Network Protocol Programming in Haskell

Kazu Yamamoto
Internet Initiative Japan Inc.

Background

- Over 7 years, I have developed several network protocols in Haskell

- My mission in our company is to contribute standardization of network protocols
- I'm one of the maintainers of the `network` library in Haskell

Why Haskell?

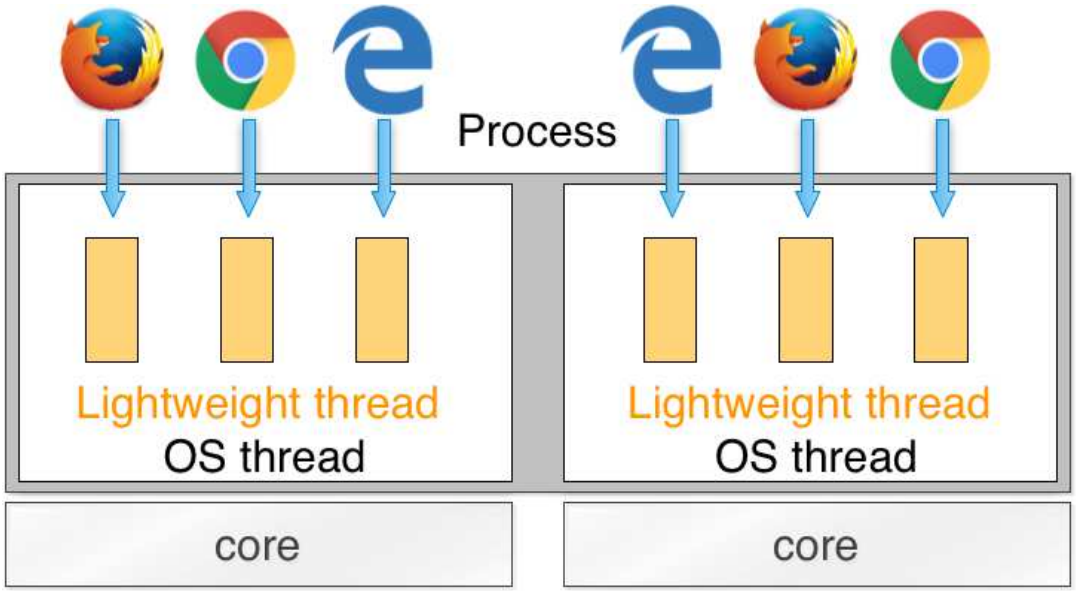
- Haskell is a statically-typed programming language
- Haskell is suitable for highly concurrent network programming

(1) Lightweight threads	Erlang, go
(2) Rich immutable data types	OCaml, Scala
(3) Strong type system	OCaml, Scala
(4) Software transactional memory	Clojure, Java

- Haskell provides everything I want

(1) Lightweight threads

- The flagship compiler of Haskell is GHC
  - Glasgow Haskell Compiler
- GHC provides lightweight (green) threads

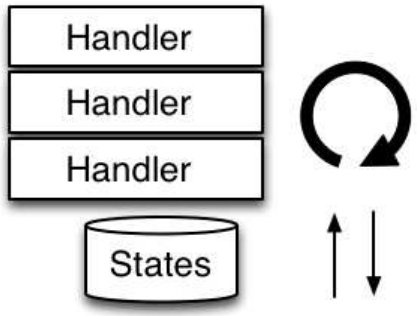


  - The overhead of a lightweight thread is about 1K bytes
  - They can migrate to another low-load core

HTTP/1.1 implementation

- Event driven programming
  - Code needs to be divided into some handlers (callbacks)
  - States need to be maintained explicitly

Event driven programming



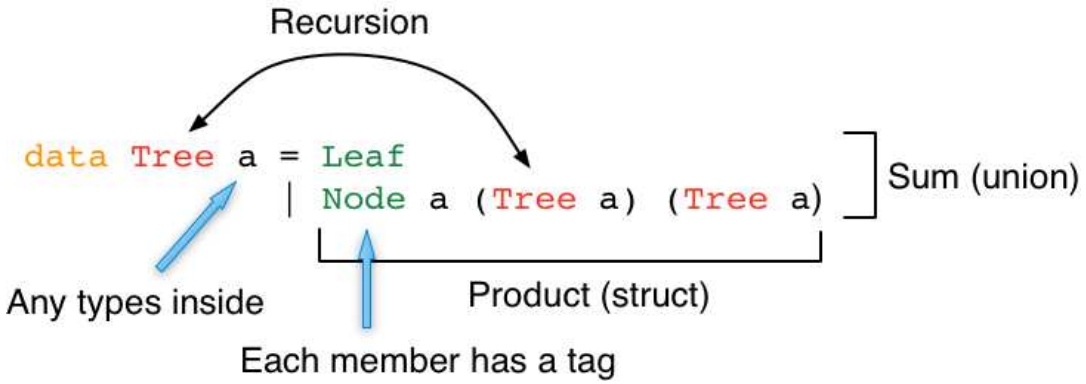
Thread programming

```
loop {
  req = receiveRequest;
  rsp = application(req);
  sendResponse(rsp);
}
```

- Lightweight thread programming
  - Straightforward
  - Tactics: one lightweight thread per connection

(2a) Rich data types

- Haskell provides integrated data types: sums of products with recursion

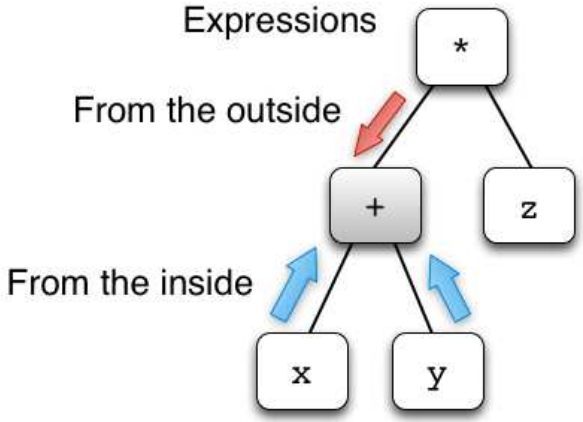


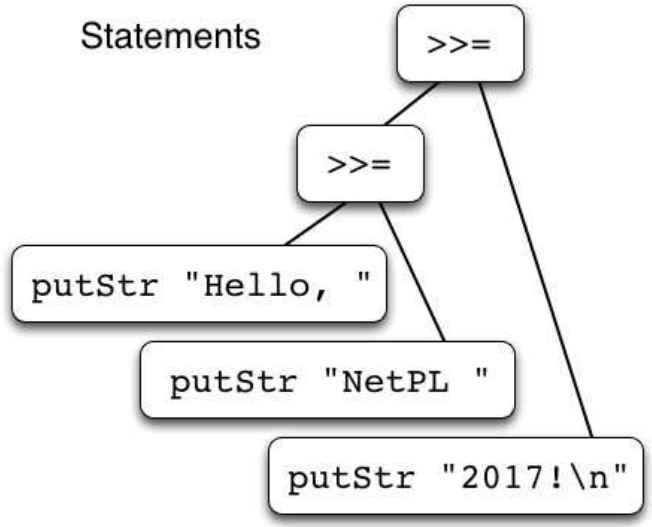
- Thanks to tags, we can cover all possible values

```
case tree of
 Leaf       -> ...
 Node x l r -> ...
```

(3) Strong type system

- Each piece of Haskell code is an expression
- Types of expressions can be checked in two ways:
  - how the expression is composed from the inside
  - how it is used from the outside





- A sequence of statements is a syntax sugar of expressions

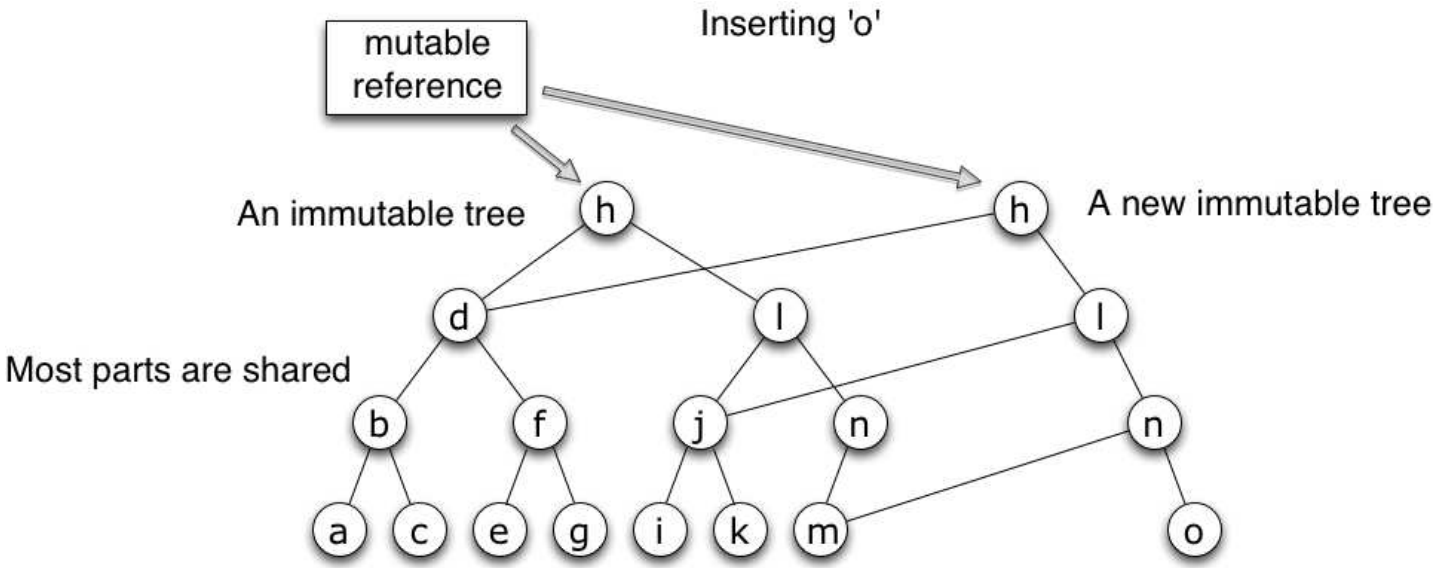
With rich data types,
the strong type system detects many errors
at compile time

If Haskell code compiles,
the code works
as its programmer intends in many cases

Debugging phase is really short

(2b) Immutable data types

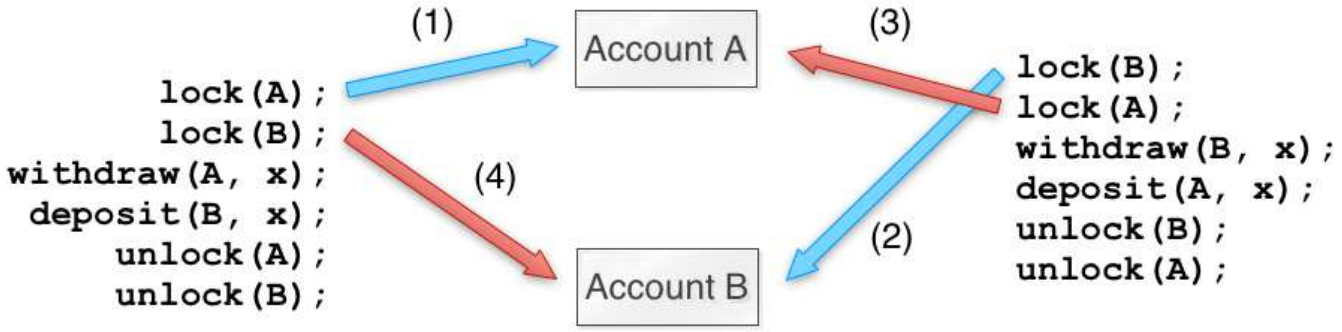
- Most data types are immutable, thus thread-safe



- Immutable data can be treated as mutable data with mutable references
- A single mutable reference can be changed w/o locking

Deadlock

- Threads may use multiple variables and need to update them in consistent manner
  - Other languages use multiple locks for this purpose
- Multiple locks sometime result in dead lock



- Common solution is to decide the total order of variables
  - But this approach is troublesome and sometime impossible

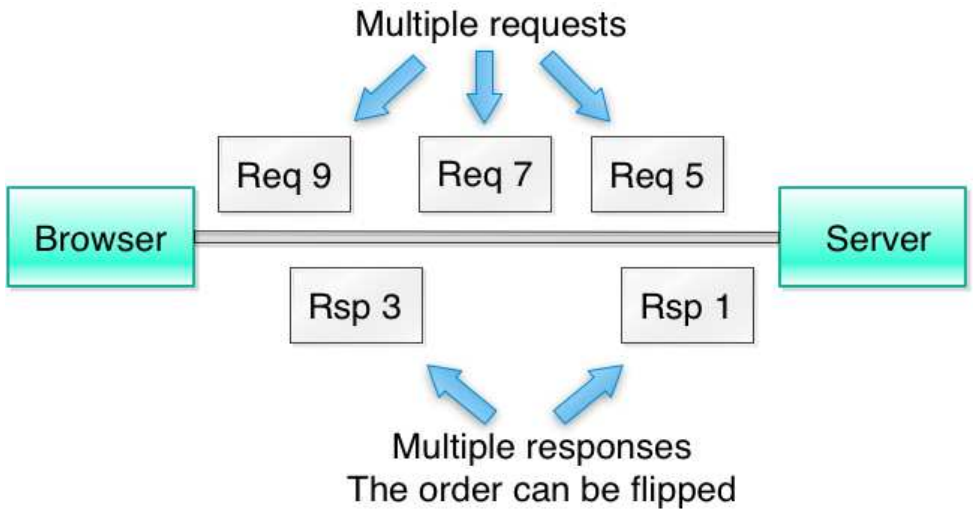
(4) Software Transactional Memory (STM)

- STM is dead-lock free
  - STM is a mechanism to make multiple locks to a single

- STM actions are retried until they succeed
  - Haskell's type system ensures that side-effects in STM actions can be rolled back
  - Retries are safe: missiles are never launched

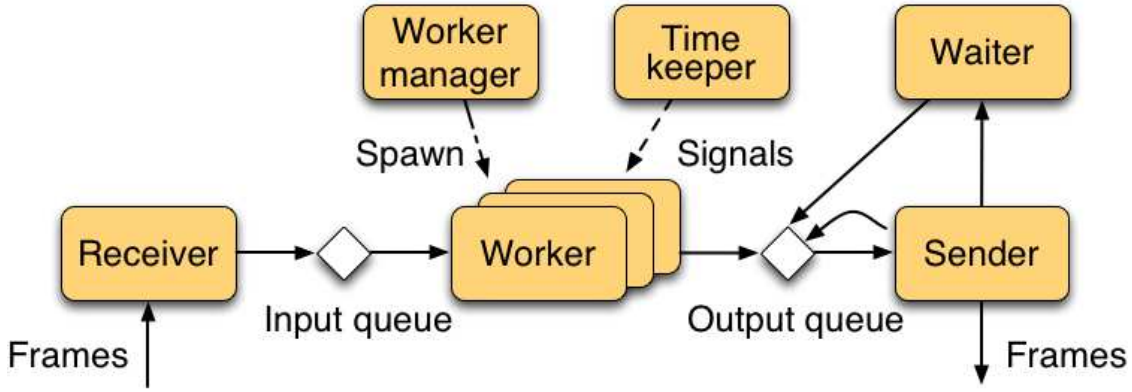
HTTP/2

- HTTP/2 is re-design of the transport layer
  - It keeps HTTP/1.1 semantics such as HTTP headers
  - Only one TCP connection is used
  - Multiple requests and responses are transported
  - The order of responses is not guaranteed



HTTP/2 implementation

- The tactics cannot be used to implement HTTP/2
- I needed to introduce several threads and some variables



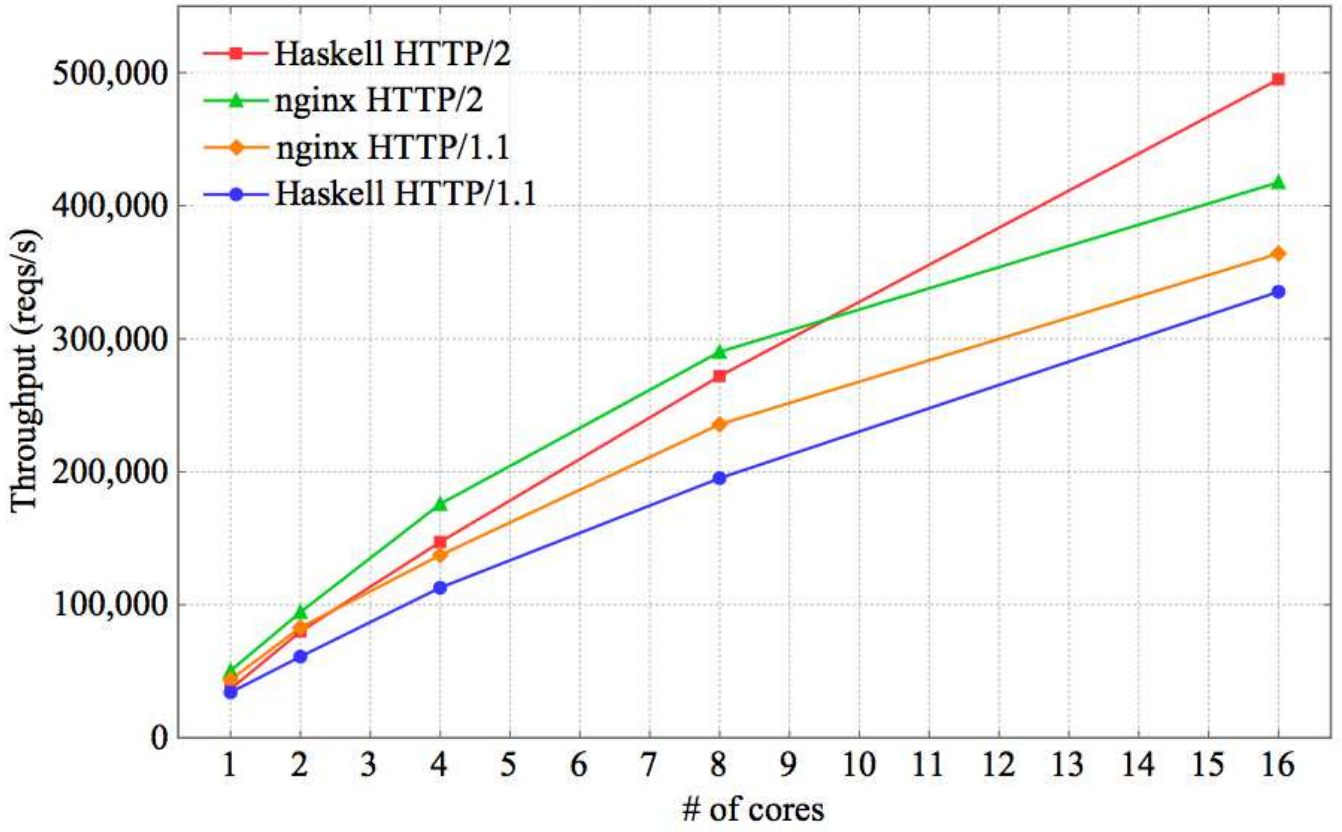
- This system is dead-lock free thanks to STM

Benchmark

- Downloading short files

- `h2load` in `nghttp2`
  - Supporting both HTTP/1.1 and HTTP/2
  - Scaling on multi-cores
- `nginx` and my server in Haskell

Performance



Further reading