Student Project Proposal
First Class Serialization for Distributed Haskells

Rob Stewart

24th October, 2013

1 Project Goals

This section describes at a high level the 3 possible directions this project could take. They are disseminated in Section 4.

1. **Port HdpH Serialization to Eden RTS** Develop a HdpH prototype that uses the Eden RTS instead of the vanilla GHC RTS. The Eden RTS provides first class serialization support. Section 4.1.

2. **Simplify HdpH API** A simplified HdpH API has been proposed [2] using the Eden serialization API. An open question remains whether the programmer should be burdened with the task of serialization, or instead have the serialization mechanisms hidden from the programmer. Section 4.2.

3. **Multi-paradigm Parallel Programming Models** A CloudHaskell fork that also uses the Eden serialization support opens up a new direction for the project. CloudHaskell is an implementation of the actor model. HdpH is an implementation of the dataflow model. The two models are quite different. Even if the Eden RTS provides a unified serialization mechanism, it is an open question whether an intersection of CloudHaskell and HdpH could reveal a useful multi-paradigm parallel programming model. Section 4.3.

2 Background on Distributed Haskells

Data serialization is a crucial feature of real-world programming languages. Questions arise when adding serialization support to a programming language that uses demand-driven lazy evaluation and higher-order functions, such as Haskell. Haskell is a purely functional lazy language, and has good support for concurrent and parallel execution on multicore machines. Extensions of the GHC runtime system (RTS) have been developed to scale Haskell program execution to multi-node architectures. Examples are GUM [12] and Eden [8]. Both support serialization at the C level.

More recent approaches to scaling Haskell over distributed-memory architectures of this kind have focused on the development of embedded domain specific languages (EDSL), written in Haskell. Two known implementations are CloudHaskell (2011) [7] and HdpH (2012) [10]. They lift the RTS scheduling, closure serialization and message passing in to Haskell. They propose subtly different solutions to data and function closure serialization at the Haskell level. Both rely on Template Haskell

```
declareStatic :: StaticDecl
declareStatic =
  mconcat
    [HdpH.declareStatic,
     declare (staticToClosure :: StaticToClosure Int),
     declare (staticToClosure :: StaticToClosure Integer),
     declare $(static 'dist_fib_abs)]
```

Figure 1: Eyesore: Example of static declaration registration in HdpH

```
remotableDecl [
    [d| fib :: (NodeId,Int) → Process Integer ;
        fib (_,0) = return 0
        fib (_,1) = return 1
        fib (myNode,n) = do
          let tsk = remoteTask ($(functionTDict 'fib)) myNode ($(mkClosure 'fib) (myNode,n-2))
          future ← async tsk
          y ← fib (myNode,n-1)
          (AsyncDone z) ← wait future
          return $ y + z
      |]
  ]
```

Figure 2: Eyesore: Template Haskell for Recursive a Cloud Haskell Process

splicing to implement a `mkClosure` primitive, which automatically generates the code necessary to invoke remote functions on other nodes.

The GHC compiler lacks support for static[1]. In a distributed setting where all nodes are running the same executable, static values can be serialized simply by transmitting a code pointer to the value. This behaviour is instead mimicked in HdpH and CloudHaskell by keeping an explicit mapping from labels to values, using Template Haskell. This introduces unnecessary boilerplate code used to define static remote tables, as demonstrated in Figure 1. The static mimickry in HdpH and CloudHaskell is separated in to separate packages, `hpdh-closure` [9] and `distributed-static` [5] respectively.

There are two limitations of the Template Haskell approach. The first is somewhat obvious — unwieldy boiler plate Template Haskell code is interspersed with application code. The Template Haskell mimickry can quickly become distracting and hard to read. An example of this is in Figure 2, which shows a recursive CloudHaskell implementation of Fibonacci. So not only does the programmer need to write Template Haskell notation for creating closures (with `mkClosure`) and in some cases defining serializable closures (Figure 2), they must also define their static declaration mapping (Figure 1).

The second limitation is a constraint on programming style when creating function closures, enforced by Template Haskell splicing. Serialized functions must either be top a static closure, or a closure abstraction applied to a tuple of local variables i.e. a closure abstraction with tuple arity of one. This constraint on programming style is demonstrated in the Fibonacci example later in Figure 7.

The consequence of second class serialization support has resulted in the emergence of largely incompatible distributed Haskell implementations, namely Eden, GUM, HdpH and CloudHaskell. The Eden and GUM RTS's and closure serialization packing code are written in C. HdpH and CloudHaskell are written entirely in Haskell and use Template Haskell mimickry for function clo-

[1]An issue ticket has been raised for this [4]

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

Figure 3: Eden Serialization API

sure creation. There has been some common ground found, between the CloudHaskell and HdpH technologies by sharing a network transport layer [3], though the serialization issue has resulted in a non-uniform closure representation at the exposed API level. As a result, the two technologies can presently not be composed to implement a solution to an application problem.

3 Serialization Support in Eden

3.1 Background

Whilst there is no first class serialization support in GHC, there is in Eden. Eden is a modest language extension to Haskell, with a supporting RTS for distributed-memory architectures. It is a fork from the GHC RTS, and continues to closely rebase on GHC releases. The Eden-7.8 release will, for example, be released shortly after the GHC-7.8 release in November 2013.

Basic support for serialization has been available in Eden since the 6.12 release in June 2011. The version based on GHC-7.8 has better fault handling with error codes, blackhole blocking semantics and improved serialization primitives. The serialization support in Eden is driven by Jost Berthold, who recently presented (September 2013) the current Eden status at Haskell Implementers Workshop [2].

3.2 Eden Serialization Primitives

The Eden serialization API is shown in Figure 3. This serialization support is orthogonal to evaluation. It supports the packing and unpacking of (almost) any kind of value, including functions and IO actions. Prohibited types for serialization are MVar's, TVar's and IORef's. Attempts to serialize these will result in a runtime system `PackException` error being thrown. The `serialize` primitive attempts to serialize a value and may block on synchronization nodes i.e. blackholes. The `trySerialize` primitive attempts to serialize a value and never blocks, returning instead a suitable error code

The simple API in Figure 3 is an adequate drop-in replacement for the Template Haskell mimickry in HdpH and CloudHaskell. This serialization API is suitable for other applications, not just for distributed programming. Other application areas suggested by Jost [1] are persistent memoization and checkpointing. The Eden serialization library could be used to alleviate computational load of frequently used applications, e.g. persisting a memoised function at shutdown, and late re-loading it result immediately to apply to another application. It could also be used to checkpoint parts of a running program as a sequence of IO actions, and storing them e.g. to file for a potential recovery. A checkpointing API has been developed [2] on top of Eden serialization support. A whole other direction for the project would be to utilize this technique for adding a new checkpoint based fault tolerance flavour to CloudHaskell or HdpH-RS [11].

```
checkpoint :: (MonadIO m, Typeable a, Typeable m) ⇒ FilePath → m a → m a
recovering :: (MonadIO m, Typeable a, Typeable m) ⇒ FilePath → m a → m a
```

Figure 4: Checkpointing API Built using Eden Serialization Primitives from Figure 3

```
spawn   ::           Closure (Par (Closure a)) → Par (IVar (Closure a))
spawnAt :: NodeId → Closure (Par (Closure a)) → Par (IVar (Closure a))
get     :: IVar (Closure a) → Par (Closure a)
```

Figure 5: Existing HdpH API

4 Project Dissemination

4.1 A HdpH Prototype with First Class Serialization Support

The project will likely begin by creating a fork of HdpH and switching the method for serialization to use Eden's RTS primitives. This will involve two phases. First, the dependency on `hdph-closure` should be removed. This will involve the removal of all calls to `mkClosure`, `toClosure` and `unClosure` in the `hdph` package. Occurrences of the first two should be replaced with a call to `serialize`, and occurrences of the third replaced with `deserialize`. The second step is to download Eden from the git repository [6] and build it. HdpH at this point should be ready for testing on multiple nodes, no doubt uncovering unforeseen technical challenges to resolve.

The proposed API is shown in Figure 6. Other than the switch from the `Closure` to `Serialized` constructor, the types are the same as before. The important difference, however, is the creation of serialized `Par` computations. No Template Haskell is necessary for creating serialized `Par` computations, or for declaring the static declaration mappings. Instead, the `serialize` primitive will be used.

A Fibonacci implementation using the current HdpH library is shown in Figure 7. The `mkClosure` function is used to create the function closure from the `dist_spawn_fib_abs` closure abstraction and tupled variables.

A Fibonacci implementation using the hypothetical Eden based HdpH library is shown in Figure 8. The restriction previously enforced by `mkClosure` is removed i.e. the `serialize` call takes the `dist_spawn_fib` function with an arity of three. No Template Haskell code is necessary.

4.2 A Simplified HdpH API

The most obvious approach to adopting the Eden serialization support is by simply switching the API from the `Closure` constructors from the `hdph-closure` package, to the `Serialized` constructor in the Eden serialization API. This is exactly what Figure 6 in Section 4.1 proposes. There are alternative approaches, for example in Erlang. Take the proposed verbose HdpH `spawn` primitive from Figure 6 using the Eden `Serialized` constructor:

```
spawn   :: Serialized (Par (Serialized a)) → Par (IVar (Serialized a))
```

```
spawn   ::           Serialized (Par (Serialized a)) → Par (IVar (Serialized a))
spawnAt :: NodeId → Serialized (Par (Serialized a)) → Par (IVar (Serialized a))
get     :: IVar (Serialized a) → Par (Serialized a)
```

Figure 6: Proposed HdpH API using `Serialized` Constructor from Eden API in Figure 3

```
-- | implementations of these omitted
fib     :: Int → Integer
par_fib :: Int → Int → Par Integer

dist_spawn_fib :: Int → Int → Int → Par Integer
dist_spawn_fib seqThreshold parThreshold n
  | n ≤ k    = force $ fib n
  | n ≤ l    = par_fib seqThreshold n
  | otherwise = do
      v ← spawn $(mkClosure [| dist_spawn_fib_abs (seqThreshold, parThreshold, n) |])
      y ← dist_spawn_fib seqThreshold parThreshold (n - 2)
      clo_x ← get v
      force $ unClosure clo_x + y
  where k = max 1 seqThreshold
        l = parThreshold

-- | closured abstraction applied to a tuple of variables
dist_spawn_fib_abs :: (Int, Int, Int) → Par (Closure Integer)
dist_spawn_fib_abs (seqThreshold, parThreshold, n) =
  dist_spawn_fib seqThreshold parThreshold (n - 1) >>= return ∘ toClosure
```

Figure 7: Eyesore: Fibonacci in HdpH using TemplateHaskell and GHC

```
-- | Using Eden serialization primitives
--   Serialization: Responsibility of the programmer
dist_spawn_fib :: Int → Int → Int → Par Integer
dist_spawn_fib seqThreshold parThreshold n
  | n ≤ k    = force $ fib n
  | n ≤ l    = par_fib seqThreshold n
  | otherwise = do
      clo_f ← serialize (dist_spawn_fib seqThreshold parThreshold (n-1))
      v ← spawn clo_f
      y ← dist_spawn_fib seqThreshold parThreshold (n - 2)
      clo_x ← get v
      deserialize clo_x >>= force ∘ flip (+) y
  where k = max 1 seqThreshold
        l = parThreshold
```

Figure 8: Fibonacci Using Eden-based Verbose HdpH API

```
fib(0) → 0;
fib(1) → 1;
fib(X) →
  Key = rpc:async_call(random_node(),fib_rpc,fib,[X-1]),
  Y = fib(X-2),
  Z = rpc:yield(Key),
  Y + Z.
```

Figure 9: Recursive Fibonacci with the Erlang `rpc` module

```
spawn   ::           (Par a) → Par (IVar a)
spawnAt :: NodeId → (Par a) → Par (IVar a)
get     :: IVar a → Par a
```

Figure 10: Simplified HdpH API Proposed in [2]

It takes a serialized `Par` computation returning a serialized value of type `a`. It returns an `IVar` holding a serialized value of type `a` in the `Par` monad. So, not only must the task be serialized before being spawned, the programmer must also de-serialize the value held within the `IVar`. These two packing requirements do not exist in other languages, for example Erlang.

The Erlang code in Figure 9 is an implementation of Fibonacci. The code design is almost identical to the HdpH code in Figure 7 and CloudHaskell code in Figure 2. The Erlang code is nevertheless shorter and cleaner. Not only is it cleaner (due to the Template Haskell boiler plate in the Haskell challengers), but serialization mechanics is hidden from the Erlang programmer. That is, the `fib` function is not packed by the programmer before being passed to the `rpc:async_call` function, and the `Z` value returned by `rpc:yield` is an already unpacked value when returned to the program.

4.3 Multi Paradigm Parallel Programming Model

Unified Closure Representation The wider goal of this work is to unify the serialization mechanisms used by the currently diverged distributed Haskells. Once the first class serialization prototype has been implemented and tested for HdpH, a marriage of distributed Haskells could be investigated. Two open questions arise. First, is it possible to pass function closures between HdpH and CloudHaskell? The supplementary question is *why* would this be a good idea…is this a desirable

```
-- | Using Eden serialization primitives
--   Serialization: Handled by HdpH primitives 'spawn' and 'get
dist_spawn_fib :: Int → Int → Int → Par Integer
dist_spawn_fib seqThreshold parThreshold n
  | n ≤ k    = force $ fib n
  | n ≤ l    = par_fib seqThreshold n
  | otherwise = do
      v ← spawn (dist_spawn_fib seqThreshold parThreshold (n-1))
      y ← dist_spawn_fib seqThreshold parThreshold (n - 2)
      x ← get v
      force $ x + y
  where k = max 1 seqThreshold
        l = parThreshold
```

Figure 11: Fibonacci Using Eden-based Cleaner HdpH API

```
-- * generate explicitly local or explicitly remote async tasks
task       :: Process a → AsyncTask a
remoteTask :: Static (SerializableDict a) → NodeId → Closure (Process a) → AsyncTask a

-- * run async task producing async action & wait on result
asyncSTM :: Serializable a ⇒ AsyncTask a → Process (Async a)
wait :: Async a → Process (AsyncResult a)
```

Figure 12: Async Library API Built using CloudHaskell Primitives

feature?

Combining Programming Models CloudHaskell and HdpH exhibit different programming models. CloudHaskell is an implementation of the actor model. HdpH is an implementation of the dataflow model. There may be technologies in these libraries that, if shared, could be mutually beneficial. HdpH for example features a load balancing scheduler. CloudHaskell has no support for load balancing. CloudHaskell features typed channels for streaming data between processes. HdpH has no support for data streaming.

Actor based programming in CloudHaskell enforces a prescribed way of thinking about shared memory. Actors do not share memory. Actors can fork threads that do share memory, however. All communication is done with explicit message passing. In contrast, HdpH does not conceptualize processes, and all node-local threads can share memory. Communication is almost entirely implicit. For example, an `rput` call invokes node to node message passing in the scheduler, and the load balancing messages is entirely hidden. Given these differences, how might an intersection of these libraries be distilled in to a multi-paradigm parallel programming model? What would it look like?

Example: An Async Library using CloudHaskell with No Load Balancing One promise is how programmers are beginning to use the CloudHaskell library. One example is the CloudHaskell platform [13]. It is a library that implements some Erlang abstractions using CloudHaskell, for example OTP behaviours. One part of the library is an implementation of Erlang's `async` module. Using this library, it is possible to write dataflow style programs similar to the HdpH model.

Scheduling is either explicitly local or explicitly remote, i.e. locally scheduled tasks cannot be load balanced elsewhere. The async API is shown in Figure 12. The `remoteTask` roughly corresponds to HdpH's `spawnAt`, `task` corresponds to `spawn` with the omission of work stealing. The is no equivalent to `asyncSTM` in HdpH, which is instead implied by calls to the spawn primitives. The `wait` primitive corresponds to the HdpH `get` call on `IVar`s.

References

[1] Jost Berthold. Orthogonal serialisation for haskell. In Jurriaan Hage and Marco T. Morazán, editors, *IFL*, volume 6647 of *Lecture Notes in Computer Science*, pages 38–53. Springer, 2010.

[2] Jost Berthold. Runtime-Supported Serialization in Haskell - an API, September 2013.

[3] Duncan Coutts, Nicolas Wu, and Edsko de Vries. Haskell library: `network-transport` package. A network abstraction layer API. `http://hackage.haskell.org/package/network-transport`.

[4] Edsko de Vries. GHC ticket #7015: Add support for 'static', 2012. `http://ghc.haskell.org/trac/ghc/ticket/7015`.

[5] Edsko de Vries. CloudHaskell Closure Library: `distributed-static` package, 2013. `http://hackage.haskell.org/package/distributed-static`.

[6] Eden Developers. Source Code for Eden. `https://github.com/jberthold/ghc`.

[7] Jeff Epstein, Andrew P. Black, and Simon L. Peyton Jones. Towards Haskell in the Cloud. In Koen Claessen, editor, *Proceedings of the 4th ACM SIGPLAN Symposium on Haskell, Haskell 2011, Tokyo, Japan, 22 September 2011*, pages 118–129. ACM, 2011.

[8] Rita Loogen, Yolanda Ortega-Mallén, and Ricardo Peña-Marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, 2005.

[9] Patrick Maier. HdpH Closure Library: `hdph-closure` package, 2013. `http://hackage.haskell.org/package/hdph-closure`.

[10] Patrick Maier, Robert Stewart, and Phil Trinder. Reliable Scalable Symbolic Computation: The Design of SymGridPar2. In *28th ACM Symposium On Applied Computing, SAC 2013, Coimbra, Portugal, March 18-22, 2013*, pages 1677–1684. ACM Press, 2013.

[11] Robert Stewart. *Reliable Massively Parallel Symbolic Computing: Fault Tolerance for a Distributed Haskell.* PhD thesis, Mathematical & Computer Sciences, Heriot-Watt University, September 2013.

[12] Philip W. Trinder, Kevin Hammond, James S. Mattson Jr., A. S. Partridge, and Simon L. Peyton Jones. Gum: A portable parallel implementation of haskell. In Charles N. Fischer, editor, *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, May 21-24, 1996*, pages 79–88. ACM, 1996.

[13] Tim Watson. CloudHaskell Platform Library, 2013. `https://github.com/haskell-distributed/distributed-process-platform`.