Nginx Haskell module
(yet another doc with examples)

Alexey Radkov

January 18, 2018

Contents

Why bother?

The *nginx-haskell-module* allows for running in Nginx written in Haskell synchronous and asynchronous tasks, per-worker and shared services, and content handlers.

Synchronous tasks

Synchronous tasks are mostly *pure* Haskell functions of various types. To make them available in Nginx configuration files, they must be exported with special declarations named *exporters*. Below is a table of *type/exporter* correspondence for all available synchronous handlers.

Type	Exporter
`String -> String`	`ngxExportSS` (`NGX_EXPORT_S_S`)
`String -> String -> String`	`ngxExportSSS` (`NGX_EXPORT_S_SS`)
`String -> Boolean`	`ngxExportBS` (`NGX_EXPORT_B_S`)
`String -> String -> Boolean`	`ngxExportBSS` (`NGX_EXPORT_B_SS`)
`[String] -> String`	`ngxExportSLS` (`NGX_EXPORT_S_LS`)
`[String] -> Boolean`	`ngxExportBLS` (`NGX_EXPORT_B_LS`)
`ByteString -> L.ByteString`	`ngxExportYY` (`NGX_EXPORT_Y_Y`)
`ByteString -> Boolean`	`ngxExportBY` (`NGX_EXPORT_B_Y`)
`ByteString -> IO L.ByteString`	`ngxExportIOYY` (`NGX_EXPORT_IOY_Y`)

All synchronous handlers may accept *strings* (one or two), a *list of strings*, or a *strict bytestring*, and return a *string*, a *boolean* or a *lazy bytestring*. The last handler from the table is *impure* or *effectful*, and it returns a *lazy bytestring* wrapped in *IO Monad*.

There are two kinds of exporters which differ only in their implementations. The first kind — *camel-cased* exporters — is implemented by means of *Template Haskell*, the other kind — exporters in braces, as they are shown in the table — is implemented using *CPP macros*. Both of them provide *FFI* declarations for functions they export, but the camel-cased exporters are available only from a separate Haskell module *ngx-export*, which can be downloaded and installed by *cabal*, whereas the CPP exporters are implemented inside the *nginx-haskell-module* in so-called *standalone* approach, where custom Haskell declarations get wrapped inside common Haskell code.

Examples

In the examples we will use *modular* approach with *camel-cased* exporters.

File test.hs

```haskell
{-# LANGUAGE TemplateHaskell #-}
```

```haskell
module NgxHaskellUserRuntime where

import           NgxExport
import qualified Data.Char as C

toUpper :: String -> String
toUpper = map C.toUpper
ngxExportSS 'toUpper

ngxExportSS 'reverse

isInList :: [String] -> Bool
isInList [] = False
isInList (x : xs) = x `elem` xs
ngxExportBLS 'isInList
```

In this module we declared three synchronous handlers: *toUpper*, *reverse*, and *isInList*. Handler *reverse* exports existing and well-known Haskell function *reverse* which reverses lists. Let's compile *test.hs* and move the library to a directory, from where we will load this.

```
ghc -O2 -dynamic -shared -fPIC -L$(ghc --print-libdir)/rts -lHSrts-ghc$(
    ghc --numeric-version) test.hs -o test.so
[1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
Linking test.so ...
cp -i test.so /var/lib/nginx/
```

File test.conf

```nginx
user                    nginx;
worker_processes        4;

events {
    worker_connections  1024;
}

http {
    default_type        application/octet-stream;
    sendfile            on;

    haskell load /var/lib/nginx/test.so;

    server {
        listen          8010;
        server_name     main;

        location / {
            haskell_run toUpper $hs_upper $arg_u;
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            echo "toUpper $arg_u = $hs_upper";
            echo "reverse $arg_r = $hs_reverse";
```

```
            echo "$arg_a ‘isInList‘ [$arg_b, $arg_c, $arg_d] = $hs_isInList";
        }
    }
}
```

Library *test.so* gets loaded by Nginx directive *haskell load*. All synchronous handlers run from directive *haskell_run*. The first argument of the directive is a name of a Haskell handler exported from the loaded library *test.so*, the second argument is an Nginx variable where the handler will put the result of its computation, the rest arguments are passed to the Haskell handler as parameters. Directive *haskell_run* has *lazy* semantics in the sense that it runs its handler only when the result is needed in a content handler or rewrite directives.

Let's test the configuration with *curl*.

```
curl 'http://127.0.0.1:8010/?u=hello&r=world&a=1&b=10&c=1'
toUpper hello = HELLO
reverse world = dlrow
1 ‘isInList‘ [10, 1, ] = 1
```

Synchronous content handlers

There are three types of exporters for synchronous content handlers.

Type	Exporter
`ByteString -> (L.ByteString, ByteString, Int)`	`ngxExportHandler` (`NGX_EXPORT_HANDLER`)
`ByteString -> L.ByteString`	`ngxExportDefHandler` (`NGX_EXPORT_DEF_HANDLER`)
`ByteString -> (ByteString, ByteString, Int)`	`ngxExportUnsafeHandler` (`NGX_EXPORT_UNSAFE_HANDLER`)

All content handlers are *pure* Haskell functions, as well as the most of other synchronous handlers. The *normal* content handler returns a *3-tuple (response-body, content-type, HTTP-status)*. The response body consists of a number of chunks packed in a *lazy bytestring*, the content type is a *strict bytestring* such as *text/html*. The *default* handler defaults the content type to *text/plain* and the HTTP status to *200*, thus returning only chunks of the response body. The *unsafe* handler returns a *3-tuple* with a single-chunked response body, the content type and the status, but the both bytestring parameters are supposed to be taken from static data, which must not be cleaned up after request termination.

Normal and *default* content handlers can be declared with two directives: *haskell_content* and *haskell_static_content*. The second directive runs its handler only once, when the first request comes, and returns the same response on further requests. The *unsafe* handler is declared with directive *haskell_unsafe_content*.

An example

Let's replace Nginx directive *echo* with our own default content handler *echo*. Add in *test.hs*,

```
import           Data.ByteString (ByteString)
import qualified Data.ByteString.Lazy as L

-- ...

echo :: ByteString -> L.ByteString
echo = L.fromStrict
ngxExportDefHandler 'echo
```

compile it and put *test.so* into */var/lib/nginx/*. Add new location */ch* into *test.conf*,

```
        location /ch {
            haskell_run toUpper $hs_upper $arg_u;
            haskell_run reverse $hs_reverse $arg_r;
            haskell_run isInList $hs_isInList $arg_a $arg_b $arg_c $arg_d;
            haskell_content echo
"toUpper $arg_u = $hs_upper
reverse $arg_r = $hs_reverse
$arg_a `isInList` [$arg_b, $arg_c, $arg_d] = $hs_isInList
";
        }
```

and test again.

```
curl 'http://127.0.0.1:8010/ch?u=content&r=handler&a=needle&b=needle&c=in
    &d=stack'
toUpper content = CONTENT
reverse handler = reldnah
needle `isInList` [needle, in, stack] = 1
```

Asynchronous tasks and request body handlers

There are only two types of Haskell handlers for per-request asynchronous tasks: the asynchronous handler and the asynchronous request body handler.

Type	Exporter
`ByteString -> IO L.ByteString`	`ngxExportAsyncIOYY` (`NGX_EXPORT_ASYNC_IOY_Y`)
`L.ByteString -> ByteString -> IO L.ByteString`	`ngxExportAsyncOnReqBody` (`NGX_EXPORT_ASYNC_ON_REQ_BODY`)

Normal asynchronous handler accepts a strict bytestring and returns a lazy bytestring. Its type exactly corresponds to that of handlers exported with *ngxExportIOYY*. Request body handlers require the request body chunks in their first parameter.

Unlike synchronous handlers, asynchronous per-request handlers are *eager*. This means that they will always run when declared in a location, no matter whether their results are going to be used in the response and rewrite directives, or not. The asynchronous handlers run in an early *rewrite phase* (before rewrite directives), and in a late rewrite phase (after rewrite directives, if in the final location there are more asynchronous tasks declared). It is possible to declare many asynchronous tasks in a single location: in this case they are spawned one by one in order of their declarations, which lets using results of early tasks in inputs of later task.

Asynchronous tasks are bound to the Nginx event loop by means of *eventfd* (or POSIX *pipes* if eventfd was not available on the platform when Nginx was being compiled). When the rewrite phase handler of this module spawns an asynchronous task, it opens an eventfd, then registers it in the event loop, and passes it to the Haskell handler. As soon as the Haskell handler finishes the task and pokes the result into buffers, it writes into the eventfd, thus informing the Nginx part that the task has been finished. Then Nginx gets back to the module's rewrite phase handler, and it spawns the next asynchronous task, or returns (when there are no more tasks), moving request processing to the next stage.

An example

Let's add two asynchronous handlers into *test.hs*: one for extracting a field from POST data, and the other for delaying response for a given number of seconds.

```haskell
import qualified Data.ByteString.Char8 as C8
import qualified Data.ByteString.Lazy.Char8 as C8L
import           Control.Concurrent
import           Safe

-- ...

reqFld :: L.ByteString -> ByteString -> IO L.ByteString
reqFld a fld = return $ maybe C8L.empty C8L.tail $
    lookup (C8L.fromStrict fld) $ map (C8L.break (== '=')) $ C8L.split '&' a
ngxExportAsyncOnReqBody 'reqFld

delay :: ByteString -> IO L.ByteString
delay v = do
    let t = readDef 0 $ C8.unpack v
    threadDelay $ t * 1000000
    return $ C8L.pack $ show t
ngxExportAsyncIOYY 'delay
```

This code must be linked with *threaded* Haskell RTS this time!

```
ghc -O2 -dynamic -shared -fPIC -L$(ghc --print-libdir)/rts ↙
    ↳ -lHSrts_thr-ghc$(ghc --numeric-version) test.hs -o test.so
[1 of 1] Compiling NgxHaskellUserRuntime ( test.hs, test.o )
Linking test.so ...
cp -i test.so /var/lib/nginx/
```

Let's make location */timer*, where we will read how many seconds to wait from the POST field *timer*, and then wait them until returning the response.

```
location /timer {
    haskell_run_async_on_request_body reqFld $hs_timeout timer;
    haskell_run_async delay $hs_waited $hs_timeout;
    echo "Waited $hs_waited sec";
}
```

Run curl tests.

```
curl -d 'timer=3' 'http://127.0.0.1:8010/timer'
Waited 3 sec
curl -d 'timer=bad' 'http://127.0.0.1:8010/timer'
Waited 0 sec
```

Asynchronous content handler

There is a special type of *impure* content handlers which allows for effectful code. The type corresponds to that of the *normal* content handler, except the result is wrapped in *IO Monad*.

Type	Exporter
`ByteString -> IO (L.ByteString, ByteString, Int)`	`ngxExportAsyncHandler` (`NGX_EXPORT_ASYNC_HANDLER`)

Such handlers are declared with directive *haskell_async_content*.

It's easy to emulate effects in a synchronous content handler by combining the latter with an asynchronous task like in the following example.

```
location /async_content {
    haskell_run_async getUrl $hs_async_httpbin "http://httpbin.org";
    haskell_content echo $hs_async_httpbin;
}
```

Here *getUrl* is an asynchronous Haskell handler that returns content of an HTTP page. This approach has at least two deficiencies related to performance and memory usage. The content may be huge and chunked, and its chunks could have been naturally used in the content handler. But they won't, because here they get collected by directive *haskell_run_async* into a single chunk, and then passed to the content handler *echo*. The other problem deals with *eagerness* of asynchronous tasks. Imagine that we put in the location a rewrite to another location: handler *getUrl* will run before redirection, but variable *hs_async_httpbin* will never be used because we'll get out from the current location.

The asynchronous task runs in a late *access phase*, and the lazy bytestring — the contents — gets used in the content handler as is, with all of its originally computed chunks.

An example

Let's rewrite our *timer* example using *haskell_async_content*.

File test.hs (*additions*)

```
{-# LANGUAGE TupleSections #-}
{-# LANGUAGE MagicHash #-}

-- ...

import          Data.ByteString.Unsafe
import          Data.ByteString.Internal (accursedUnutterablePerformIO)

-- ...

delayContent :: ByteString -> IO (L.ByteString, ByteString, Int)
delayContent v = do
    v' <- delay v
    return $ (, packLiteral 10 "text/plain"#, 200) $
        L.concat ["Waited ", v', " sec\n"]
    where packLiteral l s =
              accursedUnutterablePerformIO $ unsafePackAddressLen l s
ngxExportAsyncHandler 'delayContent
```

For the *content type* we used a static string *"text/plain"#* that ends with a *magic hash* merely to avoid any dynamic memory allocations.

File test.conf (*additions*)

```
        location /timer/ch {
            haskell_run_async_on_request_body reqFld $hs_timeout timer;
            haskell_async_content delayContent $hs_timeout;
        }
```

Run curl tests.

```
curl -d 'timer=3' 'http://127.0.0.1:8010/timer/ch'
Waited 3 sec
curl 'http://127.0.0.1:8010/timer/ch'
Waited 0 sec
```

Asynchronous services

Asynchronous tasks run in a request context, whereas asynchronous services run in a worker context. They start when the module gets initialized in a worker, and stop when a worker terminates. They are useful for gathering rarely changed data shared in many requests.

There is only one type of asynchronous services exporters.

Type	Exporter
`ByteString -> Bool -> IO L.ByteString`	`ngxExportServiceIOYY` (`NGX_EXPORT_SERVICE_IOY_Y`)

It accepts a strict bytestring and a boolean value, and returns a lazy bytestring (chunks of data). If the boolean argument is *True* then this service has never been called before in this worker process: this can be used to initialize some global data needed by the service on the first call.

Services are declared with Nginx directive *haskell_run_service*. As far as they are not bound to requests, the directive is only available on the *http* configuration level.

```
    haskell_run_service getUrlService $hs_service_httpbin "http://httpbin.org";
```

The first argument is, as ever, the name of a Haskell handler, the second — a variable where the service result will be put, and the third argument is data passed to the handler *getUrlService* in its first parameter. Notice that the third argument cannot contain variables because variable handlers in Nginx are only available in a request context, hence this argument may only be a static string.

Asynchronous services are bound to the Nginx event loop in the same way as asynchronous tasks. When a service finishes its computation, it pokes data into buffers and writes into eventfd (or a pipe's write end). Then the event handler immediately restarts the service with the boolean argument equal to *False*. This is responsibility of the author of a service handler to avoid dry runs and make sure that it is called not so often in a row. For example, if a service polls periodically, then it must delay for this time itself like in the following example.

An example

Let's retrieve content of a specific URL, say *httpbin.org*, in background. Data will update every 20 seconds.

File test.hs (*additions*)

```haskell
import           Network.HTTP.Client
import           Control.Exception
import           System.IO.Unsafe
import           Control.Monad

-- ...

httpManager :: Manager
httpManager = unsafePerformIO $ newManager defaultManagerSettings
{-# NOINLINE httpManager #-}

getUrl :: ByteString -> IO C8L.ByteString
getUrl url = catchHttpException $ getResponse url $ flip httpLbs httpManager
    where getResponse u = fmap responseBody . (parseRequest (C8.unpack u) >>=)

catchHttpException :: IO C8L.ByteString -> IO C8L.ByteString
```

```
catchHttpException = (`catch` \e ->
        return $ C8L.pack $ "HTTP EXCEPTION: " ++ show (e :: HttpException))

getUrlService :: ByteString -> Bool -> IO L.ByteString
getUrlService url firstRun = do
    unless firstRun $ threadDelay $ 20 * 1000000
    getUrl url
ngxExportServiceIOYY 'getUrlService
```

The *httpManager* defines a global state, not to say a *variable*: this is an asynchronous HTTP client implemented in module *Network.HTTP.Client*. Pragma *NOINLINE* ensures that all functions will refer to the same client object, i.e. it will nowhere be inlined. Functions *getUrl* and *catchHttpException* are used in our service handler *getUrlService*. The handler waits 20 seconds on every run except the first, and then runs the HTTP client. All HTTP exceptions are caught by *catchHttpException*, others hit the handler on top of the custom Haskell code and get logged by Nginx.

File test.conf (*additions*)

```
    haskell_run_service getUrlService $hs_service_httpbin "http://httpbin.org";

    # ...

        location /httpbin {
            echo $hs_service_httpbin;
        }
```

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin'
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv='content-type' value='text/html;charset=utf8'>
  <meta name='generator' value='Ronn/v0.7.3 (http://github.com/rtomayko/
      ronn/tree/0.7.3)'>
  <title>httpbin(1): HTTP Client Testing Service</title>

...
```

This must run really fast because it shows data that has already been retrieved by the service, requests do not trigger any network activity with *httpbin.org* by themselves!

Termination of a service

Services are killed on a worker's exit with Haskell asynchronous exception *ThreadKilled*. Then the worker waits *synchronously* until all of its services' threads exit, and calls *hs_exit()*. This scenario has two important implications.

1. The Haskell service handler may catch *ThreadKilled* on exit and make persistency actions such as writing files if they are needed.

2. *Unsafe blocking* FFI calls must be avoided in service handlers as they may hang the Nginx worker, and it won't exit. Using *interruptible* FFI fixes this problem.

Shared services

An asynchronous service may store its result in shared memory accessible from all worker processes. This is achieved with directive *haskell_service_var_in_shm*. For example, the following declaration (in *http* clause),

```
haskell_service_var_in_shm httpbin 512k /tmp $hs_service_httpbin;
```

makes service *getUrlService*, that stores its result in variable *hs_service_httpbin*, shared. The first argument of the directive — *httpbin* — is an identifier of a shared memory segment, *512k* is its maximum size, */tmp* is a directory where *file locks* will be put (see below), and *$hs_service_httpbin* is the service variable.

Shared services are called *shared* not only because they store results in shared memory, but also because at any moment of the Nginx master lifetime there is only one worker that runs a specific service. When workers start, they race to acquire a *file lock* for a service, and if a worker wins the race, it holds the lock until it exits or dies. Other workers' services of the same type wait until the lock is freed. The locks are implemented via POSIX *advisory* file locks, and so require a directory where they will be put. The directory must be *writable* to worker processes, and */tmp* seems to be a good choice in general.

Update variables

The active shared service put the value of the shared variable in a shared memory, services on other workers wait and do nothing else. Requests may come to any worker (with active or inactive services), fortunately the service result is shared and they can return it as is. But what if the result must be somehow interpreted by Haskell handlers before returning it in the response? Could the handlers just peek into the shared memory and do what they want with the shared data? Unfortunately, not: the shared memory is accessible for reading and writing only from the Nginx part!

Does it mean that we have only one option to let the Haskell part update its global state unavailable in inactive workers: passing values of shared variables into the Haskell part on every request? This would be extremely inefficient. Update variables is a trick to avoid this. They evaluate to the corresponding service variable's value only when it changes in the shared memory since the last check in the current worker, and to an empty string otherwise. Every service variable has its update variable counterpart which name is built from the service variable's name with prefix *_upd___*.

An example

Let's extend our example with loading a page in background. We are still going to load *httpbin.org*, but this time let's assume that we have another task, say extracting all links from the page and showing them in the response sorted. For that we could add a Haskell handler, say *sortLinks*, and pass to it all the page content on every request. But the page may appear huge, let's extract all the links from it and put them into a global state using update variable *_upd___hs_service_httpbin*. In this case function *sortLinks* must be impure, as it must be able to read from the global state.

File test.hs (*additions*)

```haskell
{-# LANGUAGE OverloadedStrings #-}

-- ...

import           Data.IORef
import           Text.Regex.PCRE.ByteString
import           Text.Regex.Base.RegexLike
import qualified Data.Array as A
import           Data.List

-- ...

gHttpbinLinks :: IORef [ByteString]
gHttpbinLinks = unsafePerformIO $ newIORef []
{-# NOINLINE gHttpbinLinks #-}

grepLinks :: ByteString -> [ByteString]
grepLinks v =
    map (fst . snd) . filter ((1 ==) . fst) . concatMap A.assocs .
        filter (not . null) . concatMap (matchAllText regex) $
            C8.split '\n' v
    where regex = makeRegex $ C8.pack "a href=\"([^\"]+)\"" :: Regex

grepHttpbinLinks :: ByteString -> IO L.ByteString
grepHttpbinLinks "" = return ""
grepHttpbinLinks v  = do
    writeIORef gHttpbinLinks $ grepLinks v
    return ""
ngxExportIOYY 'grepHttpbinLinks

sortLinks :: ByteString -> IO L.ByteString
sortLinks "httpbin" = do
    links <- readIORef gHttpbinLinks
    return $ L.fromChunks $ sort $ map (`C8.append` "\n") links
sortLinks _ = return ""
ngxExportIOYY 'sortLinks
```

Here *gHttpbinLinks* is the global state, *grepHttpbinLinks* is a handler for update variable *_upd___hs_service_httpbin*, almost all the time it does nothing — just returns an empty string, but when the update variable becomes not empty, it updates the global state and returns an empty string again. Handler *sortLinks* is parameterized by data identifier: when it's equal to *httpbin*, it reads the global state and returns it sorted, otherwise it returns an empty string.

File test.conf (*additions*)

```
    haskell_service_var_in_shm httpbin 512k /tmp $hs_service_httpbin;
```

```
# ...

    location /httpbin/sortlinks {
        haskell_run grepHttpbinLinks $_upd_links_ $_upd__hs_service_httpbin;
        haskell_run sortLinks $hs_links "${_upd_links_}httpbin";
        echo $hs_links;
    }
```

We have to pass variable *_upd_links_* in *sortLinks* because this will trigger update in the worker by *grepHttpbinLinks*, otherwise update won't run: remember that Nginx directives are lazy? On the other hand, *_upd_links_* is always empty and won't mess up with the rest of the argument — value *httpbin*.

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin/sortlinks'
/
/absolute-redirect/6
/anything
/basic-auth/user/passwd
/brotli
/bytes/1024

...
```

Update callbacks

There is a special type of single-shot services called update callbacks. They are declared like

```
haskell_service_var_update_callback cb_httpbin $hs_service_httpbin optional_value;
```

Here *cb_httpbin* is a Haskell handler exported by *ngxExportServiceIOYY* as always. Variable *hs_service_httpbin* must be declared in directive *haskell_service_var_in_shm* (this matches our example). Argument *optional_value* is a string, it can be omitted in which case handler *cb_httpbin* gets an empty string as its first argument.

Update callbacks do not return results. They run from a worker that holds the active service on every change of the service variable, and shall be supposedly used to integrate with other Nginx modules by signaling specific Nginx locations via an HTTP client.

Shm stats variables

Every service variable in shared memory has an associated auxiliary variable that provides basic stats in format *timestamp | size | changes | failures | failed*, where *timestamp* is a number of seconds elapsed from the beginning of the *UNIX epoch* till the last change of the variable's value, *size* is the size of the variable in bytes, *changes* is a number of changes, and *failures* is a number of memory allocation failures since the last Nginx reload, the value of flag *failed* (*0* or *1*) denotes if the last attempt of memory allocation from the shared memory pool for a new value of the variable has failed. The name of the shm stats variable is built from the service variable's name with prefix *__shm__*.

An example

Let's add a location to show shm stats about our *httpbin* service. This time only configuration file *test.conf* is affected.

File test.conf (*additions*)

```
        location /httpbin/shmstats {
            echo "Httpbin service shm stats: $_shm__hs_service_httpbin";
        }
```

Run curl tests.

```
curl 'http://127.0.0.1:8010/httpbin/shmstats'
Httpbin service shm stats: 1516274639 | 13011 | 1 | 0 | 0
```

From this output we can find that payload size of *httpbin.org* is *13011* bytes, the service variable was updated only once (less than 20 seconds elapsed from start of Nginx), and that there were no memory allocation failures.

Efficiency of data exchange between Nginx and Haskell parts

Haskell handlers may accept strings (`String` or `[String]`) and *strict* bytestrings (`ByteString`), and return strings, *lazy* bytestrings and booleans. Input C-strings are marshaled into a *String* with *peekCStringLen* which has linear complexity $O(n)$, output *Strings* are marshaled into C-strings with *newCStringLen* which is also $O(n)$. The new C-strings get freed upon the request termination in the Nginx part.

The bytestring counterparts are much faster. Both input and output are $O(1)$, using *unsafePackCStringLen* and a Haskell *stable pointer* to lazy bytestring buffers created inside Haskell handlers. If an output lazy bytestring has more than one chunk, a new single-chunked C-string will be created in variable and service handlers, but not in content handlers because the former use the chunks directly when constructing contents. Holding a stable pointer to a bytestring's chunks on the Nginx part ensures that they won't be garbage collected until the pointer gets freed. Stable pointers get freed upon the request termination for variable and content handlers, and before the next service iteration for service handlers.

Complex scenarios may require *typed exchange* between Haskell handlers and the Nginx part using *serialized* data types such as Haskell records. In this case *bytestring* flavors of the handlers would be the best choice. There are two well-known serialization mechanisms: *packing Show* / *unpacking Read* and *ToJSON* / *FromJSON* from Haskell package *aeson*. In practice, *Show* is basically faster than *ToJSON*, however in many cases *FromJSON* outperforms *Read*.

A variable handler of a shared service makes a copy of the variable's value because shared data can be altered by any worker at any moment, and there is no safe way to hold a reference to a shared data without locking. In contrast, a variable handler of a normal per-worker service shares a reference to the value with the service. Obviously, this is still not safe. Imagine that some request gets a reference to a service value from the variable handler, then lasts some time and later uses this reference again: the reference could probably be freed by this time because the service could have altered its data since the beginning of the request. This catastrophic situation could have been fixed by using a copy of the service value in every request like in shared services, but this would unnecessarily hit performance, therefore requests share *counted references*, and as soon as the count reaches *0*, the service value gets freed.

Exceptions in Haskell handlers

There is no way to catch exceptions in *pure* handlers. However they can arise from using *partial* functions such as *head* and *tail*! Switching to their *total* counterparts from module *Safe* can mitigate this issue, but it is not possible to eliminate it completely.

Fortunately, all exceptions, synchronous and asynchronous, are caught on top of the module's Haskell code. If a handler does not catch an exception itself, the exception gets caught higher and logged by Nginx. However, using exception handlers in Haskell handlers, when it's possible, should be preferred.

Summary table of all Nginx directives of the module

Directive	Level	Comment
`haskell compile`	`http`	Compile Haskell code from the last argument. Accepts arguments *threaded* (use *threaded* RTS library) and *standalone* (use *standalone* approach).
`haskell load`	`http`	Load specified Haskell library.
`haskell ghc_extra_options`	`http`	Specify extra options for GHC when the library compiles.
`haskell rts_options`	`http`	Specify options for Haskell RTS.
`haskell program_options`	`http`	Specify program options. This is just another way for passing data into Haskell handlers.
`haskell_run`	`server`, `location`, `location if`	Run a synchronous Haskell task.
`haskell_run_async`	`location`, `location if`	Run an asynchronous Haskell task.
`haskell_run_async_on_request_body`	`location`, `location if`	Run an asynchronous Haskell request body handler.
`haskell_run_service`	`http`	Run a Haskell service.
`haskell_service_var_update_callback`	`http`	Declare a callback on a service variable's update.

Directive	Level	Comment
`haskell_content`	`location`, `location if`	Declare a Haskell content handler.
`haskell_static_content`	`location`, `location if`	Declare a static Haskell content handler.
`haskell_unsafe_content`	`location`, `location if`	Declare an unsafe Haskell content handler.
`haskell_async_content`	`location`, `location if`	Declare an asynchronous Haskell content handler.
`haskell_var_nocacheable`	`http`	All variables in the list become no cacheable and safe for using in ad-hoc iterations over *error_page* cycles.
`haskell_var_compensate_uri_changes`	`http`	All variables in the list allow to cheat *error_page* when used in its redirections and make the cycle infinite.
`haskell_service_var_ignore_empty`	`http`	Do not write the service result when its value is empty.
`haskell_service_var_in_shm`	`http`	Store the service result in a shared memory. Implicitly declares a shared service.