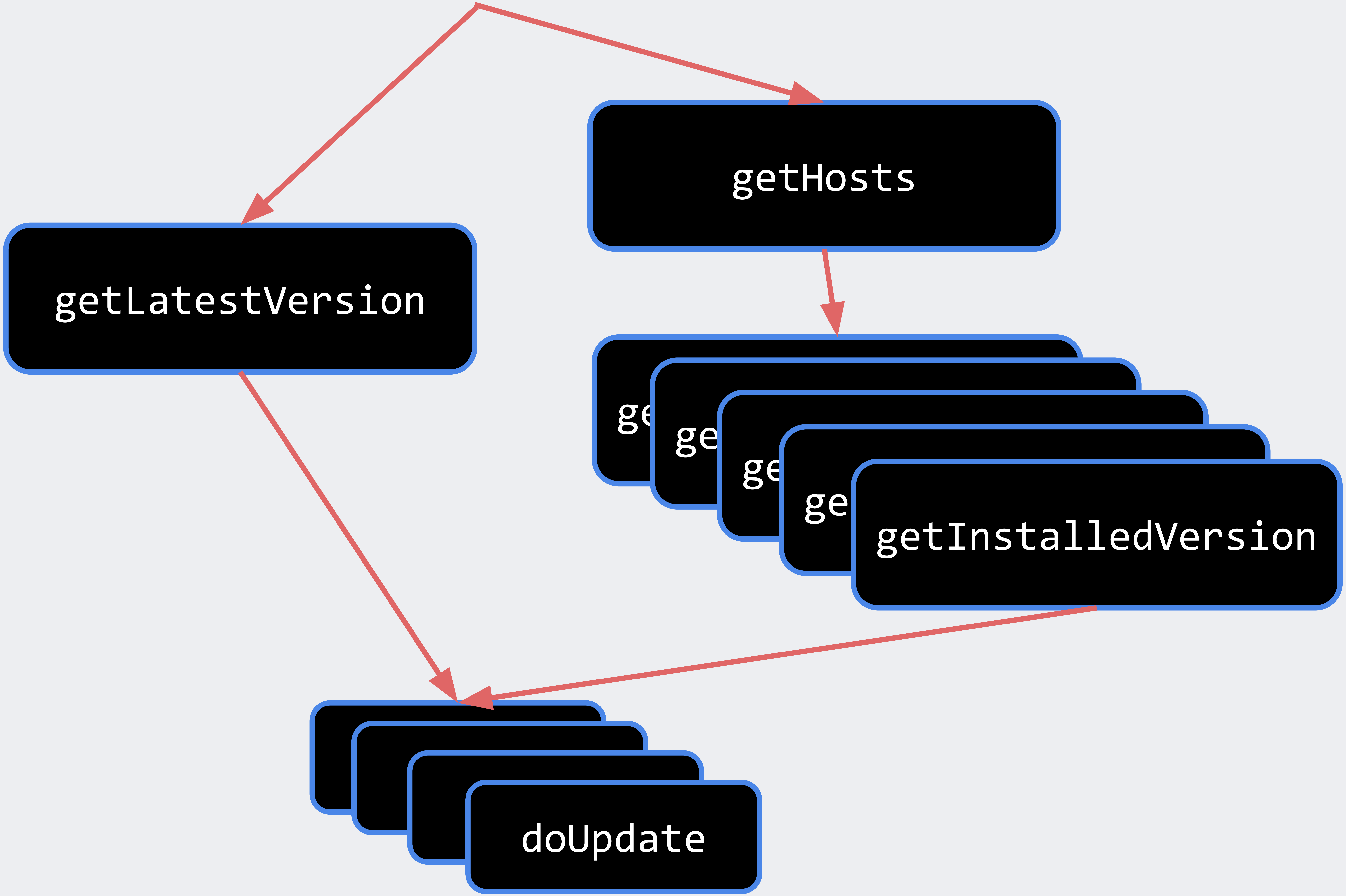
# Example: update script

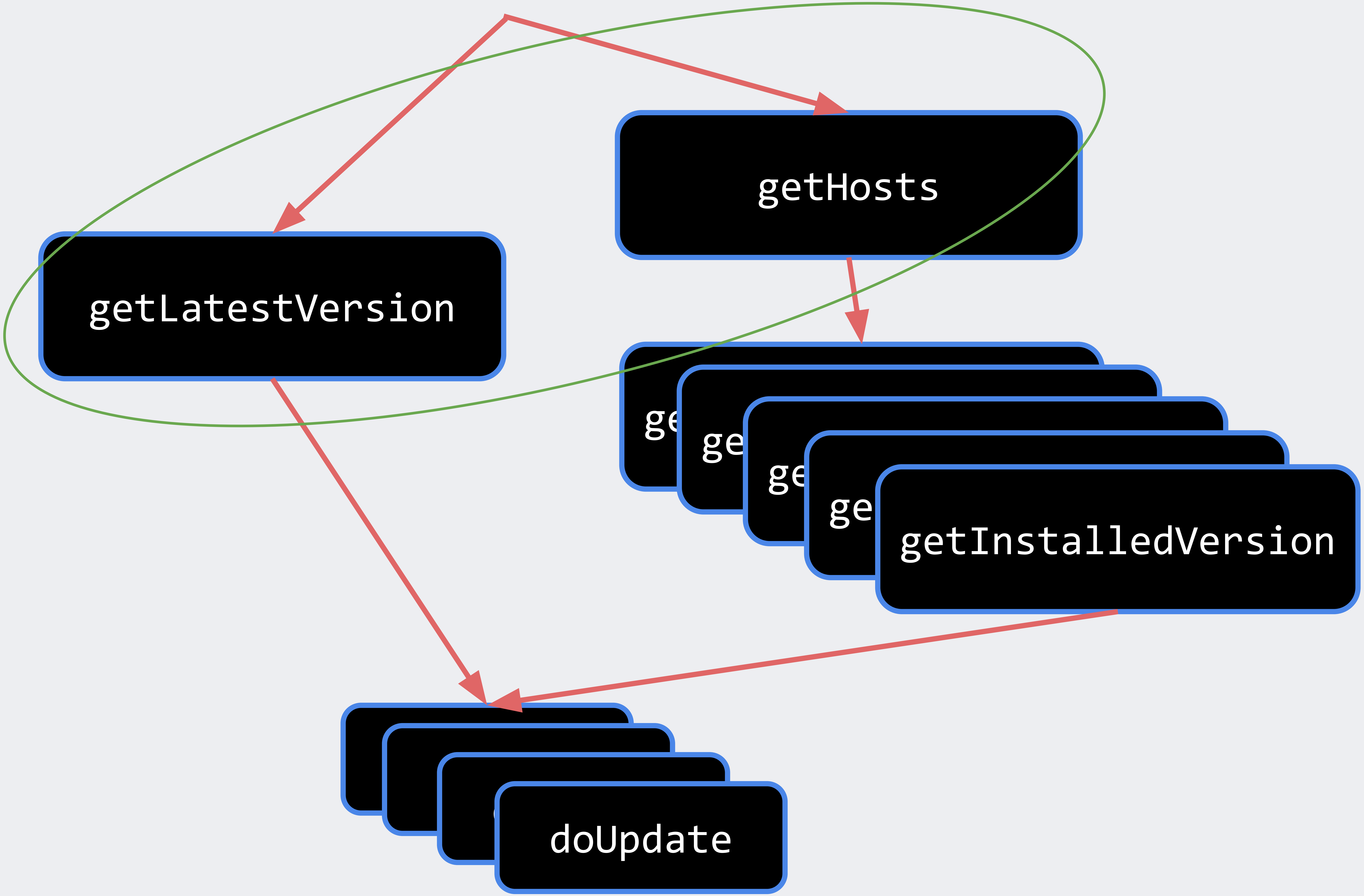
```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()

do
  latest <- getLatestVersion
  hosts <- getHosts
  installed <- mapM getInstalledVersion hosts
  let updates = [ h | (h,v) <- zip hosts installed, v < latest ]
```

# Example: update script

```
getLatestVersion    :: Haxl Version
getHosts            :: Haxl [Host]
getInstalledVersion :: Host -> Haxl Version
updateTo            :: Version -> Host -> Haxl ()

do
  latest <- getLatestVersion
  hosts <- getHosts
  installed <- mapM getInstalledVersion hosts
  let updates = [ h | (h,v) <- zip hosts installed, v < latest ]
  mapM_ (updateTo latest) updates
```

Some parts of this script could run in parallel. Which parts?


getHosts
getLatestVersion
ge
ge
ge
ge
getInstalledVersion
doUpdate



getHosts

getLatestVersion

ge

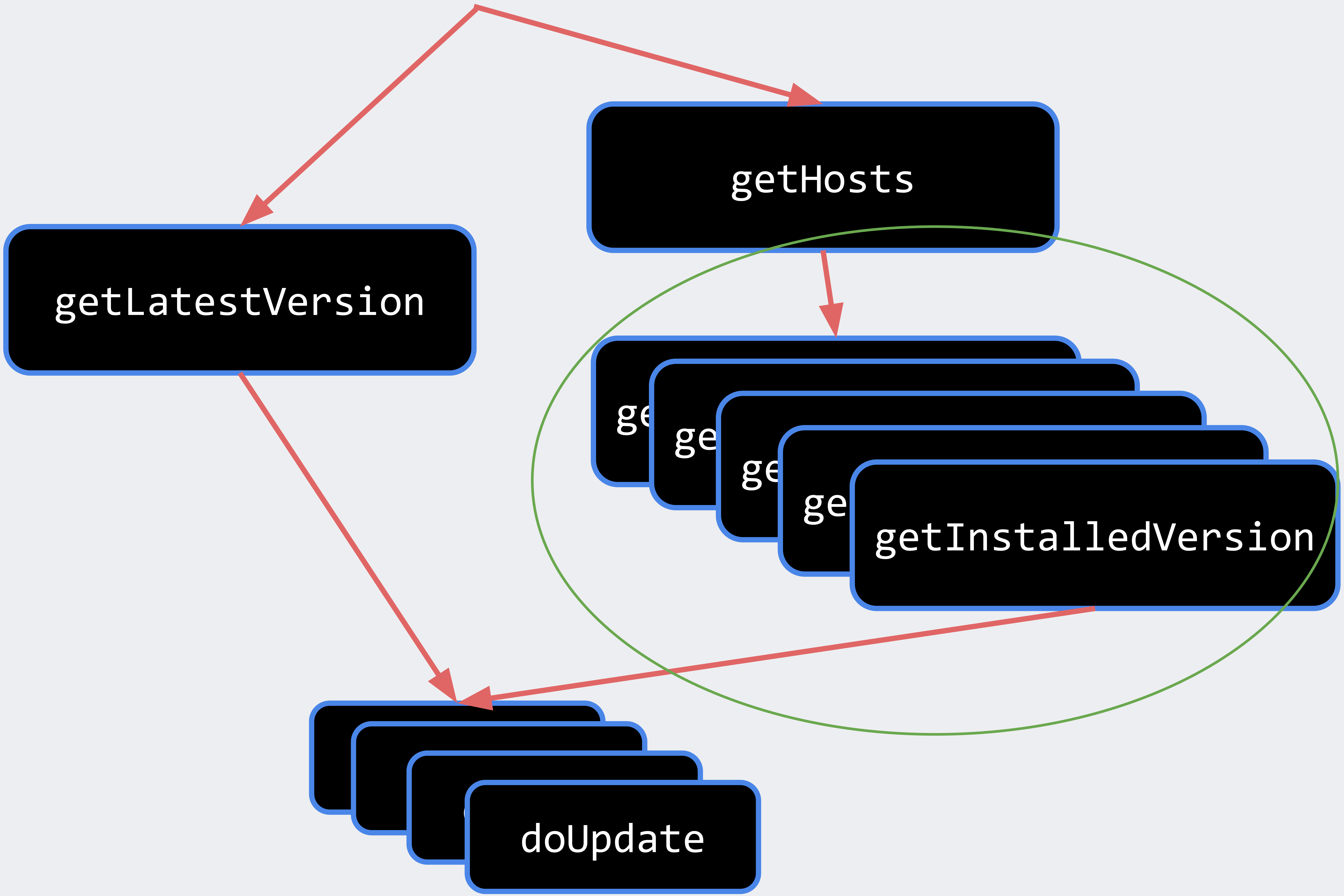
ge

ge

ge

getInstalledVersion

doUpdate

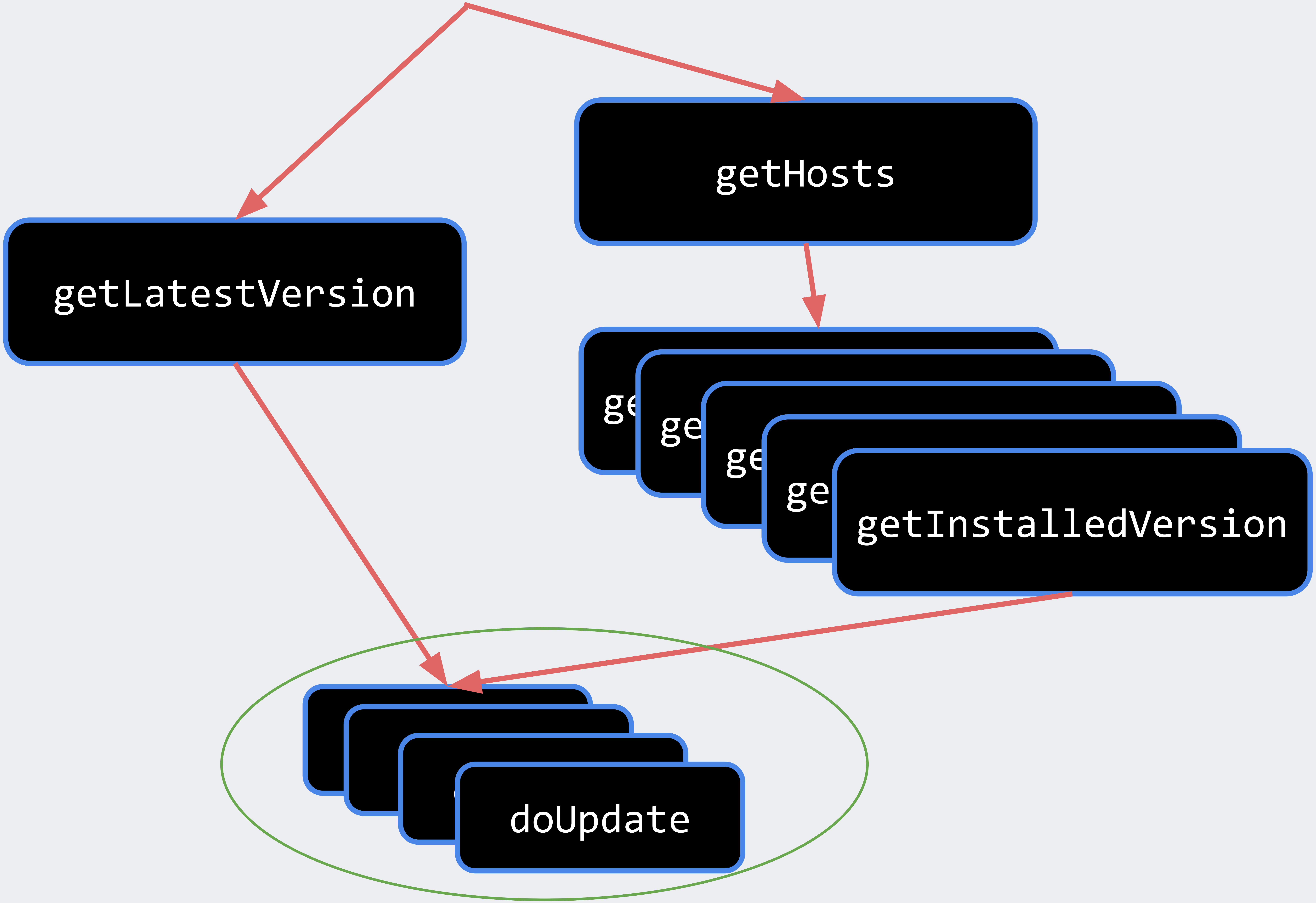


getHosts

getLatestVersion

getInstalledVersion

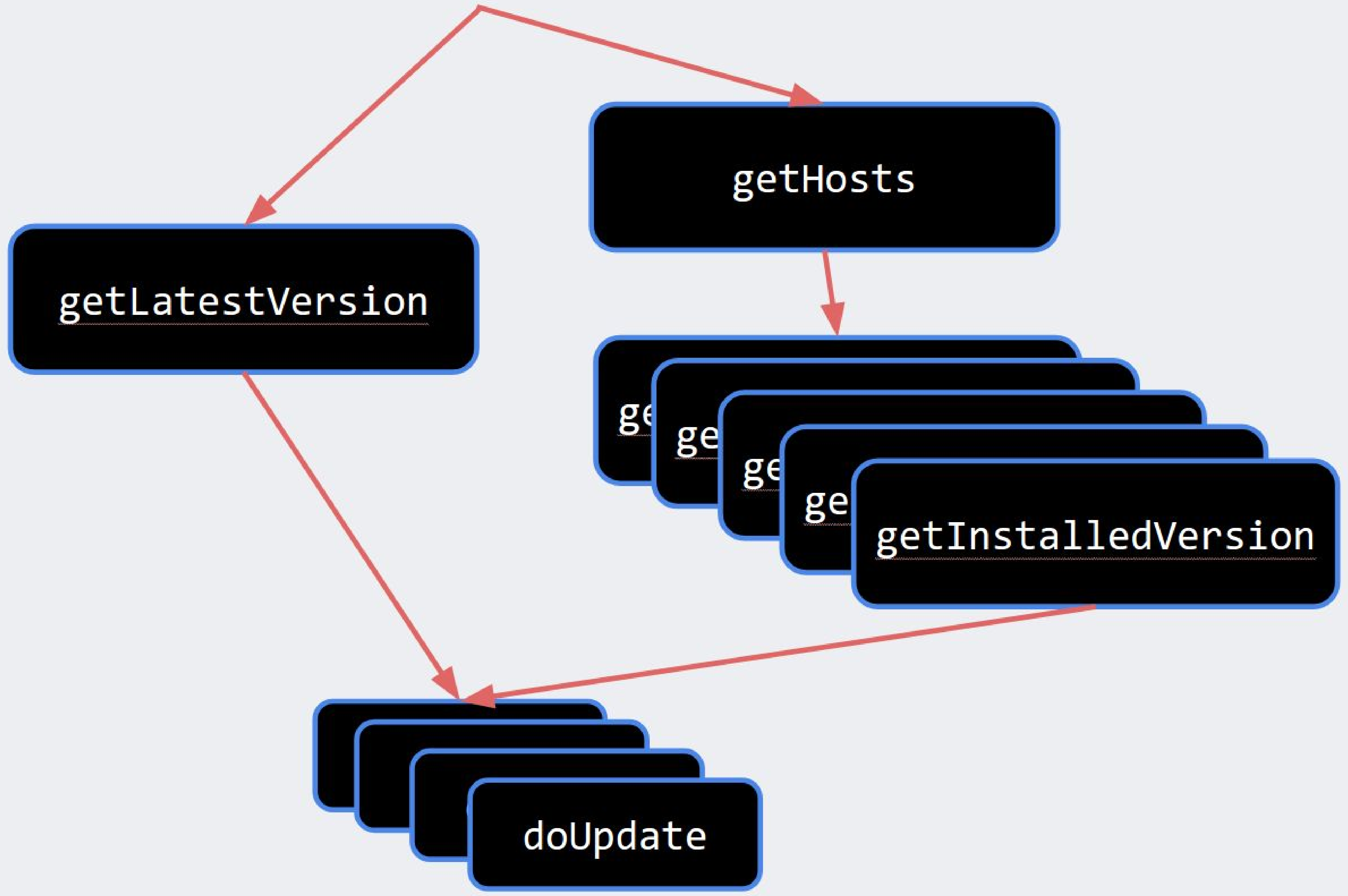
doUpdate


getHosts
getLatestVersion
getInstalledVersion
doUpdate

# Given this script:

```
do
  latest <- getLatestVersion
  hosts <- getHosts
  installed <- mapM getInstalledVersion hosts
  let updates = [ h | (h,v) <- zip hosts installed, v < latest ]
  mapM_ (updateTo latest) updates
```

# Haxl will do this:



# How?

- **Data dependencies (only) determine ordering**
  - independent computations happen in parallel
  - no explicit concurrency constructs

# How?

- **Data dependencies (only) determine ordering**
  - independent computations happen in parallel
  - no explicit concurrency constructs

- We just wrote the obvious code
  - Parallelism was no extra effort

|  | **Abstracts away from….** |
|---|---|
| **Garbage Collection** | Memory management |
| **Haxl** | Concurrency |

# To put it another way...

- We **flipped the default** from sequential to concurrent.

# To put it another way...

- We **flipped the default** from sequential to concurrent.
- Use it when concurrent is the right default!

# How does it work?

# The main problem

- Data-dependencies aren’t first-class
  - so we need compiler support
- We don’t want to build Haxl into the compiler
  - so we searched for a more general solution...

# Example

```
do
  latest <- getLatestVersion
  hosts <- getHosts
  ...
```

These two statements are independent, but only the compiler can know this.

# Example

```
do
  latest <- getLatestVersion
  hosts <- getHosts
  ...
```

**do** is just syntactic sugar:

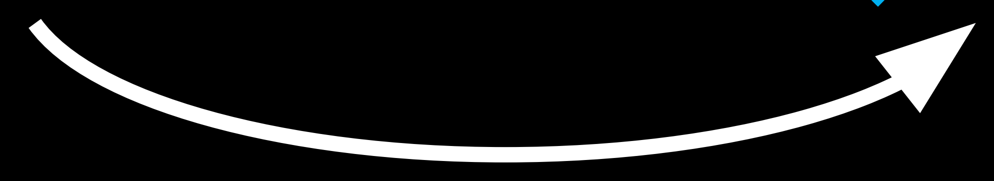
```
getLatestVersion
  >>=
  (\latest ->
     getHosts
       >>=
       (\hosts -> … )
  )
```

The monad bind operator

Every monad implements >>=

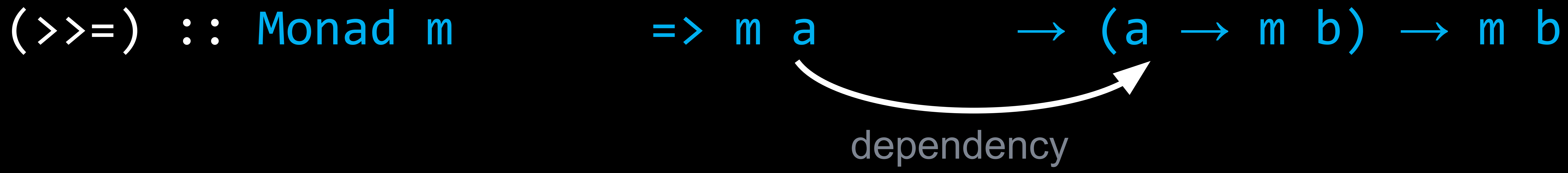
# But we have already lost...

```
(>>=) :: Monad m       => m a       → (a → m b) → m b
```



dependency

# But we have already lost...



```
(>>=) :: Monad m => m a → (a → m b) → m b
```

dependency

>>= combines things sequentially

# This can *only* be sequential:

```
getLatestVersion
  >>=
  (\latest ->
     getHosts
       >>=
       (\hosts -> … )
  )
```

We need to use a different abstraction

# Applicative

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

Applicative “ap” (or “splat”)

# Applicative

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

Applicative “ap” (or “splat”)

<*> combines things in parallel (given a suitable implementation)

# Applicative

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

Applicative “ap” (or “splat”)

```
(<*>) :: Applicative f => f (a → b) → f a       → f b
```

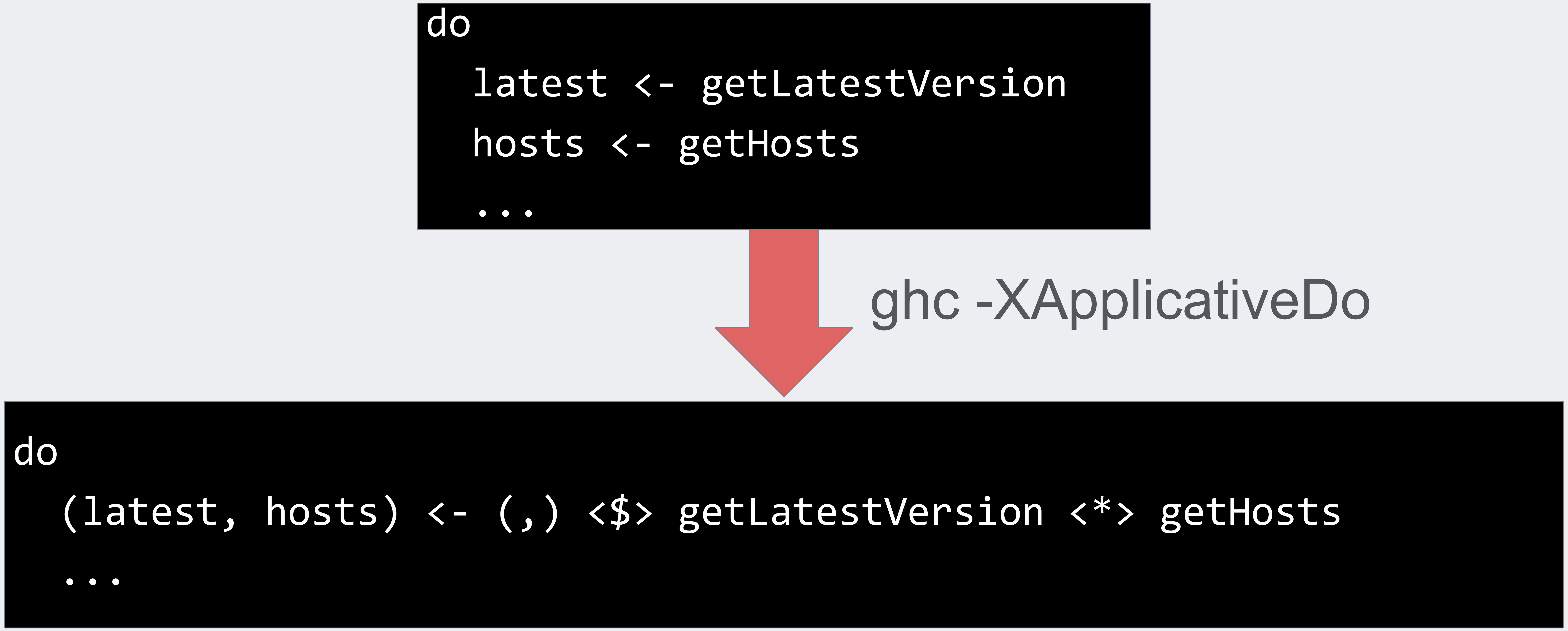
independent

# Applicative

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

But I don’t want to write this by hand

# ApplicativeDo



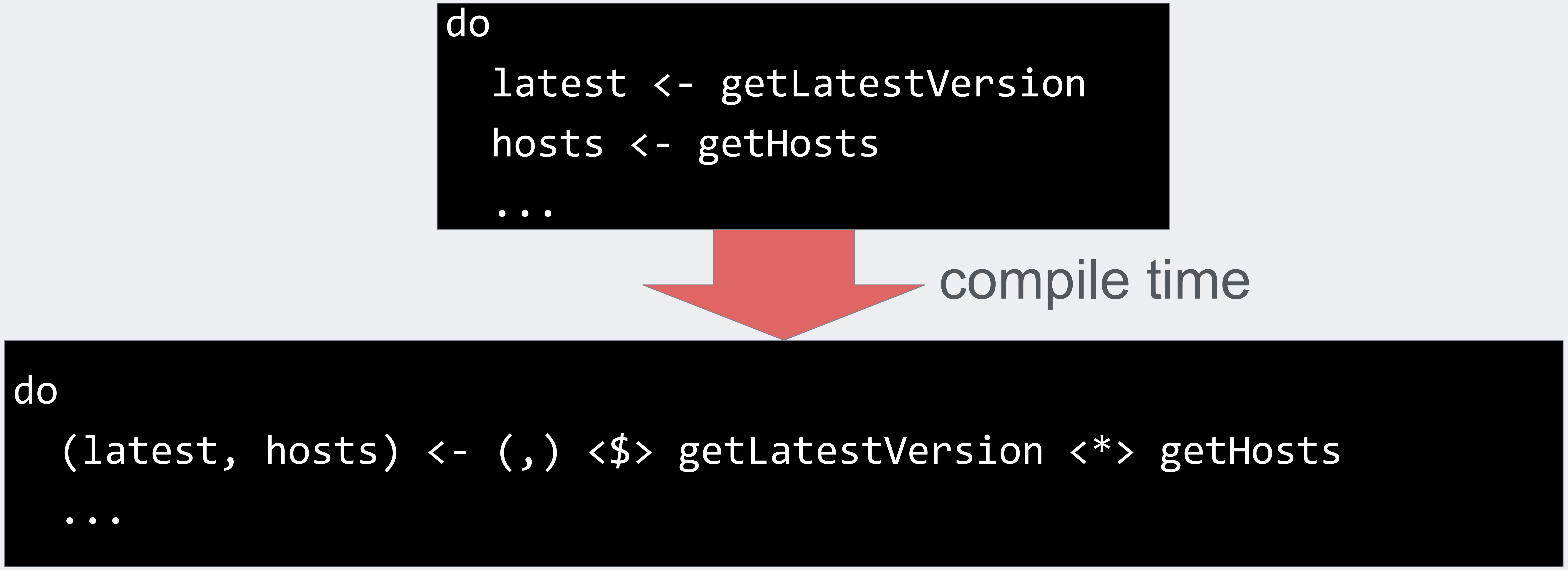
```
do
  latest <- getLatestVersion
  hosts <- getHosts
  ...
```

ghc -XApplicativeDo

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

Haxl library implements <*>

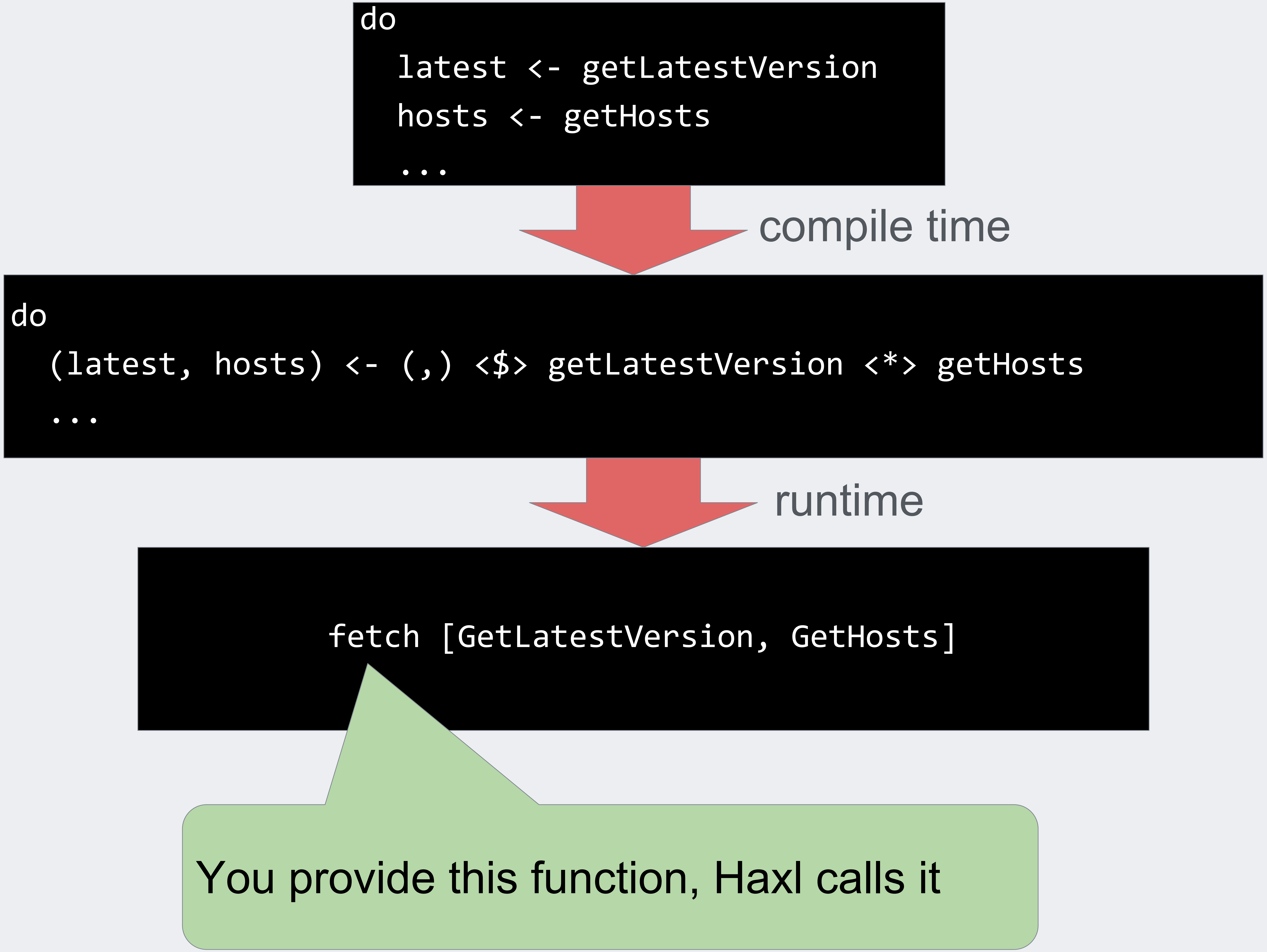
# Implements <*> to do *what*?



```
do
  latest <- getLatestVersion
  hosts <- getHosts
  ...
```

compile time

```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```



```
do
  latest <- getLatestVersion
  hosts <- getHosts
  ...
```

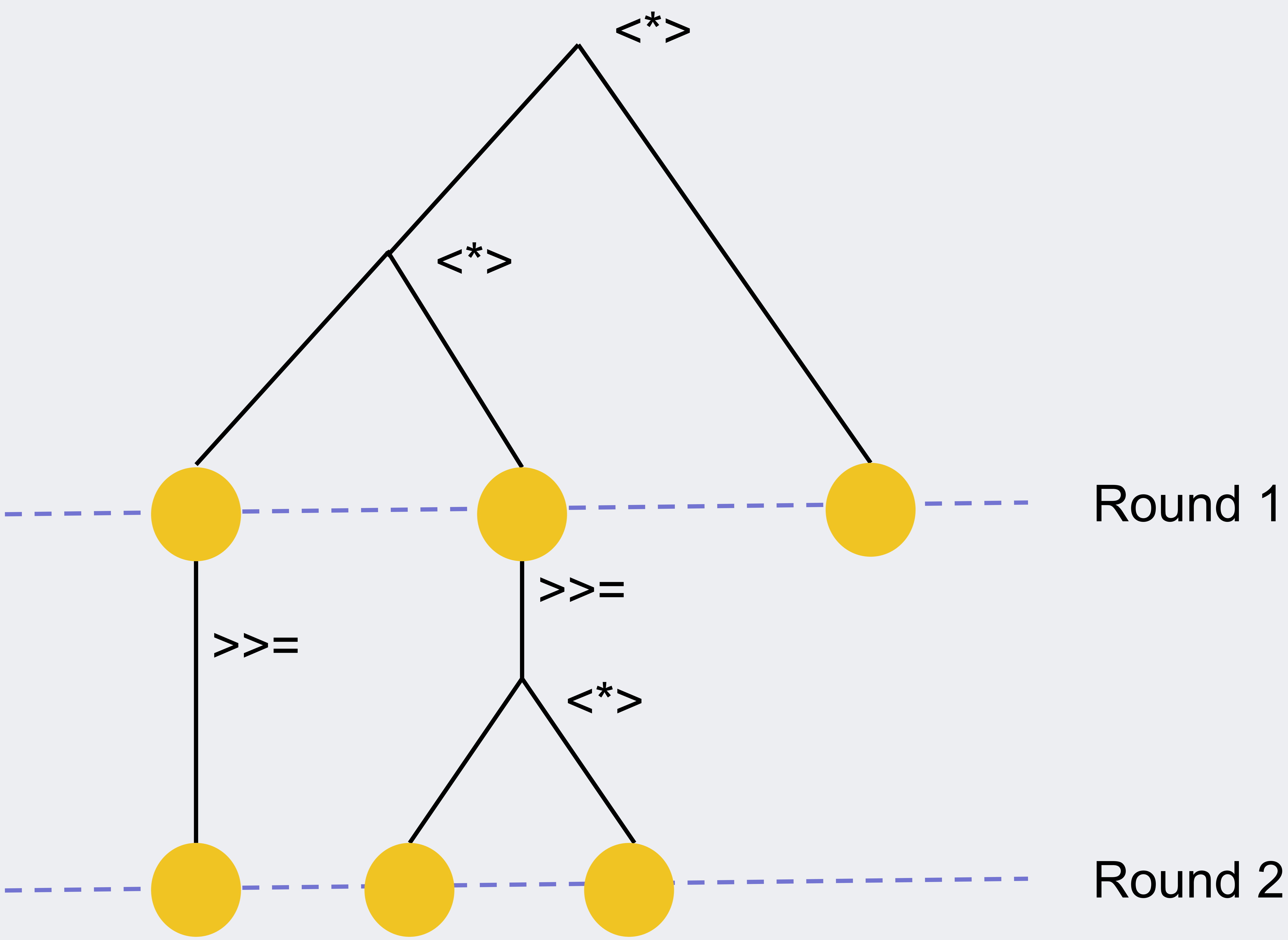
compile time

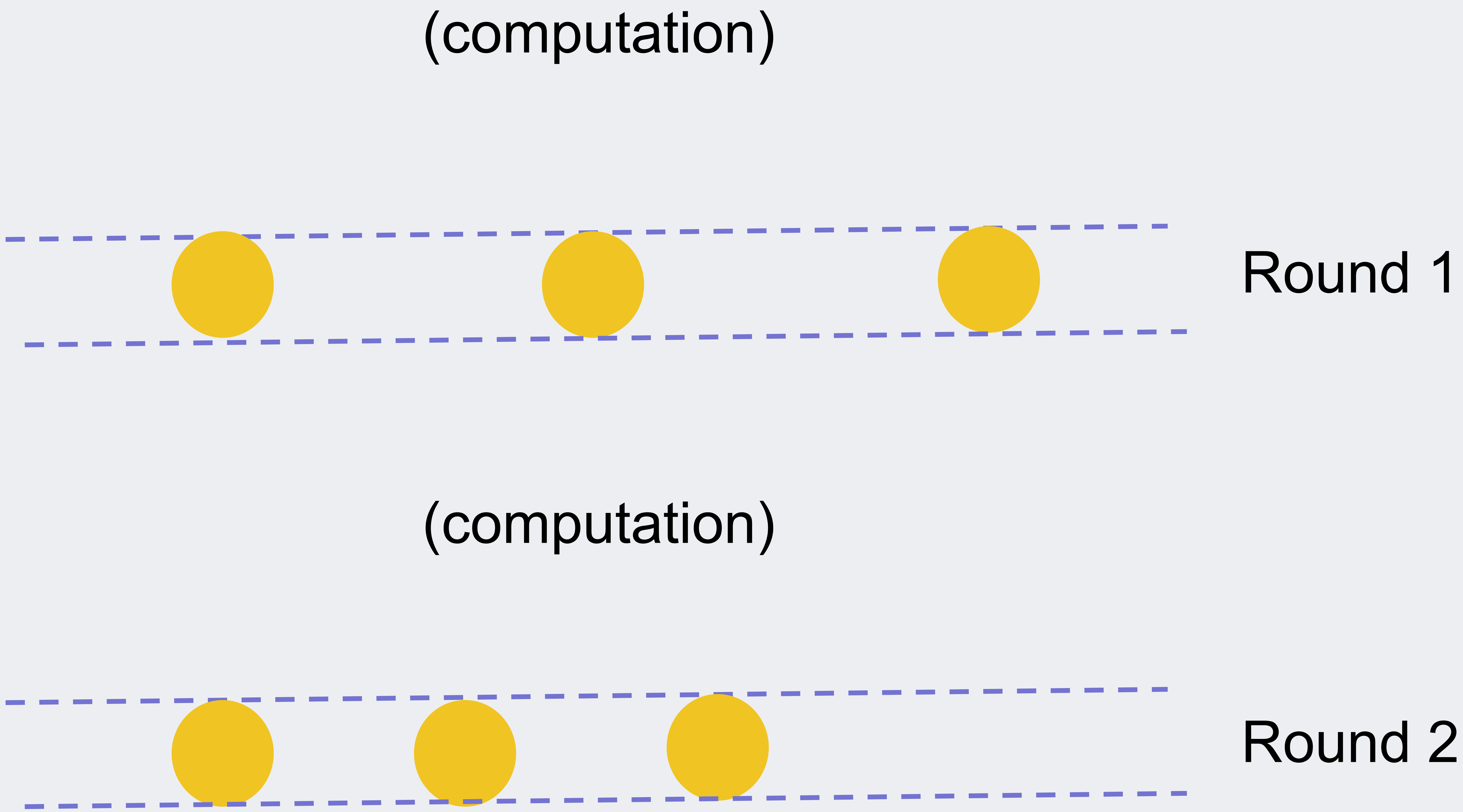
```
do
  (latest, hosts) <- (,) <$> getLatestVersion <*> getHosts
  ...
```

runtime

```
fetch [GetLatestVersion, GetHosts]
```

You provide this function, Haxl calls it


<*>
<*>
Round 1
>>=
>>=
<*>
Round 2

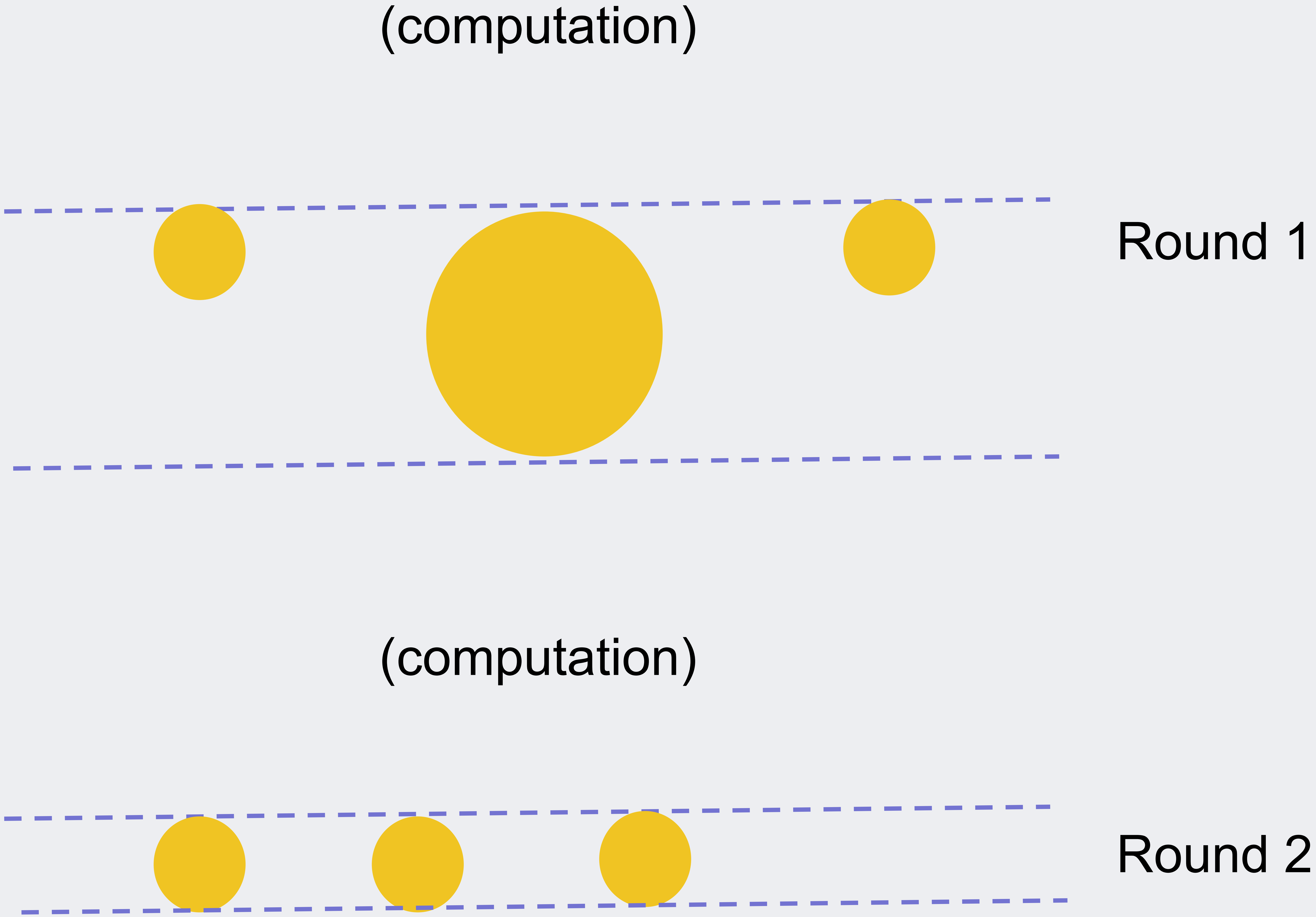


(computation)

Round 1

(computation)

Round 2



(computation)

Round 1

(computation)

Round 2

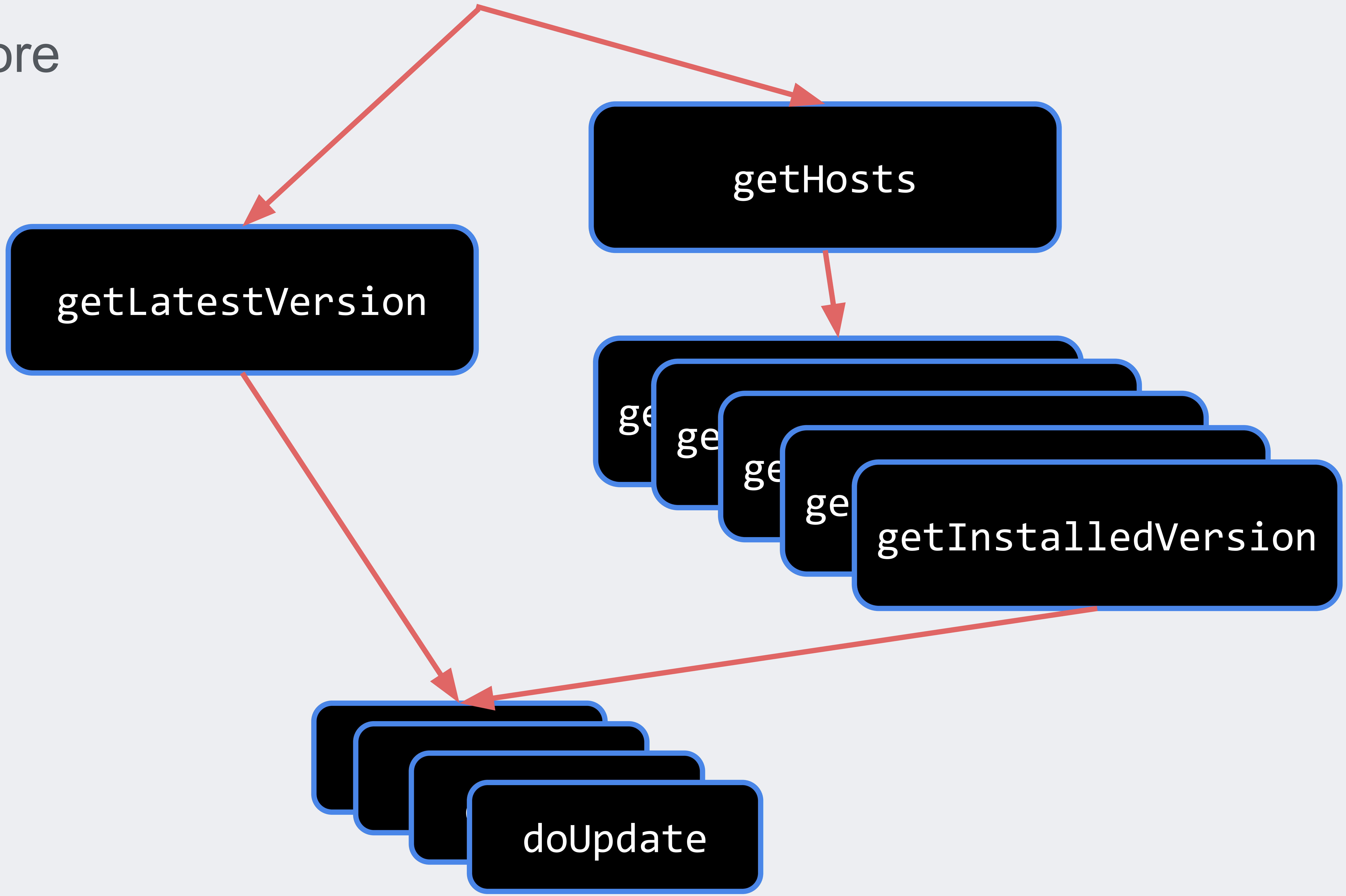
# This is not optimal:

```
do
  latest <- getLatestVersion
  hosts <- getHosts
  installed <- mapM getInstalledVersion hosts
  let updates = [ h | (h,v) <- zip hosts installed, v < latest ]
  mapM_ (updateTo latest) updates
```

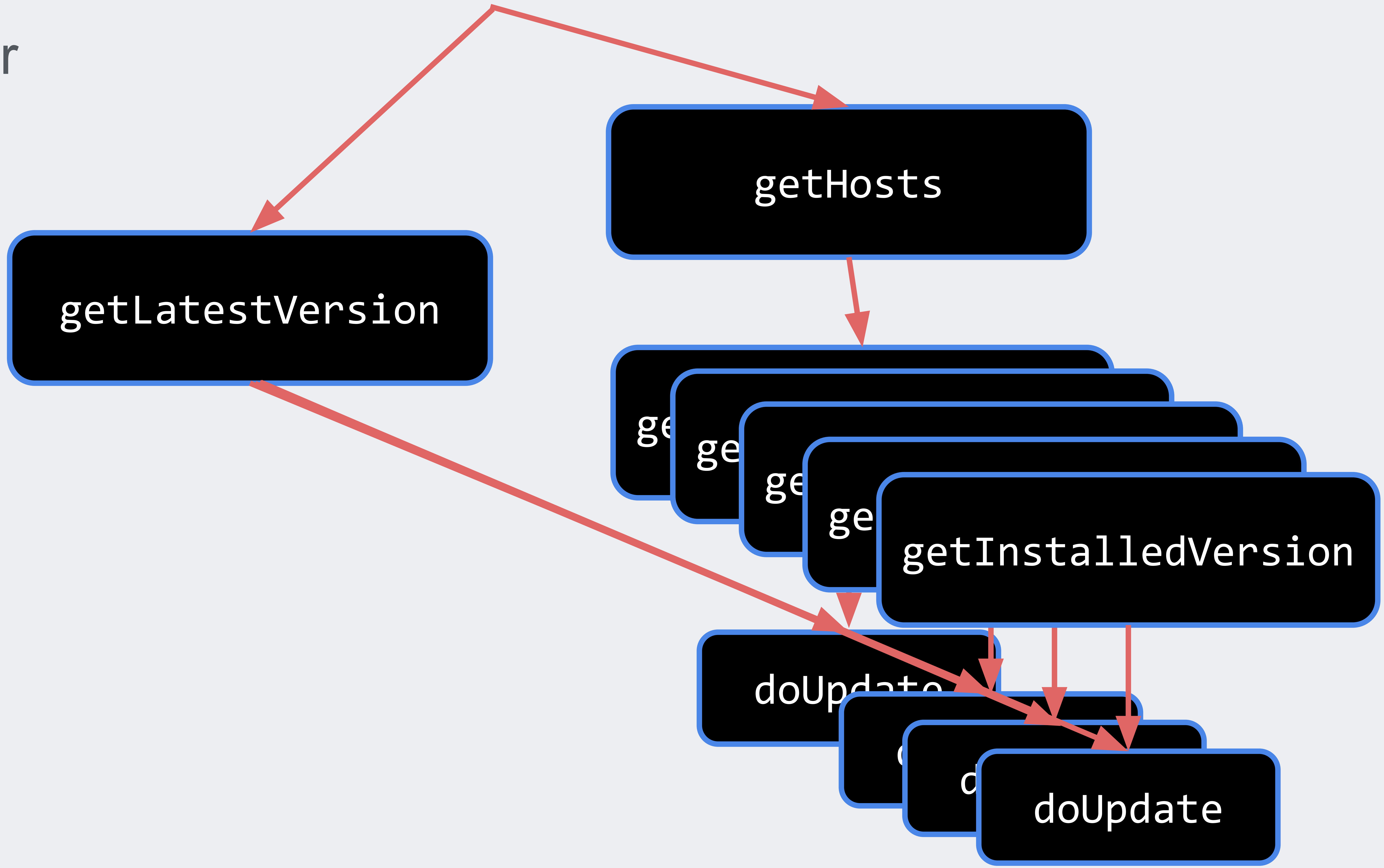
# This is better:

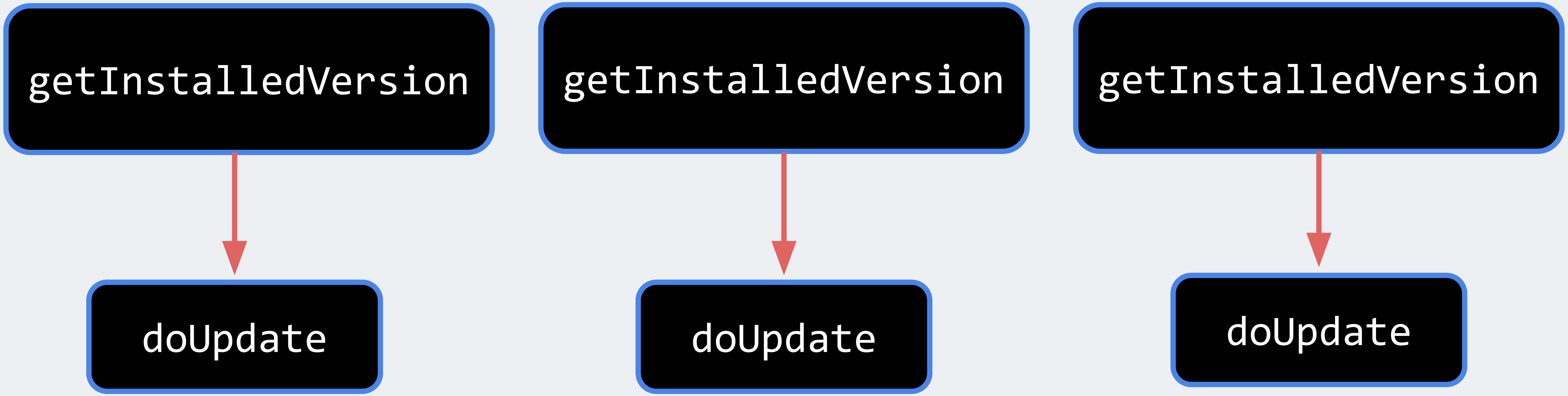
```
do
  latest <- getLatestVersion
  hosts <- getHosts
  let
    perHost h = do
      v <- getInstalledVersion h
      when (v < latest) (updateTo latest h)

  mapM_ perHost hosts
```

# Before


getHosts
getLatestVersion
ge
ge
ge
ge
getInstalledVersion
doUpdate

# After


getHosts
getLatestVersion
ge
ge
ge
ge
getInstalledVersion
doUpdate
d
d
doUpdate



getInstalledVersion

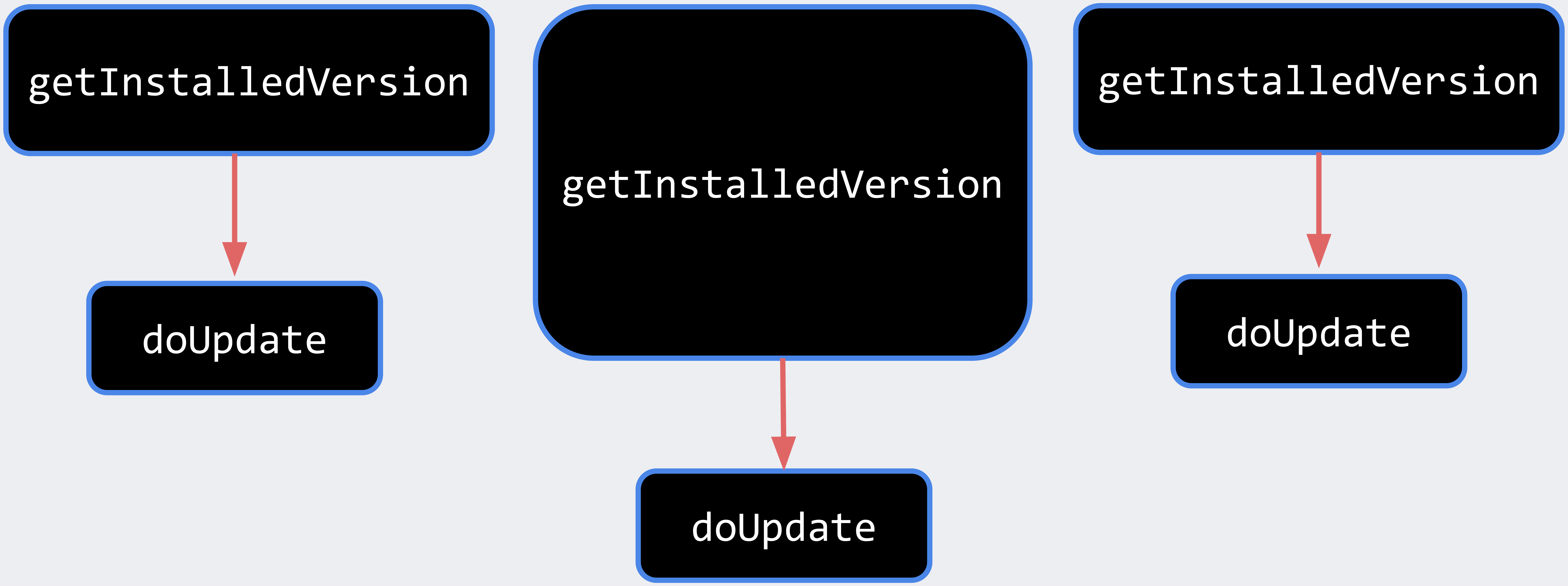
doUpdate

getInstalledVersion

doUpdate

getInstalledVersion

doUpdate



getInstalledVersion

doUpdate

getInstalledVersion

doUpdate

getInstalledVersion

doUpdate

# Haxl 2

- Drops requirement to complete all I/O in a “round”
- I/O can be arbitrarily overlapped
  - with other I/O and computation

# New contract with the I/O provider

```
fetch
  [ (GetLatestVersion, resultVar)
  , (GetHosts, resultVar)
  ]
```

Provider calls “putResult resultVar”
when the result is ready

# Haxl 2

- Tradeoff: possibly less batching
- We saw latency reductions after deploying this

# Haxl 2

- Tradeoff: possibly less batching
- We saw latency reductions after deploying this
- Haxl 2 is available on github & Hackage

# Recap: our example

- Original code had lots of inherent parallelism
- To squeeze out even more, we had to refactor

# Recap: our example

- Original code had lots of inherent parallelism
- To squeeze out even more, we had to refactor
- Big Hammer got us 80% of the way for 20% of the effort

# What about testing & debugging?

# To use Haxl for your I/O

- Make a data type:

```haskell
data HTTP a where
  HTTP :: HTTPRequest -> HTTP Text

deriving instance Eq (HTTP a)
deriving instance Show (HTTP a)
instance Hashable (HTTP a) where ...
```

- Add misc boilerplate instances
- Implement `fetch`

Now, our I/O is data.

We can do a lot of things with data.

# Haxl stores I/O request & results

- In a per-request cache
- If you fetch data again, you get the same answer
- Good for
  - perf
  - correctness (resistant to changes in external data)
  - modularity...

# Wait, modularity?

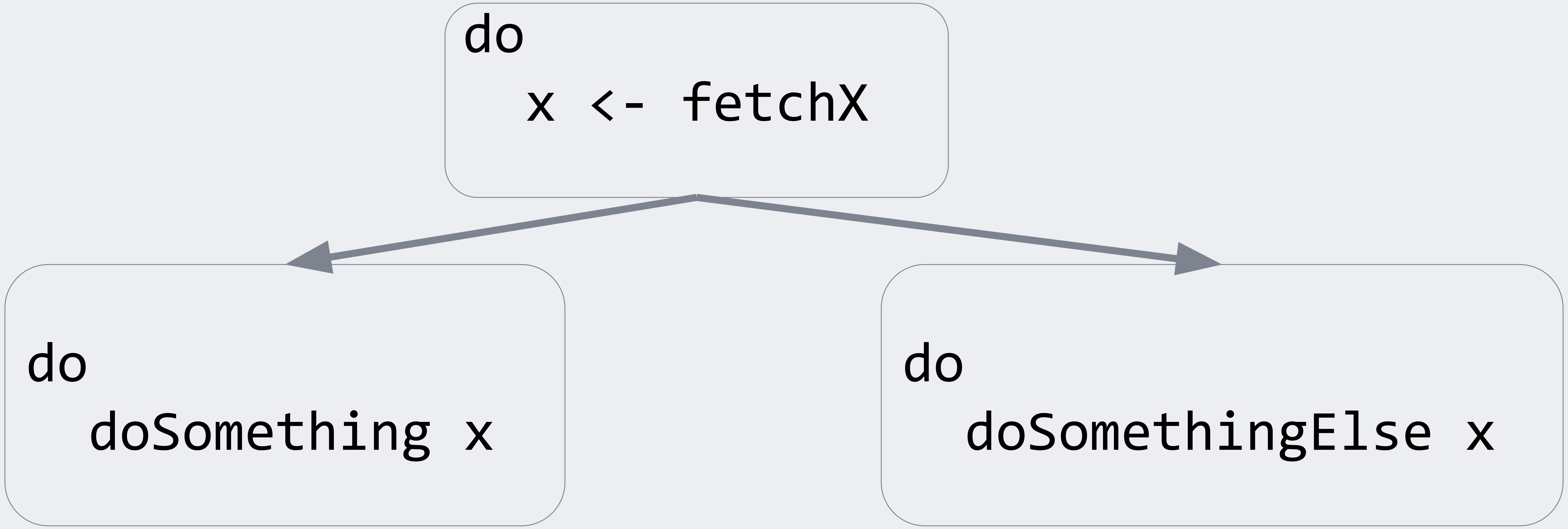
- Yes!
- Without caching, you would have to factor out the fetching and pass the results around

```
do
  x <- fetchX
  doSomething x
```

```
do
  x <- fetchX
  doSomethingElse x
```

# Wait, modularity?

- Yes!
- Without caching, you would have to factor out the fetching and pass the results around



```
do
  x <- fetchX
```

```
do
  doSomething x
```

```
do
  doSomethingElse x
```

# The cache records all the I/O

- Without the I/O, everything else is deterministic
- If we run the same thing again...
  - All the I/O is already cached
  - We must get the same result
- Only works if you do this for *all* your I/O

# The cache records all the I/O

- We can dump it:

```
dumpCacheAsHaskell :: Haxl String
```

```
loadCache :: GenHaxl u ()
loadCache = do
  cacheRequest (HTTPRequest “http://www.example.com”) (Right “...”)
  ...
```

# To make a unit test...

- Run the code
- Dump the cache into a file
- You now have a unit test

- Can also write libraries to pre-populate the cache
  - for synthetic test data

# … and debug things

- Dump the cache on failure
- Use it later to reproduce what happened

# Why did we make Haxl?

https://code.facebook.com/posts/745068642270222/fighting-spam-with-haskell/

Code
Search
Open Source
Platforms ▾
Infrastructure Systems ▾
Hardware Infrastructure ▾
Video & VR ▾
Artificial Intelligence ▾

June 26, 2015 SECURITY · BACKEND

# Fighting spam with Haskell

Simon Marlow

One of our weapons in the fight against spam, malware, and other abuse on Facebook is a system called Sigma. Its job is to proactively identify malicious actions on Facebook, such as spam, phishing attacks, posting links to malware, etc. Bad content detected by Sigma is removed automatically so that it doesn't show up in your News Feed.

We recently completed a two-year-long major redesign of Sigma, which involved replacing the **in-house FXL language** previously used to program Sigma with **Haskell**. The Haskell-powered Sigma now runs in production, serving more than one million requests per second.

# Sigma

- Rule engine for abuse detection / remediation
- System + rules implemented in Haskell

# Sigma

- Rule engine for abuse detection / remediation
- System + rules implemented in Haskell
- Haxl means that
  - rule authors don’t worry about concurrency
  - caching and memoization happen automatically

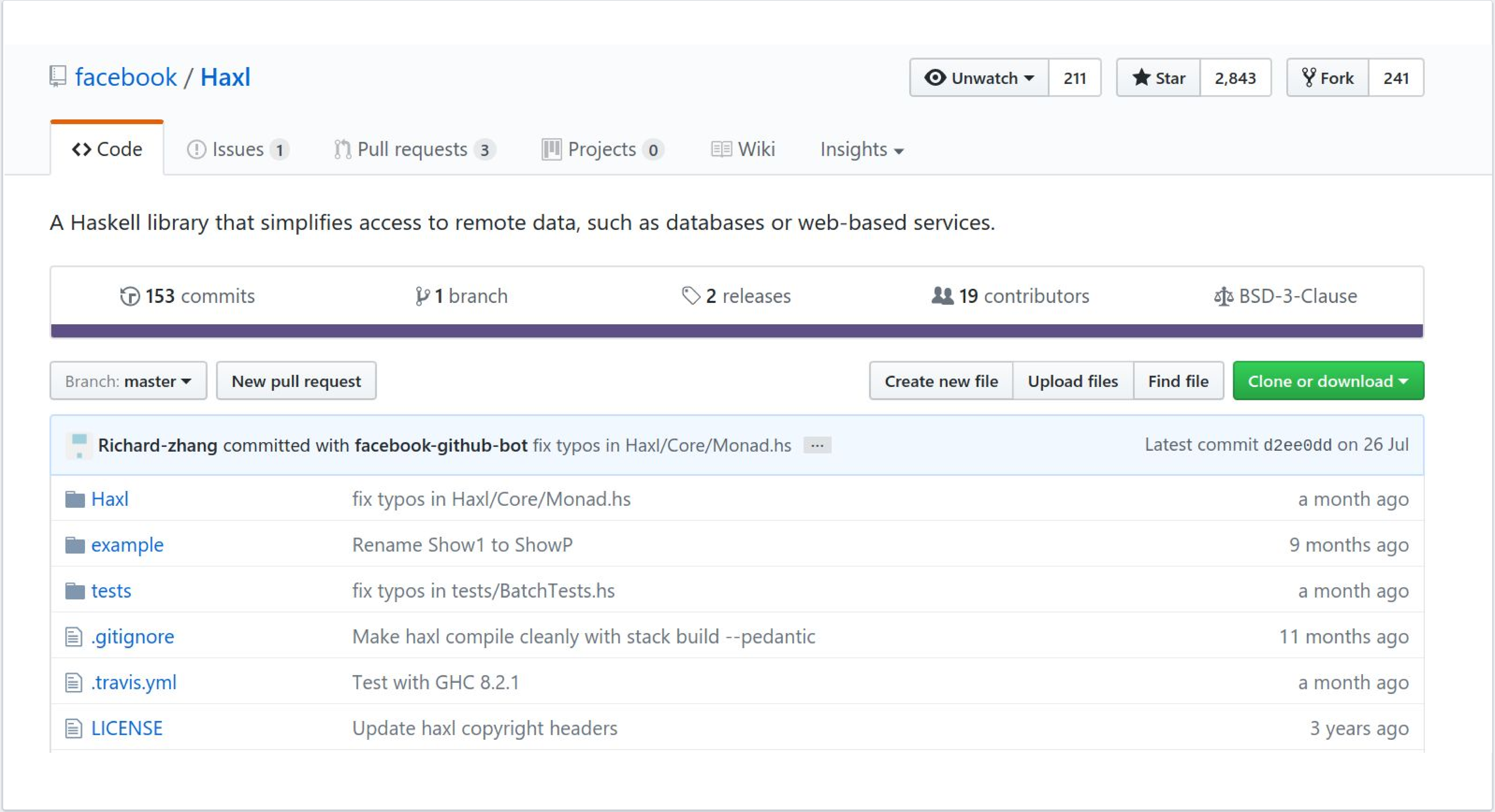
# Sigma

- Rule engine for abuse detection / remediation
- System + rules implemented in Haskell
- Haxl means that
  - rule authors don’t worry about concurrency
  - caching and memoization happen automatically
- Haskell means that
  - rules are type-safe and side-effect free
  - new rules can be deployed quickly and safely

# Stats

- Serving over 1M requests/sec
- With thousands of machines
- Hundreds of Kloc of Haskell code
- Hundreds of changes per day
  - deployed immediately
- Dozens of developers regularly committing

# Haxl is open source



facebook / Haxl

A Haskell library that simplifies access to remote data, such as databases or web-based services.

153 commits | 1 branch | 2 releases | 19 contributors | BSD-3-Clause

Richard-zhang committed with facebook-github-bot fix typos in Haxl/Core/Monad.hs ··· Latest commit d2ee0dd on 26 Jul

| | | |
|---|---|---|
| Haxl | fix typos in Haxl/Core/Monad.hs | a month ago |
| example | Rename Show1 to ShowP | 9 months ago |
| tests | fix typos in tests/BatchTests.hs | a month ago |
| .gitignore | Make haxl compile cleanly with stack build --pedantic | 11 months ago |
| .travis.yml | Test with GHC 8.2.1 | a month ago |
| LICENSE | Update haxl copyright headers | 3 years ago |

# Clones

- Stitch (Scala; @Twitter; not open source)
- clump (Scala; open source clone of Stitch)
- Fetch (Scala; open source)
- Fetch (PureScript; open source)
- muse (Clojure; open source)
- urania (Clojure; open source; based on muse)
- HaxlSharp (C#; open source)
- fraxl (Haskell; using Free Applicatives)

# Summary

- You write:
  - Boilerplate for your I/O
- You get:
  - Concurrency
  - Caching
  - Testability (record/replay, mocking)
  - Debuggability (capture run, repro later)

# Questions?