DNA

Safe Haskell	None
Language	Haskell2010

DNA is a data flow DSL aimed at expressing data movement and initiation of computational kernels for numerical calculations. We use the "actor/channel" paradigm, which allows for a descriptive approach that separates definition from execution strategy. Our target is data intensive high performance computing applications that employ hierarchical cluster scheduling techniques. Furthermore, we provide infrastructure for detailed profiling and high availability, allowing recovery for certain types of failures.

DNA is presently implemented as an embedded monadic DSL on top of the well-established distributed programming framework "Cloud Haskell". This document describes the structure of the language at a high level, followed by detailed specifications and use cases for the introduced primitives. We will give examples at several points.

Contents

High level structure

DNA programs are composed of actors and channels. DNA provides means for defining an abstract data flow graph using programming language primitives.

Actors are executed concurrently, don't share state and can only communicate using a restricted message passing scheme.

Every actor can receive either one or many inputs of the same type and produce either one or multiple outputs. This depends on the type of the actor: For example, a single `Actor` will only ever accept one input parameter and produce one result. On the other hand, a group of `Actor`s will produce an unordered set of values of same type. Finally, a `CollectActor` receives a group of values while producing a single result. In general, actors have no knowledge where their input parameters come from or where result will be sent, these connections will be made from the outside.

Actors are spawned hierarchically, so every actor but the first will be created by a parent actor. Communication is forced to flow along these hierarchies: Both inputs and results can only be sent to and received from either the parent actor or sibling actors on the same level.

actors in a group, they get assigned ranks from *0* to *N-1*. Conceptually, a group of actors is treated as single actor which runs on several execution elements simultaneously.

To illustrate this, here is example of distributed dot product. We assume that `ddpComputeVector`, `ddpReadVector` and `splitSlice` are already defined:

```
-- Calculate dot product of slice of full vector
ddpProductSlice = actor $ \(fullSlice) -> duration "vector slice" $ do
   -- Calculate offsets
   slices <- scatterSlice <$> groupSize
   slice  <- (slices !!)  <$> rank
   -- First we need to generate files on tmpfs
   fname <- duration "generate" $ eval ddpGenerateVector n
   -- Start local processes
   shellVA <- startActor (N 0) $
       useLocal >> return $(mkStaticClosure 'ddpComputeVector)
   shellVB <- startActor (N 0) $
       useLocal >> return $(mkStaticClosure 'ddpReadVector)
   -- Connect actors
   sendParam slice              shellVA
   sendParam (fname, Slice 0 n) shellVB
   futVA <- delay Local shellVA
   futVB <- delay Local shellVB
   -- Await results
   va <- duration "receive compute" $ await futVA
   vb <- duration "receive read"    $ await futVB
   -- Clean up, compute sum
   kernel "compute sum" [FloatHint 0 (2 * fromIntegral n)] $
     return (S.sum $ S.zipWith (*) va vb :: Double)

-- Calculate dot product of full vector
ddpDotProduct :: Actor Int64 Double
ddpDotProduct = actor $ \size -> do
    -- Chunk & send out
    shell <- startGroup (Frac 1) (NNodes 1) $ do
        useLocal
        return $(mkStaticClosure 'ddpProductSlice)
    broadcast (Slice 0 size) shell
    -- Collect results
    partials <- delayGroup shell
    duration "collecting vectors" $ gather partials (+) 0

main :: IO ()
main = dnaRun (...) $
  liftIO . print =<<  eval ddpDotProduct (400*1000*1000)
```

This generates an actor tree of the following shape:

```
        ddpDotProduct
              |
       ddpProductSlice
       /             \
ddpComputeVector  ddpReadVector
```

Here `ddpDotProduct` is a single actor, which takes exactly one parameter `size` and produces exactly the sum as its output. On the other hand, `ddpProductSlice` is an actor group, which sums up a portion of the full dot-product. Each actor in group spawns two child actors: `ddpComputeVector` and `ddpReadVector` are two child actors, which for our example are supposed to generate or read the requested vector slice from the hard desk, respectively.

Scheduling data flow programs for execution.

Scheduling, spawning and generation of the runtime data flow graph are handled separately. The starting point for scheduling is the cluster architecture descriptor, which describes the resources available to the program.

For DNA, we are using the following simple algorithm: First a control actor starts the program. It's actor which passed to `runDna` as parameter. This actor will be assigned exclusively all resources available to the program, which it can then in turn allocate to it spawn child actors. When a child actor finishes execution (either normally or abnormally), its resources are returned to parent actor's resource pool and can be reused.

High Availability

We must account for the fact that every actor could fail at any point. This could not only happen because of hardware failures, but also due to programming errors. In order to maintain the liveness of the data flow network, we must detect such failures, no matter the concrete reason. In the worst case, our only choice is to simply terminate all child processes and propagate the error to actors which depend on the failed actor. This approach is obviously problematic for achieving fault tolerance since we always have a single point of failure.

To improve stability, we need to make use of special cases. For example, let us assume that a single actor instance in large group fails. Then in some case it makes sense to simply ignore the failure and discard the partial result. This is the "failout" model. To use these semantics in the DNA program, all we need to do is to specify `failout` when spawning the actor with `startGroup`. To make use of failout example above should be changed to:

```
...
shell <- startGroup (Frac 1) (NNodes 1) $ do
    useLocal
    failout
    return $(mkStaticClosure 'ddpProductSlice)
...
```

Another important recovery technique is restarting failed processes. This obviously loses the current state of the restarted process, so any accumulated data is lost. In the current design, we only support this approach for `CollectActor`s. Similarly only change to program is addition of `respawnOnFail` to parameters of actors.

Profiling

For maintaing a robust system performance, we track the performance of all actors and channels. This should allow us to assess exactly how performance is shaped by not only scheduling and resource allocation, but also performance of individual software and hardware components. For example, we might decide to change the scheduling with the goal of eliminating idle times, optimise kernels better or decide to run a kernel on more suitable computation hardware were available.

However, in order to facilitate making informed decisions about such changes, it is not only important to collect raw performance numbers such as time spent or memory consumed. For understanding the performance of the whole system we need to put our measurements into context. This means that we should associate them from the ground up with the data flow structure of the program.

Our approach is therefore to implement profiling as an integral service of the DNA runtime. The generated profile will automatically track the overall performance of the system, capturing timings of all involved actors and channels. Furthermore, wherever possible the data flow program should contribute extra information about its activity, such as number of floating point operations expected or amount of raw data transferred. In the end, we will use the key performance metrics derived from these values in order to visualise the whole system performance in a way that will hopefully allow for painless optimisation of the whole system.

DNA monad

```
data DNA a
```

Monad for defining the behaviour of a cluster application. This concerns resource allocations as well as steering data and control flow.

Instances

```
Monad DNA
Functor DNA
Applicative DNA
```

---

dnaRun

```
:: (RemoteTable -> RemoteTable)
-> DNA ()
-> IO ()
```

Cloud haskell's remote tablse

DNA program

Execute DNA program. First parameter is list of remote tables. Each invocation of `remotable` generate `__remoteTable` top level identifier with type `RemoteTable -> RemoteTable`. All such remote tables must composed using `.` and passed to dnaRun as in following example:

```
dnaRun (ModuleA.__remoteTable . ModuleB.__remoteTable) program
```

UNIX startup. If command line parameter '--nprocs=N' is given. Program will create N processes on same machine and execute program using these processes as cloud haskell's nodes.

SLURM startup. Jobs of starting processes is handled to SLURM and processes learn addresses of other processes from environment variables set by SLURM. No command line parameters is required in this case.

Groups of actors

Actor could run in groups. These groups are treated as single logical actor. Each actor in group is assigned rank from *0* to *N-1* where *N* is group size. For uniformity single actors are treated as members of group of size 1. Both group size and rank could be accessed using `rank` and `groupSize`

`rank` `:: DNA Int`

Obtains the rank of the current process in its group. Every process in a group of size *N* has assigned a rank from *0* to *N-1*. Single processes always have rank *0*. It should be used as follows:

```
do ...
   n <- rank
   ...
```

`groupSize` `:: DNA Int`

Obtains the size of the group that the current process belongs to. For single processes this is always *1*. It should be used as follows:

```
do ...
   n <- groupSize
   ...
```

Logging and profiling

DNA programs write logs in GHC's eventlog format for recording execution progress and performance monitoring. Logs are written in following locations: `~/_dna/logs/PID-u/{N}/program-name.eventlog` if program was started using UNIX startup or `~/_dna/logs/SLURM_JOB_ID-s/{N}/program-name.eventlog` if it was started by SLURM (see `runDna` for detail of starting DNA program). They're stored in GHC's eventlog format.

`logMessage` `:: String -> DNA ()`

Outputs a message to the eventlog as well as `stdout`. Useful for documenting progress and providing debugging information.

For example, we could have an actor log the amount of resource it has available:

```
do avail <- availableNodes
   logMessage $ "Actor is running on " ++ show (avail+1) ++ " nodes."
```

It will produce eventlog output similar to this

```
713150762: cap 0: MSG [pid=pid://localhost:40000:0:10] Actor is running c
```

---

duration

```
:: String   Computation name for profiling
-> DNA a    DNA code to profile
-> DNA a
```

Basic profiling for DNA actions. Works basically the same way as kernel, but without the specialised profiling support. Instead, the profiling report will only contain the wall clock time the contained DNA action took.

For example, in the DNA example we used duration to profile how long a Promise was awaited:

```
   va <- duration "receive compute" $ await futVA
```

It will result in eventlog output similar to:

```
941813583: cap 0: START [pid=pid://localhost:40000:0:12] receive compute
...
945372376: cap 0: END [pid=pid://localhost:40000:0:12] receive compute
```

Kernels

---

data **Kern** a

Monad for actual calculation code. We expect all significant work of the cluster application to be encapsulated in this monad. In fact, the only way to perform arbitrary IO actions from DNA is to use kernel or unboundKernel and then liftIO the desired code:

```
  kernel "do IO" $ liftIO $ do
    someIoComputation
```

Pure computations should be lifted into the Kern monad as well whenever they are likely to require a significant amount of computation. However care needs to be taken that no thunks escape due to lazy evaluation. Ideally, the result should be fully evaluated:

```
  kernel "pure computation" $ do
    let pure = pureCode
```

```
    return $! pure `using` rdeepseq
```

Instances

Monad Kern

Functor Kern

Applicative Kern

MonadIO Kern

---

kernel

:: String	Kernel name. This name will be used in profile analysis to refer to profiling data collected about the contained code.
-> [ProfileHint]	Kernel performance characteristics. This will prompt the framework to track specialised performance metrics, allowing in-depth analysis later.
-> Kern a	Th kernel code to execute.
-> DNA a	

Executes a kernel computation. The computation will be bound to an operating system thread by default (see also unboundKernel). The function will block until computation is done. Profile hints can be used to request profiling where desired.

For example, we could define ddpReadVector as used in the DNA example as follows:

```
ddpReadVector = actor $ \(fname, Slice off n) ->
  kernel "read vector" [iOHint{hintReadBytes = fromIntegral (n * 8)}] $
    liftIO $ readData n off fname
```

This "actor" reads a certain slice of a file from the disk, which is implemented using a "kernel" calling the readData IO action. As with most kernels, this could potentially become a bottleneck, therefore we supply DNA with a meaningful name (read vector) as well as a hint about how much I/O activity we expect. This will prompt the profiling framework to gather evidence about the actual I/O activity so we can compare it with our expectations.

---

unboundKernel

:: String	Kernel name
-> [ProfileHint]	Kernel performance characteristics
-> Kern a	Kernel code
-> DNA a	

A variant of kernel that executes the kernel in an *unbound* thread. Haskell runtime could migrate unbound haskell threads between OS threads. This is generally faster, but less safe. Especially profiling can be unreliable in this mode.

The most likely use for this is cheap kernels that are unlikely to run for a significant

time. For example, we could use an unbound kernel for cleaning up data:

```
unboundKernel "delete vector" [] $
  liftIO $ removeFile fname
```

Here we know that `removeFile` is safe to be called from unbound kernels, and likely cheap enough that allocating a full operating system thread can be considered overkill.

`data` **`ProfileHint`**

A program annotation providing additional information about how much work we expect the program to be doing in a certain phase. The purpose of this hint is that we can set-up measurements to match these numbers to the program's real performance. Note that the hint must only be a best-effort estimate. As a rule of thumb, it is better to use a more conservative estimate, as this will generally result in lower performance estimates. These hints are passed to `kernel` or `unboundKernel`.

Hints should preferably be constructed using the default constructors: `floatHint`, `memHint`, `ioHint`, `haskellHint` and `cudaHint`. See their definitions for examples.

Constructors

`FloatHint` Estimate for how many floating point operations the code is executing. Profiling will use `perf_event` in order to take measurements. Keep in mind that this has double-counting issues (20%-40% are not uncommon for SSE or AVX code).

`hintFloatOps :: !Int`

`hintDoubleOps :: !Int`

`MemHint` Estimate for the amount of data that will have to be read from RAM over the course of the kernel calculation.

`hintMemoryReadBytes :: !Int`

`IOHint` Estimate for how much data the program is reading or writing from/to external sources.

`hintReadBytes :: !Int`

`hintWriteBytes :: !Int`

`HaskellHint` Rough estimate for how much Haskell work we are doing

`hintAllocation :: !Int`

`CUDAHint` CUDA statistics. The values are hints about how much data transfers we expect to be targetting the device and the host respectively.

The FLOP hints will only be checked if logging is running in either `"float-ops"` or `"double-ops"` mode, respectively. Note that this requires instrumentation, which will reduce overall performance!

`hintCopyBytesHost :: !Int`

`hintCopyBytesDevice :: !Int`

`hintCudaFloatOps :: !Int`

hintCudaDoubleOps :: !Int

---

floatHint :: ProfileHint

Default constructor for FloatHint with hint 0 for all metrics. Can be used for requesting FLOP profiling. Hints can be added by overwriting fields values.

For example, we can use this ProfileHint to declare the amount of floating point operations involved in computing a sum:

```
kernel "compute sum" [floatHint{hintDoubleOps = fromIntegral (2*n)} ] $
    return $ (S.sum $ S.zipWith (*) va vb :: Double)
```

---

memHint :: ProfileHint

Default constructor for MemHint with hint 0 for all metrics. Can be used for requesting memory bandwidth profiling. Hints can be added by overwriting field values.

This could be used to track the bandwidth involved in copying a large buffer:

```
let size = Vec.length in * sizeOf (Vec.head in)
kernel "copy buffer" [memHint{hintMemoryReadBytes=size}] $ liftIO $
    VecMut.copy in out
```

---

ioHint :: ProfileHint

Default constructor for IOHint with hint 0 for all metrics. Can be used for requesting I/O bandwidth profiling. Hints can be added by overwriting field values.

This can be used to document the amount of data that we expect to read from a hard drive:

```
kernel "read vector" [iOHint{hintReadBytes = fromIntegral (n * 8)}] $
    liftIO $ readData n off fname
```

---

haskellHint :: ProfileHint

Default constructor for IOHint with hint 0 for all metrics. Can be used for requesting Haskell allocation profiling. Hints can be added by overwriting field values.

Useful for tracking the amount of allocation Haskell does in a certain computation. This can often be a good indicator for whether it has been compiled in an efficient way.

```
unboundKernel "generate vector" [HaskellHint (fromIntegral $ n * 8)] $
  liftIO $ withFileChan out "data" WriteMode $ \h ->
    BS.hPut h $ runPut $
      replicateM_ (fromIntegral n) $ putFloat64le 0.1
```

For example, this HaskellHint specifies that Haskell is allowed to only heap-allocate

one Double-object per value written.

---

cudaHint :: ProfileHint

Default constructor for CUDAHint with hint 0 for all metrics. Can be used for requesting Haskell allocation profiling. Hints can be added by overwriting field values.

For instance, we could wrap an accelerate computation as follows:

```
let size = S.length va
kernel "accelerate dot product" [cudaHint{hintCudaDoubleOps=size*2}] $ li
  let sh = S.length va
      va' = A.fromVectors (A.Z A.:. size) ((), va) :: A.Vector Double
      vb' = A.fromVectors (A.Z A.:. size) ((), vb) :: A.Vector Double
   return $ head $ A.toList $ CUDA.run $
     A.fold (+) 0 $ A.zipWith (*) (A.use va') (A.use vb')
```

Actors

---

data **Actor** a b

This is the simplest kind of actor. It receives exactly one message of type a and produce a result of type b. It could only be constructed using actor function.

Instances

Typeable (* -> * -> *) Actor

---

actor

```
:: (Serializable a, Serializable b)
=> (a -> DNA b)                          data flow definition
-> Actor a b
```

Smart constructor for Actors. As the type signature shows, an Actor is constructed from a function that takes a parameter a and returns a result b. The DNA monad allows the actor to take further actions, such as spawning other actors or starting data transfers.

For example following actor adds one to its parameter

```
succActor :: Actor Int Int
succActor = actor $ \i -> return (i+1)
```

---

data **CollectActor** a b

In contrast to a simple Actor, actors of this type can receive a group of messages. However, it will still produce just a singular message. In functional programming terms,

this actor corresponds to a `fold`, which reduces an unordered set of messages into an aggregate output value. It could only be constructed using `collectActor` function.

Instances

```
Typeable (* -> * -> *) CollectActor
```

---

collectActor

```
:: (Serializable a, Serializable b, Serializable s)
=> (s -> a -> Kern s)                    stepper function
-> Kern s                               start value
-> (s -> Kern b)                        termination function
-> CollectActor a b
```

Just like a `fold`, a `CollectorActor` is defined in terms of an internal state which gets updated for every message received. To be precise, the state first gets initialised using a start value, then gets updated successively using the stepper function. Once all results have been received, the termination function generates the overall result value of the actor.

In this example actor sums its parameters. It's very simple actor. In this case type of accumulator (s above) is same as type of resulting value (`Double`) but this isn't necessary. It also doesn't do any IO.

```
sumActor :: CollectorActor Double Double
sumActor = collectActor
    (\sum a -> return (sum + a))
    (return 0)
    (\sum -> return sum)
```

Spawning

Actors could be spawned using start* functions. They spawn new actors which are executed asynchronously and usually on remote nodes. Nodes for newly spawned actor(s) are taken from pool of free nodes. If there's not enough nodes it's runtime error. eval* functions allows to execute actor synchronously.

Eval

---

eval

```
:: (Serializable a, Serializable b)
=> Actor a b                             Actor to execute
-> a                                    Value which is passed to an actor as
                                        parameter
```

```
-> DNA b
```

If one don't want to create new actor it's possible to execute simple `Actor` inside current actor. For example:

```
do ...
   b <- eval someActor 42
   ...
```

---

evalClosure

```
:: (Typeable a, Typeable b)
=> Closure (Actor a b)        Actor to execute
-> a                          Value which is passed to an actor as parameter
-> DNA b
```

Like `eval`, but uses a `Closure` of the actor code.

Spawn parameters

`data` **Spawn** `a`

Monad for accumulating optional parameters for spawning processes. It exists only to (ab)use do-notation and meant to be used as follows:

```
do useLocal
   return $(mkStaticClosure 'actorName)
```

Instances

```
Monad Spawn
Functor Spawn
Applicative Spawn
```

---

useLocal `:: Spawn ()`

With this parameter new actor will be spawned on same node as parent actor. In case of group of actors one of newly spawned actors will run on local node. Otherwise it will be spawned on other node. See documentation for `Res` for description of interaction of this flag with resource allocation.

---

failout `:: Spawn ()`

Spawn the process using the "failout" fault-tolerance model. Only valid for group of processes (it's ignored for spawning single process actors). If some actor in group fails group will still continue.

respawnOnFail :: Spawn ()

Try to respawn actor in case of crash.

debugFlags :: [DebugFlag] -> Spawn ()

Set debugging flags. They are mostly useful for debugging DNA itself.

data **DebugFlag**

Flags which could be passed to actors for debugging purposes

Constructors

CrashProbably Double	Crash during startup with given probability. Not all actors will honor that request
EnableDebugPrint Bool	Enable debug printing. If parameter is true child actors will have debug printing enabled too.

Instances

Eq DebugFlag

Show DebugFlag

Generic DebugFlag

Binary DebugFlag

Typeable * DebugFlag

type Rep DebugFlag

Resources

These data types are used for describing how much resources should be allocated to nodes and are passed as parameters to start* functions.

data **Res**

This describes how many nodes we want to allocate either to a single actor process or to the group of processes as whole. We can either request exactly *n* nodes or a fraction of the total pool of free nodes. If there isn't enough nodes in the pool to satisfy request it will cause runtime error.

For example `N 4` requests exactly for nodes. And `Frac 0.5` requests half of all currently available nodes.

Local node (which could be added using `useLocal`) is added in addition to this. If in the end 0 nodes will be allocated it will cause runtime error.

Constructors

N `Int` Fixed number of nodes

Frac `Double` Fraction of nodes. Should lie in *(0,1]* range.

Instances

`Show Res`

`Generic Res`

`Binary Res`

`Typeable * Res`

`type Rep Res`

`data` **ResGroup**

Describes how to divide allocated nodes between worker processes.

Constructors

NWorkers `Int` divide nodes evenly between *n* actors.

NNodes `Int` Allocate no less that *n* nodes for each actors. DSL will try to create as many actor as possible under given constraint

Instances

`Show ResGroup`

`Generic ResGroup`

`Binary ResGroup`

`Typeable * ResGroup`

`type Rep ResGroup`

`data` **Location**

Describes whether some entity should be local to node or could be possibly on remote node.

Constructors

Remote

Local

Instances

`Eq Location`

`Ord Location`

`Show Location`

`Generic Location`

`Binary Location`

```
Typeable * Location
type Rep Location
```

---

availableNodes :: DNA Int

Returns the number of nodes that are available at the moment for spawning of remote processes.

---

waitForResources :: Shell a b -> DNA ()

Barrier that ensures that all resources associated with the given actor have been returned to pool and can be re-allocated. It will block until resources are returned.

```
do a <- startActor ...
   ...
   waitForResources a
```

After `waitForResources a` it's guaranteed that resources allocated to actor a have been returned.

N.B. It only ensures that actor released resources. They could be taken by another start* function.

Function to spawn new actors

All functions for starting new actors following same pattern. They take parameter which describe how many nodes should be allocated to actor(s) and `Closure` to actor to be spawned. They all return handle to running actor (see documentation of `Shell` for details).

Here is example of spawning single actor on remote node. To be able to create `Closure` to execute actor on remote node we need to make it "remotable". For details of `remotable` semantics refer to distributed-process documentation,. (This could change in future version of `distributed-process` when it start use StaticPointers language extension)

```
someActor :: Actor Int Int
someActor = actor $ \i -> ...

remotable [ 'someActor ]
```

Finally we start actor and allocate 3 nodes to it:

```
do a <- startActor (N 3) (return $(mkStaticClosure 'someActor))
   ...
```

In next example we start group of actors, use half of available nodes and local node in addition to that. These nodes will be evenly divided between 4 actors:

```
do a <- startGroup (Frac 0.5) (NWorkers 4) $ do
          useLocal
          return $(mkStaticClosure 'someActor)
   ...
```

All other start* functions share same pattern and could be used in similar manner.

---

startActor

```
:: (Serializable a, Serializable b)
```

`=> Res`	How many nodes do we want to allocate for actor
`-> Spawn (Closure (Actor a b))`	Actor to spawn
`-> DNA (Shell (Val a) (Val b))`	Handle to spawned actor

Starts a single actor as a new process, and returns the handle to the running actor. Spawned actor will receive single message and produce single result as described by `Val` type tags.

---

startGroup

```
:: (Serializable a, Serializable b)
```

`=> Res`	How many nodes do we want to allocate for actor
`-> ResGroup`	How to divide nodes between actors in group
`-> Spawn (Closure (Actor a b))`	Actor to spawn
`-> DNA (Shell (Scatter a) (Grp b))`	Handle to spawned actor

Start a group of actor processes. They receive set of values which could be sent to them using `broadcast` or `distributeWork` and produce group of values as result.

---

startCollector

```
:: (Serializable a, Serializable b)
```

`=> Res`	How many nodes do we want to allocate for actor
`-> Spawn (Closure (CollectActor a b))`	Actor to spawn
`-> DNA (Shell (Grp a) (Val b))`	Handle to spawned actor

As `startActor`, but starts collector actor. It receives groups of messages from group of actors and produces single result.

---

startCollectorTree

```
:: Serializable a
```

```
  => Spawn (Closure (CollectActor a a))   Actor to spawn
  -> DNA (Shell (Grp a) (Val a))          Handle to spawned actor
```

Start a group of collector actor processes. It always require one node.

---

startCollectorTreeGroup

```
  :: Serializable a
  => Res                                  How many nodes do we want to
                                          allocate for group of actors
  -> Spawn (Closure (CollectActor a a))   Actor to spawn
  -> DNA (Shell (Grp a) (Grp a))          Handle to spawned actor
```

Start a group of collector actor processes to collect data in tree-like fashion. They collect data from group of actors and divide it between themselves. So if we have 12 worker actors in a group and 3 actor in group of collectors collector with rank 0 will collect results from workers with rank 0..3 etc. Collectors will produce 3 result which in turn should be aggregated by another collector.

Shell

`data` **Shell** `a b`

Handle of a running actor or group. Note that we treat actors and groups of actors uniformly here. Shell data type has two type parameters which describe what kind of data actor receives or produces. For example:

```
Shell (InputTag a) (OutputTag b)
```

Also both input and output types have tags which describe how many messages data type produces and how this actor could be connected with others. It means that shell receives message(s) of type a and produce message(s) of type b. We support tags `Val`, `Grp` and `Scatter`.

Instances

```
Generic (Shell a b)
Binary (Shell a b)
Typeable (* -> * -> *) Shell
type Rep (Shell a b)
```

`data` **Val** `a`

The actor receives/produces a single value, respectively.

Instances

Typeable (* -> *) Val

---

data **Grp** a

The actor receives/produces an unordered group of values.

Instances

Typeable (* -> *) Grp

---

data **Scatter** a

Only appears as an input tag. It means that we may want to scatter values to a set of running actors.

Instances

Typeable (* -> *) Scatter

Connecting actors

Each actor must be connected to exactly one destination and consequently could only receive input from a single source. Trying to connect an actor twice will result in a runtime error. Functions `sendParam`, `broadcast`, `distributeWork`, `connect`, `delay`, and `delayGroup` count to this.

---

sendParam

```
:: Serializable a
=> a                  Parameter to send
-> Shell (Val a) b    Actor to send parameter to
-> DNA ()
```

Send input parameter to an actor. Calling this function twice will result in runtime error.

```
do ...
   a <- startActor (N 1) (return $(mkStaticClosure 'someActor))
   sendParam 100
   ...
```

---

broadcast

```
:: Serializable a
=> a                       Parameter to send
-> Shell (Scatter a) b     Group of actors to send parameter to
-> DNA ()
```

Send same value to all actors in group. Essentially same as `sendParam` but works for group of actors.

```
do ...
   a <- startGroup (Frac 0.5) (NNodes 1) (return $(mkStaticClosure 'someA
   broadcast 100
   ...
```

---

distributeWork

```
:: Serializable b
=> a                          Parameter we want to send
-> (Int -> a -> [b])          Function which distribute work between actors. First
                              parameter is length of list to produce. It must generate
                              list of required length.
-> Shell (Scatter b) c        Group of actors to send parameter to
-> DNA ()
```

Distribute work between group of actors. `distributeWork a f` will send values produced by function `f` to each actor in group. Computation is performed locally.

---

connect

```
:: (Serializable b, Typeable tag)
=> Shell a (tag b)                Actor which produce message(s)
-> Shell (tag b) c                Actor which receives message(s)
-> DNA ()
```

Connect output of one actor to input of another actor. In example we connect output of group of actors to collect actor.

```
do ...
   a <- startGroupN (N 10) (NNodes 1) (return $(mkStaticClosure 'worker))
   c <- startCollector (N 1) (return $(mkStaticClosure 'collector))
   connect a c
   ...
```

File channels.

`data` **FileChan** `a`

File channel for communication between actors. It uses file system to store data and it's assumed that different actors have access to same file. It could be either placed on network FS or all actors are running on same computer.

Instances

```
Show (FileChan a)
Generic (FileChan a)
Binary (FileChan a)
Typeable (* -> *) FileChan
type Rep (FileChan a)
```

---

createFileChan

```
:: Location
```
If `Local` will try to create channel in `/ramdisks` or `/tmp` if possible.

```
-> String
```
Channel name

```
-> DNA (FileChan a)
```

Allocates a new file channel for sharing data between actors.

Promises

`data` **Promise** `a`

Result of an actor's computation. It could be generated by `delay` and actual value extracted by `await`

```
do ...
   p <- delay someActor
   ...
   a <- await p
```

---

delay

```
:: Serializable b
=> Location
-> Shell a (Val b)
```
Actor to obtain promise from.

```
-> DNA (Promise b)
```

Obtains a promise from a shell. This amounts to connecting the actor.

---

await

```
:: Serializable a
=> Promise a
```
Promise to extract value from

```
-> DNA a
```

Extract value from `Promise`, will block until value arrives

```
data Group a
```

Like `Promise`, but stands for the a group of results, as generated by an actor group. It could be used in likewise manner. In example below values produced by group of actors `grp` are summed in call to `gather`.

```
do ...
   p <- delayGroup grp
   ...
   a <- gather p (+) 0
```

delayGroup

```
:: Serializable b
=> Shell a (Grp b)      Actor to obtain promise from
-> DNA (Group b)
```

Like `delay`, but for a `Grp` of actors. Consequently, we produce a promise `Group`.

gather

```
:: Serializable a
=> Group a              Promise to use.
-> (b -> a -> b)        Stepper function (called for each message)
-> b                    Initial value
-> DNA b
```

Obtains results from a group of actors by folding over the results. It behaves like `CollectActor` but all functions are evaluated locally. It will block until all messages are collected.

Reexports

```
class Monad m => MonadIO m where
```

Methods

liftIO :: IO a -> m a

Instances

MonadIO IO

MonadIO Process

MonadIO NC

MonadIO Kern

MonadIO m => MonadIO (MaybeT m)

```
MonadIO m => MonadIO (ListT m)
MonadIO m => MonadIO (IdentityT m)
MonadIO (MxAgent s)
MonadIO m => MonadIO (StateT s m)
(Error e, MonadIO m) => MonadIO (ErrorT e m)
(Monoid w, MonadIO m) => MonadIO (WriterT w m)
(Monoid w, MonadIO m) => MonadIO (WriterT w m)
MonadIO m => MonadIO (StateT s m)
MonadIO m => MonadIO (ReaderT r m)
MonadIO m => MonadIO (ExceptT e m)
MonadIO m => MonadIO (ContT r m)
MonadIO m => MonadIO (ProgramT instr m)
(Monoid w, MonadIO m) => MonadIO (RWST r w s m)
(Monoid w, MonadIO m) => MonadIO (RWST r w s m)
```

```
remotable :: [Name] -> Q [Dec]
```

```
mkStaticClosure :: Name -> Q Exp
```