Some Haskell users have interesting questions...

Can Haskell functions be serialized?

...This must be important for distributed haskell computations.
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:

```haskell
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

```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

```
1: graphroot
  CLO hdr 1, 2

2: closure 2
  CLO hdr 1, 2
  D 1

3: closure 3
  CLO hdr 1, 2

4: closure 4
  CLO hdr 1, 2

5: closure 5
  CLO hdr 1, 2
  REF 5

6: closure 6
  CLO hdr 1, 2

7: closure 7
  CLO hdr 1, 2

8: closure 8
  CLO hdr 1, 2

9: closure 9
  CLO hdr 1, 2

10: closure 10
    CLO hdr 1, 2
    REF 5

11: closure 11
    CLO hdr 1, 2
```

- 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).
- Deserialise 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)`
- `instance Typeable a => Read (Serialized a)`
- `instance Typeable a => Binary (Serialized a)`

- 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?

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)

Problematic with lazy IO operations!

Concurrent evaluation and serialisation

What if serialisation finds a blackhole?1

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.

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 |

copy <- deserialize (decode bin')

... -- must check format!

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 15/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 16/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 17/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 18/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 19/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 20/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 21/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 22/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 23/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 24/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 25/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 26/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 27/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 28/29

J.Berthold — Haskell RTS-Serialisation API — HIW'13, Boston
Slide 29/29
Refined serialisation support in the runtime

Primitive operations returning RTS 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

```
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 ...
```

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

Persist memoised function at shutdown, load when running again

```
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]

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 Havkels

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 a) -> Par a -- eager explicit
spark :: Serialized(Par a) -> Par a -- lazy implicit
```

HdpH: Communication via IVars

```
type Par 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) -> Serialised a -> Par a -- remote write
probe:: IVar a -> Par Bool -- local test
get :: IVar a -> Par a -- local read
```

Distributed Haskell

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 a -> Par a -- eager explicit
spark :: Par a -> Par a -- lazy implicit
```

HdpH: Communication via IVars

```
type Par 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 a -- 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)

```haskell
 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

```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 explicit failure handling)
- Useful applications: (some specific to this approach)
  Memoisation, Checkpointing, Distributed programming

Your contributions are most welcome!
- using it – improving it – revising it –
Orthogonal Serialisation for Haskell.

Towards haskell in the cloud.

Reliable scalable symbolic computation: the design of SymGridPar2.

GUM: a Portable Parallel Implementation of Haskell.