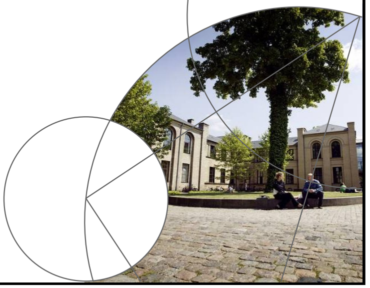
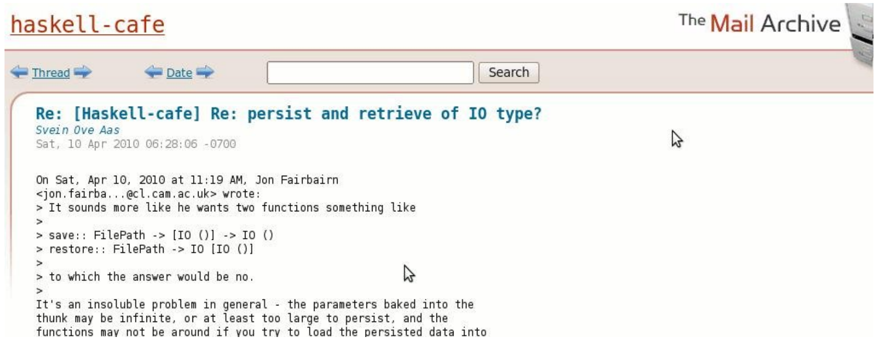
Faculty of Science

Runtime-Supported Serialisation in Haskell - an API

Jost Berthold
berthold@diku.dk
Department of Computer Science



Some Haskell users have interesting questions ...



haskell-cafe The Mail Archive

Thread Date Search

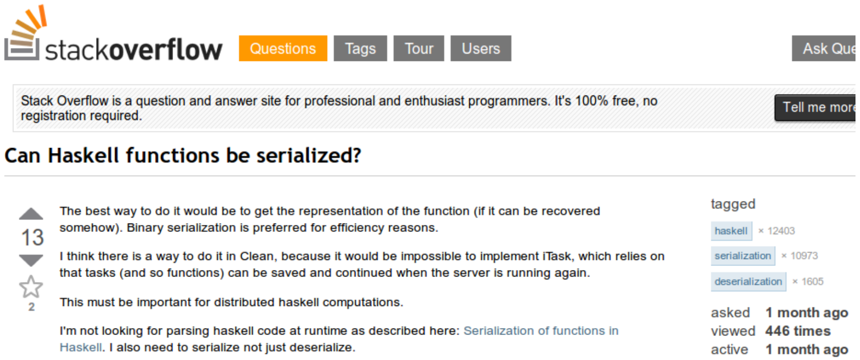
Re: [Haskell-cafe] Re: persist and retrieve of IO type?
Svein Ove Aas
Sat, 10 Apr 2010 06:28:06 -0700

```
On Sat, Apr 10, 2010 at 11:19 AM, Jon Fairbairn
<jon.fairba...@cl.cam.ac.uk> wrote:
> It sounds more like he wants two functions something like
>
> save:: FilePath -> [IO ()] -> IO ()
> restore:: FilePath -> IO [IO ()]
>
> to which the answer would be no.
>
It's an insoluble problem in general - the parameters baked into the
thunk may be infinite, or at least too large to persist, and the
functions may not be around if you try to load the persisted data into
```

...two functions something like

```
save::  FilePath -> [IO ()] -> IO()
restore::  FilePath -> IO [IO ()]
```


Some Haskell users have interesting questions ...



stackoverflow Questions Tags Tour Users Ask Que

Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required. Tell me mor

Can Haskell functions be serialized?

13

2

The best way to do it would be to get the representation of the function (if it can be recovered somehow). Binary serialization is preferred for efficiency reasons.

I think there is a way to do it in Clean, because it would be impossible to implement iTask, which relies on that tasks (and so functions) can be saved and continued when the server is running again.

This must be important for distributed haskell computations.

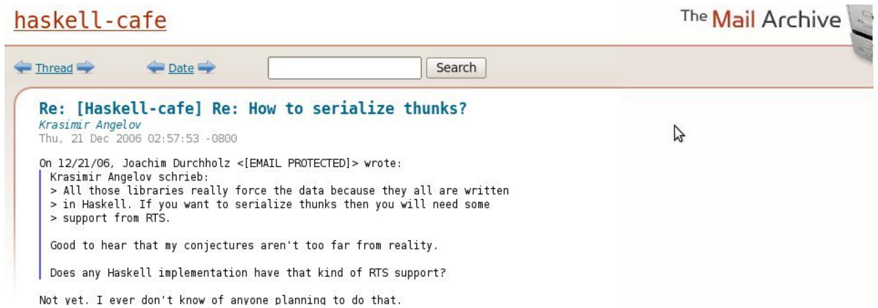
I'm not looking for parsing haskell code at runtime as described here: Serialization of functions in Haskell. I also need to serialize not just deserialize.

tagged
haskell × 12403
serialization × 10973
deserialization × 1605

asked 1 month ago
viewed 446 times
active 1 month ago

Can Haskell functions be serialized? ...
...This must be important for distributed haskell computations.

Some Haskell users have interesting questions ...



haskell-cafe The Mail Archive

Thread Date Search

Re: [Haskell-cafe] Re: How to serialize thunks?
Krasimir Angelov
Thu, 21 Dec 2006 02:57:53 -0800

```
On 12/21/06, Joachim Durchholz <[EMAIL PROTECTED]> wrote:
Krasimir Angelov schrieb:
> All those libraries really force the data because they all are written
> in Haskell. If you want to serialize thunks then you will need some
> support from RTS.

Good to hear that my conjectures aren't too far from reality.

Does any Haskell implementation have that kind of RTS support?

Not yet. I ever don't know of anyone planning to do that.
```

All those libraries really force the data ...
...to serialize thunks [..] you will need some support from RTS ...

Support for serialisation

- Converting a data structure into a form which can be externally stored and later retrieved.
- Focus can be:
  - language interoperability (e.g. XML, JSON) (not addressed today)
  - easy and efficient load/store for applications, persistence
  - communication in a distributed application
- Standard Answers given for Haskell:
  - `Read` and `Show` provide serialisation.
  - The Binary package is faster and more elegant.
- Not a good match for Haskell:
  Not purely functional. How to treat functions?
  Undesired strictness. How to serialise thunks?

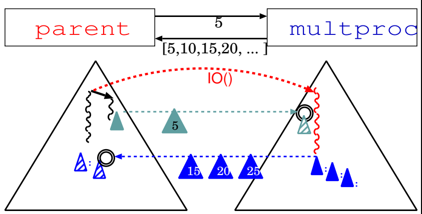
Another route to serialisation support

- Use parallel Haskell runtime system support for data transfer
- Separable from other aspects of parallelism support [Ber11].
- Problems: No safety net, not even types.
- This talk:
  - Presents basic technique and limitations
  - Makes proposals for an extended and more robust API
  - Briefly outlines applications

Serialisation in a parallel Haskell runtime (Eden)

Parallel Haskell dialect Eden:

```
let multproc = process (\n -> [n,2*n..])
    result = multproc # 5
in zipWith f result [1..limit]
```



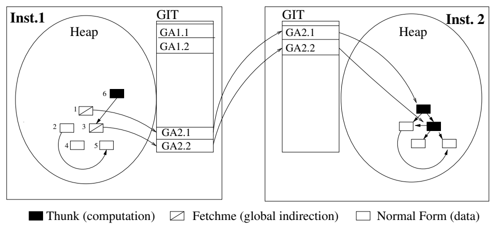
- Parallel Processes, applying a function to one argument
- Hyperstrict in argument and result

  Typed communication channels between processes (no sharing)
  - Stream communication for lists
  - Concurrency for tuples
- Same mechanism for process instantiation (IO-monadic internally)
  Runtime support orthogonal to evaluation.

Serialisation in a parallel Haskell runtime (GUM) [THM+95]

Glasgow Parallel Haskell

```
let result1 = map (*5) [1..limit]
              `using` seqList rnf
    result2 = ...
in result1 `par` result2 `seq` ...
```



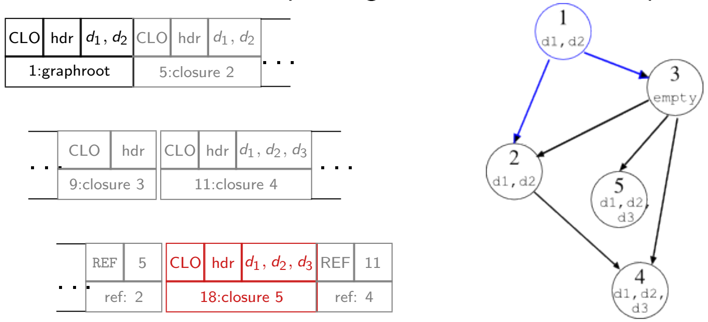
- Sparks: Subexpressions for parallel evaluation
- Fishing: requesting sparks from other nodes

Exporting sparks (thunks) to other nodes

- relocates unevaluated thunks (avoids work duplication), but duplicates evaluated data (avoids overhead),
- allows to fetch results through global addresses

Runtime Support for Serialisation ("Packing")

- Haskell data is graph of closures in the heap
- Breadth-first traversal, packing header data and non-pointers



- Back references for closures already packed
- Cannot touch mutable structures (MVar,TVar, IORef).
- Contains code pointers, can only be deserialised by same binary.

Access to serialisation from Haskell

Access to packing routine by primitive operations

Primitive Operations (2010 version)

```
serialize#   :: a -> State# s -> (# State# s, ByteArray# #)
deserialize# :: ByteArray# -> State# s -> (# State# s, a #)
```

- Haskell heap structure representing the `a` is serialised
- ...into a byte array (itself allocated in the Haskell heap).
- Deserialisation constructs (a copy of) the serialised structure.
- Serialisation operations monadic (`State#` for sequencing).
  `deserialize` conceptually pure, but certainly used in monadic context

(Too) Simple IO Monad Wrapper

```
serialize0   :: a -> IO (UArray Int Word)
deserialize0 :: UArray Int Word -> IO a
```


Trust me, I know what I'm doing...

```
let myNums = [1..10] -- :: [Integer]
blob <- serialize0 myNums
 ...
copy <- deserialize0 blob
let num = length copy + head copy
 ...             -- copy :: [Int]
```

Type defaults can be unlucky...

Phantoms to the rescue!

Typed Serialisation Data (a "packet")

```
data Serialized a = Serialized { packetData :: ByteArray# }

serialize :: a -> IO (Serialized a)
deserialize :: Serialized a -> IO a
```

No tampering with the serialised type.

But if we want to persist values?

Enabling persistence – adding additional information

Persistence – externalising data to retain across program runs

- Reading and writing serialised data externally

Instances of `Serialized` for IO

```
instance Typeable a => Show (Serialized a) --  adds  type fingerprint
   where ...    -- writing ascii format    --    and executable hash
instance Typeable a => Read (Serialized a) -- checks type fingerprint
   where ...    -- parsing ascii format         -- read . show == id --      and execut
instance Typeable a => Binary (Serialized a)
   where ...    -- as above..., uses type fingerprint and executable hash
```

- Save type when writing (in `Show` instance and `put`)
- Check type when reading back in (in `Read` instance and `get`)
- `Typeable` restricts the approach to monomorphic types.

Also: includes a fingerprint of executable

- ensuring that only the same executable can safely decode.

What can possibly go wrong?

```
hdl <- openFile "/etc/passwd" ReadMode
blob <- serialize hdl
  ...            -- must fail!
hClose hdl
  ...
hdl' <- deserialize blob -- ???
  ...
```

Some types just do not make sense to serialise...

...especially those representing: impurity, state, location, effects.

Want operational safety and reliable behaviour

- Generate exceptions for prohibited and internal types
  - no mutable types (MVar, IORef, TVar)
  - no system types (thread id, RTS internal data)

```
txt  <- readFile "/etc/passwd"
 ...
putStrLn (head (lines txt))
 ...
blob <- serialize txt
 ...      -- might fail! :-(
```

Problematic with lazy IO operations!

Concurrent evaluation and serialisation

What if serialisation finds a blackhole?[1]

```
let bigStuff = f input
  ...
buddy <- forkIO (compute bigStuff)
  ...
blob <-trySerialize bigStuff
  ...
```

Two choices:

1. Behave as an evaluator: Block serialising thread on blackhole, retry when evaluated.
2. Behave as an observer: Indicate blocking by an exception to the caller.

- Pro blocking: clear semantics, no leakage
- Pro observing: can have useful applications (clearly not pure ones)

[1] Blackhole: synchronisation node in the heap, representing data currently under evaluation

...and there are more ways to go wrong!

Size of the serialised data should be restricted

```
let size   = 10^6 :: Integer
    big = listArray (0,size) [0 .. size]
    bigger = amap (*2) big
    reallyBig = amap (*3) bigger
print  $ reallyBig!0 -- forces arrays
  ...   -- but leaves elements unevaluated
blob <- serialize reallyBig
  ...          -- 3 x 1M thunks
```

- can produce large arrays (not under programmer's control)
- Packet size should be limited (considerably less than heap)

Implementation uses fixed internal buffers

```
  :
let bin  = encode blob-- use Binary instance
    bin' = tamperWith bin -- have some fun
  ...
copy <- deserialize (decode bin')
  ...       -- must check format!
```

Code must handle corrupted data

- Binary `decode` can fail in Haskell
- `deserialize#` can fail in the runtime system

Summary: Possible Exceptions related to Packing

Pack Exceptions

```
instance Exception PackException
data PackException = P_SUCCESS  -- never used
```

...occurring inside the runtime system

```
    | P_BLACKHOLE      -- found data under evaluation (trySerialize only)
    | P_NOBUFFER       -- buffer too small (size configurable)
    | P_CANNOT_PACK              -- prohibited type found
    | P_UNSUPPORTED | P_IMPOSSIBLE  -- unsupported/impossible type found
    | P_GARBLED                  -- garbled data (deserialize only)
```

...occurring inside Haskell (Read or Binary instances)

```
    | P_ParseError      -- error while reading in serialised data
    | P_BinaryMismatch  -- serialised by a different executable
    | P_TypeMismatch    -- unexpected data type
  deriving ( Eq, Ord, Typeable )
```

Refined serialisation support in the runtime

Primitive operations returning RTS error codes

Primitive Operations with error codes

```
serialize#    :: a -> State# s -> (# State# s, Int#, ByteArray# #)
trySerialize# :: a -> State# s -> (# State# s, Int#, ByteArray# #)
deserialize#  :: ByteArray# -> State# s -> (# State# s, Int#, a #)
```

- Occurrence of prohibited closure types (MVar, TVar, IORef) and other internal errors indicated by error codes
- `deserialize#` indicates packet format failures
- `serialize#` may block on synchronisation nodes (blackholes)
- `trySerialize#` never blocks (returns suitable error code)

Serialisation API in Haskell

Haskell API

```
serialize    :: a -> IO (Serialized a) -- throws PackException (RTS)
trySerialize :: a -> IO (Serialized a) -- throws PackException (RTS)
deserialize  :: Serialized a -> IO a   -- throws PackException (RTS)
```

Instances

```
instance Typeable a => Binary (Serialized a)
    where ... -- throws PackException (Haskell)      -- adding / checking
instance Typeable a => Show (Serialized a)           -- type and executable
    where ... -- throws PackException (Haskell)      -- fingerprints
instance Typeable a => Read (Serialized a)
    where ... -- throws PackException (Haskell)
```

Exception type

```
data PackException = P_SUCCESS                        -- never used
    | P_BLACKHOLE | P_NOBUFFER | P_CANNOT_PACK  -- RTS errors
    | P_UNSUPPORTED | P_IMPOSSIBLE | P_GARBLED  -- RTS errors
    | P_ParseError | P_BinaryMismatch | P_TypeMismatch -- Haskell errors
```


Potential Applications

The feature – effectively:

- In a single program: Creating deep copies
- With Binary instance: Persistence, orthogonal to evaluation
- With distribution: Communication and remote execution

Potential applications for runtime-supported serialisation:

- Persistent memoisation of functions across program runs
  Persist memoised function at shutdown, load when running again
- Checkpointing (long-running) monadic action sequences
  Persist intermediate states (with bindings), recover after interruptions
- Easy distributed programming
  Communicate serialised data to evaluate or execute remotely

Persistent function memoisation

- Using off-the-shelf memoisation as a HOF from a library...
  `memo ::  (a -> b) -> a -> b`
- Memoised function can be globally in scope (CAF memoisation):

Persistent memoisation pattern

```
{-# NOINLINE f_memo #-}
f_memo = unsafePerformIO $ decodeFromFile "f_memo.cache" `catch`
                          (\e -> print (e::SomeException) >> return f)
                               where {-# NOINLINE f #-}
                                     f = memo f'
                                     f' x = ... -- can use f recursively
```

- Memoised f loaded at first use, in global scope

```
main = do let x = f_memo ... -- first use
          ...
          let y = f_memo ... -- in memory
          ...
          f_memo `seq` encodeToFile "f_memo.cache" f_memo
```

Checkpointed versions for monad combinators

Serialise m a to a file before running

```
checkpoint :: (MonadIO m, Typeable a, Typeable m) => FilePath -> m a -> m a
```

Try to deserialise m a from file and run it, else use second arg.

```
recovering :: (MonadIO m, Typeable a, Typeable m) => FilePath -> m a -> m a
```

Checkpointed Monad Combinators

```
sequenceC :: (Typeable a, Typeable m, MonadIO m) => FilePath -> [m a] -> m [a]
sequenceC  _    [] = return []                                 --------------
sequenceC name ms = recovering name (seqC_acc [] ms) -- should use Traversable!
    where seqC_acc acc    []    = return (reverse acc)
          seqC_acc acc (m:ms) = do x <- m
                                   checkpoint name $
                                     seqC_acc (x:acc) ms

mapMC    file  f   xs = sequenceC file (map f xs)
filterMC file pred xs = do flgs <- mapMC ("filterMC"++file) pred xs
                           return [ x | (x,True) <- zip xs flgs ]
  ...
```


Distributed Haskells

Haskell-distributed parallel Haskell (Maier, Stewart, Trinder)[MST13]

HdpH: task distribution (Par monad)

```
type   Par a        -- Par monad computation returning type 'a'
type   Closure  a  -- serialisable closure of type 'a'
pushTo :: PE ->  Closure  (Par ()) -> Par () -- eager explicit
spark  ::        Closure  (Par ()) -> Par () -- lazy implicit
```

HdpH: Communication via IVars

```
type IVar a     -- write-once buffer of type 'a'
type GIVar a    -- global handle to an 'IVar a'

new   :: Par(IVar a)              -- creation
glob :: IVar a -> Par(GIVar a) -- globalisation
rput :: GIVar(  Closure  a) ->  Closure  a  -> Par() -- remote write
probe:: IVar a -> Par Bool      -- local test
get   :: IVar a -> Par a        -- local read
```


Distributed Haskells

Haskell-distributed parallel Haskell (Maier, Stewart, Trinder)[MST13]

HdpH: task distribution (Par monad)

```
type   Par a        -- Par monad computation returning type 'a'
-- using Serialized a instead of Closure a
pushTo :: PE -> Serialized(Par ()) -> Par () -- eager explicit
spark  ::       Serialized(Par ()) -> Par () -- lazy implicit
```

HdpH: Communication via IVars

```
type IVar a     -- write-once buffer of type 'a'
type GIVar a    -- global handle to an 'IVar a'

new   :: Par(IVar a)              -- creation
glob :: IVar a -> Par(GIVar a) -- globalisation
rput :: GIVar(Serialized a) -> Serialized a -> Par() -- remote write
probe:: IVar a -> Par Bool      -- local test
get   :: IVar a -> Par a        -- local read
```


Distributed Haskells

Haskell-distributed parallel Haskell (Maier, Stewart, Trinder)[MST13]

HdpH: task distribution (Par monad)

```
type   Par a        -- Par monad computation returning type 'a'
-- using serialisation internally (inside pushTo and spark)
pushTo :: PE ->   Par ()           -> Par () -- eager explicit
spark  ::         Par ()           -> Par () -- lazy implicit
```

HdpH: Communication via IVars

```
type IVar a     -- write-once buffer of type 'a'
type GIVar a    -- global handle to an 'IVar a'

new   :: Par(IVar a)              -- creation
glob :: IVar a -> Par(GIVar a) -- globalisation
rput :: GIVar(     a       ) ->      a       -> Par() -- remote write
probe:: IVar a -> Par Bool      -- local test
get   :: IVar a -> Par a        -- local read
```

Distributed Haskells

Haskell-distributed parallel Haskell (Maier, Stewart, Trinder)[MST13]

HdpH: task distribution (Par monad)

```
type   Par a       -- Par monad computation returning type 'a'
-- using Serialized a instead of Closure a
pushTo :: PE -> Serialized(Par ()) -> Par () -- eager explicit
spark  ::       Serialized(Par ()) -> Par () -- lazy implicit
```

Similar option for Cloud Haskell (Epstein, Peyton-Jones, Black)[EBPJ11]

Cloud Haskell

```
-- core operation , here with Serialized instead of Closure
spawn :: NodeId -> Serialized (Process ()) -> Process ProcessId -- remote exec.
```

On the other hand: Closure/Static approach created to restrict serialisation (avoiding prohibited types)

Directions for future distributed Haskell?

Closure/Static approach in Cloud Haskell and HdpH

- Compile-time closure conversion (code inserted by programmer)
- Avoids capturing prohibited types – and other failures

Runtime-supported serialisation explained here

- Exceptions and runtime checks (handlers inserted by programmer)
- Fully delivers on call-by-need
- The application code itself is typically short and simple

There should be a useful combination!

And more exciting work to do:
Adaptive scheduling, GUM global addresses – within Haskell

Status and perspective

- Basic support was available since Eden-6.12.
  (no error handling, blackhole blocking semantics, not thread-safe)
- New version will be included in Eden-7.8 (just around the corner)
  Modified primitive operations, better fault tolerance, error codes
- Source code:

  Parallel Haskell runtime Eden main development repository
  `http://james.mathematik.uni-marburg.de:8080/gitweb/`
  (also here: `https://github.com/jberthold/ghc/`)

  Haskell parts (as described here) soon available as a package
  (runtime support required for installation)
  `https://github.com/jberthold/rts-serialisation/`

Conclusions

- Alternative approach to Haskell serialisation
- Proposed an extended Haskell API to recover from failures
  (advocating explict failure handling)
- Useful applications: (some specific to this approach)
  Memoisation, Checkpointing, Distributed programming

Your contributions are most welcome!
– using it – improving it – revising it –

Jost Berthold.
Orthogonal Serialisation for Haskell.
In Jurriaan Hage and Marco Morazan, editors, *IFL'10, 22nd Symposium on Implementation and Application of Functional Languages*, Springer LNCS 6647, pages 38–53, 2011.

Jeff Epstein, Andrew P. Black, and Simon Peyton-Jones.
Towards haskell in the cloud.
In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 118–129, New York, NY, USA, 2011. ACM.

Patrick Maier, Rob Stewart, and Phil Trinder.
Reliable scalable symbolic computation: the design of SymGridPar2.
In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1674–1681, New York, NY, USA, 2013. ACM.

Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., Andrew S. Partridge, and Simon L. Peyton Jones.
GUM: a Portable Parallel Implementation of Haskell.
In *IFL'95: International Workshop on the Implementation of Functional Languages*, 1995.