Network Protocol Programming in Haskell

Kazuhiko Yamamoto
Internet Initiative Japan Inc.
kazu@iij.ad.jp

1 HASKELL LANGUAGE

Over seven years, we have developed several network protocol libraries including anti-spam, DNS, HTTP/1.1, HTTP/2 and TLS 1.3 in Haskell. Based on these experiences, we regard Haskell as a safe and highly-concurrent network programming language thanks to its strong type system and lightweight threads (i.e. green threads). This paper describes advantages and disadvantages of Haskell and reports our experiences.

Haskell is typically described as a purely functional programming language. Unfortunately, this catch-phrase gives programmers an impression that is difficult to approach. Roughly speaking, *pure* means that Haskell's type system clearly separates pure functions (i.e. without side effects) and impure functions (i.e. with side effects), and prevents pure functions from calling impure functions. *Functional* indicates that Haskell encourages modular programming with *immutable* data which provides consistency for concurrent programming.

1.1 Advantages

Haskell provides integrated data types which are sums (e.g. `union` and `enum`) of products (e.g. `struct`) with recursion. Since each member of data types has a unique tag, the compiler can check the coverage of values. This prevents null exceptions. The rich data types enable us to express target problems directly and to define embedded domain specific languages (EDSL) easily.

Each piece of Haskell code is an *expression*, which means that the type of an expression can in checked by two ways; how the expression is composed from the inside and how it is used from the outside. Note that a sequence of *statements* can be emulated by an expression combining impure functions whose main purposes are side-effects.

With the rich data types and strong type checking, Haskell code works as its programmer intends in many cases if the code compiles. The compiled code in other compiled languages does not always run well, for instance, due to null exceptions, implicit type conversions, incomplete coverage of values, etc.

Most data types are immutable. Thus, they can be shared by threads in a consistent manner. Impure functions can make use of *mutable* data, such as arrays, and immutable data can be treated as mutable data by wrapping it with mutable references.

Glasgow Haskell Compiler (GHC) [1], the flagship compiler of Haskell, provides lightweight threads, whose memory overhead is around 1 KiB, based on the NxM model. Thanks to a well-defined foreign function interface (FFI), if a lightweight thread calls a blocking external function (such as system calls), a dedicated OS thread (native thread) takes over this procedure. Other lightweight threads are not blocked and can continue running on the OS thread. Lightweight threads are even able to migrate to another low-load CPU core if multiple cores are available. Comparing event driven programming where the code is divided into handlers, lightweight thread programming makes code straightforward and easy to maintain.

Software Transactional Memory (STM), which provides dead-lock free locks, is seamlessly integrated. Again, the type system distinguishes STM functions, whose side-effects are constrained, so that the transactions can be rolled back. Lightweight threads can share multiple locks without suffering from dead-lock (but live-lock is still possible).

1.2 Disadvantages

Unformtunately, Haskell has disadvantages; 1) Compile speed of GHC is slow. As GHC provides more features, GHC gets slower. A project to make GHC faster started in last year. 2) Typical generated code is faster than typical dynamic languages but is roughly 5 time slower than that in C. 3) Cross compilation is possible but not easy.

2 NETWORK PROGRAMMING

It is important for network servers to utilize multiple cores and to be robust. Also, rapid prototyping is the key to selecting proper data structures for performance and understanding the target domain.

2.1 Multi-core utilization

The IO manager is the heart of GHC's lightweight threads. It multiplexes output requests from lightweight threads to devices and dispatches arrived inputs to corresponding sleeping lightweight threads. Even though GHC provides a multicore scheduler, a parallel garbage collector and efficient multicore memory allocation, the IO manager of GHC 7.6 and earlier did not scale to a multicore environment. To make use of the potential of multicores, we needed to use the *prefork* technique where one process is forked for each core before the server starts its services.

After the evolution of the IO manager in GHC 7.8 by our work [2], lightweight threads can scale to up to, at least, 40 cores. Since the prefork technique is not necessary anymore, we were able to remove the code of prefork and inter-process communication for graceful shutdown, etc.

2.2 Safe concurrency

In 2010, we were developing an anti-spam server in Haskell, which informs a mail server about evaluation scores based on SPF, Sender ID, DomainKeys and DKIM through the Milter interface. First this server concurrently used a famous asynchronous DNS library written in C through FFI. Unfortunately, we got a lot of assertion failures in the real world operation. So, we gave up on the asynchronous DNS library and developed a DNS library entirely in Haskell.

All pure functions are reentrant and impure functions in Haskell standard library are carefully designed as thread-safe. Thanks to Haskell's safe concurrency, our DNS library is quite stable even under highly concurrent environments. We received a similar experience report from the programmer of a web-based RSS reader service in 2013. The service was suffering from the same assertion failures from the asynchronous DNS library in C under 100 concurrent resolvers. Switching to our DNS library, the resolver's behavior became much more stable.

With lightweight threads, implementation of synchronous application protocols is straightforward. For instance, an HTTP/1.1 server just spawns one lightweight thread for every TCP connection and each thread repeats the cycle of reading a request, executing a web application and sending a response. Our high-performance HTTP/1.1 server library, Warp [4], follows this common tactic. Our big question was whether lightweight threads were useful for implementing an asynchronous application protocol such as HTTP/2.

HTTP/2 redesigned HTTP/1.1's transport while the semantics of HTTP/1.1 (such as HTTP headers) are preserved. Multiple frames containing requests and responses are transferred asynchronously over a single TCP connection. We supported HTTP/2 in Warp by using multiple lightweight threads with two queues [3]. A receiver repeatedly decodes received frames, produces a request value, and enqueues it to an STM based input queue. In a symmetric fashion, a sender repeatedly dequeues a response value from an STM based output queue, encodes its data to frames until the buffer is filled or a flow control window is exhausted, sends the frames, and if data remains to be sent, re-enqueues the response on the output queue.

A worker thread pool is prepared between the queues. The role of workers is to dequeue a request value from the input queue, pass it to a web application, and enqueue the application's response value onto the output queue. Even though there are other lightweight threads for house keeping and STM values for flow control, etc, this architecture is free from dead-lock thanks to STM.

2.3 Rapid prototyping

To deliver important content on a priority basis, *priority* is defined in HTTP/2. The output queue should be a priority queue proportional to content weights. In a few days, we implemented 7 data structures in Haskell to compare their performance. Our conclusion is that mutable binary heaps based on arrays are a reasonable option to implement HTTP/2's priority queue in most programming languages while immutable priority search queues (PSQ) are suitable for highly concurrent environment such as Haskell[3].

During the standardization process of HTTP/2 in IETF, we explored the design space of header compression. The final specification, RFC 7541, defines a static table, dynamic tables and Huffman encoding as compression methods. Each end maintains those tables in a synchronous manner. If a header name or a pair of header name and value is found in a table, it can be represented as an index number whose typical length is 7 bits. Huffman encoding shortens header names and/or values when they are conveyed on a connection. The draft 08 or earlier also defines the reference set which implements *difference* based on index numbers.

To understand effectiveness of each method, we developed an EDSL to express compression strategies and defined 8 strategies to cover all possible combinations of four compression methods. This experiment uncovered that the reference set, which is complicated and has a special corner case, contributes little to compression ratio. Thus, we proposed to remove the reference set from the header compression. The reference set was removed in the draft 09. This made the specification and implementations much simpler. 24.5% (704/932) lines of code were removed from the main part of HTTP/2 library in Haskell.

REFERENCES

[1] S. Marlow and S. Peyton Jones. The Glasgow Haskell Compiler. In *the Architecture of Open Source Applications*, volume 2. 2012. http://www.aosabook.org/en/ghc.html.

[2] A. Voellmy, J. Wang, P. Hudak, and K. Yamamoto. Mio: A High-Performance Multicore IO Manager for GHC. In *Proceedings of Haskell Symposium*, 2013.

[3] K. Yamamoto. Experience Report: Developing High Performance HTTP/2 Server in Haskell. In *Proceedings of Haskell Symposium*, 2016.

[4] K. Yamamoto, M. Snoyman, and A. Voellmy. Warp. In *The Performance of Open Source Applications*. 2013. http://www.aosabook.org/en/posa/warp.html.