Cardano.BM - logging, benchmarking and monitoring

Alexander Diemand Denis Shevchenko Andreas Triantafyllos

August 2019

Abstract

This framework combines logging, benchmarking and monitoring. Complex evaluations of STM or monadic actions can be observed from outside while reading operating system counters before and after, and calculating their differences, thus relating resource usage to such actions.

Through interactive configuration, the runtime behaviour of logging or the measurement of resource usage can be altered.

Further reduction in logging can be achieved by redirecting log messages to an aggregation function which will output the running statistics with less frequency than the original message.

Contents

Chapter 1

Logging, benchmarking and monitoring

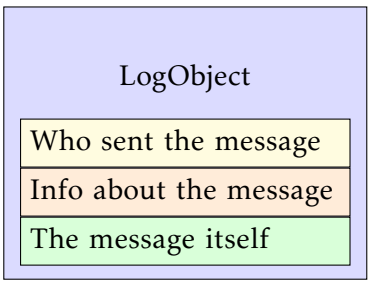
1.1 Main concepts

The main concepts of the framework:

1. LogObject - captures the observable information
2. Trace - transforms and delivers the observables
3. Backend - receives and outputs observables
4. Configuration - defines behaviour of traces, routing of observables

1.1.1 LogObject

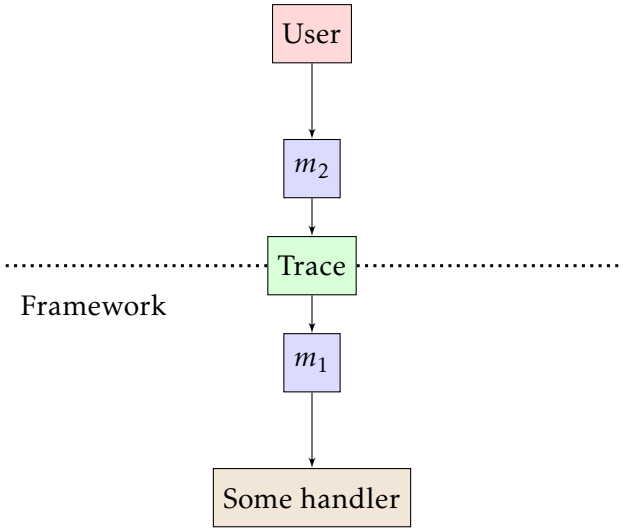
LogObject represents an observation to be logged or otherwise further processed. It is annotated with a logger name, meta information (timestamp and severity level), and some particular message:



Please see Cardano.BM.Data.LogItem for more details.

1.1.2 Trace

You can think of Trace as a pipeline for messages. It is a *consumer* of messages from a user's point of view, but a *source* of messages from the framework's point of view. A user traces an observable to a Trace, which ends in the framework that further processes the message.

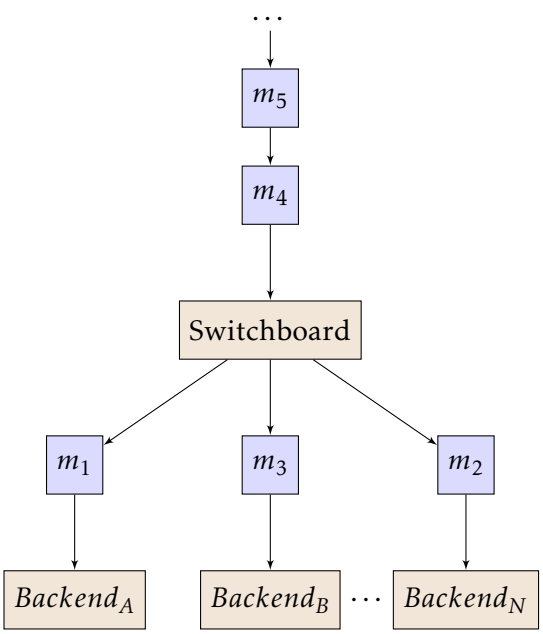


Please see the section 1.4.1 for more details about the ideas behind Trace.

1.1.3 Backend

A Backend must implement functions to process incoming messages of type LogObject. It is an instance of IsEffectuator. Moreover, a backend is also life-cycle managed. The class IsBackend ensures that every backend implements the *realize* and *unrealize* functions.

The central backend in the framework is the Switchboard. It sets up all the other backends and redirects incoming messages to these backends according to configuration:



1.1.4 Configuration

Configuration defines how the message flow in the framework is routed and the behaviour of distinct Traces. It can be parsed from a file in YAML format, or it can explicitly be defined in code.

Please note that Configuration can be changed at runtime using the interactive editor (see *Cardano.BM.*Configuration.Editor for more details).

1.2 Overview

Figure 1.1 displays the relationships among modules in *Cardano.BM*.

1.2.1 Backends

As was mentioned above, the central backend is the Switchboard that redirects incoming log messages to selected backends according to Configuration.

The backend EKGView displays runtime counters and user-defined values in a browser.

The Log backend makes use of the katip package to output log items to files or the console. The format can be chosen to be textual or JSON representation.

The Aggregation backend computes simple statistics over incoming log items (e.g. last, min, max, mean) (see Cardano.BM.Data.Aggregated). Alternatively, Aggregation can also estimate the average of the values passed in using *EWMA*, the exponentially weighted moving average. This works for numerical values, that is if the content of a LogObject is a LogValue.

The backend LogBuffer keeps the latest message per context name and shows these collected messages in the GUI (Editor), or outputs them to the switchboard.

Output selection determines which log items of a named context are routed to which backend. In the case of the Log output, this includes a configured output sink, *scribe* in *katip* parlance.

Items that are aggregated lead to the creation of an output of their current statistics. To prevent a potential infinite loop these aggregated statistics cannot be routed again back into Aggregation.

1.2.2 Trace

Log items are created in the application's context and passed in via a hierarchy of Traces. Such a hierarchy of named traces can be built with the function appendName. The newly added child Trace will add its name to the logging context and behave as configured. Among the different kinds of Traces implemented are:

1. NoTrace which suppresses all log items,
2. SetSeverity which sets a specific severity to all log items,
3. FilterTrace which filters the log items passing through it,
4. ObservableTrace which allows capturing of operating system counters.

(further behaviour types are implemented in Cardano.BM.Data.SubTrace)

1.2.3 Monitoring

With *Monitoring* we aim to shortcut the logging-analysis cycle and immediately evaluate monitors on logged values when they become available. In case a monitor is triggered a number of actions can be run: either internal actions that can alter the Configuration, or actions that can lead to alerting in external systems.

1.2.4 IMPORTANT!

It is not the intention that this framework should (as part of normal use) record sufficient information so as to make the sequence of events reproducible, i.e. it is not an audit or transaction log.

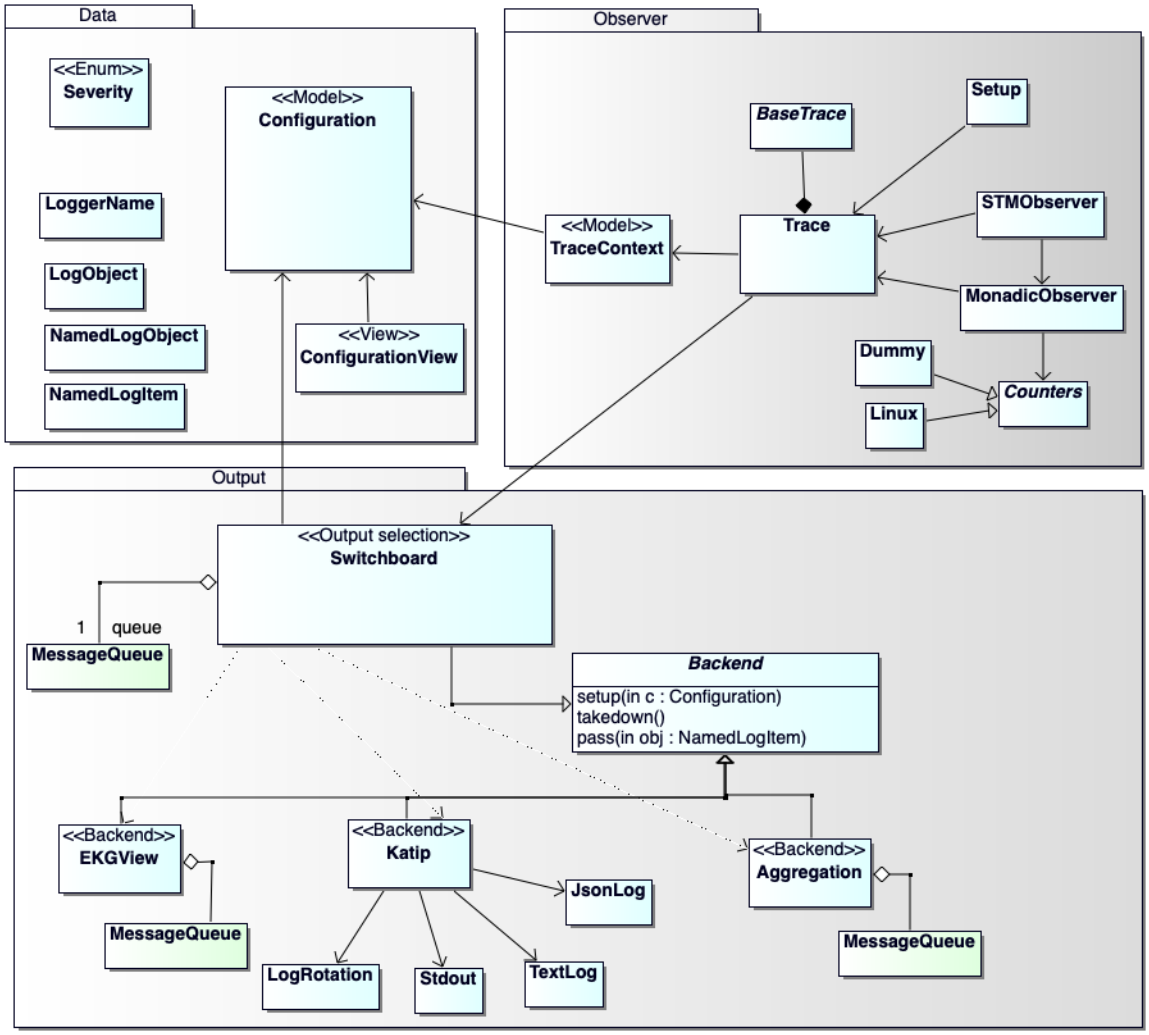


Figure 1.1: Overview of module relationships. The arrows indicate import of a module. The arrows with a triangle at one end would signify ”inheritance” in object-oriented programming, but we use it to show that one module replaces the other in the namespace, thus specializes its interface.

1.3 Requirements

1.3.1 Observables

We can observe the passage of the flow of execution through particular points in the code (really the points at which the graph is reduced). Typically observables would be part of an outcome (which has a start and an end). Where the environment permits these outcomes could also gather additional environmental context (e.g read system counters, ‘know’ the time). The proposed framework would be able to aggregate, filter such outcome measures so as to calculation things (where appropriate) such as:

- min/max/mean/variance of the resource costs of achieving an outcome
- elapsed wall-clock time

- CPU cycles
- memory allocations, etc
- exponentially weighted moving average of outcomes, events
- min/max/mean/variance of inter-arrival times of demand for service (the arrival pattern)
- measuring offered load against the system (e.g rate/distribution of requests against the wallet by an exchange, transactions being forwarded between nodes)

STM evaluation

We treat STM evaluation as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. Logging in an STM will keep a list of log items which at the exit of the evaluation will be passed to the logging subsystem. Since we do not know the exact time an event occurred in the STM action, we annotate the event afterwards with the time interval of the STM action.

Function evaluation

We treat a function call as a black box and register measurables (counters) before entering, and report the difference at exit together with the result. The function is expected to accept a 'Trace' argument which receives the events.

QuickCheck properties *tentatively*

The function

```
quickCheckResult :: Testable prop => prop -> IO Result
```

will return a *Result* data structure from which we can extract the number of tests performed. Recording the start and end times allows us to derive the time spent for a single test. (although this measurement is wrong as it includes the time spent in QuickCheck setting up the test case (and shrinking?))

1.3.2 Traces

Log items are sent as streams of events to the logging system for processing (aggregation, ..) before output. Functions that need to log events must accept a *Trace* argument. There is no monad related to logging in the monad stack, thus this can work in any monadic environment.

Trace Context

A Trace maintains a named context stack. A new name can be put onto it, and all subsequent log messages are labeled with this named context. This is also true to all downstream functions which receive the modified Trace. We thus can see the call tree and how the evaluation entered the context where a logging function was called. The context also maintains a mapping from name to Severity: this way a logging function call can early end and not produce a log item when the minimum severity is not reached.

SubTrace

A Trace is created in *IO* within setupTrace with the intent to pass the traced items to a downstream logging framework for outputting to various destinations in different formats. Apart from adding a name to the naming stack we can also alter the behaviour of the Trace. The newly created Trace with a specific function to process the recorded items will forward these to the upstream Trace. This way we can, for example, locally turn on aggregation of observables and only report a summary to the logs.

1.3.3 Aggregation

Log items contain a named context, severity and a payload (message, structured value). Thinking of a relation

```
(name, severity) -> value
```

, folding a summarizing function over it outputs

```
(name, severity) -> Summary
```

. Depending on the type of *value*, the summary could provide for example:

- * : first, last, count, the time between events (mean, sigma)
- Num : min, max, median, quartiles, mean, sigma, the delta between events (mean, sigma)

Other possible aggregations:

- exponentially weighted moving average
- histograms

1.3.4 Monitoring

- Enable (or disable) measuring events and performance at runtime (e.g. measure how block holding time has changed).
- Send alarms when observables give evidence for abnormalities
- Observe actions in progress, i.e. have started and not yet finished
- Bridge to *Datadog*?

1.3.5 Reporting

We might want to buffer events in case an exception is detected. This FIFO queue could then be output to the log for post-factum inspection.

1.3.6 Visualisation

EKG

https://hackage.haskell.org/package/ekg
This library allows live monitor a running instance over HTTP. There is a way we can add our own metrics to it and update them.

Log files

The output of observables immediately or aggregated to log files. The format is chosen to be JSON for easier post-processing.

Web app

Could combine EKG, log files and parameterization into one GUI.
(e.g. `https://github.com/HeinrichApfelmus/threepenny-gui`)

1.4 Description

1.4.1 Contravariant Functors Explanation

Tracer's implementations is based on a contravariant package.

Please see the presentation in `docs/pres-20190409/contravariant-idea` to understand the core idea of the contravariant functor.

1.4.2 Logging with Trace

Setup procedure

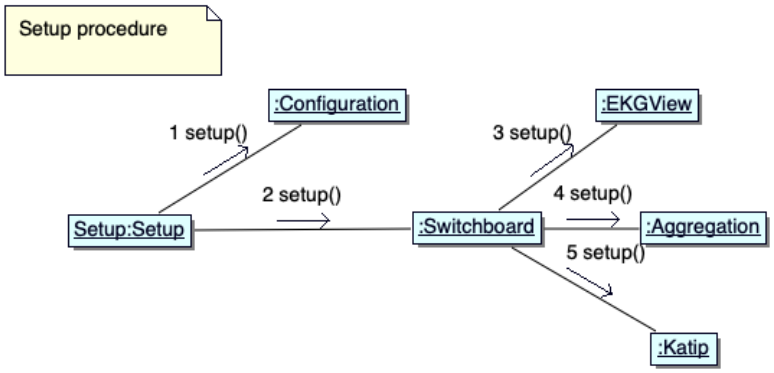


Figure 1.2: Setup procedure

Hierarchy of Traces

1.4.3 Micro-benchmarks record observables

Micro-benchmarks are recording observables that measure resource usage of the whole program for a specific time. These measurements are then associated with the subsystem that was observed at that time. Caveat: if the executable under observation runs on a multiprocessor computer where more than one parallel thread executes at the same time, it becomes difficult to associate resource usage to a single function. Even more so, as Haskell's thread do not map directly to operating system threads. So the expressiveness of our approach is only valid statistically when a large number of observables have been captured.

Counters

The framework provides access to the following O/S counters (defined in ObservableInstance) on *Linux*:

- monotonic clock (see MonotonicClock)
- CPU or total time (*/proc/<pid >/stat*) (see ProcessStats)
- memory allocation (*/proc/<pid >/statm*) (see MemoryStats)
- network bytes received/sent (*/proc/<pid >/net/netstat*) (see NetStats)
- disk input/output (*/proc/<pid >/io*) (see IOStats)

On all platforms, access is provided to the *RTS* counters (see GhcRtsStats).

Implementing micro-benchmarks

In a micro-benchmark we capture operating system counters over an STM evaluation or a function, before and afterwards. Then, we compute the difference between the two and report all three measurements via a *Trace* to the logging system. Here we refer to the example that can be found in complex example.

STM.bracketObserveIO *trace* `"observeSTM"` (*stmAction args*)

The capturing of STM actions is defined in Cardano.BM.Observer.STM and the function *STM.bracketObserveIO* has type:

```
bracketObserveIO
  :: Configuration
  → Trace IO a
  → Severity
  → Text
  → STM.STM t
  → IO t
```

It accepts a Trace to which it logs, adds a name to the context name and enters this with a SubTrace, and finally the STM action which will be evaluated. Because this evaluation can be retried, we cannot pass to it a Trace to which it could log directly. A variant of this function bracketObserveLogIO also captures log items in its result, which then are threaded through the Trace.
Capturing observables for a function evaluation in *IO*, the type of bracketObserveIO (defined in Cardano.BM.Observer.Monadic) is:

```
bracketObserveIO
  :: Configuration
  → Trace IO a
  → Severity
  → Text
  → IO t
  → IO t
```

It accepts a Trace to which it logs items, adds a name to the context name and enters this with a SubTrace, and then the IO action which will be evaluated.

```
bracketObserveIO trace "observeDownload" $ do
  license ← openURI "http://www.gnu.org/licenses/gpl.txt"
  case license of
    Right bs → logInfo trace $ pack $ BS8.unpack bs
    Left e → logError trace $ "failed to download; error: " ++ (show e)
  threadDelay 50000-- .05 second
  pure ()
```

Counters are evaluated before the evaluation and afterwards. We trace these as log items ObserveOpen and ObserveClose, as well as the difference with type ObserveDiff.

Configuration of mu-benchmarks

Observed STM actions or functions enter a new named context with a SubTrace. Thus, they need a configuration of the behaviour of this SubTrace in the new context. We can define this in the configuration for our example:

```
CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTrace [NetStats, IOStats])
```

This enables the capturing of network and I/O stats from the operating system. Other Observables are implemented in Cardano.BM.Data.Observable.
Captured observables need to be routed to backends. In our example we configure:

```
CM.setBackends c "complex.observeIO" (Just [AggregationBK])
```

to direct observables from named context *complex.observeIO* to the Aggregation backend.

1.4.4 Configuration

Format

The configuration is parsed from a file in *Yaml* format (see https://en.wikipedia.org/wiki/YAML) on startup. In a first parsing step the file is loaded into an internal *Representation*. This structure is then further processed and validated before copied into the runtime Configuration.

Configuration editor

The configuration editor (figure 1.3) provides a minimalistic GUI accessible through a browser that directly modifies the runtime configuration of the logging system. Most importantly, the global minimum severity filter can be set. This will suppress all log messages that have a severity assigned that is lower than this setting. Moreover, the following behaviours of the logging system can be changed through the GUI:

- *Backends*: relates the named logging context to a BackendKind
- *Scribes*: if the backend is KatipBK, defines to which outputs the messages are directed (see *ScribeId*)
- *Severities* a local minimum severity filter for just the named context (see Severity)
- *SubTrace* entering a new named context can create a new Trace with a specific behaviour (see SubTrace)
- *Aggregation* if the backend is *AggregationBK*, defines which aggregation method to use (see AggregatedKind)

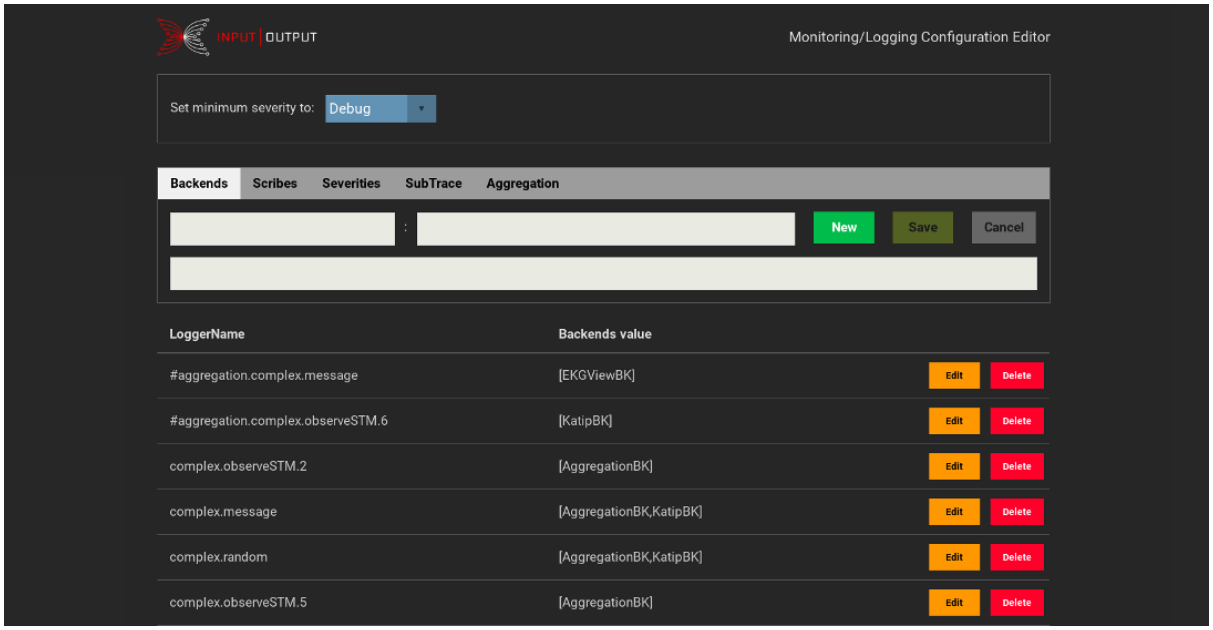


Figure 1.3: The configuration editor is listening on *localhost* and can be accessed through a browser. At the top is the setting for the global minimum severity filter, that drops all messages that have a severity lower than this setting. Below are the settings for various behaviours of the logging system.

1.4.5 Information reduction in Aggregation

Statistics

Configuration

1.4.6 Output selection

Configuration

1.4.7 Monitoring

Configuration

Evaluation of monitors

Actions fired

1.5 Examples

1.5.1 Simple example showing plain logging

```
{-# LANGUAGE CPP #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif
module Main
```

```
    (main)
    where
import Control.Concurrent (threadDelay)
import Control.Concurrent.MVar (MVar, newMVar, modifyMVar_, withMVar)
import Data.Aeson (FromJSON)
import Cardano.BM.Backend.Switchboard (addUserDefinedBackend)
import Cardano.BM.Data.Backend
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Configuration.Static (defaultConfigStdout)
#ifdef LINUX
import Cardano.BM.Scribe.Systemd (plugin)
import Cardano.BM.Data.Output (ScribeDefinition (..),
        ScribePrivacy (..), ScribeKind (..), ScribeFormat (..))
import Cardano.BM.Plugin (loadPlugin)
#endif
import Cardano.BM.Setup (setupTrace_)
import Cardano.BM.Trace (Trace, appendName, logDebug, logError,
        logInfo, logNotice, logWarning)
```

a simple backend

```
type MyBackendMVar a = MVar (MyBackendInternal a)
newtype MyBackend a = MyBackend {myBE :: MyBackendMVar a}
data MyBackendInternal a = MyBackendInternal {
  counter :: Int
  }
instance (FromJSON a) => IsBackend MyBackend a where
  bekind _ = UserDefinedBK "MyBackend"
  realize _ = MyBackend <$> newMVar (MyBackendInternal 0)
  unrealize be = putStrLn $ "unrealize " <> show (bekind be)
instance IsEffectuator MyBackend a where
  effectuate be _item = do
    modifyMVar_ (myBE be) $ λmybe →
      return $ mybe {counter = counter mybe + 1}
  handleOverflow _ = putStrLn "Error: MyBackend's queue full!"
```

Entry procedure

```
main :: IO ()
main = do
  c ← defaultConfigStdout
  CM.setDefaultBackends c [KatipBK, UserDefinedBK "MyBackend"]
#ifdef LINUX
  CM.setSetupBackends c [KatipBK, GraylogBK]
  CM.setDefaultBackends c [KatipBK, GraylogBK, UserDefinedBK "MyBackend"]
  CM.setGraylogPort c 3456
  CM.setSetupScribes c [ScribeDefinition {
                          scName = "text"
```

```
                    ,scFormat = ScText
                    ,scKind = StdoutSK
                    ,scPrivacy = ScPublic
                    ,scRotation = Nothing
                    }
                   ,ScribeDefinition {
                      scName = "json"
                    ,scFormat = ScJson
                    ,scKind = StdoutSK
                    ,scPrivacy = ScPublic
                    ,scRotation = Nothing
                    }
                   ]
  CM.setScribes c "simple.systemd" (Just ["JournalSK::cardano"])
# endif
  CM.setScribes c "simple.json" (Just ["StdoutSK::json"])
  (tr :: Trace IO String, sb) ← setupTrace_ c "simple"
  be :: MyBackend String ← realize c
  let mybe = MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be}
  addUserDefinedBackend sb mybe "MyBackend"
# ifdef LINUX
  -- inspect log with 'journalctl -t cardano'
  Cardano.BM.Scribe ∘ Systemd.plugin c tr sb "cardano"
    ⩥ loadPlugin sb
# endif
  let trText = appendName "text" tr
    trJson = appendName "json" tr
# ifdef LINUX
    trSystemd = appendName "systemd" tr
# endif
  logDebug trText   "this is a debug message\nwith a second line"
  logDebug trJson   "this is a debug message\nwith a second line"
  logInfo   trText   "this is an information."
  logInfo   trJson   "this is an information."
  logNotice trText   "this is a notice!"
  logNotice trJson   "this is a notice!"
  logWarning trText "this is a warning!"
  logWarning trJson "this is a warning!"
  logError  trText   "this is an error!"
  logError  trJson   "this is an error!"
# ifdef LINUX
  logError  trSystemd "this is an error!"
# endif
  threadDelay 80000
  withMVar (myBE be) $ λbackend →
    putStrLn $ "read in total " ++ (show $ counter backend) ++ " messages."
  return ()
```

1.5.2 Complex example showing logging, aggregation, and observing *IO* actions

Module header and import directives

```
 {-# LANGUAGE CPP #-}
 {-# LANGUAGE FlexibleContexts #-}
 {-# LANGUAGE MultiParamTypeClasses #-}
 {-# LANGUAGE ScopedTypeVariables #-}
#if defined (linux_HOST_OS)
#define LINUX
#endif
 {-define the parallel procedures that create messages -}
#define RUN_ProcMessageOutput
#define RUN_ProcObserveIO
#undef RUN_ProcObseverSTM
#undef RUN_ProcObseveDownload
#define RUN_ProcRandom
#define RUN_ProcMonitoring
#define RUN_ProcBufferDump
#define RUN_ProcCounterOutput
module Main
  (main)
  where
import Control.Concurrent (threadDelay)
import qualified Control.Concurrent.Async as Async
import Control.Monad (forM_, when)
import Data.Aeson (ToJSON (..), Value (..), (. =))
import qualified Data.HashMap.Strict as HM
import Data.Maybe (isJust)
import Data.Text (Text, pack)
#ifdef ENABLE_OBSERVABLES
import Control.Monad (forM)
import GHC.Conc.Sync (atomically, STM, TVar, newTVar, readTVar, writeTVar)
#ifdef LINUX
import qualified Data.ByteString.Char8 as BS8
import Network.Download (openURI)
#endif
#endif
import System.Random
import Cardano.BM.Backend.Aggregation
import Cardano.BM.Backend.Editor
import Cardano.BM.Backend.EKGView
import Cardano.BM.Backend.Monitoring
import Cardano.BM.Backend.Switchboard (Switchboard, readLogBuffer)
import Cardano.BM.Backend.TraceForwarder
#ifdef LINUX
import Cardano.BM.Scribe.Systemd
#endif
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Counters (readCounters)
```

```
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.AggregatedKind
import Cardano.BM.Data.BackendKind
import Cardano.BM.Data.Configuration (RemoteAddr (..))
import Cardano.BM.Data.Counter
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.MonitoringEval
import Cardano.BM.Data.Output
import Cardano.BM.Data.Rotation
import Cardano.BM.Data.Severity
import Cardano.BM.Data.SubTrace
import Cardano.BM.Data.Trace
import Cardano.BM.Data.Tracer
# ifdef ENABLE_OBSERVABLES
import Cardano.BM.Configuration
import Cardano.BM.Data.Observable
import Cardano.BM.Observer.Monadic (bracketObserveIO)
import qualified Cardano.BM.Observer.STM as STM
# endif
import Cardano.BM.Plugin
import Cardano.BM.Setup
import Cardano.BM.Trace
```

Define configuration

Selected values can be viewed in EKG on `http://localhost:12790`. And, the *Prometheus* interface is accessible at `http://localhost:12800/metrics`
The configuration editor listens on `http://localhost:13790`.

```
prepare_configuration :: IO CM.Configuration
prepare_configuration = do
  c ← CM.empty
  CM.setMinSeverity c Info
  CM.setSetupBackends c [KatipBK
      ,AggregationBK
      ,MonitoringBK
      -- , TraceForwarderBK -- testing for pipe
      ]
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
      scName = "stdout"
      ,scKind = StdoutSK
      ,scFormat = ScText
      ,scPrivacy = ScPublic
      ,scRotation = Nothing
      }
    ,ScribeDefinition {
      scName = "logs/out.odd.json"
      ,scKind = FileSK
      ,scFormat = ScJson
      ,scPrivacy = ScPublic
```

```
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000-- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ,ScribeDefinition {
    scName = "logs/out.even.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000-- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ,ScribeDefinition {
    scName = "logs/downloading.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000-- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ,ScribeDefinition {
    scName = "logs/out.txt"
    ,scKind = FileSK
    ,scFormat = ScText
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 5000-- 5kB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 3
      }
    }
  ,ScribeDefinition {
    scName = "logs/out.json"
    ,scKind = FileSK
    ,scFormat = ScJson
    ,scPrivacy = ScPublic
    ,scRotation = Just $ RotationParameters
      {rpLogLimitBytes = 50000000-- 50 MB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = 13
      }
    }
  ]
```

```
# ifdef LINUX
  CM.setDefaultScribes c ["StdoutSK::stdout","JournalSK::example-complex"]
# else
  CM.setDefaultScribes c ["StdoutSK::stdout"]
# endif
  CM.setScribes c "complex.random" (Just ["StdoutSK::stdout","FileSK::logs/out.txt"])
  forM_ [(1::Int)..10] $ λx →
    if odd x
    then
      CM.setScribes c ("complex.#aggregation.complex.observeSTM." <> pack (show x)) $ Just ["File
    else
      CM.setScribes c ("complex.#aggregation.complex.observeSTM." <> pack (show x)) $ Just ["File
# ifdef LINUX
# ifdef ENABLE_OBSERVABLES
  CM.setSubTrace c "complex.observeDownload" (Just $ ObservableTraceSelf [IOStats,NetStats])
# endif
  CM.setBackends c "complex.observeDownload" (Just [KatipBK])
  CM.setScribes c "complex.observeDownload" (Just ["FileSK::logs/downloading.json"])
# endif
  CM.setSubTrace c "#messagecounters.switchboard" $ Just NoTrace
  CM.setSubTrace c "#messagecounters.katip"       $ Just NoTrace
  CM.setSubTrace c "#messagecounters.aggregation" $ Just NoTrace
  CM.setSubTrace c "#messagecounters.ekgview"     $ Just Neutral
  CM.setBackends c "#messagecounters.switchboard" $ Just [EditorBK,KatipBK]
  CM.setSubTrace c "#messagecounters.monitoring" $ Just NoTrace
  CM.setSubTrace c "complex.random" (Just $ TeeTrace "ewma")
  CM.setSubTrace c "#ekgview"
    (Just $ FilterTrace [(Drop (StartsWith "#ekgview.complex.#aggregation.complex.random"),
        Unhide [EndsWith ".count",
          EndsWith ".avg",
          EndsWith ".mean"]),
      (Drop (StartsWith "#ekgview.complex.#aggregation.complex.observeIO"),
        Unhide [Contains "diff.RTS.cpuNs.timed."]),
      (Drop (StartsWith "#ekgview.complex.#aggregation.complex.observeSTM"),
        Unhide [Contains "diff.RTS.gcNum.timed."]),
      (Drop (StartsWith "#ekgview.complex.#aggregation.complex.message"),
        Unhide [Contains ".timed.m"])
      ])
# ifdef ENABLE_OBSERVABLES
  CM.setSubTrace c "complex.observeIO" (Just $ ObservableTraceSelf [GhcRtsStats,MemoryStats])
  forM_ [(1::Int)..10] $ λx →
    CM.setSubTrace
      c
      ("complex.observeSTM." <> (pack $ show x))
      (Just $ ObservableTraceSelf [GhcRtsStats,MemoryStats])
# endif
  CM.setBackends c "complex.message" (Just [AggregationBK,KatipBK,TraceForwarderBK])
  CM.setBackends c "complex.random" (Just [KatipBK,EKGViewBK])
  CM.setBackends c "complex.random.ewma" (Just [AggregationBK])
  CM.setBackends c "complex.observeIO" (Just [AggregationBK,MonitoringBK])
```

```
  forM_ [(1 :: Int)..10] $ λx → do
    CM.setBackends c
      ("complex.observeSTM." <> pack (show x))
      (Just [AggregationBK])
    CM.setBackends c
      ("complex.#aggregation.complex.observeSTM." <> pack (show x))
      (Just [KatipBK])
  CM.setAggregatedKind c "complex.random.rr" (Just StatsAK)
  CM.setAggregatedKind c "complex.random.ewma.rr" (Just (EwmaAK 0.22))
  CM.setBackends c "complex.#aggregation.complex.random" (Just [EditorBK])
  CM.setBackends c "complex.#aggregation.complex.random.ewma" (Just [EKGViewBK, EditorBK])
  CM.setBackends c "complex.#aggregation.complex.message" (Just [EKGViewBK, MonitoringBK])
  CM.setBackends c "complex.#aggregation.complex.monitoring" (Just [MonitoringBK])
  CM.setBackends c "complex.#aggregation.complex.observeIO" (Just [EKGViewBK])
  CM.setScribes c "complex.counters" (Just ["StdoutSK::stdout","FileSK::logs/out.json"])
  CM.setEKGport c 12790
  CM.setPrometheusBindAddr c $ Just ("localhost",12800)
  CM.setGUIport c 13790
```

output could also be forwarded using a pipe:

```
  CM.setForwardTo c (Just $ RemotePipe "logs/pipe")
  CM.setForwardTo c (Just $ RemotePipe "\\\\.\\pipe\\acceptor")-- on Windows


  CM.setForwardTo c (Just $ RemoteSocket "127.0.0.1" "42999")
  CM.setTextOption c "forwarderMinSeverity" "Warning"-- sets min severity filter in forward
  CM.setMonitors c $ HM.fromList
    [("complex.monitoring"
      ,(Just (Compare "monitMe" (GE, OpMeasurable 10))
        ,Compare "monitMe" (GE, OpMeasurable 42)
        ,[CreateMessage Warning "MonitMe is greater than 42!"]
        )
      )
    ,("complex.#aggregation.complex.monitoring"
      ,(Just (Compare "monitMe.fcount" (GE, OpMeasurable 8))
        ,Compare "monitMe.mean" (GE, OpMeasurable 41)
        ,[CreateMessage Warning "MonitMe.mean is greater than 41!"]
        )
      )
    ,("complex.observeIO.close"
      ,(Nothing
        ,Compare "complex.observeIO.close.Mem.size" (GE, OpMeasurable 25)
        ,[CreateMessage Warning "closing mem size is greater than 25!"]
        )
      )
    ]
  CM.setBackends c "complex.monitoring" (Just [AggregationBK, KatipBK, MonitoringBK])
  return c
```

Dump the log buffer periodically

```
dumpBuffer :: Switchboard Text → Trace IO Text → IO (Async.Async ())
dumpBuffer sb trace = do
  logInfo trace "starting buffer dump"
  Async.async (loop trace)
where
  loop tr = do
    threadDelay 25000000-- 25 seconds
    buf ← readLogBuffer sb
    forM_ buf $ λ(logname, LogObject _ lometa locontent) → do
      let tr' = modifyName (λn → "#buffer" <> "." <> n <> "." <> logname) tr
      traceNamedObject tr' (lometa, locontent)
    loop tr
```

Thread that outputs a random number to a Trace

```
randomThr :: Trace IO Text → IO (Async.Async ())
randomThr trace = do
  logInfo trace "starting random generator"
  let trace' = appendName "random" trace
  Async.async (loop trace')
where
  loop tr = do
    threadDelay 500000-- 0.5 second
    num ← randomRIO (42 − 42, 42 + 42) :: IO Double
    lo ← (,) <$> mkLOMeta Info Public <*> pure (LogValue "rr" (PureD num))
    traceNamedObject tr lo
    loop tr
```

Thread that outputs a random number to monitoring Trace

```
# ifdef RUN_ProcMonitoring
monitoringThr :: Trace IO Text → IO (Async.Async ())
monitoringThr trace = do
  logInfo trace "starting numbers for monitoring..."
  let trace' = appendName "monitoring" trace
  Async.async (loop trace')
where
  loop tr = do
    threadDelay 500000-- 0.5 second
    num ← randomRIO (42 − 42, 42 + 42) :: IO Double
    lo ← (,) <$> mkLOMeta Warning Public <*> pure (LogValue "monitMe" (PureD num))
    traceNamedObject tr lo
    loop tr
# endif
```

Thread that observes an *IO* action

```
#ifdef ENABLE_OBSERVABLES
observeIO :: Configuration → Trace IO Text → IO (Async.Async ())
observeIO config trace = do
  logInfo trace "starting observer"
  proc ← Async.async (loop trace)
  return proc
  where
    loop tr = do
      threadDelay 5000000-- 5 seconds
      let tr' = appendName "observeIO" tr
      _ ← bracketObserveIO config tr' Warning "complex.observeIO" $ do
        num ← randomRIO (100000, 200000) :: IO Int
        ls ← return $ reverse $ init $ reverse $ 42 : [1..num]
        pure $ const ls ()
      loop tr
#endif
```

Threads that observe *STM* actions on the same TVar

```
#ifdef ENABLE_OBSERVABLES
observeSTM :: Configuration → Trace IO Text → IO [Async.Async ()]
observeSTM config trace = do
  logInfo trace "starting STM observer"
  tvar ← atomically $ newTVar ([1..1000] :: [Int])
  -- spawn 10 threads
  proc ← forM [(1 :: Int)..10] $ λx → Async.async (loop trace tvar (pack $ show x))
  return proc
  where
    loop tr tvarlist trname = do
      threadDelay 10000000-- 10 seconds
      STM.bracketObserveIO config tr Warning ("observeSTM." <> trname) (stmAction tvarlist)
      loop tr tvarlist trname
stmAction :: TVar [Int] → STM ()
stmAction tvarlist = do
  list ← readTVar tvarlist
  writeTVar tvarlist $! (++) [42] $ reverse $ init $ reverse $ list
  pure ()
#endif
```

Thread that observes an *IO* action which downloads a text in order to observe the I/O statistics

```
#ifdef LINUX
#ifdef ENABLE_OBSERVABLES
observeDownload :: Configuration → Trace IO Text → IO (Async.Async ())
observeDownload config trace = do
  proc ← Async.async (loop trace)
```

```
    return proc
    where
      loop tr = do
        threadDelay 1000000-- 1 second
        let tr' = appendName "observeDownload" tr
        bracketObserveIO config tr' Warning "complex.observeDownload" $ do
          license ← openURI "http://www.gnu.org/licenses/gpl.txt"
          case license of
            Right bs → logNotice tr' $ pack $ BS8.unpack bs
            Left _ → return ()
          threadDelay 50000-- .05 second
          pure ()
        loop tr
# endif
# endif
```

Thread that periodically outputs a message

```
data Pet = Pet {name :: Text, age :: Int}
  deriving (Show)
instance ToObject Pet where
    toObject MinimalVerbosity _ = emptyObject-- do not log
    toObject NormalVerbosity (Pet _ _) =
      mkObject ["kind" .= String "Pet"]
    toObject MaximalVerbosity (Pet n a) =
      mkObject ["kind" .= String "Pet"
        ,"name" .= toJSON n
        ,"age" .= toJSON a]
instance HasTextFormatter Pet where
    formatText pet _o = "Pet " <> name pet <> " is " <> pack (show (age pet)) <> " years old."
instance Transformable Text IO Pet where
    -- transform to JSON Object
    trTransformer MaximalVerbosity tr = trStructuredText MaximalVerbosity tr
    trTransformer MinimalVerbosity _tr = nullTracer
    -- transform to textual representation using show
    trTransformer _v tr = Tracer $ λpet → do
      meta ← mkLOMeta Info Public
      traceWith tr $ ("pet",LogObject "pet" meta $ (LogMessage ∘ pack ∘ show) pet)
-- default privacy annotation:  Public
instance HasPrivacyAnnotation Pet
instance HasSeverityAnnotation Pet where
    getSeverityAnnotation _ = Critical
# ifdef RUN_ProcMessageOutput
msgThr :: Trace IO Text → IO (Async.Async ())
msgThr trace = do
  logInfo trace "start messaging .."
  let trace' = appendName "message" trace
  Async.async (loop trace')
  where
    loop tr = do
```

```
        threadDelay 3000000-- 3 seconds
        logNotice tr "N O T I F I C A T I O N ! ! !"
        logDebug tr "a detailed debug message."
        logError tr "Boooommm .."
        traceWith (toLogObject MaximalVerbosity tr) (Pet "bella" 8)
        loop tr
#endif
```

Thread that periodically outputs operating system counters

```
#ifdef RUN_ProcCounterOutput
countersThr :: Trace IO Text → IO (Async.Async ())
countersThr trace = do
  let trace' = appendName "counters" trace
  Async.async (loop trace')
  where
    loop tr = do
      threadDelay 3000000-- 3 seconds
      let counters = [MemoryStats, ProcessStats, NetStats, IOStats, SysStats]
      cts ← readCounters (ObservableTraceSelf counters)
      mle ← mkLOMeta Info Confidential
      forM_ cts $ λc@(Counter _ct cn cv) →
        traceNamedObject tr (mle, LogValue (nameCounter c <> "." <> cn) cv)
      loop tr
#endif
```

Main entry point

```
main :: IO ()
main = do
  -- create configuration
  c ← prepare_configuration
  -- create initial top-level Trace
  (tr :: Trace IO Text, sb) ← setupTrace_ c "complex"
  -- load plugins
  Cardano.BM.Backend ∘ Editor.plugin c tr sb
    ≫= loadPlugin sb
  Cardano.BM.Backend ∘ EKGView.plugin c tr sb
    ≫= loadPlugin sb
  forwardTo ← CM.getForwardTo c
  when (isJust forwardTo) $
    Cardano.BM.Backend ∘ TraceForwarder.plugin c tr sb "forwarderMinSeverity"
      ≫= loadPlugin sb
  Cardano.BM.Backend ∘ Aggregation.plugin c tr sb
    ≫= loadPlugin sb
  Cardano.BM.Backend ∘ Monitoring.plugin c tr sb
    ≫= loadPlugin sb
#ifdef LINUX
  -- inspect logs with 'journalctl -t example-complex'
```

```
  Cardano.BM.Scribe ∘ Systemd.plugin c tr sb "example-complex"
    ≫= loadPlugin sb
# endif
  logNotice tr "starting program; hit CTRL-C to terminate"
-- user can watch the progress only if EKG is enabled.
  logInfo tr "watch its progress on http://localhost:12790"
# ifdef RUN_ProcBufferDump
  procDump ← dumpBuffer sb tr
# endif
# ifdef RUN_ProcRandom
   {-start thread sending unbounded sequence of random numbers to a trace which aggregates them ir
  procRandom ← randomThr tr
# endif
# ifdef RUN_ProcMonitoring
  procMonitoring ← monitoringThr tr
# endif
# ifdef RUN_ProcObserveIO
  -- start thread endlessly reversing lists of random length
# ifdef ENABLE_OBSERVABLES
  procObsvIO ← observeIO c tr
# endif
# endif
# ifdef RUN_ProcObseverSTM
  -- start threads endlessly observing STM actions operating on the same TVar
# ifdef ENABLE_OBSERVABLES
  procObsvSTMs ← observeSTM c tr
# endif
# endif
# ifdef LINUX
# ifdef RUN_ProcObseveDownload
  -- start thread endlessly which downloads sth in order to check the I/O usage
# ifdef ENABLE_OBSERVABLES
  procObsvDownload ← observeDownload c tr
# endif
# endif
# endif
# ifdef RUN_ProcMessageOutput
  -- start a thread to output a text messages every n seconds
  procMsg ← msgThr tr
# endif
# ifdef RUN_ProcCounterOutput
  procCounters ← countersThr tr
# endif
# ifdef RUN_ProcCounterOutput
  _ ← Async.waitCatch procCounters
# endif
# ifdef RUN_ProcMessageOutput
  -- wait for message thread to finish, ignoring any exception
  _ ← Async.waitCatch procMsg
# endif
```

```
# ifdef LINUX
# ifdef RUN_ProcObseveDownload
  -- wait for download thread to finish, ignoring any exception
# ifdef ENABLE_OBSERVABLES
  _ ← Async.waitCatch procObsvDownload
# endif
# endif
# endif
# ifdef RUN_ProcObseverSTM
  -- wait for observer thread to finish, ignoring any exception
# ifdef ENABLE_OBSERVABLES
  _ ← forM procObsvSTMs Async.waitCatch
# endif
# endif
# ifdef RUN_ProcObserveIO
  -- wait for observer thread to finish, ignoring any exception
# ifdef ENABLE_OBSERVABLES
  _ ← Async.waitCatch procObsvIO
# endif
# endif
# ifdef RUN_ProcRandom
  -- wait for random thread to finish, ignoring any exception
  _ ← Async.waitCatch procRandom
# endif
# ifdef RUN_ProcMonitoring
  _ ← Async.waitCatch procMonitoring
# endif
# ifdef RUN_ProcBufferDump
  _ ← Async.waitCatch procDump
# endif
  return ()
```

1.5.3 Performance example for time measurements

Module header and import directives

```
{-# LANGUAGE ScopedTypeVariables #-}
module Main
  (main)
  where
import qualified Control.Concurrent.Async as Async
import Control.Monad (forM_)
import qualified Data.HashMap.Strict as HM
import Data.Text (Text)
import Criterion (Benchmark, bench, nfIO)
import Criterion.Main (defaultMain)
import Cardano.BM.Backend.Switchboard
import qualified Cardano.BM.Configuration.Model as CM
import Cardano.BM.Data.Aggregated (Measurable (..))
import Cardano.BM.Data.BackendKind
```

```
import Cardano.BM.Data.LogItem
import Cardano.BM.Data.MonitoringEval
import Cardano.BM.Data.Severity
import Cardano.BM.Setup
import Cardano.BM.Trace
```

Define configuration

```
prepare_configuration :: IO CM.Configuration
prepare_configuration = do
  c ← CM.empty
  CM.setMinSeverity c Warning
  CM.setSetupBackends c [MonitoringBK]
  CM.setDefaultBackends c [MonitoringBK]
  CM.setMonitors c $ HM.fromList
    [("performance.monitoring"
      ,(Nothing
        ,Compare "monitMe" (GE,(OpMeasurable 42))
        ,[SetGlobalMinimalSeverity Debug]
        )
      )
    ]
  CM.setBackends c "performance.monitoring" (Just [MonitoringBK])
  return c
```

Thread that outputs a value to monitoring Trace

```
monitoringThr :: Trace IO Text → Int → IO (Async.Async ())
monitoringThr trace objNumber = do
  let trace' = appendName "monitoring" trace
  obj ← (,) <$> (mkLOMeta Warning Public) <*> pure (LogValue "monitMe" (PureD 123.45))
  proc ← Async.async (loop trace' obj)
  return proc
  where
    loop tr lo = do
      forM_ [1..objNumber] $ \_ → traceNamedObject tr lo
        -- terminate Switchboard
      killPill ← (,) <$> (mkLOMeta Warning Public) <*> pure KillPill
      traceNamedObject tr killPill
```

Main entry point

```
main :: IO ()
main = defaultMain
  [benchMain 1000
  ,benchMain 10000
  ,benchMain 100000
  ,benchMain 1000000
```

```
    ]
benchMain :: Int → Benchmark
benchMain objNumber = bench (show objNumber ++ " objects") $ nfIO $ do
  c ← prepare_configuration
  (tr :: Trace IO Text, sb) ← setupTrace_ c "performance"
  procMonitoring ← monitoringThr tr objNumber
  _ ← Async.wait procMonitoring
  _ ← waitForTermination sb
  return ()
```

1.6 Code listings - contra-tracer package

1.6.1 Examples

Tracing using the contravariant Tracer naturally reads:

```
let logTrace = traceWith $ showTracing $ stdoutTracer
in logTrace "hello world"
```

1.6.2 Contravariant Tracer

The notion of a Tracer is an action that can be used to observe information of interest during evaluation. Tracers can capture (and annotate) such observations with additional information from their execution context.

```
newtype Tracer m a = Tracer {runTracer :: a → m ()}
```

A Tracer is an instance of *Contravariant*, which permits new Tracers to be constructed that feed into the existing Tracer by use of *contramap*.

```
instance Contravariant (Tracer m) where
  contramap f (Tracer t) = Tracer (t ∘ f)
```

Although a Tracer is invoked in a monadic context (which may be *Identity*), the construction of a new Tracer is a pure function. This brings with it the constraint that the derived Tracers form a hierachy which has its root at the top level tracer.

In principle a Tracer is an instance of *Semigroup* and *Monoid*, by sequential composition of the tracing actions.

```
instance Applicative m ⇒ Semigroup (Tracer m s) where
  Tracer a1 <> Tracer a2 = Tracer $ λs → a1 s *> a2 s
instance Applicative m ⇒ Monoid (Tracer m s) where
  mappend = (<>)
  mempty = nullTracer
```

nullTracer

The simplest tracer - one that suppresses all output.

```
nullTracer :: Applicative m ⇒ Tracer m a
nullTracer = Tracer $ \_ → pure ()
```

traceWith

```
traceWith :: Tracer m a → a → m ()
traceWith = runTracer
```

1.6.3 Transformers

Contravariant transformers using Kleisli arrows

Tracers can be transformed using Kleisli arrows, e.g. arrows of the type $Monad\ m \Rightarrow a \rightarrow m\ b$, technically this makes Tracer a contravariant functor over *Kleisli* category. The important difference from using 'contramap' is that the monadic action runs when a tracer is called, this might be the prefered behaviour when trying to trace timeing information.

```
contramapM :: Monad m
  ⇒ (a → m b)
  → Tracer m b
  → Tracer m a
contramapM f (Tracer tr) = Tracer (f >=> tr)
```

Applying *show* on a Tracer's messages

The Tracer transformer exploiting Show.

```
showTracing :: (Show a) ⇒ Tracer m String → Tracer m a
showTracing = contramap show
```

Conditional tracing - statically defined

The Tracer transformer that allows for on/off control of tracing at trace creation time.

```
condTracing :: (Monad m) ⇒ (a → Bool) → Tracer m a → Tracer m a
condTracing active tr = Tracer $ λs →
  when (active s) (traceWith tr s)
```

Conditional tracing - dynamically evaluated

The tracer transformer that can exercise dynamic control over tracing, the dynamic decision being made using the context accessible in the monadic context.

```
condTracingM :: (Monad m) ⇒ m (a → Bool) → Tracer m a → Tracer m a
condTracingM activeP tr = Tracer $ λs → do
  active ← activeP
  when (active s) (traceWith tr s)
```

natTrace

Natural transformation from monad m to monad n.

```
natTracer :: (forall x ∘ m x → n x) → Tracer m s → Tracer n s
natTracer nat (Tracer tr) = Tracer (nat ∘ tr)
```

1.6.4 Output

Directing a Tracer's output to stdout

The Tracer that prints a string (as a line) to stdout (usual caveats about interleaving should be heeded).

```
stdoutTracer :: (MonadIO m) ⇒ Tracer m String
stdoutTracer = Tracer $ liftIO ∘ putStrLn
```

Outputting a Tracer with *Debug.Trace*

A Tracer that uses *TraceM* (from Debug.Trace) as its output mechanism.

```
debugTracer :: (Applicative m) ⇒ Tracer m String
debugTracer = Tracer Debug.Trace.traceM
```

1.7 Code listings - iohk-monitoring package

1.7.1 Cardano.BM.Observer.STM

```
stmWithLog :: STM.STM (t, [(LOMeta, LOContent a)]) → STM.STM (t, [(LOMeta, LOContent a)])
stmWithLog action = action
```

Observe *STM* action in a named context

With given name, create a SubTrace according to Configuration and run the passed *STM* action on it.

```
bracketObserveIO :: Config.Configuration → Trace IO a → Severity → Text → STM.STM t → IO t
bracketObserveIO config trace severity name action = do
    subTrace ← fromMaybe Neutral <$> Config.findSubTrace config name
    bracketObserveIO' subTrace severity trace action
  where
    bracketObserveIO' :: SubTrace → Severity → Trace IO a → STM.STM t → IO t
    bracketObserveIO' NoTrace _ _ act =
      STM.atomically act
    bracketObserveIO' subtrace sev logTrace act = do
      mCountersid ← observeOpen subtrace sev logTrace
      -- run action; if an exception is caught, then it will be logged and rethrown.
      t ← (STM.atomically act) `catch` (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) ≫ thr
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClo
          -- however the result of the action is returned
          TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace sev logTrace countersid []
          case res of
            Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
            _ → pure ()
      pure t
```

Observe *STM* action in a named context and output captured log items

The *STM* action might output messages, which after "success" will be forwarded to the logging trace. Otherwise, this function behaves the same as bracketObserveIO.

```
bracketObserveLogIO :: Config.Configuration → Trace IO a → Severity → Text → STM.STM (t, [(LOMeta
bracketObserveLogIO config trace severity name action = do
    subTrace ← fromMaybe Neutral <$> Config.findSubTrace config name
    bracketObserveLogIO' subTrace severity trace action
  where
    bracketObserveLogIO' :: SubTrace → Severity → Trace IO a → STM.STM (t, [(LOMeta, LOContent a)])
    bracketObserveLogIO' NoTrace _ _ act = do
      (t, _) ← STM.atomically $ stmWithLog act
      pure t
    bracketObserveLogIO' subtrace sev logTrace act = do
      mCountersid ← observeOpen subtrace sev logTrace
      -- run action, return result and log items; if an exception is
      -- caught, then it will be logged and rethrown.
      (t, as) ← (STM.atomically $ stmWithLog act) `catch`
          (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) ≫ throwM e))
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClo
          -- however the result of the action is returned
          TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace sev logTrace countersid as
          case res of
            Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
            _ → pure ()
      pure t
```

1.7.2 Cardano.BM.Observer.Monadic

Monadic.bracketObserverIO

Observes an *IO* action. The subtrace type is found in the configuration with the passed-in name.

Microbenchmarking steps:

1. Create a *trace* which will have been configured to observe things besides logging.

```
import qualified Cardano.BM.Configuration.Model as CM
 ∘ ∘ ∘
c ← config
trace ← setupTrace (Right c) "demo-playground"
  where
    config :: IO CM.Configuration
    config = do
      c ← CM.empty
      CM.setMinSeverity c Debug
```

```
CM.setSetupBackends c [KatipBK, AggregationBK]
CM.setDefaultBackends c [KatipBK, AggregationBK]
CM.setSetupScribes c [ScribeDefinition {
  scName = "stdout"
  ,scKind = StdoutSK
  ,scRotation = Nothing
  }
  ]
CM.setDefaultScribes c ["StdoutSK::stdout"]
return c
```

2. *c* is the Configuration of *trace*. In order to enable the collection and processing of measurements (min, max, mean, std-dev) *AggregationBK* is needed.

```
CM.setDefaultBackends c [KatipBK, AggregationBK]
```

in a configuration file (YAML) means

```
defaultBackends:
  - KatipBK
  - AggregationBK
```

3. Set the measurements that you want to take by changing the configuration of the *trace* using setSubTrace, in order to declare the namespace where we want to enable the particular measurements and the list with the kind of measurements.

```
CM.setSubTrace
    config
    "submit-tx"
    (Just $ ObservableTraceSelf observablesSet)
  where
    observablesSet = [MonotonicClock, MemoryStats]
```

4. Find an action to measure. e.g.:

```
runProtocolWithPipe x hdl proto `catch` (λProtocolStopped → return ())
```

and use bracketObserveIO. e.g.:

```
bracketObserveIO trace "submit-tx" $
  runProtocolWithPipe x hdl proto `catch` (λProtocolStopped → return ())


bracketObserveIO :: Config.Configuration → Trace IO a → Severity → Text → IO t → IO t
bracketObserveIO config trace severity name action = do
    subTrace ← fromMaybe Neutral <$> Config.findSubTrace config name
    bracketObserveIO' subTrace severity trace action
  where
    bracketObserveIO' :: SubTrace → Severity → Trace IO a → IO t → IO t
    bracketObserveIO' NoTrace _ _ act = act
    bracketObserveIO' subtrace sev logTrace act = do
      mCountersid ← observeOpen subtrace sev logTrace
      -- run action; if an exception is caught it will be logged and rethrown.
      t ← act `catch` (λ(e :: SomeException) → (TIO.hPutStrLn stderr (pack (show e)) ≫ throwM e))
```

```
    case mCountersid of
      Left openException →
        -- since observeOpen faced an exception there is no reason to call observeClo
        -- however the result of the action is returned
        TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
      Right countersid → do
        res ← observeClose subtrace sev logTrace countersid [ ]
        case res of
          Left ex → TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex))
          _ → pure ()
    pure t
```

Monadic.bracketObserverM

Observes a *MonadIO m* ⇒ *m* action.

```
bracketObserveM :: (MonadCatch m, MonadIO m) ⇒ Config.Configuration → Trace m a → Severity → Text
bracketObserveM config trace severity name action = do
    subTrace ← liftIO $ fromMaybe Neutral < $ > Config.findSubTrace config name
    bracketObserveM' subTrace severity trace action
  where
    bracketObserveM' :: (MonadCatch m, MonadIO m) ⇒ SubTrace → Severity → Trace m a → m t → m t
    bracketObserveM' NoTrace _ _ act = act
    bracketObserveM' subtrace sev logTrace act = do
      mCountersid ← observeOpen subtrace sev logTrace
      -- run action; if an exception is caught it will be logged and rethrown.
      t ← act `catch` (λ(e :: SomeException) → liftIO (TIO.hPutStrLn stderr (pack (show e)) ≫ throwM e))
      case mCountersid of
        Left openException →
          -- since observeOpen faced an exception there is no reason to call observeClo
          -- however the result of the action is returned
          liftIO $ TIO.hPutStrLn stderr ("ObserveOpen: " <> pack (show openException))
        Right countersid → do
          res ← observeClose subtrace sev logTrace countersid [ ]
          case res of
            Left ex → liftIO (TIO.hPutStrLn stderr ("ObserveClose: " <> pack (show ex)))
            _ → pure ()
      pure t
```

Monadic.bracketObserver

Observes a *MonadIO m* ⇒ *m* action. This observer bracket does not interfere on exceptions.

```
bracketObserveX :: (MonadIO m) ⇒ Config.Configuration → Trace m a → Severity → Text → m t → m t
bracketObserveX config trace severity name action = do
    subTrace ← liftIO $ fromMaybe Neutral < $ > Config.findSubTrace config name
    bracketObserveX' subTrace severity trace action
  where
    bracketObserveX' :: (MonadIO m) ⇒ SubTrace → Severity → Trace m a → m t → m t
    bracketObserveX' NoTrace _ _ act = act
    bracketObserveX' subtrace sev logTrace act = do
```

```
    countersid ← observeOpen0 subtrace sev logTrace
    -- run action
    t ← act
    observeClose0 subtrace sev logTrace countersid [ ]
    pure t
```

observerOpen

```
observeOpen :: (MonadCatch m, MonadIO m) ⇒ SubTrace → Severity → Trace m a → m (Either SomeExcept
observeOpen subtrace severity logTrace = (do
  state ← observeOpen0 subtrace severity logTrace
  return (Right state)) `catch` (return ∘ Left)
observeOpen0 :: (MonadIO m) ⇒ SubTrace → Severity → Trace m a → m CounterState
observeOpen0 subtrace severity logTrace = do
  -- take measurement
  counters ← liftIO $ readCounters subtrace
  let state = CounterState counters
  if counters ≡ [ ]
  then return ()
  else do
    -- send opening message to Trace
    meta ← mkLOMeta severity Confidential
    traceNamedObject logTrace (meta, ObserveOpen state)
  return state
```

observeClose

```
observeClose
  :: (MonadCatch m, MonadIO m) ⇒ SubTrace → Severity → Trace m a
  → CounterState → [(LOMeta, LOContent a)]
  → m (Either SomeException ())
observeClose subtrace sev logTrace initState logObjects = (do
  observeClose0 subtrace sev logTrace initState logObjects
  return (Right ())) `catch` (return ∘ Left)
observeClose0 :: (MonadIO m) ⇒ SubTrace → Severity → Trace m a
  → CounterState → [(LOMeta, LOContent a)]
  → m ()
observeClose0 subtrace sev logTrace initState logObjects = do
  let initialCounters = csCounters initState
  -- take measurement
  counters ← liftIO $ readCounters subtrace
  if counters ≡ [ ]
  then return ()
  else do
    mle ← mkLOMeta sev Confidential
    -- send closing message to Trace
    traceNamedObject logTrace $
      (mle, ObserveClose (CounterState counters))
```

```
    -- send diff message to Trace
    traceNamedObject logTrace $
      (mle, ObserveDiff (CounterState (diffCounters initialCounters counters)))
  -- trace the messages gathered from inside the action
  forM_ logObjects $ traceNamedObject logTrace
  return ()
```

1.7.3 Cardano.BM.Tracing

1.7.4 Cardano.BM.Tracer

Divisible and Decidable instances of Tracer

A *Divisible* contravariant functor is the contravariant analogue of *Applicative*. A *Divisible* contravariant functor has the ability to be composed "beside" another contravariant. It gives a way to combine two contravariant functors that focus on different parts of a structure. (see https://hackage.haskell.org/package/contravariant-1.5/docs/Data-Functor-Contravariant-Divi

```
instance Applicative m ⇒ Divisible (Tracer m) where
  divide :: (a → (b,c)) → Tracer m b → Tracer m c → Tracer m a
  divide f (Tracer g) (Tracer h) = Tracer $ λa → case f a of
    (b,c) → g b *> h c
  conquer :: Tracer m a
  conquer = nullTracer
```

A *Decidable* contravariant functor is the contravariant analogue of *Alternative*. Noting the superclass constraint that the contravariant functor must also be *Divisible*, a *Decidable* functor has the ability to "fan out" input, under the intuition that contravariant functors consume input. It chooses the appropriate contravariant functor for a data structure that is an alternative choice (sum) of two different parts. (see https://hackage.haskell.org/package/contravariant-1.5/docs/Data-

```
instance Applicative m ⇒ Decidable (Tracer m) where
  lose :: (a → Void) → Tracer m a
  lose _ = nullTracer
  choose :: (a → Either b c) → Tracer m b → Tracer m c → Tracer m a
  choose f (Tracer g) (Tracer h) = Tracer $ either g h ∘ f
```

bracketObserve

Indicates the beginning and the end of an action. *matchObservations* can be used if we want a Tracer which produces the difference between the starting and the ending observations of the action.

```
bracketObserve :: forall m s e b d ∘ Monad m
  ⇒ (m s, m e, Tracer m (Observable s e d))
  → m b
  → m b
bracketObserve (getStart, getEnd, tr) action = do
  let transform :: Tracer m (Observable s e d) → Tracer m ObserveIndicator
      transform trace = Tracer $ λcase
        ObserveBefore → do
          start ← getStart
```

```
            traceWith trace $ OStart start
          ObserveAfter → do
            end ← getEnd
            traceWith trace $ OEnd end Nothing
      tr' = transform tr
    traceWith tr' ObserveBefore
    res ← action
    traceWith tr' ObserveAfter
    return res
```

example

```
  data AddSub a = Add a
    | Sub a
    deriving Show
  type Time = Word64
  type ObservableS t = Observable t t t
  example :: IO Int
  example = do
      let trInt :: Tracer IO (AddSub Int)
          trInt = showTracing stdoutTracer
          trObserve :: Tracer IO (ObservableS Time)
          trObserve = showTracing stdoutTracer
      _ ← bracketObserve (getMonotonicTimeNSec, getMonotonicTimeNSec, trObserve) (actionAdd trInt)
      bracketObserve (getMonotonicTimeNSec, getMonotonicTimeNSec, trObserve) (actionSub trInt)
    where
      actionAdd :: Tracer IO (AddSub Int) → IO Int
      actionAdd tr = do
        let res = 1 + 2
        traceWith tr $ Add res
        return res
      actionSub :: Tracer IO (AddSub Int) → IO Int
      actionSub tr = do
        let res = 1 - 2
        traceWith tr $ Sub res
        return res
  exampleWithChoose :: IO Int
  exampleWithChoose = do
      let trInt :: Tracer IO (AddSub Int)
          trInt = showTracing stdoutTracer
          trObserve :: Tracer IO (ObservableS (AddSub Time))
          trObserve = showTracing stdoutTracer
          trace :: Tracer IO (Either (ObservableS (AddSub Time)) (AddSub Int))
          trace = choose id trObserve trInt
          bracketObserve' (getTime, tr) = bracketObserve (getTime, getTime, tr)
      _ ← bracketObserve' (Add <$> getMonotonicTimeNSec, contramap Left trace) $ actionAdd $ contramap
      bracketObserve' (Sub <$> getMonotonicTimeNSec, contramap Left trace) $ actionSub $ contramap Right
    where
```

```
    actionAdd :: Tracer IO (AddSub Int) → IO Int
    actionAdd tr = do
      let res = 1 + 2
      traceWith tr $ Add res
      return res
    actionSub :: Tracer IO (AddSub Int) → IO Int
    actionSub tr = do
      let res = 1 - 2
      traceWith tr $ Sub res
      return res
instance Show (ObservableS Time) where
  show (OStart time) = "OStart " ++ show time
  show (OEnd time mTime) = "OEnd " ++ show time ++ ", ODiff " ++ show mTime
instance Show (ObservableS (AddSub Time)) where
  show (OStart a) = "OStart " ++ show a
  show (OEnd a b) = "OEnd " ++ show a ++ ", ODiff " ++ show b
```

1.7.5 Cardano.BM.Trace

Utilities

Natural transformation from monad m to monad n.

```
natTrace :: (forall x ∘ m x → n x) → Tracer m (LoggerName, LogObject a) → Tracer n (LoggerName, LogOb
natTrace = natTracer
```

Enter new named context

A new context name is added.

```
appendName :: LoggerName → Trace m a → Trace m a
appendName name tr = Tracer $ λ(names0, lo) →
  let names = if names0 ≡ T.empty then name else name <> "." <> names0
  in
  traceWith tr (names, lo)
```

Change named context

The context name is overwritten.

```
modifyName
    :: (LoggerName → LoggerName)
    → Trace m a
    → Trace m a
modifyName k = contramap f
  where
    f (names0, lo) = (k names0, lo)
```

Contramap a trace and produce the naming context

```
named :: Tracer m (LoggerName, LogObject a) → Tracer m (LOMeta, LOContent a)
named = contramap $ λ(meta, loc) → (mempty, LogObject mempty meta loc)
```

Trace a LogObject through

```
traceNamedObject
  :: MonadIO m
  ⇒ Trace m a
  → (LOMeta, LOContent a)
  → m ()
traceNamedObject logTrace lo =
  traceWith (named logTrace) lo
```

Concrete Trace on stdout

This function returns a trace with an action of type "LogObject $a \rightarrow IO\ ()$" which will output a text message as text and all others as JSON encoded representation to the console.

TODO remove *locallock*

```
locallock :: MVar ()
locallock = unsafePerformIO $ newMVar ()
```

```
stdoutTrace :: Trace IO T.Text
stdoutTrace = Tracer $ λ(ctx, LogObject _loname _ lc) →
    withMVar locallock $ \_ →
      case lc of
        (LogMessage logItem) →
          output ctx logItem
        obj →
          output ctx $ toStrict (encodeToLazyText obj)
  where
    output nm msg = TIO.putStrLn $ nm <> "  ::  " <> msg
```

Concrete Trace into a *TVar*

```
traceInTVar :: STM.TVar [a] → Tracer STM.STM a
traceInTVar tvar = Tracer $ λa → STM.modifyTVar tvar ((:) a)
traceInTVarIO :: STM.TVar [a] → Tracer IO a
traceInTVarIO tvar = Tracer $ λa →
  STM.atomically $ STM.modifyTVar tvar ((:) a)
```

Enter message into a trace

The function traceNamedItem creates a LogObject and threads this through the action defined in the Trace.

```
traceNamedItem
  :: MonadIO m
  ⇒ Trace m a
  → PrivacyAnnotation
  → Severity
  → a
  → m ()
traceNamedItem logTrace p s m =
  traceNamedObject logTrace =≪
    (,) <$> liftIO (mkLOMeta s p)
        <*> pure (LogMessage m)
```

Logging functions

```
logDebug, logInfo, logNotice, logWarning, logError, logCritical, logAlert, logEmergency
  :: MonadIO m ⇒ Trace m a → a → m ()
logDebug   logTrace = traceNamedItem logTrace Public Debug
logInfo    logTrace = traceNamedItem logTrace Public Info
logNotice  logTrace = traceNamedItem logTrace Public Notice
logWarning logTrace = traceNamedItem logTrace Public Warning
logError   logTrace = traceNamedItem logTrace Public Error
logCritical logTrace = traceNamedItem logTrace Public Critical
logAlert   logTrace = traceNamedItem logTrace Public Alert
logEmergency logTrace = traceNamedItem logTrace Public Emergency
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS, logCriticalS, logAlertS, logEmergencyS
  :: MonadIO m ⇒ Trace m a → a → m ()
logDebugS   logTrace = traceNamedItem logTrace Confidential Debug
logInfoS    logTrace = traceNamedItem logTrace Confidential Info
logNoticeS  logTrace = traceNamedItem logTrace Confidential Notice
logWarningS logTrace = traceNamedItem logTrace Confidential Warning
logErrorS   logTrace = traceNamedItem logTrace Confidential Error
logCriticalS logTrace = traceNamedItem logTrace Confidential Critical
logAlertS   logTrace = traceNamedItem logTrace Confidential Alert
logEmergencyS logTrace = traceNamedItem logTrace Confidential Emergency
```

1.7.6 Cardano.BM.ElidingTracer

Tracer transformer for eliding messages

The eliding tracer transformer depends on two predicates to decide on which observable type eliding messages is active (**??**), and whether two messages can be considered equivalent and thus be elided (**??**).

```
class ElidingTracer a where
```

This predicate is *True* for message types for which eliding is enabled. Needs to be overwritten in instances of *ElidingTracer*.

```
doelide :: a → Bool
```

The predicate to determine if two messages are *equivalent*. This needs to be overwritten in instances of *ElidingTracer*.

```
isEquivalent :: a → a → Bool
```

Create a new state *MVar*.

```
newstate :: IO (MVar (Maybe a, Integer))
default newstate :: IO (MVar (Maybe a, Integer))
newstate = newMVar (Nothing, 0)
```

Internal state transitions.

```
starteliding :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → Trace IO t
    → a → IO (Maybe a, Integer)
default starteliding :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → Trace IO t
    → a → IO (Maybe a, Integer)
starteliding tverb tr ev = do
  traceWith (toLogObject tverb tr) ev
  return (Just ev, 0)
conteliding :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → Trace IO t
    → a → (Maybe a, Integer) → IO (Maybe a, Integer)
default conteliding :: Transformable t IO a
    ⇒ TracingVerbosity → Trace IO t
    → a → (Maybe a, Integer) → IO (Maybe a, Integer)
conteliding _tverb _tr _ (Nothing, _count) = return (Nothing, 0)
conteliding _tverb _tr ev (_old, count) = return (Just ev, count + 1)
stopeliding :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → Trace IO t
    → a → (Maybe a, Integer) → IO (Maybe a, Integer)
default stopeliding :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → Trace IO t
    → a → (Maybe a, Integer) → IO (Maybe a, Integer)
stopeliding tverb tr ev (Nothing, _count) = do
  traceWith (toLogObject tverb tr) ev
  return (Nothing, 0)
stopeliding tverb tr ev (Just ev0, count) = do
  when (count > 1) $ do -- report the number of elided messages
    meta ← mkLOMeta (getSeverityAnnotation ev0) (getPrivacyAnnotation ev0)
    traceNamedObject tr (meta, LogValue "before next, messages elided" (PureI $ toInteger (count –
  when (count > 0) $ -- output last elided message
    traceWith (toLogObject tverb tr) ev0
  traceWith (toLogObject tverb tr) ev
  return (Nothing, 0)
```

The transformer from a Tracer IO empha to Trace *IO t* contains the main logic of eliding messages.

```
elideToLogObject
    :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → MVar (Maybe a, Integer)
    → Trace IO t → Tracer IO a
default elideToLogObject
    :: (ToObject t, Transformable t IO a)
    ⇒ TracingVerbosity → MVar (Maybe a, Integer)
    → Trace IO t → Tracer IO a
elideToLogObject tverb mvar tr = Tracer $ λev →
  modifyMVar_ mvar $ λs@(old, _count) →
  if doelide ev
    then
      case old of
        Nothing → starteliding tverb tr ev
        Just ev0 →
          if ev `isEquivalent` ev0
            then
              conteliding tverb tr ev s ≫= λcase
                (Nothing, _) → stopeliding tverb tr ev s
                newpair → return newpair
            else
              stopeliding tverb tr ev s
    else
      stopeliding tverb tr ev s
```

1.7.7 Cardano.BM.Setup

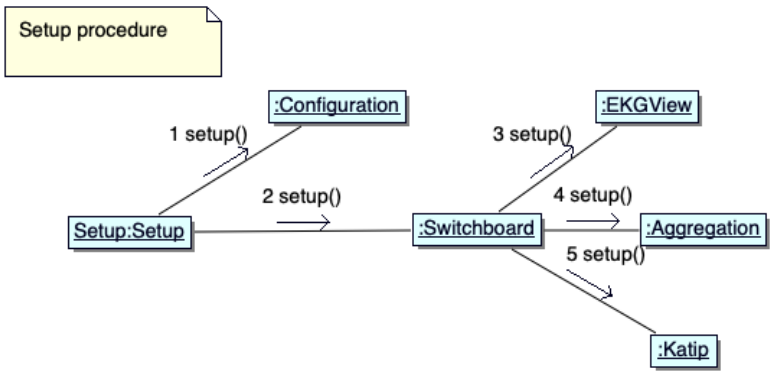


Figure 1.4: Setup procedure

setupTrace

Setup a new Trace with either a given Configuration or a *FilePath* to a configuration file. After all tracing operations have ended; *shutdownTrace* must be called.

```
setupTrace :: (MonadIO m, ToJSON a, FromJSON a, ToObject a) ⇒ Either FilePath Config.Configuration →
setupTrace (Left cfgFile) name = do
```

```
    c ← liftIO $ Config.setup cfgFile
    fst <$> setupTrace_ c name
setupTrace (Right c) name = fst <$> setupTrace_ c name

setupTrace_ :: (MonadIO m, ToJSON a, FromJSON a, ToObject a) ⇒ Config.Configuration → Text → m (Trac
setupTrace_ c name = do
    sb ← liftIO $ Switchboard.realize c
    let tr = appendName name $ natTrace liftIO (Switchboard.mainTraceConditionally c sb)
    return (tr, sb)
```

shutdown

Shut down the Switchboard and all the Traces related to it.

```
shutdown :: (ToJSON a, FromJSON a, ToObject a) ⇒ Switchboard.Switchboard a → IO ()
shutdown = Switchboard.unrealize
```

withTrace

Setup a Trace from Configuration and pass it to the action. At the end, shutdown all the components and close the trace.

```
withTrace :: (MonadIO m, MonadMask m, ToJSON a, FromJSON a, ToObject a) ⇒ Config.Configuration → T
withTrace cfg name action =
    bracket
        (setupTrace_ cfg name)              -- aquire
        (λ(_, sb) → liftIO $ shutdown sb) -- release
        (λ(tr, _) → action tr)              -- action
```

1.7.8 Cardano.BM.Counters

The platform is chosen on which we compile this library.

Currently, we mainly support *Linux* with its 'proc' filesystem, but also partially support *Windows*.

```
{-# LANGUAGE CPP #-}
module Cardano.BM.Counters
    (
      Platform.readCounters
    , getMonoClock
    ) where
#if defined (linux_HOST_OS)
import qualified Cardano.BM.Counters.Linux as Platform
#elif defined (mingw32_HOST_OS)
import qualified Cardano.BM.Counters.Windows as Platform
#elif defined (darwin_HOST_OS)
import qualified Cardano.BM.Counters.Darwin as Platform
#else
import qualified Cardano.BM.Counters.Dummy as Platform
#endif
import Cardano.BM.Counters.Common (getMonoClock)
```

1.7.9 Cardano.BM.Counters.Common

Common functions that serve *readCounters* on all platforms.

```
nominalTimeToMicroseconds :: Word64 → Microsecond
nominalTimeToMicroseconds = fromMicroseconds ∘ toInteger ∘ (`div` 1000)
```

Read monotonic clock

```
getMonoClock :: IO [Counter]
getMonoClock = do
  t ← getMonotonicTimeNSec
  return [Counter MonotonicClockTime "monoclock" $ Microseconds (t `div` 1000)]
```

Read GHC RTS statistics

Read counters from GHC's *RTS* (runtime system). The values returned are as per the last GC (garbage collection) run.

```
readRTSStats :: IO [Counter]
readRTSStats = do
    iscollected ← GhcStats.getRTSStatsEnabled
    if iscollected
      then ghcstats
      else return [ ]
  where
    ghcstats :: IO [Counter]
    ghcstats = do
      -- need to run GC?
      rts ← GhcStats.getRTSStats
      let getrts = ghcval rts
      return [getrts (Bytes ∘ fromIntegral ∘ GhcStats.allocated_bytes, "bytesAllocated")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.cumulative_live_bytes, "liveBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_live_bytes, "maxLiveBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_large_objects_bytes, "maxLargeBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_compact_bytes, "maxCompactBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_slop_bytes, "maxSlopBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.max_mem_in_use_bytes, "maxUsedMemBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.gcdetails_live_bytes ∘ GhcStats.gc, "gcLiveBytes")
        ,getrts (Bytes ∘ fromIntegral ∘ GhcStats.gcdetails_copied_bytes ∘ GhcStats.gc, "gcCopiedBytes")
        ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_cpu_ns, "gcCpuNs")
        ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.gc_elapsed_ns, "gcElapsedNs")
        ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.cpu_ns, "cpuNs")
        ,getrts (Nanoseconds ∘ fromIntegral ∘ GhcStats.elapsed_ns, "elapsedNs")
        ,getrts (PureI ∘ toInteger ∘ GhcStats.gcs, "gcNum")
        ,getrts (PureI ∘ toInteger ∘ GhcStats.major_gcs, "gcMajorNum")
        ]
    ghcval :: GhcStats.RTSStats → ((GhcStats.RTSStats → Measurable), Text) → Counter
    ghcval s (f, n) = Counter RTSStats n $ (f s)
```

1.7.10 Cardano.BM.Counters.Dummy

This is a dummy definition of *readCounters* on platforms that do not support the 'proc' filesystem from which we would read the counters.

The only supported measurements are monotonic clock time and RTS statistics for now.

```
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace                = return []
readCounters Neutral                = return []
readCounters (TeeTrace _)           = return []
readCounters (FilterTrace _)        = return []
readCounters UntimedTrace           = return []
readCounters DropOpening            = return []
readCounters (SetSeverity _)        = return []
#ifdef ENABLE_OBSERVABLES
readCounters (ObservableTraceSelf tts) = readCounters' tts []
readCounters (ObservableTrace _ tts) = readCounters' tts []
readCounters' :: [ObservableInstance] → [Counter] → IO [Counter]
readCounters' [] acc = return acc
readCounters' (MonotonicClock : r) acc = getMonoClock >>= λxs → readCounters' r $ acc ++ xs
readCounters' (GhcRtsStats : r) acc = readRTSStats >>= λxs → readCounters' r $ acc ++ xs
readCounters' (SysStats      : r) acc = readCounters' r $ acc ++ [Counter SysInfo "Platform" (PureI $ fromInte
readCounters' (_             : r) acc = readCounters' r acc
#else
readCounters (ObservableTraceSelf _) = return [Counter SysInfo "Platform" (PureI $ fromIntegral $ fromE
readCounters (ObservableTrace _ _) = return []
#endif
```

1.7.11 Cardano.BM.Counters.Linux

we have to expand the *readMemStats* function
to read full data from *proc*

```
readCounters :: SubTrace → IO [Counter]
readCounters NoTrace                = return []
readCounters Neutral                = return []
readCounters (TeeTrace _)           = return []
readCounters (FilterTrace _)        = return []
readCounters UntimedTrace           = return []
readCounters DropOpening            = return []
readCounters (SetSeverity _)        = return []
#ifdef ENABLE_OBSERVABLES
readCounters (ObservableTraceSelf tts) = do
    pid ← getProcessID
    takeMeasurements pid tts
readCounters (ObservableTrace pid tts) =
    takeMeasurements pid tts
takeMeasurements :: ProcessID → [ObservableInstance] → IO [Counter]
takeMeasurements pid tts =
    foldrM (λ(sel,fun) a →
      if any (≡ sel) tts
```

```
        then (fun >>= λxs → return $ a ++ xs)
        else return a) [ ] selectors
  where
    selectors = [(MonotonicClock, getMonoClock)
      ,(MemoryStats, readProcStatM pid)
      ,(ProcessStats, readProcStats pid)
      ,(NetStats, readProcNet pid)
      ,(SysStats, readSysStats pid)
      ,(IOStats, readProcIO pid)
      ,(GhcRtsStats, readRTSStats)
      ]
#else
readCounters (ObservableTraceSelf _) = return [ ]
readCounters (ObservableTrace _ _) = return [ ]
#endif

#ifdef ENABLE_OBSERVABLES
pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID → FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID → FilePath
pathProcStatM pid = pathProc </> (show pid) </> "statm"
pathProcIO :: ProcessID → FilePath
pathProcIO pid = pathProc </> (show pid) </> "io"
pathProcNet :: ProcessID → FilePath
pathProcNet pid = pathProc </> (show pid) </> "net" </> "netstat"
#endif
```

Reading from a file in /proc/<pid >

```
#ifdef ENABLE_OBSERVABLES
readProcList :: FilePath → IO [Integer]
readProcList fp = do
    fs ← getFileStatus fp
    if readable fs
    then do
      cs ← readFile fp
      return $ map (λs → maybe 0 id $ (readMaybe s :: Maybe Integer)) (words cs)
    else
      return [ ]
  where
    readable fs = intersectFileModes (fileMode fs) ownerReadMode ≡ ownerReadMode
#endif
```

readSysStats - generic platform specific information

```
#ifdef ENABLE_OBSERVABLES
readSysStats :: ProcessID → IO [Counter]
```

```
readSysStats pid = do
  return [Counter SysInfo "Pid" (PureI $ fromIntegral pid)
    ,Counter SysInfo "Platform" (PureI $ fromIntegral $ fromEnum Linux)
    ]
# endif
```

readProcStatM - /proc/<pid >/statm

```
/proc/[pid]/statm
       Provides information about memory usage, measured in pages.  The columns are:
              size       (1) total program size
                            (same as VmSize in /proc/[pid]/status)
              resident   (2) resident set size
                            (same as VmRSS in /proc/[pid]/status)
              shared     (3) number of resident shared pages (i.e., backed by a file)
                            (same as RssFile+RssShmem in /proc/[pid]/status)
              text       (4) text (code)
              lib        (5) library (unused since Linux 2.6; always 0)
              data       (6) data + stack
              dt         (7) dirty pages (unused since Linux 2.6; always 0)
```

```
# ifdef ENABLE_OBSERVABLES
readProcStatM :: ProcessID → IO [Counter]
readProcStatM pid = do
    ps0 ← readProcList (pathProcStatM pid)
    let ps = zip colnames ps0
      psUseful = filter (("unused" ≢) ∘ fst) ps
    return $ map (λ(n,i) → Counter MemoryCounter n (PureI i)) psUseful
  where
    colnames :: [Text]
    colnames = ["size","resident","shared","text","unused","data","unused"]
# endif
```

readProcStats - //proc//<pid >//stat

```
/proc/[pid]/stat
       Status  information about the process.  This is used by ps(1).  It is defined in the kernel source file
       fs/proc/array.c.

       The fields, in order, with their proper scanf(3) format specifiers, are listed below.  Whether  or  not
       certain  of  these  fields  display  valid information is governed by a ptrace access mode
       PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT check (refer to ptrace(2)).  If the check denies access,
       then the field value is displayed as 0.  The affected fields are indicated with the marking [PT].

       (1) pid  %d
                   The process ID.

       (2) comm  %s
                   The  filename  of  the  executable,  in parentheses.  This is visible whether or not the exe-
                   cutable is swapped out.

       (3) state  %c
                   One of the following characters, indicating process state:

                   R  Running

                   S  Sleeping in an interruptible wait

                   D  Waiting in uninterruptible disk sleep

                   Z  Zombie
```

```
              T  Stopped (on a signal) or (before Linux 2.6.33) trace stopped

              t  Tracing stop (Linux 2.6.33 onward)

              W  Paging (only before Linux 2.6.0)

              X  Dead (from Linux 2.6.0 onward)

              x  Dead (Linux 2.6.33 to 3.13 only)

              K  Wakekill (Linux 2.6.33 to 3.13 only)

              W  Waking (Linux 2.6.33 to 3.13 only)

              P  Parked (Linux 3.9 to 3.13 only)

(4) ppid  %d
              The PID of the parent of this process.

(5) pgrp  %d
              The process group ID of the process.

(6) session  %d
              The session ID of the process.

(7) tty_nr  %d
              The controlling terminal of the process.  (The minor device number is contained in the combi-
              nation of bits 31 to 20 and 7 to 0; the major device number is in bits 15 to 8.)

(8) tpgid  %d
              The ID of the foreground process group of the controlling terminal of the process.

(9) flags  %u
              The  kernel  flags  word of the process.  For bit meanings, see the PF_* defines in the Linux
              kernel source file include/linux/sched.h.  Details depend on the kernel version.

              The format for this field was %lu before Linux 2.6.

(10) minflt  %lu
              The number of minor faults the process has made which have not required loading a memory page
              from disk.

(11) cminflt  %lu
              The number of minor faults that the process's waited-for children have made.

(12) majflt  %lu
              The  number  of  major  faults the process has made which have required loading a memory page
              from disk.

(13) cmajflt  %lu
              The number of major faults that the process's waited-for children have made.

(14) utime  %lu
              Amount of time that this process has been scheduled in user mode,  measured  in  clock  ticks
              (divide by sysconf(_SC_CLK_TCK)).  This includes guest time, guest_time (time spent running a
              virtual CPU, see below), so that applications that are not aware of the guest time  field  do
              not lose that time from their calculations.

(15) stime  %lu
              Amount  of  time that this process has been scheduled in kernel mode, measured in clock ticks
              (divide by sysconf(_SC_CLK_TCK)).

(16) cutime  %ld
              Amount of time that this process's waited-for children have been scheduled in user mode, mea-
              sured  in  clock ticks (divide by sysconf(_SC_CLK_TCK)).  (See also times(2).)  This includes
              guest time, cguest_time (time spent running a virtual CPU, see below).

(17) cstime  %ld
              Amount of time that this process's waited-for children have been scheduled  in  kernel  mode,
              measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(18) priority  %ld
              (Explanation  for  Linux  2.6)  For  processes  running a real-time scheduling policy (policy
```

```
              below; see sched_setscheduler(2)), this is the negated scheduling priority, minus  one;  that
              is,  a  number  in  the range -2 to -100, corresponding to real-time priorities 1 to 99.  For
              processes running under a non-real-time scheduling policy, this is the raw nice  value  (set-
              priority(2))  as  represented in the kernel.  The kernel stores nice values as numbers in the
              range 0 (high) to 39 (low), corresponding to the user-visible nice range of -20 to 19.

      (19) nice  %ld
              The nice value (see setpriority(2)), a value in the range 19 (low priority) to -20 (high pri-
              ority).

      (20) num_threads  %ld
              Number of threads in this process (since Linux 2.6).  Before kernel 2.6, this field was  hard
              coded to 0 as a placeholder for an earlier removed field.

      (21) itrealvalue  %ld
              The  time in jiffies before the next SIGALRM is sent to the process due to an interval timer.
              Since kernel 2.6.17, this field is no longer maintained, and is hard coded as 0.

      (22) starttime  %llu
              The time the process started after system boot.  In kernels before Linux 2.6, this value  was
              expressed  in  jiffies.   Since  Linux  2.6, the value is expressed in clock ticks (divide by
              sysconf(_SC_CLK_TCK)).

              The format for this field was %lu before Linux 2.6.

      (23) vsize  %lu
              Virtual memory size in bytes.

      (24) rss  %ld
              Resident Set Size: number of pages the process has in real memory.  This is  just  the  pages
              which  count  toward  text, data, or stack space.  This does not include pages which have not
              been demand-loaded in, or which are swapped out.

      (25) rsslim  %lu
              Current soft limit in bytes on the rss of the process; see the description of  RLIMIT_RSS  in
              getrlimit(2).

      (26) startcode  %lu  [PT]
              The address above which program text can run.

      (27) endcode  %lu  [PT]
              The address below which program text can run.

      (28) startstack  %lu  [PT]
              The address of the start (i.e., bottom) of the stack.

      (29) kstkesp  %lu  [PT]
              The current value of ESP (stack pointer), as found in the kernel stack page for the process.

      (30) kstkeip  %lu  [PT]
              The current EIP (instruction pointer).

      (31) signal  %lu
              The  bitmap of pending signals, displayed as a decimal number.  Obsolete, because it does not
              provide information on real-time signals; use /proc/[pid]/status instead.

      (32) blocked  %lu
              The bitmap of blocked signals, displayed as a decimal number.  Obsolete, because it does  not
              provide information on real-time signals; use /proc/[pid]/status instead.

      (33) sigignore  %lu
              The  bitmap of ignored signals, displayed as a decimal number.  Obsolete, because it does not
              provide information on real-time signals; use /proc/[pid]/status instead.

      (34) sigcatch  %lu
              The bitmap of caught signals, displayed as a decimal number.  Obsolete, because it  does  not
              provide information on real-time signals; use /proc/[pid]/status instead.

      (35) wchan  %lu  [PT]
              This  is  the  "channel" in which the process is waiting.  It is the address of a location in
              the kernel where the process is sleeping.  The corresponding symbolic name can  be  found  in
              /proc/[pid]/wchan.
```

```
(36) nswap  %lu
             Number of pages swapped (not maintained).

(37) cnswap  %lu
             Cumulative nswap for child processes (not maintained).

(38) exit_signal  %d  (since Linux 2.1.22)
             Signal to be sent to parent when we die.

(39) processor  %d  (since Linux 2.2.8)
             CPU number last executed on.

(40) rt_priority  %u  (since Linux 2.5.19)
             Real-time  scheduling priority, a number in the range 1 to 99 for processes scheduled under a
             real-time policy, or 0, for non-real-time processes (see sched_setscheduler(2)).

(41) policy  %u  (since Linux 2.5.19)
             Scheduling policy  (see  sched_setscheduler(2)).  Decode  using  the  SCHED_*  constants  in
             linux/sched.h.

             The format for this field was %lu before Linux 2.6.22.

(42) delayacct_blkio_ticks  %llu  (since Linux 2.6.18)
             Aggregated block I/O delays, measured in clock ticks (centiseconds).

(43) guest_time  %lu  (since Linux 2.6.24)
             Guest  time  of  the process (time spent running a virtual CPU for a guest operating system),
             measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(44) cguest_time  %ld  (since Linux 2.6.24)
             Guest  time  of  the  process's  children,  measured  in  clock  ticks  (divide  by
             sysconf(_SC_CLK_TCK)).

(45) start_data  %lu  (since Linux 3.3)  [PT]
             Address above which program initialized and uninitialized (BSS) data are placed.

(46) end_data  %lu  (since Linux 3.3)  [PT]
             Address below which program initialized and uninitialized (BSS) data are placed.

(47) start_brk  %lu  (since Linux 3.3)  [PT]
             Address above which program heap can be expanded with brk(2).

(48) arg_start  %lu  (since Linux 3.5)  [PT]
             Address above which program command-line arguments (argv) are placed.

(49) arg_end  %lu  (since Linux 3.5)  [PT]
             Address below program command-line arguments (argv) are placed.

(50) env_start  %lu  (since Linux 3.5)  [PT]
             Address above which program environment is placed.

(51) env_end  %lu  (since Linux 3.5)  [PT]
             Address below which program environment is placed.

(52) exit_code  %d  (since Linux 3.5)  [PT]
             The thread's exit status in the form reported by waitpid(2).
```

```
# ifdef ENABLE_OBSERVABLES
readProcStats :: ProcessID → IO [Counter]
readProcStats pid = do
    ps0 ← readProcList (pathProcStat pid)
    let ps = zip colnames ps0
      psUseful = filter ((/= "unused") ∘ fst) ps
    return $ map (λ(n,i) → Counter StatInfo n (PureI i)) psUseful
  where
    colnames :: [Text]
    colnames = ["pid","unused","unused","ppid","pgrp","session","ttynr","tpgid","flags","mi
      ,"cminflt","majflt","cmajflt","utime","stime","cutime","cstime","priority","nice","n
```

```
      ,"itrealvalue","starttime","vsize","rss","rsslim","startcode","endcode","startstack
      ,"signal","blocked","sigignore","sigcatch","wchan","nswap","cnswap","exitsignal","p
      ,"policy","blkio","guesttime","cguesttime","startdata","enddata","startbrk","argsta
      ,"envend","exitcode"
      ]
```

 # endif

readProcIO - //proc//<pid >//io

```
/proc/[pid]/io (since kernel 2.6.20)
       This file contains I/O statistics for the process, for example:

              # cat /proc/3828/io
              rchar: 323934931
              wchar: 323929600
              syscr: 632687
              syscw: 632675
              read_bytes: 0
              write_bytes: 323932160
              cancelled_write_bytes: 0

       The fields are as follows:

       rchar: characters read
              The number of bytes which this task has caused to be read from storage.  This is simply the  sum
              of bytes which this process passed to read(2) and similar system calls.  It includes things such
              as terminal I/O and is unaffected by whether or not actual physical disk I/O was  required  (the
              read might have been satisfied from pagecache).

       wchar: characters written
              The  number  of bytes which this task has caused, or shall cause to be written to disk.  Similar
              caveats apply here as with rchar.

       syscr: read syscalls
              Attempt to count the number of read I/O operations-that is, system calls  such  as  read(2)  and
              pread(2).

       syscw: write syscalls
              Attempt  to  count the number of write I/O operations-that is, system calls such as write(2) and
              pwrite(2).

       read_bytes: bytes read
              Attempt to count the number of bytes which this process really did cause to be fetched from  the
              storage layer.  This is accurate for block-backed filesystems.

       write_bytes: bytes written
              Attempt to count the number of bytes which this process caused to be sent to the storage layer.

       cancelled_write_bytes:
              The  big  inaccuracy  here  is truncate.  If a process writes 1MB to a file and then deletes the
              file, it will in fact perform no writeout.  But it will have been accounted as having caused 1MB
              of  write.   In other words: this field represents the number of bytes which this process caused
              to not happen, by truncating pagecache.  A task can cause "negative"  I/O  too.   If  this  task
              truncates  some  dirty  pagecache,  some  I/O  which another task has been accounted for (in its
              write\_bytes) will not be happening.

       Note: In the current implementation, things are a bit racy  on  32-bit  systems:  if  process  A  reads
       process  B's  /proc/[pid]/io  while process B is updating one of these 64-bit counters, process A could
       see an intermediate result.

       Permission to access this file is governed by a ptrace access mode PTRACE\_MODE\_READ\_FSCREDS check;  see
       ptrace(2).
```

 # ifdef ENABLE_OBSERVABLES
readProcIO :: *ProcessID* → *IO* [*Counter*]
readProcIO pid = **do**

```
    ps0 ← readProcList (pathProcIO pid)
    let ps = zip3 colnames ps0 units
      ps2 = filter (λ(n, _i, _u) → "ign" ≢ n) ps
    return $ map (λ(n,i,u) → Counter IOCounter n (u i)) ps2
  where
    colnames :: [Text]
    colnames = ["ign","rchar","ign","wchar","ign","syscr","ign","syscw","ign","rbytes","ign
    units = [PureI,Bytes ∘ fromInteger,PureI,Bytes ∘ fromInteger,PureI,PureI,PureI,PureI,PureI,Bytes ∘ fro
# endif
```

Network TCP/IP counters

```
example:
\\
cat /proc/<pid>/net/netstat
\\
TcpExt: SyncookiesSent SyncookiesRecv SyncookiesFailed EmbryonicRsts PruneCalled RcvPruned OfoPruned OutOfWindowIcmps Lo!
!ckDroppedIcmps ArpFilter TW TWRecycled TWKilled PAWSActive PAWSEstab DelayedACKs DelayedACKLocked DelayedACKLost ListenO!
!verflows ListenDrops TCPHPHits TCPPureAcks TCPHPAcks TCPRenoRecovery TCPSackRecovery TCPSACKReneging TCPSACKReorder TCPR!
!enoReorder TCPTSReorder TCPFullUndo TCPPartialUndo TCPDSACKUndo TCPLossUndo TCPLostRetransmit TCPRenoFailures TCPSackFai!
!lures TCPLossFailures TCPFastRetrans TCPSlowStartRetrans TCPTimeouts TCPLossProbes TCPLossProbeRecovery TCPRenoRecoveryF!
!ail TCPSackRecoveryFail TCPRcvCollapsed TCPDSACKOldSent TCPDSACKOfoSent TCPDSACKRecv TCPDSACKOfoRecv TCPAbortOnData TCPA!
!bortOnClose TCPAbortOnMemory TCPAbortOnTimeout TCPAbortOnLinger TCPAbortFailed TCPMemoryPressures TCPMemoryPressuresChro!
!no TCPSACKDiscard TCPDSACKIgnoredOld TCPDSACKIgnoredNoUndo TCPSpuriousRTOs TCPMD5NotFound TCPMD5Unexpected TCPMD5Failure!
! TCPSackShifted TCPSackMerged TCPSackShiftFallback TCPBacklogDrop PFMemallocDrop TCPMinTTLDrop TCPDeferAcceptDrop IPReve!
!rsePathFilter TCPTimeWaitOverflow TCPReqQFullDoCookies TCPReqQFullDrop TCPRetransFail TCPRcvCoalesce TCPOFOQueue TCPOFOD!
!rop TCPOFOMerge TCPChallengeACK TCPSYNChallenge TCPFastOpenActive TCPFastOpenActiveFailTCPFastOpenPassive TCPFastOpenPas!
!siveFail TCPFastOpenListenOverflow TCPFastOpenCookieReqd TCPFastOpenBlackhole TCPSpuriousRtxHostQueues BusyPollRxPackets!
! TCPAutoCorking TCPFromZeroWindowAdv TCPToZeroWindowAdv TCPWantZeroWindowAdv TCPSynRetrans TCPOrigDataSent TCPHystartTra!
!inDetect TCPHystartTrainCwnd TCPHystartDelayDetect TCPHystartDelayCwnd TCPACKSkippedSynRecv TCPACKSkippedPAWS TCPACKSkip!
!pedSeq TCPACKSkippedFinWait2 TCPACKSkippedTimeWait TCPACKSkippedChallenge TCPWinProbe TCPKeepAlive TCPMTUPFail TCPMTUPSu!
!ccess TCPDelivered TCPDeliveredCE TCPAckCompressed
TcpExt: 0 0 0 0 28 0 0 0 0 0 1670 1 0 0 6 6029 1 1766 0 0 384612 66799 105553 0 21 0 638 0 1 7 1 1 32 128 0 1 0 22 0 116!
! 383 19 0 0 0 1788 224 178 0 435 224 0 13 0 0 0 0 0 0 67 0 0 0 0 3 1 668 0 0 0 4 0 0 0 0 0 91870 4468 0 224 22 23 0 0 0 !
!0 0 0 0 6 0 21492 0 0 11 188 188680 6 145 13 425 0 3 4 0 0 1 117 22984 0 0 192495 0 4500
IpExt: InNoRoutes InTruncatedPkts InMcastPkts OutMcastPkts InBcastPkts OutBcastPkts InOctets OutOctets InMcastOctets Out!
!McastOctets InBcastOctets OutBcastOctets InCsumErrors InNoECTPkts InECT1Pkts InECT0Pkts InCEPkts
IpExt: 0 0 20053 8977 2437 23 3163525943 196480057 2426648 1491754 394285 5523 0 3513269 0 217426 0
```

```
# ifdef ENABLE_OBSERVABLES
readProcNet :: ProcessID → IO [Counter]
readProcNet pid = do
    ipexts0 ← words <$> lastline <$> lines <$> readFile (pathProcNet pid)
    let ipexts1 = map (λi → readMaybe i :: Maybe Integer) ipexts0
    return $
      if length ipexts1 ⩾ 9-- enough fields available
      then mkCounters [("IpExt:InOctets",ipexts1 !! 7),("IpExt:OutOctets",ipexts1 !! 8)]
      else []
  where
    lastline ls | length ls ≡ 4 = last ls-- ensures we read the fourth line
      | otherwise = []
    mkCounters = catMaybes ∘ map (λ(n,c) → mkCounter n c)
    mkCounter _n Nothing = Nothing
    mkCounter n (Just i) = Just (Counter NetCounter (pack n) (Bytes $ fromInteger i))
# endif
```

1.7.12 Cardano.BM.Data.Aggregated

Measurable

A Measurable may consist of different types of values. Time measurements are strict, so are *Bytes* which are externally measured. The real or integral numeric values are lazily linked, so we can decide later to drop them.

```
data Measurable = Microseconds  {-# UNPACK #-}  !Word64
  | Nanoseconds  {-# UNPACK #-}  !Word64
  | Seconds      {-# UNPACK #-}  !Word64
  | Bytes        {-# UNPACK #-}  !Word64
  | PureD        !Double
  | PureI        !Integer
  | Severity     S.Severity
  deriving (Eq, Read, Generic, ToJSON, FromJSON)
```

Measurable can be transformed to an integral value.

```
instance Ord Measurable where
  compare (Seconds a) (Seconds b)                = compare a b
  compare (Microseconds a) (Microseconds b) = compare a b
  compare (Nanoseconds a) (Nanoseconds b)   = compare a b
  compare (Seconds a) (Microseconds b)       = compare (a * 1000 * 1000) b
  compare (Nanoseconds a) (Microseconds b)  = compare a (b * 1000)
  compare (Seconds a) (Nanoseconds b)        = compare (a * 1000 * 1000 * 1000) b
  compare (Microseconds a) (Nanoseconds b)  = compare (a * 1000) b
  compare (Microseconds a) (Seconds b)       = compare a (b * 1000 * 1000)
  compare (Nanoseconds a) (Seconds b)        = compare a (b * 1000 * 1000 * 1000)
  compare (Bytes a) (Bytes b)                = compare a b
  compare (PureD a) (PureD b)                = compare a b
  compare (PureI a) (PureI b)                = compare a b
  compare (Severity a) (Severity b)          = compare a b
  compare (PureI a) (Seconds b)       | a >= 0 = compare a (toInteger b)
  compare (PureI a) (Microseconds b) | a >= 0 = compare a (toInteger b)
  compare (PureI a) (Nanoseconds b)  | a >= 0 = compare a (toInteger b)
  compare (PureI a) (Bytes b)        | a >= 0 = compare a (toInteger b)
  compare (Seconds a)      (PureI b) | b >= 0 = compare (toInteger a) b
  compare (Microseconds a) (PureI b) | b >= 0 = compare (toInteger a) b
  compare (Nanoseconds a) (PureI b)  | b >= 0 = compare (toInteger a) b
  compare (Bytes a)        (PureI b) | b >= 0 = compare (toInteger a) b
  compare a@(PureD _) (PureI b)              = compare (getInteger a) b
  compare (PureI a) b@(PureD _)              = compare a (getInteger b)
  compare _a _b                              = LT
```

Measurable can be transformed to an integral value.

```
getInteger :: Measurable → Integer
getInteger (Microseconds a) = toInteger a
getInteger (Nanoseconds a) = toInteger a
getInteger (Seconds a)      = toInteger a
getInteger (Bytes a)        = toInteger a
getInteger (PureI a)        = a
```

```
getInteger (PureD a)      = round a
getInteger (Severity a)   = toInteger (fromEnum a)
```

Measurable can be transformed to a rational value.

```
getDouble :: Measurable → Double
getDouble (Microseconds a) = fromIntegral a
getDouble (Nanoseconds a) = fromIntegral a
getDouble (Seconds a)     = fromIntegral a
getDouble (Bytes a)       = fromIntegral a
getDouble (PureI a)       = fromInteger a
getDouble (PureD a)       = a
getDouble (Severity a)    = fromIntegral (fromEnum a)
```

It is a numerical value, thus supports functions to operate on numbers.

```
instance Num Measurable where
  (+) (Microseconds a) (Microseconds b) = Microseconds (a + b)
  (+) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a + b)
  (+) (Seconds a)     (Seconds b)     = Seconds     (a + b)
  (+) (Bytes a)       (Bytes b)       = Bytes       (a + b)
  (+) (PureI a)       (PureI b)       = PureI       (a + b)
  (+) (PureD a)       (PureD b)       = PureD       (a + b)
  (+) a               _               = a

  (*) (Microseconds a) (Microseconds b) = Microseconds (a * b)
  (*) (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a * b)
  (*) (Seconds a)     (Seconds b)     = Seconds     (a * b)
  (*) (Bytes a)       (Bytes b)       = Bytes       (a * b)
  (*) (PureI a)       (PureI b)       = PureI       (a * b)
  (*) (PureD a)       (PureD b)       = PureD       (a * b)
  (*) a               _               = a

  abs (Microseconds a) = Microseconds (abs a)
  abs (Nanoseconds a) = Nanoseconds (abs a)
  abs (Seconds a)     = Seconds     (abs a)
  abs (Bytes a)       = Bytes       (abs a)
  abs (PureI a)       = PureI       (abs a)
  abs (PureD a)       = PureD       (abs a)
  abs a               = a

  signum (Microseconds a) = Microseconds (signum a)
  signum (Nanoseconds a) = Nanoseconds (signum a)
  signum (Seconds a)     = Seconds     (signum a)
  signum (Bytes a)       = Bytes       (signum a)
  signum (PureI a)       = PureI       (signum a)
  signum (PureD a)       = PureD       (signum a)
  signum a               = a

  negate (Microseconds a) = Microseconds (negate a)
  negate (Nanoseconds a) = Nanoseconds (negate a)
  negate (Seconds a)     = Seconds     (negate a)
  negate (Bytes a)       = Bytes       (negate a)
  negate (PureI a)       = PureI       (negate a)
  negate (PureD a)       = PureD       (negate a)
  negate a               = a
```

```
  fromInteger = PureI
subtractMeasurable :: Measurable → Measurable → Measurable
subtractMeasurable (Microseconds a) (Microseconds b) = Microseconds (a – b)
subtractMeasurable (Nanoseconds a) (Nanoseconds b) = Nanoseconds (a – b)
subtractMeasurable (Seconds a)     (Seconds b)     = Seconds     (a – b)
subtractMeasurable (Bytes a)       (Bytes b)       = Bytes       (a – b)
subtractMeasurable (PureI a)       (PureI b)       = PureI       (a – b)
subtractMeasurable (PureD a)       (PureD b)       = PureD       (a – b)
subtractMeasurable a               _               = a
```

Pretty printing of Measurable.

```
instance Show Measurable where
  show v@(Microseconds a) = show a ++ showUnits v
  show v@(Nanoseconds a) = show a ++ showUnits v
  show v@(Seconds a)     = show a ++ showUnits v
  show v@(Bytes a)       = show a ++ showUnits v
  show v@(PureI a)       = show a ++ showUnits v
  show v@(PureD a)       = show a ++ showUnits v
  show v@(Severity a)    = show a ++ showUnits v
showUnits :: Measurable → String
showUnits (Microseconds _) = " µs"
showUnits (Nanoseconds _) = " ns"
showUnits (Seconds _)     = " s"
showUnits (Bytes _)       = " B"
showUnits (PureI _)       = ""
showUnits (PureD _)       = ""
showUnits (Severity _)    = ""
-- show in S.I. units
showSI :: Measurable → String
showSI (Microseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000 :: Float) / (1000 :: Float))) ++
                          showUnits (Seconds a)
showSI (Nanoseconds a) = show (fromFloatDigits ((fromIntegral a) / (1000 :: Float) / (1000 :: Float) / (1000 :: F
                          showUnits (Seconds a)
showSI v@(Seconds a)   = show a ++ showUnits v
showSI v@(Bytes a)     = show a ++ showUnits v
showSI v@(PureI a)     = show a ++ showUnits v
showSI v@(PureD a)     = show a ++ showUnits v
showSI v@(Severity a)  = show a ++ showUnits v
```

Stats

A Stats statistics is strictly computed.

```
data BaseStats = BaseStats {
  fmin :: !Measurable,
  fmax :: !Measurable,
  fcount :: {-# UNPACK #-} !Word64,
  fsum_A :: {-# UNPACK #-} !Double,
  fsum_B :: {-# UNPACK #-} !Double
  } deriving (Show, Generic, ToJSON, FromJSON)
```

```
instance Eq BaseStats where
  (BaseStats mina maxa counta sumAa sumBa) ≡ (BaseStats minb maxb countb sumAb sumBb) =
    mina ≡ minb ∧ maxa ≡ maxb ∧ counta ≡ countb ∧
    abs (sumAa − sumAb) < 1.0e−4 ∧
    abs (sumBa − sumBb) < 1.0e−4
data Stats = Stats {
  flast  :: !Measurable,
  fold   :: !Measurable,
  fbasic :: !BaseStats,
  fdelta :: !BaseStats,
  ftimed :: !BaseStats
  } deriving (Show, Eq, Generic, ToJSON, FromJSON)


meanOfStats :: BaseStats → Double
meanOfStats = fsum_A


stdevOfStats :: BaseStats → Double
stdevOfStats s =
    calculate (fcount s)
  where
    calculate :: Word64 → Double
    calculate n =
      if n ⩾ 2
      then sqrt $ (fsum_B s) / (fromInteger $ fromIntegral (n − 1))
      else 0
```

instance *Semigroup* Stats disabled for the moment, because not needed.

We use a parallel algorithm to update the estimation of mean and variance from two sample statistics. (see https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Parallel_a

```
instance Semigroup Stats where
  (<>) a b = let counta = fcount a
    countb = fcount b
    newcount = counta + countb
    delta = fsum_A b − fsum_A a
    in
    Stats {flast = flast b-- right associative
      ,fmin    = min (fmin a) (fmin b)
      ,fmax    = max (fmax a) (fmax b)
      ,fcount = newcount
      ,fsum_A = fsum_A a + (delta / fromInteger newcount)
      ,fsum_B = fsum_B a + fsum_B b + (delta * delta) * (fromInteger (counta * countb) / fromInteger newcount
      }


stats2Text :: Stats → Text
stats2Text (Stats slast _ sbasic sdelta stimed) =
    pack $
      "{ last=" ++ show slast ++
      ", basic-stats=" ++ showStats' (sbasic) ++
```

```
        ", delta-stats=" ++ showStats' (sdelta) ++
        ", timed-stats=" ++ showStats' (stimed) ++
        " }"
    where
      showStats' :: BaseStats → String
      showStats' s =
        ", { min=" ++ show (fmin s) ++
        ", max=" ++ show (fmax s) ++
        ", mean=" ++ show (meanOfStats s) ++ showUnits (fmin s) ++
        ", std-dev=" ++ show (stdevOfStats s) ++
        ", count=" ++ show (fcount s) ++
        " }"
```

Exponentially Weighted Moving Average (EWMA)

Following https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average we calculate the exponential moving average for a series of values Y_t according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1 - \alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

```
data EWMA = EmptyEWMA {alpha :: !Double}
  | EWMA {alpha :: !Double
    ,avg :: !Measurable
    } deriving (Show, Eq, Generic, ToJSON, FromJSON)
```

Aggregated

```
data Aggregated = AggregatedStats !Stats
  | AggregatedEWMA !EWMA
  deriving (Eq, Generic, ToJSON, FromJSON)
```

instance *Semigroup* Aggregated disabled for the moment, because not needed.

```
instance Semigroup Aggregated where
  (<>) (AggregatedStats a) (AggregatedStats b) =
    AggregatedStats (a <> b)
  (<>) a _ = a


singletonStats :: Measurable → Aggregated
singletonStats a =
  let stats = Stats {flast = a
    ,fold                  = Nanoseconds 0
    ,fbasic = BaseStats
      {fmin = a
      ,fmax = a
      ,fcount = 1
      ,fsum_A = getDouble a
```

```
      ,fsum_B = 0}
    ,fdelta = BaseStats
      {fmin = 0
      ,fmax = 0
      ,fcount = 1
      ,fsum_A = 0
      ,fsum_B = 0}
    ,ftimed = BaseStats
      {fmin = Nanoseconds 0
      ,fmax = Nanoseconds 0
      ,fcount = 1
      ,fsum_A = 0
      ,fsum_B = 0}
    }
  in
  AggregatedStats stats


instance Show Aggregated where
  show (AggregatedStats astats) =
    "{ stats = " ++ show astats ++ " }"
  show (AggregatedEWMA a) = show a
```

Update aggregation

We distinguish an unitialized from an already initialized aggregation. The latter is properly initialized.
We use Welford's online algorithm to update the estimation of mean and variance of the sample statistics. (see https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance#Welford's_

```
updateAggregation :: Measurable → Aggregated → Word64 → Either Text Aggregated
updateAggregation v (AggregatedStats s) tstamp =
    Right $ AggregatedStats $! Stats {flast = v
      ,fold = mkTimestamp
      ,fbasic = updateBaseStats 1 v (fbasic s)
      ,fdelta = updateBaseStats 2 deltav (fdelta s)
      ,ftimed = updateBaseStats 2 timediff (ftimed s)
      }
  where
    deltav = subtractMeasurable v (flast s)
    mkTimestamp = Nanoseconds $ tstamp
    timediff = Nanoseconds $ fromInteger $ (getInteger mkTimestamp) - (getInteger $ fold s)
updateAggregation v (AggregatedEWMA e) _ =
    let !eitherAvg = ewma e v
    in
      AggregatedEWMA <$> eitherAvg
updateBaseStats :: Word64 → Measurable → BaseStats → BaseStats
updateBaseStats startAt v s =
    let newcount = fcount s + 1 in
    if (startAt > newcount)
    then s {fcount = fcount s + 1}
```

```
    else
      let newcountRel = newcount - startAt + 1
          newvalue = getDouble v
          delta = newvalue - fsum_A s
          dincr = (delta / fromIntegral newcountRel)
          delta2 = newvalue - fsum_A s - dincr
          (minim, maxim) =
            if startAt ≡ newcount
            then (v, v)
            else (min v (fmin s), max v (fmax s))
      in
      BaseStats {fmin = minim
          ,fmax           = maxim
          ,fcount = newcount
          ,fsum_A = fsum_A s + dincr
          ,fsum_B = fsum_B s + (delta * delta2)
          }
```

Calculation of EWMA

Following https://en.wikipedia.org/wiki/Moving_average#Exponential_moving_average we calculate the exponential moving average for a series of values Y_t according to:

$$S_t = \begin{cases} Y_1, & t = 1 \\ \alpha \cdot Y_t + (1-\alpha) \cdot S_{t-1}, & t > 1 \end{cases}$$

The pattern matching below ensures that the EWMA will start with the first value passed in, and will not change type, once determined.

```
ewma :: EWMA → Measurable → Either Text EWMA
ewma (EmptyEWMA a) v = Right $ EWMA a v
ewma (EWMA a s@(Microseconds _)) y@(Microseconds _) =
  Right $ EWMA a $ Microseconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Seconds _)) y@(Seconds _) =
  Right $ EWMA a $ Seconds $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a s@(Bytes _)) y@(Bytes _) =
  Right $ EWMA a $ Bytes $ round $ a * (getDouble y) + (1 - a) * (getDouble s)
ewma (EWMA a (PureI s)) (PureI y) =
  Right $ EWMA a $ PureI $ round $ a * (fromInteger y) + (1 - a) * (fromInteger s)
ewma (EWMA a (PureD s)) (PureD y) =
  Right $ EWMA a $ PureD $ a * y + (1 - a) * s
ewma _ _ = Left "EWMA: Cannot compute average on values of different types"
```

1.7.13 Cardano.BM.Data.AggregatedKind

AggregatedKind

This identifies the type of Aggregated.

```
data AggregatedKind = StatsAK
  | EwmaAK {alpha :: !Double}
    deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)
```

1.7.14 Cardano.BM.Data.Backend

BackendId

A backend is identified by BackendKind *x Name*

```
type BackendId = Text
```

Accepts a LogObject

Instances of this type class accept a LogObject and deal with it.

```
class IsEffectuator t a where
  effectuate :: t a → LogObject a → IO ()
  effectuatefrom :: forall s ∘ (IsEffectuator s a) ⇒ t a → LogObject a → s a → IO ()
  default effectuatefrom :: forall s ∘ (IsEffectuator s a) ⇒ t a → LogObject a → s a → IO ()
  effectuatefrom t nli _ = effectuate t nli
  handleOverflow :: t a → IO ()
```

Declaration of a Backend

A backend is life-cycle managed, thus can be *realize*d and *unrealize*d.

```
class (IsEffectuator t a
  , FromJSON a
  , Exception (BackendFailure t)
  ) ⇒ IsBackend t a where
  type BackendFailure t :: *
  type BackendFailure t = GenericBackendFailure
  bekind     :: t a → BackendKind
  realize    :: Configuration → IO (t a)
  realizefrom :: forall s ∘ (IsEffectuator s a) ⇒ Configuration → Trace IO a → s a → IO (t a)
  default realizefrom :: forall s ∘ (IsEffectuator s a) ⇒ Configuration → Trace IO a → s a → IO (t a)
  realizefrom cfg _ _ = realize cfg
  unrealize :: t a → IO ()
```

Backend

This data structure for a backend defines its behaviour as an IsEffectuator when processing an incoming message, and as an IsBackend for unrealizing the backend.

```
data Backend a = MkBackend
  { bEffectuate :: LogObject a → IO ()
  , bUnrealize :: IO ()
  }
```

GenericBackendFailure

A default type for backend-specific failures, when they wouldn't care to define their own.

```
newtype GenericBackendFailure =
  GenericBackendFailure {unGenericBackendFailure :: String}
```

```
instance Exception GenericBackendFailure
instance Show GenericBackendFailure where
  show x = "Generic backend failure: " <> unGenericBackendFailure x
```

1.7.15 Cardano.BM.Data.BackendKind

BackendKind

This identifies the backends that can be attached to the Switchboard.

```
data BackendKind =
  AggregationBK
  | EditorBK
  | EKGViewBK
  | GraylogBK
  | KatipBK
  | LogBufferBK
  | MonitoringBK
  | TraceAcceptorBK
  | TraceForwarderBK
  | UserDefinedBK Text
  | SwitchboardBK
  deriving (Eq, Ord, Show, Read)
instance ToJSON BackendKind where
  toJSON AggregationBK        = String "AggregationBK"
  toJSON EditorBK             = String "EditorBK"
  toJSON EKGViewBK            = String "EKGViewBK"
  toJSON GraylogBK            = String "GraylogBK"
  toJSON KatipBK              = String "KatipBK"
  toJSON LogBufferBK          = String "LogBufferBK"
  toJSON MonitoringBK         = String "MonitoringBK"
  toJSON TraceForwarderBK     = String "TraceForwarderBK"
  toJSON TraceAcceptorBK      = String "TraceAcceptorBK"
  toJSON (UserDefinedBK name) = object [ "kind" .= String "UserDefinedBK"
                                       , "name" .= toJSON name
                                       ]
  toJSON SwitchboardBK        = String "SwitchboardBK"
instance FromJSON BackendKind where
  parseJSON v = withObject
        "BackendKind"
        (λvalue → do
          c ← value .: "kind" :: Parser Text
          case c of
            "UserDefinedBK" →
              UserDefinedBK <$> value .: "name"
            _               → fail "not expected kind"
        )
        v
      <|> withText
        "BackendKind"
        (λcase
```

```
      "AggregationBK"      → pure AggregationBK
      "EditorBK"           → pure EditorBK
      "EKGViewBK"          → pure EKGViewBK
      "GraylogBK"          → pure GraylogBK
      "KatipBK"            → pure KatipBK
      "LogBufferBK"        → pure LogBufferBK
      "MonitoringBK"       → pure MonitoringBK
      "TraceAcceptorBK"    → pure TraceAcceptorBK
      "TraceForwarderBK" → pure TraceForwarderBK
      "SwitchboardBK"      → pure SwitchboardBK
      _                    → fail "not expected BackendKind"
    )
    v
```

1.7.16 Cardano.BM.Data.Configuration

Data structure to help parsing configuration files.

Representation

```
type Port = Int
type HostPort = (String, Port)
data Representation = Representation
  { minSeverity     :: Severity
  , rotation        :: Maybe RotationParameters
  , setupScribes    :: [ScribeDefinition]
  , defaultScribes  :: [(ScribeKind, Text)]
  , setupBackends   :: [BackendKind]
  , defaultBackends :: [BackendKind]
  , hasEKG          :: Maybe Port
  , hasGraylog      :: Maybe Port
  , hasPrometheus   :: Maybe HostPort
  , hasGUI          :: Maybe Port
  , traceForwardTo  :: Maybe RemoteAddr
  , traceAcceptAt   :: Maybe [RemoteAddrNamed]
  , options         :: HM.HashMap Text Value
  }
  deriving (Generic, Show, ToJSON, FromJSON)
data RemoteAddr
  = RemotePipe FilePath
  | RemoteSocket String String
  deriving (Generic, Eq, Show, ToJSON, FromJSON)
data RemoteAddrNamed = RemoteAddrNamed
  { nodeName :: Text
  , remoteAddr :: RemoteAddr
  } deriving (Generic, Eq, Show, ToJSON, FromJSON)
```

readRepresentation

```
readRepresentation :: FilePath → IO Representation
readRepresentation fp =
  either throwIO pure =≪ parseRepresentation < $ > BS.readFile fp
```

parseRepresentation

```
parseRepresentation :: ByteString → Either ParseException Representation
parseRepresentation =
  fmap implicit_fill_representation ∘ decodeEither'
```

after parsing the configuration representation we implicitly correct it.

```
implicit_fill_representation :: Representation → Representation
implicit_fill_representation =
    remove_ekgview_if_not_defined ∘
    filter_duplicates_from_backends ∘
    filter_duplicates_from_scribes ∘
    union_setup_and_usage_backends ∘
    add_ekgview_if_port_defined ∘
    add_katip_if_any_scribes
  where
    filter_duplicates_from_backends r =
      r {setupBackends = mkUniq $ setupBackends r}
    filter_duplicates_from_scribes r =
      r {setupScribes = mkUniq $ setupScribes r}
    union_setup_and_usage_backends r =
      r {setupBackends = setupBackends r <> defaultBackends r}
    remove_ekgview_if_not_defined r =
      case hasEKG r of
      Nothing → r {defaultBackends = filter (λbk → bk ≢ EKGViewBK) (defaultBackends r)
        , setupBackends = filter (λbk → bk ≢ EKGViewBK) (setupBackends r)
        }
      Just _ → r
    add_ekgview_if_port_defined r =
      case hasEKG r of
      Nothing → r
      Just _ → r {setupBackends = setupBackends r <> [EKGViewBK]}
    add_katip_if_any_scribes r =
      if (any ¬ [null $ setupScribes r, null $ defaultScribes r])
      then r {setupBackends = setupBackends r <> [KatipBK]}
      else r
    mkUniq :: Ord a ⇒ [a] → [a]
    mkUniq = Set.toList ∘ Set.fromList
```

1.7.17 Cardano.BM.Data.Counter

Counter

```
data Counter = Counter
        {cType  :: !CounterType
```

```
        , cName :: !Text
        , cValue :: !Measurable
        }
        deriving (Show, Eq, Generic, ToJSON, FromJSON)
data CounterType = MonotonicClockTime
  | MemoryCounter
  | SysInfo
  | StatInfo
  | IOCounter
  | NetCounter
  | RTSStats
    deriving (Eq, Show, Generic, ToJSON, FromJSON)
instance ToJSON Microsecond where
  toJSON = toJSON ∘ toMicroseconds
  toEncoding = toEncoding ∘ toMicroseconds
```

Names of counters

```
nameCounter :: Counter → Text
nameCounter (Counter MonotonicClockTime _ _) = "Clock"
nameCounter (Counter MemoryCounter _ _) = "Mem"
nameCounter (Counter SysInfo _ _) = "Sys"
nameCounter (Counter StatInfo _ _) = "Stat"
nameCounter (Counter IOCounter _ _) = "IO"
nameCounter (Counter NetCounter _ _) = "Net"
nameCounter (Counter RTSStats _ _) = "RTS"
```

CounterState

```
data CounterState = CounterState {
  csCounters :: [Counter]
  }
  deriving (Show, Eq, Generic, ToJSON, FromJSON)
```

Difference between counters

```
diffCounters :: [Counter] → [Counter] → [Counter]
diffCounters openings closings =
    getCountersDiff openings closings
  where
    getCountersDiff :: [Counter]
                    → [Counter]
                    → [Counter]
    getCountersDiff as bs =
      let
        getName counter = nameCounter counter <> cName counter
        asNames = map getName as
```

```
    aPairs = zip asNames as
    bsNames = map getName bs
    bs' = zip bsNames bs
    bPairs = HM.fromList bs'
  in
    catMaybes $ (flip map) aPairs $ λ(name, Counter _ _ startValue) →
      case HM.lookup name bPairs of
        Nothing      → Nothing
        Just counter → let endValue = cValue counter
                       in Just counter {cValue = endValue – startValue}
```

Platform information

```
data Platform = UnknownPlatform | Linux | Darwin | Windows
  deriving (Show, Eq, Ord, Enum)
newtype PlatformCode = PlatformCode {platform :: Platform}
instance Show PlatformCode where
  show (PlatformCode p) = show p
```

1.7.18 Cardano.BM.Data.LogItem

LoggerName

A LoggerName has currently type *Text*.

```
type LoggerName = Text
```

Logging of outcomes with LogObject

```
data LogObject a = LogObject
  {loName :: LoggerName
  ,loMeta  :: !LOMeta
  ,loContent :: !(LOContent a)
  } deriving (Show, Eq)
instance ToJSON a ⇒ ToJSON (LogObject a) where
  toJSON (LogObject _loname _lometa _locontent) =
    object ["loname" .= _loname
      ,"lometa"     .= _lometa
      ,"locontent" .= _locontent
      ]
instance (FromJSON a) ⇒ FromJSON (LogObject a) where
  parseJSON = withObject "LogObject" $ λv →
    LogObject <$> v .: "loname"
      <*> v .: "lometa"
      <*> v .: "locontent"
```

Meta data for a LogObject. Text was selected over ThreadId in order to be able to use the logging system under SimM of ouroboros-network because ThreadId from Control.Concurrent lacks a Read instance.

```
data LOMeta = LOMeta {
  tstamp   :: {-# UNPACK #-} !UTCTime
  ,tid     :: {-# UNPACK #-} !Text
  ,hostname :: {-# UNPACK #-} !Text
  ,severity :: !Severity
  ,privacy :: !PrivacyAnnotation
  }
instance ToJSON LOMeta where
    toJSON (LOMeta _tstamp _tid _hn _sev _priv) =
      object ["tstamp" .= _tstamp
        ,"tid"       .= _tid
        ,"hostname" .= _hn
        ,"severity" .= show _sev
        ,"privacy"   .= show _priv
        ]
instance FromJSON LOMeta where
    parseJSON = withObject "LOMeta" $ λv →
      LOMeta <$> v .: "tstamp"
          <*> v .: "tid"
          <*> v .: "hostname"
          <*> v .: "severity"
          <*> v .: "privacy"
instance Show LOMeta where
    show (LOMeta tstamp1 tid1 hn1 _sev1 _priv1) =
      "LOMeta@" ++ show tstamp1 ++ "  tid=" ++ show tid1 ++ if (¬ $ null $ show hn1) then "  on  " ++ show h
instance Eq LOMeta where
    (≡) (LOMeta tstamp1 tid1 hn1 sev1 priv1) (LOMeta tstamp2 tid2 hn2 sev2 priv2) =
      tstamp1 ≡ tstamp2 ∧ tid1 ≡ tid2 ∧ hn1 ≡ hn2 ∧ sev1 ≡ sev2 ∧ priv1 ≡ priv2
mkLOMeta :: MonadIO m ⇒ Severity → PrivacyAnnotation → m LOMeta
mkLOMeta sev priv =
    LOMeta <$> liftIO getCurrentTime
      <*> (cleantid <$> liftIO myThreadId)
      <*> pure ""
      <*> pure sev
      <*> pure priv
  where
    cleantid threadid = do
      let prefixText = "ThreadId "
        condStripPrefix s = fromMaybe s $ stripPrefix prefixText s
      condStripPrefix $ (pack ∘ show) threadid
```

Convert a timestamp to ns since epoch:

```
utc2ns :: UTCTime → Word64
utc2ns utctime = fromInteger ∘ round $ 1000 _000_000 * utcTimeToPOSIXSeconds utctime


data MonitorAction = MonitorAlert Text
  | MonitorAlterGlobalSeverity Severity
  | MonitorAlterSeverity LoggerName Severity
  deriving (Show, Eq)
instance ToJSON MonitorAction where
```

```
    toJSON (MonitorAlert m) =
      object ["kind" .= String "MonitorAlert"
        ,"message" .= toJSON m]
    toJSON (MonitorAlterGlobalSeverity s) =
      object ["kind" .= String "MonitorAlterGlobalSeverity"
        ,"severity" .= toJSON s]
    toJSON (MonitorAlterSeverity n s) =
      object ["kind" .= String "MonitorAlterSeverity"
        ,"name" .= toJSON n
        ,"severity" .= toJSON s]
  instance FromJSON MonitorAction where
    parseJSON = withObject "MonitorAction" $ λv →
      (v .: "kind" :: Parser Text)
      >>=
      λcase "MonitorAlert" →
                  MonitorAlert <$> v .: "message"
        "MonitorAlterGlobalSeverity" →
                  MonitorAlterGlobalSeverity <$> v .: "severity"
        "MonitorAlterSeverity" →
                  MonitorAlterSeverity <$> v .: "name" <*> v .: "severity"
        _ → fail "unknown MonitorAction"
```

LogStructured could also be:

```
forall b ∘ (ToJSON b) ⇒ LogStructured b
```

Payload of a LogObject:

```
data LOContent a = LogMessage a
          | LogError !Text
          | LogValue !Text !Measurable
          | LogStructuredText Object Text
          | LogStructured Object
          | ObserveOpen !CounterState
          | ObserveDiff !CounterState
          | ObserveClose !CounterState
          | AggregatedMessage [(Text, Aggregated)]
          | MonitoringEffect !MonitorAction
          | Command !CommandValue
          | KillPill
          deriving (Show, Eq)
-- WARNING: update 'locTypeEq' when extending this!
instance ToJSON a ⇒ ToJSON (LOContent a) where
  toJSON (LogMessage m) =
    object ["kind" .= String "LogMessage"
      ,"message" .= toJSON m]
  toJSON (LogError m) =
    object ["kind" .= String "LogError"
      ,"message" .= toJSON m]
  toJSON (LogValue n v) =
    object ["kind" .= String "LogValue"
      ,"name" .= toJSON n
```

```
      , "value" .= toJSON v]
  toJSON (LogStructured m) =
    object ["kind" .= String "LogStructured"
      , "data" .= m]
  toJSON (LogStructuredText o t) =
    object ["kind" .= String "LogStructuredText"
      , "data" .= o
      , "text" .= t]
  toJSON (ObserveOpen c) =
    object ["kind" .= String "ObserveOpen"
      , "counters" .= toJSON c]
  toJSON (ObserveDiff c) =
    object ["kind" .= String "ObserveDiff"
      , "counters" .= toJSON c]
  toJSON (ObserveClose c) =
    object ["kind" .= String "ObserveClose"
      , "counters" .= toJSON c]
  toJSON (AggregatedMessage ps) =
    object ["kind" .= String "AggregatedMessage"
      , "pairs" .= toJSON ps]
  toJSON (MonitoringEffect a) =
    object ["kind" .= String "MonitoringEffect"
      , "action" .= toJSON a]
  toJSON (Command c) =
    object ["kind" .= String "Command"
      , "command" .= toJSON c]
  toJSON KillPill =
    String "KillPill"
instance (FromJSON a) ⇒ FromJSON (LOContent a) where
  parseJSON j = withObject "LOContent"
      (λv → (v .: "kind" :: Parser Text)
        ≫=
        λcase "LogMessage" → LogMessage <$> v .: "message"
          "LogError" → LogError <$> v .: "message"
          "LogValue" → LogValue <$> v .: "name" <*> v .: "value"
          "LogStructured" → LogStructured <$> v .: "data"
          "LogStructuredText" → LogStructuredText <$> v .: "data" <*> v .: "text"
          "ObserveOpen" → ObserveOpen <$> v .: "counters"
          "ObserveDiff" → ObserveDiff <$> v .: "counters"
          "ObserveClose" → ObserveClose <$> v .: "counters"
          "AggregatedMessage" → AggregatedMessage <$> v .: "pairs"
          "MonitoringEffect" → MonitoringEffect <$> v .: "action"
          "Command" → Command <$> v .: "command"
          _ → fail "unknown LOContent")
      j
    <|>
      withText "LOContent"
      (λcase "KillPill" → pure KillPill
        _ → fail "unknown LOContent (String)")
      j
loType :: LogObject a → Text
```

```
loType (LogObject _ _ content) = loType2Name content
-- Equality between LogObjects based on their log content types.
loTypeEq :: LogObject a → LogObject a → Bool
loTypeEq = locTypeEq `on` loContent
locTypeEq :: LOContent a → LOContent a → Bool
locTypeEq LogMessage {}       LogMessage {}       = True
locTypeEq LogError {}         LogError {}         = True
locTypeEq LogValue {}         LogValue {}         = True
locTypeEq LogStructured {}    LogStructured {}    = True
locTypeEq ObserveOpen {}      ObserveOpen {}      = True
locTypeEq ObserveDiff {}      ObserveDiff {}      = True
locTypeEq ObserveClose {}     ObserveClose {}     = True
locTypeEq AggregatedMessage {} AggregatedMessage {} = True
locTypeEq MonitoringEffect {} MonitoringEffect {} = True
locTypeEq Command {}          Command {}          = True
locTypeEq KillPill {}         KillPill {}         = True
locTypeEq _ _ = False
```

Name of a message content type

```
loType2Name :: LOContent a → Text
loType2Name = λcase
  LogMessage _          → "LogMessage"
  LogError _            → "LogError"
  LogValue _ _          → "LogValue"
  LogStructured _       → "LogStructured"
  LogStructuredText _ _ → "LogStructuredText"
  ObserveOpen _         → "ObserveOpen"
  ObserveDiff _         → "ObserveDiff"
  ObserveClose _        → "ObserveClose"
  AggregatedMessage _   → "AggregatedMessage"
  MonitoringEffect _    → "MonitoringEffect"
  Command _             → "Command"
  KillPill              → "KillPill"
```

Backends can enter commands to the trace. Commands will end up in the Switchboard, which will interpret them and take action.

```
newtype CommandValue = DumpBufferedTo BackendKind
  deriving (Show, Eq)
instance ToJSON CommandValue where
  toJSON (DumpBufferedTo be) =
    object ["kind" .= String "DumpBufferedTo"
      , "backend" .= toJSON be]
instance FromJSON CommandValue where
  parseJSON = withObject "CommandValue" $ λv →
    (v .: "kind" :: Parser Text)
    >>=
    λcase "DumpBufferedTo" → DumpBufferedTo <$> v .: "backend"
      _ → fail "unknown CommandValue"
```

Privacy annotation

```
data PrivacyAnnotation =
  Confidential-- confidential information - handle with care
  | Public-- indifferent - can be public.
  deriving (Show, Eq)
instance FromJSON PrivacyAnnotation where
  parseJSON = withText "PrivacyAnnotation" $
    λcase "Confidential" → pure Confidential
      "Public" → pure Public
      _ → fail "unknown PrivacyAnnotation"
```

Data structure for annotating the severity and privacy of an object.

```
data PrivacyAndSeverityAnnotated a
  = PSA {psaSeverity :: !Severity
    ,psaPrivacy :: !PrivacyAnnotation
    ,psaPayload :: a
    }
  deriving (Show)
```

Mapping Log Objects

This provides a helper function to transform log items. It would often be used with *contramap*.

```
mapLogObject :: (a → b) → LogObject a → LogObject b
mapLogObject f (LogObject nm me loc) = LogObject nm me (mapLOContent f loc)
instance Functor LogObject where
  fmap = mapLogObject
mapLOContent :: (a → b) → LOContent a → LOContent b
mapLOContent f = λcase
  LogMessage msg       → LogMessage (f msg)
  LogError a           → LogError a
  LogStructured o      → LogStructured o
  LogStructuredText o m → LogStructuredText o m
  LogValue n v         → LogValue n v
  ObserveOpen st       → ObserveOpen st
  ObserveDiff st       → ObserveDiff st
  ObserveClose st      → ObserveClose st
  AggregatedMessage ag → AggregatedMessage ag
  MonitoringEffect act → MonitoringEffect act
  Command v            → Command v
  KillPill             → KillPill
-- Equality between LogObjects based on their log content values.
loContentEq :: Eq a ⇒ LogObject a → LogObject a → Bool
loContentEq = (≡) `on` loContent
```

Render context name as text

```
loname2text :: [LoggerName] → Text
loname2text nms = T.init $ foldl' (λel acc → acc <> "." <> el) "" nms
```

1.7.19 Cardano.BM.Data.Observable

ObservableInstance

```
data ObservableInstance = MonotonicClock
  | MemoryStats
  | ProcessStats
  | NetStats
  | IOStats
  | SysStats
  | GhcRtsStats
    deriving (Generic, Eq, Show, FromJSON, ToJSON, Read)
```

1.7.20 Cardano.BM.Data.Rotation

RotationParameters

```
data RotationParameters = RotationParameters
  {rpLogLimitBytes :: !Word64-- max size of file in bytes
  ,rpMaxAgeHours :: !Word   -- hours
  ,rpKeepFilesNum :: !Word  -- number of files to keep
  } deriving (Generic, Show, Eq, Ord, FromJSON, ToJSON)
```

1.7.21 Cardano.BM.Data.Severity

Severity

The intended meaning of severity codes:
Debug *detailed information about values and decision flow* Info general information of events; progressing properly Notice *needs attention; something ¬ progressing properly* Warning may continue into an error condition if continued Error *unexpected set* **of** *event or condition occurred* Critical error condition causing degrade of operation Alert *a subsystem is no longer operating correctly, likely requires ma* at this point, the system can never progress without additional intervention
We were informed by the *Syslog* taxonomy: https://en.wikipedia.org/wiki/Syslog#Severity_level

```
data Severity = Debug
  | Info
  | Notice
  | Warning
  | Error
  | Critical
  | Alert
  | Emergency
    deriving (Show, Eq, Ord, Bounded, Enum, Generic, ToJSON, Read)
instance FromJSON Severity where
  parseJSON = withText "severity" $ λcase
    "Debug"     → pure Debug
    "Info"      → pure Info
    "Notice"    → pure Notice
    "Warning"   → pure Warning
    "Error"     → pure Error
```

```
    "Critical" → pure Critical
    "Alert"    → pure Alert
    "Emergency" → pure Emergency
    _          → pure Info-- catch all
```

1.7.22 Cardano.BM.Data.SubTrace

SubTrace

```
data NameSelector = Exact Text | StartsWith Text | EndsWith Text | Contains Text
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data DropName     = Drop NameSelector
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data UnhideNames = Unhide [NameSelector]
                  deriving (Generic, Show, FromJSON, ToJSON, Read, Eq)
data SubTrace = Neutral
  | UntimedTrace
  | NoTrace
  | TeeTrace LoggerName
  | FilterTrace [(DropName, UnhideNames)]
  | DropOpening
  | ObservableTraceSelf [ObservableInstance]
  | ObservableTrace ProcessID [ObservableInstance]
  | SetSeverity Severity
    deriving (Generic, Show, Read, Eq)
#ifdef WINDOWS
-- Wrap the Win32 DWORD type alias so that it can be logged
newtype ProcessID = ProcessID ProcessId
  deriving (Generic, Show, Read, Eq)
instance ToJSON ProcessID where
  toJSON (ProcessID pid) = Number $ fromIntegral pid
instance FromJSON ProcessID where
  parseJSON v = ProcessID <$> parseJSON v
#else
instance ToJSON ProcessID where
  toJSON (CPid pid) = Number $ fromIntegral pid
instance FromJSON ProcessID where
  parseJSON v = CPid <$> parseJSON v
#endif
instance FromJSON SubTrace where
  parseJSON = withObject "SubTrace" $ λo → do
                  subtrace :: Text ← o .: "subtrace"
                  case subtrace of
                    "Neutral"       → return $ Neutral
                    "UntimedTrace"  → return $ UntimedTrace
                    "NoTrace"       → return $ NoTrace
                    "TeeTrace"      → TeeTrace      <$> o .: "contents"
                    "FilterTrace"   → FilterTrace   <$> o .: "contents"
                    "DropOpening"   → return $ DropOpening
```

```
                "ObservableTraceSelf" → ObservableTraceSelf <$> o .: "contents"
                "ObservableTrace" → ObservableTrace <$> o .: "pid"
                                                    <*> o .: "contents"
                "SetSeverity"     → SetSeverity     <$> o .: "contents"
                other             → fail $ "unexpected subtrace: " ++ (unpack other)
    instance ToJSON SubTrace where
      toJSON Neutral =
        object ["subtrace" .= String "Neutral"          ]
      toJSON UntimedTrace =
        object ["subtrace" .= String "UntimedTrace"     ]
      toJSON NoTrace =
        object ["subtrace" .= String "NoTrace"          ]
      toJSON (TeeTrace name) =
        object ["subtrace" .= String "TeeTrace"         ,"contents" .= toJSON name]
      toJSON (FilterTrace dus) =
        object ["subtrace" .= String "FilterTrace"      ,"contents" .= toJSON dus     ]
      toJSON DropOpening =
        object ["subtrace" .= String "DropOpening"      ]
      toJSON (ObservableTraceSelf os) =
        object ["subtrace" .= String "ObservableTraceSelf","contents" .= toJSON os]
      toJSON (ObservableTrace pid os) =
        object ["subtrace" .= String "ObservableTrace","pid" .= toJSON pid
          ,"contents" .= toJSON os                      ]
      toJSON (SetSeverity sev) =
        object ["subtrace" .= String "SetSeverity"      ,"contents" .= toJSON sev     ]
```

1.7.23 Cardano.BM.Data.Trace

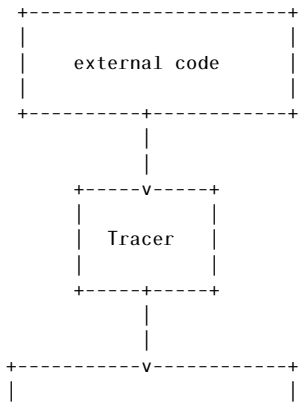
Trace

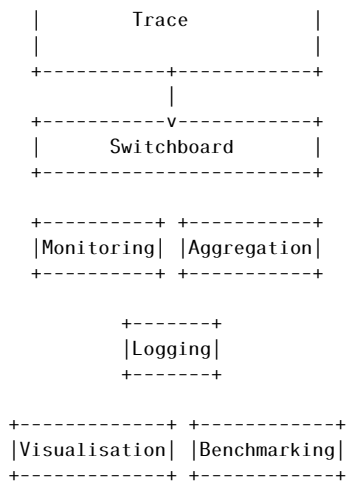
A Trace *m a* is a Tracer containing the context name and a LogObject *a*.

```
type Trace m a = Tracer m (LoggerName, LogObject a)
```

1.7.24 Cardano.BM.Data.Tracer

This module extends the basic Tracer with one that keeps a list of context names to create the basis for Trace which accepts messages from a Tracer and ends in the Switchboard for further processing of the messages.





ToLogObject - transforms a logged item to LogObject

The transformer toLogObject accepts any type for which a ToObject instance is available and returns a LogObject which can be forwarded into the Switchboard. It adds a verbosity hint of NormalVerbosity.
A verbosity level TracingVerbosity can be passed to the transformer toLogObject.

```
class Monad m ⇒ ToLogObject m where
  toLogObject :: (ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  toLogObject :: (ToObject a, Transformable a m b)
    ⇒ TracingVerbosity → Trace m a → Tracer m b
  toLogObjectVerbose :: (ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  default toLogObjectVerbose :: (ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  toLogObjectVerbose = trTransformer MaximalVerbosity
  toLogObjectMinimal :: (ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  default toLogObjectMinimal :: (ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  toLogObjectMinimal = trTransformer MinimalVerbosity
instance ToLogObject IO where
  toLogObject :: (MonadIO m, ToObject a, Transformable a m b)
    ⇒ Trace m a → Tracer m b
  toLogObject = trTransformer NormalVerbosity
  toLogObject :: (MonadIO m, ToObject a, Transformable a m b)
    ⇒ TracingVerbosity → Trace m a → Tracer m b
  toLogObject = trTransformer

To be placed in ouroboros – network ∘
instance (MonadFork m, MonadTimer m) ⇒ ToLogObject m where
  toLogObject tr = Tracer $ λa → do
    lo ← LogObject <$> pure ""
      <*> (LOMeta <$> getMonotonicTime-- must be evaluated at the calling site
        <*> (pack ∘ show <$> myThreadId)
        <*> pure Debug
```

```
        <*> pure Public)
      <*> pure (LogMessage a)
  traceWith tr lo
```

Verbosity levels

The tracing verbosity will be passed to instances of ToObject for rendering the traced item accordingly.

```
data TracingVerbosity = MinimalVerbosity | NormalVerbosity | MaximalVerbosity
  deriving (Eq, Read, Ord)
```

ToObject - transforms a logged item to a JSON Object

Katip requires JSON objects to be logged as context. This typeclass provides a default instance which uses *ToJSON* and produces an empty object if 'toJSON' results in any type other than *Object*. If you have a type you want to log that produces an Array or Number for example, you'll want to write an explicit instance of ToObject. You can trivially add a ToObject instance for something with a *ToJSON* instance like:

```
instance ToObject Foo
```

The toObject function accepts a TracingVerbosity level as argument and can render the traced item differently depending on the verbosity level.

```
class ToObject a where
  toObject :: TracingVerbosity → a → Object
  default toObject :: ToJSON a ⇒ TracingVerbosity → a → Object
  toObject _ v = case toJSON v of
    Object o → o
    s@(String _) → HM.singleton "string" s
    _            → mempty
  textTransformer :: a → Object → Text
  default textTransformer :: a → Object → Text
  textTransformer _ o = TL.toStrict $ encodeToLazyText o
```

A helper function for creating an *Object* given a list of pairs, named items, or the empty *Object*.

```
mkObject :: ToObject a ⇒ [(Text, a)] → HM.HashMap Text a
mkObject = HM.fromList

emptyObject :: ToObject a ⇒ HM.HashMap Text a
emptyObject = HM.empty
```

default instances:

```
instance ToObject () where
  toObject _ _ = mempty

instance ToObject String
instance ToObject Text
instance ToObject Value
instance ToJSON a ⇒ ToObject (LogObject a)
instance ToJSON a ⇒ ToObject (LOContent a)
```

A transformable Tracer

Parameterised over the source Tracer (*b*) and the target Tracer (*a*).
The default definition of trTransformer is the nullTracer. This blocks output of all items which lack a corresponding instance of Transformable.
Depending on the input type it can create objects of LogValue for numerical values, LogMessage for textual messages, and for all others a LogStructured of their ToObject representation.

```
class (Monad m, HasPrivacyAnnotation b, HasSeverityAnnotation b) ⇒ Transformable a m b where
  trTransformer :: TracingVerbosity → Trace m a → Tracer m b
  default trTransformer :: TracingVerbosity → Trace m a → Tracer m b
  trTransformer _ _ = nullTracer

trFromIntegral :: (Integral b, MonadIO m, HasPrivacyAnnotation b, HasSeverityAnnotation b)
  ⇒ LoggerName → Trace m a → Tracer m b
trFromIntegral name tr = Tracer $ λarg →
    traceWith tr =≪ do
      meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
      return (mempty
        , LogObject mempty meta (LogValue name $ PureI $ fromIntegral arg)
        )
trFromReal :: (Real b, MonadIO m, HasPrivacyAnnotation b, HasSeverityAnnotation b)
  ⇒ LoggerName → Trace m a → Tracer m b
trFromReal name tr = Tracer $ λarg →
    traceWith tr =≪ do
      meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
      return (mempty
        , LogObject mempty meta (LogValue name $ PureD $ realToFrac arg)
        )
instance Transformable a IO Int where
  trTransformer MinimalVerbosity = trFromIntegral ""
  trTransformer _ = trFromIntegral "int"
instance Transformable a IO Integer where
  trTransformer MinimalVerbosity = trFromIntegral ""
  trTransformer _ = trFromIntegral "integer"
instance Transformable a IO Word64 where
  trTransformer MinimalVerbosity = trFromIntegral ""
  trTransformer _ = trFromIntegral "word64"
instance Transformable a IO Double where
  trTransformer MinimalVerbosity = trFromReal ""
  trTransformer _ = trFromReal "double"
instance Transformable a IO Float where
  trTransformer MinimalVerbosity = trFromReal ""
  trTransformer _ = trFromReal "float"
instance Transformable Text IO Text where
  trTransformer _ tr = Tracer $ λarg →
    traceWith tr =≪ do
      meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
      return (mempty
        , LogObject mempty meta (LogMessage arg)
        )
instance Transformable String IO String where
  trTransformer _ tr = Tracer $ λarg →
```

```
      traceWith tr =≪ do
        meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
        return (mempty
          , LogObject mempty meta (LogMessage arg)
          )
instance Transformable Text IO String where
  trTransformer _ tr = Tracer $ λarg →
    traceWith tr =≪ do
      meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
      return (mempty
        , LogObject mempty meta (LogMessage $ T.pack arg)
        )
instance Transformable String IO Text where
  trTransformer _ tr = Tracer $ λarg →
    traceWith tr =≪ do
      meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
      return (mempty
        , LogObject mempty meta (LogMessage $ T.unpack arg)
        )
```

The function trStructured is a tracer transformer which transforms traced items to their ToObject representation and further traces them as a LogObject of type LogStructured. If the ToObject representation is empty, then no tracing happens.

```
trStructured :: (ToObject b, MonadIO m, HasPrivacyAnnotation b, HasSeverityAnnotation b)
  ⇒ TracingVerbosity → Trace m a → Tracer m b
trStructured verb tr = Tracer $ λarg →
let
  obj = toObject verb arg
in traceWith tr =≪ do
  meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
  return (mempty
    , LogObject mempty meta (LogStructuredText obj (T.pack $ show $ obj))
    )


class HasTextFormatter a where
  formatText :: a → Object → Text
  default formatText :: a → Object → Text
  formatText _a = T.pack ∘ show
trStructuredText :: (ToObject b, MonadIO m, HasTextFormatter b
  , HasPrivacyAnnotation b, HasSeverityAnnotation b)
    ⇒ TracingVerbosity → Trace m a → Tracer m b
trStructuredText verb tr = Tracer $ λarg →
let
  obj = toObject verb arg
in traceWith tr =≪ do
  meta ← mkLOMeta (getSeverityAnnotation arg) (getPrivacyAnnotation arg)
  return (mempty
    , LogObject mempty meta (LogStructuredText obj (formatText arg obj))
    )
```

Transformers for setting severity level

The log Severity level of a LogObject can be altered.

```
setSeverity :: Severity → Trace m a → Trace m a
setSeverity sev tr = Tracer $ λ(ctx, lo@(LogObject _nm meta@(LOMeta _ts _tid _hn _sev _pr) _lc)) →
  traceWith tr $ (ctx, lo {loMeta = meta {severity = sev}})
severityDebug, severityInfo, severityNotice,
  severityWarning, severityError, severityCritical,
  severityAlert, severityEmergency :: Trace m a → Trace m a
severityDebug     = setSeverity Debug
severityInfo      = setSeverity Info
severityNotice    = setSeverity Notice
severityWarning   = setSeverity Warning
severityError     = setSeverity Error
severityCritical  = setSeverity Critical
severityAlert     = setSeverity Alert
severityEmergency = setSeverity Emergency
```

The Severity of any Tracer can be set with wrapping it in *WithSeverity*. The traced types need to be of class *HasSeverityAnnotation*.

```
annotateSeverity :: HasSeverityAnnotation a ⇒ Tracer m (WithSeverity a) → Tracer m a
annotateSeverity tr = Tracer $ λarg →
  traceWith tr $ WithSeverity (getSeverityAnnotation arg) arg
```

Transformers for setting privacy annotation

The privacy annotation (PrivacyAnnotation) of the LogObject can be altered with the following functions.

```
setPrivacy :: PrivacyAnnotation → Trace m a → Trace m a
setPrivacy prannot tr = Tracer $ λ(ctx, lo@(LogObject _nm meta _lc)) →
  traceWith tr $ (ctx, lo {loMeta = meta {privacy = prannot}})
annotateConfidential, annotatePublic :: Trace m a → Trace m a
annotateConfidential = setPrivacy Confidential
annotatePublic = setPrivacy Public
```

The PrivacyAnnotation of any Tracer can be set with wrapping it in *WithPrivacyAnnotation*. The traced types need to be of class *DefinePrivacyAnnotation*.

```
annotatePrivacyAnnotation :: HasPrivacyAnnotation a ⇒ Tracer m (WithPrivacyAnnotation a) → Tracer m a
annotatePrivacyAnnotation tr = Tracer $ λarg →
  traceWith tr $ WithPrivacyAnnotation (getPrivacyAnnotation arg) arg
```

Transformer for filtering based on *Severity*

This structure wraps a Severity around traced observables.

```
data WithSeverity a = WithSeverity Severity a
```

The traced observables with annotated severity are filtered.

```
filterSeverity :: forall m a ∘ (Monad m, HasSeverityAnnotation a)
  ⇒ (a → m Severity)
  → Tracer m a
  → Tracer m a
filterSeverity msevlimit tr = Tracer $ λarg → do
  sevlimit ← msevlimit arg
  when (getSeverityAnnotation arg ⩾ sevlimit) $
    traceWith tr arg
```

General instances of *WithSeverity* wrapped observable types.

```
instance forall m a t ∘ (Monad m, Transformable t m a) ⇒ Transformable t m (WithSeverity a) where
  trTransformer verb tr = Tracer $ λ(WithSeverity sev arg) →
    let transformer :: Tracer m a
        transformer = trTransformer verb $ setSeverity sev tr
    in traceWith transformer arg
```

Transformer for filtering based on *PrivacyAnnotation*

This structure wraps a Severity around traced observables.

```
data WithPrivacyAnnotation a = WithPrivacyAnnotation PrivacyAnnotation a
```

The traced observables with annotated severity are filtered.

```
filterPrivacyAnnotation :: forall m a ∘ (Monad m, HasPrivacyAnnotation a)
  ⇒ (a → m PrivacyAnnotation)
  → Tracer m a
  → Tracer m a
filterPrivacyAnnotation mpa tr = Tracer $ λarg → do
  pa ← mpa arg
  when (getPrivacyAnnotation arg ≡ pa) $
    traceWith tr arg
```

General instances of *WithPrivacyAnnotation* wrapped observable types.

```
instance forall m a t ∘ (Monad m, Transformable t m a) ⇒ Transformable t m (WithPrivacyAnnotation a) wh
  trTransformer verb tr = Tracer $ λ(WithPrivacyAnnotation pa arg) →
    let transformer :: Tracer m a
        transformer = trTransformer verb $ setPrivacy pa tr
    in traceWith transformer arg
```

The properties of being annotated with severity and privacy

From a type with the property of *HasSeverityAnnotation*, one will be able to extract its severity annotation.

```
class HasSeverityAnnotation a where
  getSeverityAnnotation :: a → Severity
  default getSeverityAnnotation :: a → Severity
  getSeverityAnnotation _ = Debug
instance HasSeverityAnnotation (WithSeverity a) where
```

```
  getSeverityAnnotation (WithSeverity sev _) = sev
instance HasSeverityAnnotation a ⇒ HasSeverityAnnotation (WithPrivacyAnnotation a) where
  getSeverityAnnotation (WithPrivacyAnnotation _ a) = getSeverityAnnotation a
-- default instances
instance HasSeverityAnnotation Double
instance HasSeverityAnnotation Float
instance HasSeverityAnnotation Int
instance HasSeverityAnnotation Integer
instance HasSeverityAnnotation String
instance HasSeverityAnnotation Text
instance HasSeverityAnnotation Word64
```

And, privacy annotation can be extracted from types with the property *HasPrivacyAnnotation*.

```
class HasPrivacyAnnotation a where
  getPrivacyAnnotation :: a → PrivacyAnnotation
  default getPrivacyAnnotation :: a → PrivacyAnnotation
  getPrivacyAnnotation _ = Public
instance HasPrivacyAnnotation (WithPrivacyAnnotation a) where
  getPrivacyAnnotation (WithPrivacyAnnotation pva _) = pva
instance HasPrivacyAnnotation a ⇒ HasPrivacyAnnotation (WithSeverity a) where
  getPrivacyAnnotation (WithSeverity _ a) = getPrivacyAnnotation a
-- default instances
instance HasPrivacyAnnotation Double
instance HasPrivacyAnnotation Float
instance HasPrivacyAnnotation Int
instance HasPrivacyAnnotation Integer
instance HasPrivacyAnnotation String
instance HasPrivacyAnnotation Text
instance HasPrivacyAnnotation Word64
```

1.7.25 Cardano.BM.Configuration

see Cardano.BM.Configuration.Model for the implementation.

```
getTextOptionOrDefault :: CM.Configuration → Text → Text → IO Text
getTextOptionOrDefault cg name def = fromMaybe def <$> CM.getTextOption cg name
```

Test severities

Test severity of the given LOMeta to be greater or equal to those of the specific LoggerName.

```
testSeverity :: CM.Configuration → LoggerName → LOMeta → IO Bool
testSeverity config loggername meta = do
  globminsev ← CM.minSeverity config
  globnamesev ← CM.inspectSeverity config loggername
  let minsev = max globminsev $ fromMaybe Debug globnamesev
  return $ (severity meta) ⩾ minsev
```

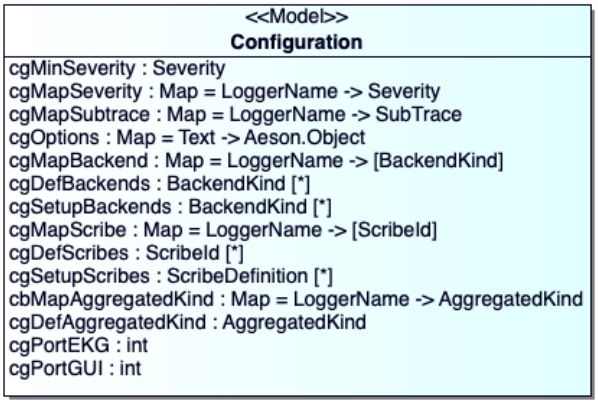


Figure 1.5: Configuration model

1.7.26 Cardano.BM.Configuration.Model

Configuration.Model

```
type ConfigurationMVar = MVar ConfigurationInternal
newtype Configuration = Configuration
  {getCG :: ConfigurationMVar}
-- Our internal state; see -"Configuration model"-
data ConfigurationInternal = ConfigurationInternal
  {cgMinSeverity     :: Severity
  -- minimum severity level of every object that will be output
  ,cgDefRotation     :: Maybe RotationParameters
  -- default rotation parameters
  ,cgMapSeverity     :: HM.HashMap LoggerName Severity
  -- severity filter per loggername
  ,cgMapSubtrace     :: HM.HashMap LoggerName SubTrace
  -- type of trace per loggername
  ,cgOptions         :: HM.HashMap Text Value
  -- options needed for tracing, logging and monitoring
  ,cgMapBackend      :: HM.HashMap LoggerName [BackendKind]
  -- backends that will be used for the specific loggername
  ,cgDefBackendKs    :: [BackendKind]
  -- backends that will be used if a set of backends for the
  -- specific loggername is not set
  ,cgSetupBackends   :: [BackendKind]
  -- backends to setup; every backend to be used must have
  -- been declared here
  ,cgMapScribe       :: HM.HashMap LoggerName [ScribeId]
  -- katip scribes that will be used for the specific loggername
  ,cgMapScribeCache  :: HM.HashMap LoggerName [ScribeId]
  -- map to cache info of the cgMapScribe
  ,cgDefScribes      :: [ScribeId]
  -- katip scribes that will be used if a set of scribes for the
  -- specific loggername is not set
  ,cgSetupScribes    :: [ScribeDefinition]
```

```
  -- katip scribes to setup; every scribe to be used must have
  -- been declared here
  ,cgMapAggregatedKind :: HM.HashMap LoggerName AggregatedKind
  -- kind of Aggregated that will be used for the specific loggername
  ,cgDefAggregatedKind :: AggregatedKind
  -- kind of Aggregated that will be used if a set of scribes for the
  -- specific loggername is not set
  ,cgMonitors          :: HM.HashMap LoggerName (MEvPreCond, MEvExpr, [MEvAction])
  ,cgPortEKG           :: Int
  -- port for EKG server
  ,cgPortGraylog       :: Int
  -- port to Graylog server
  ,cgBindAddrPrometheus :: Maybe (String, Int)
  -- host/port to bind Prometheus server at
  ,cgForwardTo         :: Maybe RemoteAddr
  -- trace acceptor to forward to
  ,cgAcceptAt          :: Maybe [RemoteAddrNamed]
  -- accept remote traces at this address
  ,cgPortGUI           :: Int
  -- port for changes at runtime
  } deriving (Show, Eq)
```

Backends configured in the Switchboard

For a given context name return the list of backends configured, or, in case no such configuration exists, return the default backends.

```
getBackends :: Configuration → LoggerName → IO [BackendKind]
getBackends configuration name = do
  cg ← readMVar $ getCG configuration
  -- let outs = HM.lookup name (cgMapBackend cg)
  -- case outs of
  -- Nothing -> return (cgDefBackendKs cg)
  -- Just os -> return os
  let defs = cgDefBackendKs cg
  let mapbks = cgMapBackend cg
  let find_s [ ] = defs
      find_s lnames = case HM.lookup (T.intercalate "." lnames) mapbks of
        Nothing → find_s (init lnames)
        Just os → os
  return $ find_s $ T.split (≡ '.') name

getDefaultBackends :: Configuration → IO [BackendKind]
getDefaultBackends configuration =
  cgDefBackendKs <$> (readMVar $ getCG configuration)

setDefaultBackends :: Configuration → [BackendKind] → IO ()
setDefaultBackends configuration bes =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgDefBackendKs = bes}

setBackends :: Configuration → LoggerName → Maybe [BackendKind] → IO ()
setBackends configuration name be =
```

```
    modifyMVar_ (getCG configuration) $ λcg →
      return cg {cgMapBackend = HM.alter (\_ → be) name (cgMapBackend cg)}
```

Backends to be setup by the Switchboard

Defines the list of Backends that need to be setup by the Switchboard.

```
setSetupBackends :: Configuration → [BackendKind] → IO ()
setSetupBackends configuration bes =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgSetupBackends = bes}
getSetupBackends :: Configuration → IO [BackendKind]
getSetupBackends configuration =
  cgSetupBackends <$> (readMVar $ getCG configuration)
```

Scribes configured in the Log backend

For a given context name return the list of scribes to output to, or, in case no such configuration exists, return the default scribes to use.

```
getScribes :: Configuration → LoggerName → IO [ScribeId]
getScribes configuration name = do
  cg ← readMVar (getCG configuration)
  (updateCache, scribes) ← do
    let defs = cgDefScribes cg
    let mapscribes = cgMapScribe cg
    let find_s [] = defs
        find_s lnames = case HM.lookup (T.intercalate "." lnames) mapscribes of
          Nothing → find_s (init lnames)
          Just os → os
    let outs = HM.lookup name (cgMapScribeCache cg)
    -- look if scribes are already cached
    return $ case outs of
      -- if no cached scribes found; search the appropriate scribes that
      -- they must inherit and update the cached map
      Nothing → (True, find_s $ T.split (≡ '.') name)
      Just os → (False, os)
  when updateCache $ setCachedScribes configuration name $ Just scribes
  return scribes
getCachedScribes :: Configuration → LoggerName → IO (Maybe [ScribeId])
getCachedScribes configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name $ cgMapScribeCache cg
setScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapScribe = HM.alter (\_ → scribes) name (cgMapScribe cg)}
setCachedScribes :: Configuration → LoggerName → Maybe [ScribeId] → IO ()
setCachedScribes configuration name scribes =
  modifyMVar_ (getCG configuration) $ λcg →
```

```
    return cg {cgMapScribeCache = HM.alter (\_ → scribes) name (cgMapScribeCache cg)}
setDefaultScribes :: Configuration → [ScribeId] → IO ()
setDefaultScribes configuration scs =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgDefScribes = scs}
```

Scribes to be setup in the Log backend

Defines the list of *Scribe*s that need to be setup in the Log backend.

```
setSetupScribes :: Configuration → [ScribeDefinition] → IO ()
setSetupScribes configuration sds =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgSetupScribes = sds}
getSetupScribes :: Configuration → IO [ScribeDefinition]
getSetupScribes configuration =
  cgSetupScribes <$> readMVar (getCG configuration)
```

AggregatedKind to define the type of measurement

For a given context name return its AggregatedKind or in case no such configuration exists, return the default AggregatedKind to use.

```
getAggregatedKind :: Configuration → LoggerName → IO AggregatedKind
getAggregatedKind configuration name = do
  cg ← readMVar $ getCG configuration
  let outs = HM.lookup name (cgMapAggregatedKind cg)
  case outs of
    Nothing → return $ cgDefAggregatedKind cg
    Just os → return $ os
setDefaultAggregatedKind :: Configuration → AggregatedKind → IO ()
setDefaultAggregatedKind configuration defAK =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgDefAggregatedKind = defAK}
setAggregatedKind :: Configuration → LoggerName → Maybe AggregatedKind → IO ()
setAggregatedKind configuration name ak =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapAggregatedKind = HM.alter (\_ → ak) name (cgMapAggregatedKind cg)}
```

Access port numbers of EKG, Prometheus, GUI

```
getEKGport :: Configuration → IO Int
getEKGport configuration =
  cgPortEKG <$> (readMVar $ getCG configuration)
setEKGport :: Configuration → Int → IO ()
setEKGport configuration port =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgPortEKG = port}
getGraylogPort :: Configuration → IO Int
```

```
getGraylogPort configuration =
  cgPortGraylog <$> (readMVar $ getCG configuration)
setGraylogPort :: Configuration → Int → IO ()
setGraylogPort configuration port =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgPortGraylog = port}
getPrometheusBindAddr :: Configuration → IO (Maybe (String, Int))
getPrometheusBindAddr configuration =
  cgBindAddrPrometheus <$> (readMVar $ getCG configuration)
setPrometheusBindAddr :: Configuration → Maybe (String, Int) → IO ()
setPrometheusBindAddr configuration mHostPort =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgBindAddrPrometheus = mHostPort}
getGUIport :: Configuration → IO Int
getGUIport configuration =
  cgPortGUI <$> (readMVar $ getCG configuration)
setGUIport :: Configuration → Int → IO ()
setGUIport configuration port =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgPortGUI = port}
getAcceptAt :: Configuration → IO (Maybe [RemoteAddrNamed])
getAcceptAt = fmap cgAcceptAt ∘ readMVar ∘ getCG
getForwardTo :: Configuration → IO (Maybe RemoteAddr)
getForwardTo = fmap cgForwardTo ∘ readMVar ∘ getCG
setForwardTo :: Configuration → Maybe RemoteAddr → IO ()
setForwardTo cf mra =
  modifyMVar_ (getCG cf) $ λcg →
    return cg {cgForwardTo = mra}
```

Options

```
getMapOption' :: HM.HashMap Text Value → Text → Maybe Object
getMapOption' m (flip HM.lookup m → Just (Object x)) = Just x
getMapOption' _ _ = Nothing
getTextOption' :: HM.HashMap Text Value → Text → Maybe Text
getTextOption' m (flip HM.lookup m → Just (String x)) = Just x
getTextOption' _ _ = Nothing
getOption :: Configuration → Text → IO (Maybe Value)
getOption configuration name =
  HM.lookup name ∘ cgOptions <$> readMVar (getCG configuration)
getTextOption :: Configuration → Text → IO (Maybe Text)
getTextOption configuration name =
  flip getTextOption' name ∘ cgOptions <$> readMVar (getCG configuration)
getMapOption :: Configuration → Text → IO (Maybe Object)
getMapOption configuration name =
  flip getMapOption' name ∘ cgOptions <$> readMVar (getCG configuration)
updateOption :: Configuration → Text → (Maybe Value → Value) → IO ()
```

```
updateOption configuration name f =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgOptions = HM.alter (Just ∘ f) name (cgOptions cg)}
setOption :: Configuration → Text → Value → IO ()
setOption configuration name = updateOption configuration name ∘ const
setTextOption :: Configuration → Text → Text → IO ()
setTextOption configuration name = setOption configuration name ∘ String
```

Global setting of minimum severity

```
minSeverity :: Configuration → IO Severity
minSeverity configuration =
  cgMinSeverity <$> (readMVar $ getCG configuration)
setMinSeverity :: Configuration → Severity → IO ()
setMinSeverity configuration sev =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMinSeverity = sev}
```

Relation of context name to minimum severity

```
inspectSeverity :: Configuration → Text → IO (Maybe Severity)
inspectSeverity configuration name = do
  cg ← readMVar $ getCG configuration
  return $ HM.lookup name (cgMapSeverity cg)
setSeverity :: Configuration → Text → Maybe Severity → IO ()
setSeverity configuration name sev =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapSeverity = HM.alter (\_ → sev) name (cgMapSeverity cg)}
```

Relation of context name to SubTrace

A new context may contain a different type of Trace. The function appendName will look up the SubTrace for the context's name.

```
findSubTrace :: Configuration → Text → IO (Maybe SubTrace)
findSubTrace configuration name =
  HM.lookup name <$> cgMapSubtrace <$> (readMVar $ getCG configuration)
setSubTrace :: Configuration → Text → Maybe SubTrace → IO ()
setSubTrace configuration name trafo =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMapSubtrace = HM.alter (\_ → trafo) name (cgMapSubtrace cg)}
```

Monitors

```
Just (
  fromList [
```

```
      ("chain.creation.block",Array [
        Object (fromList [("monitor",String "((time > (23 s)) Or (time < (17 s)))")]),
        Object (fromList [("actions",Array [
          String "AlterMinSeverity \"chain.creation\" Debug"])])])
    ,("#aggregation.critproc.observable",Array [
        Object (fromList [("monitor",String "(mean >= (42))")]),
        Object (fromList [("actions",Array [
          String "CreateMessage \"exceeded\" \"the observable has been too long too high!\
          String "AlterGlobalMinSeverity Info"])])])

getMonitors :: Configuration → IO (HM.HashMap LoggerName (MEvPreCond,MEvExpr,[MEvAction]))
getMonitors configuration = do
  cg ← readMVar $ getCG configuration
  return (cgMonitors cg)
setMonitors :: Configuration → HM.HashMap LoggerName (MEvPreCond,MEvExpr,[MEvAction]) → IO (
setMonitors configuration monitors =
  modifyMVar_ (getCG configuration) $ λcg →
    return cg {cgMonitors = monitors}
```

Parse configuration from file

Parse the configuration into an internal representation first. Then, fill in Configuration after refinement.

```
setup :: FilePath → IO Configuration
setup fp = do
    r ← R.readRepresentation fp
    setupFromRepresentation r
parseMonitors :: Maybe (HM.HashMap Text Value) → HM.HashMap LoggerName (MEvPreCond,MEvExpr,
parseMonitors Nothing = HM.empty
parseMonitors (Just hmv) = HM.mapMaybe mkMonitor hmv
    where
    mkMonitor :: Value → Maybe (MEvPreCond,MEvExpr,[MEvAction])
    mkMonitor = parseMaybe $ λv →
      (withObject "" $ λo →
        (,,) <$> o .:? "monitor-if"
             <*> o .: "monitor"
             <*> o .: "actions") v
       <|> parseJSON v
setupFromRepresentation :: R.Representation → IO Configuration
setupFromRepresentation r = do
    let getMap       = getMapOption' (R.options r)
        mapscribes   = parseScribeMap $ getMap "mapScribes"
        defRotation  = R.rotation r
    cgref ← newMVar $ ConfigurationInternal
        {cgMinSeverity     = R.minSeverity r
        ,cgDefRotation     = defRotation
        ,cgMapSeverity     = parseSeverityMap $ getMap "mapSeverity"
        ,cgMapSubtrace     = parseSubtraceMap $ getMap "mapSubtrace"
        ,cgOptions         = R.options r
```

```
        ,cgMapBackend      = parseBackendMap $ getMap "mapBackends"
        ,cgDefBackendKs    = R.defaultBackends r
        ,cgSetupBackends   = R.setupBackends r
        ,cgMapScribe       = mapscribes
        ,cgMapScribeCache  = mapscribes
        ,cgDefScribes      = r_defaultScribes r
        ,cgSetupScribes    = fillRotationParams defRotation (R.setupScribes r)
        ,cgMapAggregatedKind = parseAggregatedKindMap $ getMap "mapAggregatedkinds"
        ,cgDefAggregatedKind = StatsAK
        ,cgMonitors        = parseMonitors $ getMap "mapMonitors"
        ,cgPortEKG         = r_hasEKG r
        ,cgPortGraylog     = r_hasGraylog r
        ,cgBindAddrPrometheus = r_hasPrometheus r
        ,cgPortGUI         = r_hasGUI r
        ,cgForwardTo       = r_forward r
        ,cgAcceptAt        = r_accept r
        }
    return $ Configuration cgref
where
    parseSeverityMap :: Maybe (HM.HashMap Text Value) -> HM.HashMap Text Severity
    parseSeverityMap Nothing = HM.empty
    parseSeverityMap (Just hmv) = HM.mapMaybe mkSeverity hmv
      where
        mkSeverity (String s) = Just (read (unpack s) :: Severity)
        mkSeverity _ = Nothing
    fillRotationParams :: Maybe RotationParameters -> [ScribeDefinition] -> [ScribeDefinition]
    fillRotationParams defaultRotation = map $ λsd ->
        if scKind sd == FileSK
        then
          sd {scRotation = maybe defaultRotation Just (scRotation sd)}
        else
          -- stdout, stderr, /dev/null and systemd cannot be rotated
          sd {scRotation = Nothing}
    parseBackendMap Nothing = HM.empty
    parseBackendMap (Just hmv) = HM.map mkBackends hmv
      where
        mkBackends (Array bes) = catMaybes $ map mkBackend $ Vector.toList bes
        mkBackends _ = [ ]
        mkBackend :: Value -> Maybe BackendKind
        mkBackend = parseMaybe parseJSON
    parseScribeMap Nothing = HM.empty
    parseScribeMap (Just hmv) = HM.map mkScribes hmv
      where
        mkScribes (Array scs) = catMaybes $ map mkScribe $ Vector.toList scs
        mkScribes (String s) = [(s :: ScribeId)]
        mkScribes _ = [ ]
        mkScribe :: Value -> Maybe ScribeId
        mkScribe = parseMaybe parseJSON
    parseSubtraceMap :: Maybe (HM.HashMap Text Value) -> HM.HashMap Text SubTrace
    parseSubtraceMap Nothing = HM.empty
```

```
    parseSubtraceMap (Just hmv) = HM.mapMaybe mkSubtrace hmv
      where
        mkSubtrace :: Value → Maybe SubTrace
        mkSubtrace = parseMaybe parseJSON
    r_hasEKG repr = case (R.hasEKG repr) of
      Nothing → 0
      Just p → p
    r_hasGraylog repr = case (R.hasGraylog repr) of
      Nothing → 0
      Just p → p
    r_hasPrometheus repr = R.hasPrometheus repr
    r_hasGUI repr = case (R.hasGUI repr) of
      Nothing → 0
      Just p → p
    r_forward repr = R.traceForwardTo repr
    r_accept repr = R.traceAcceptAt repr
    r_defaultScribes repr = map (λ(k,n) → pack (show k) <> "::" <> n) (R.defaultScribes repr)
parseAggregatedKindMap :: Maybe (HM.HashMap Text Value) → HM.HashMap LoggerName AggregatedKi
parseAggregatedKindMap Nothing = HM.empty
parseAggregatedKindMap (Just hmv) = HM.mapMaybe mkAggregatedKind hmv
    where
    mkAggregatedKind :: Value → Maybe AggregatedKind
    mkAggregatedKind (String s) = Just $ read $ unpack s
    mkAggregatedKind v = (parseMaybe parseJSON) v
```

Setup empty configuration

```
empty :: IO Configuration
empty = do
  cgref ← newMVar $ ConfigurationInternal
    {cgMinSeverity      = Debug
    ,cgDefRotation      = Nothing
    ,cgMapSeverity      = HM.empty
    ,cgMapSubtrace      = HM.empty
    ,cgOptions          = HM.empty
    ,cgMapBackend       = HM.empty
    ,cgDefBackendKs     = []
    ,cgSetupBackends    = []
    ,cgMapScribe        = HM.empty
    ,cgMapScribeCache   = HM.empty
    ,cgDefScribes       = []
    ,cgSetupScribes     = []
    ,cgMapAggregatedKind = HM.empty
    ,cgDefAggregatedKind = StatsAK
    ,cgMonitors         = HM.empty
    ,cgPortEKG          = 0
    ,cgPortGraylog      = 0
    ,cgBindAddrPrometheus = Nothing
    ,cgPortGUI          = 0
    ,cgForwardTo        = Nothing
```

```
    , cgAcceptAt          = Nothing
    }
  return $ Configuration cgref
```

toRepresentation

```
toRepresentation :: Configuration -> IO R.Representation
toRepresentation (Configuration c) = do
  cfg <- readMVar c
  let portEKG = cgPortEKG cfg
    portGraylog = cgPortGraylog cfg
    portGUI = cgPortGUI cfg
    otherOptions = cgOptions cfg
    defScribes = cgDefScribes cfg
    splitScribeId :: ScribeId -> (ScribeKind, Text)
    splitScribeId x =
      -- "(ScribeId)" = "(ScribeKind) ::  (Filename)"
      let (a, b) = T.breakOn "::" x
      in
        (read $ unpack a, T.drop 2 b)
    createOption :: Text -> (a -> Value) -> HM.HashMap Text a -> HM.HashMap Text Value
    createOption name f hashmap =
      if null hashmap
      then HM.empty
      else HM.singleton name $ Object (HM.map f hashmap)
    toString :: Show a => a -> Value
    toString = String . pack . show
    toObject :: (MEvPreCond, MEvExpr, [MEvAction]) -> Value
    toObject (Nothing, expr, actions) =
      object ["monitor" .= expr
        , "actions" .= actions
        ]
    toObject (Just precond, expr, actions) =
      object ["monitor-if" .= precond
        , "monitor"        .= expr
        , "actions"        .= actions
        ]
    toJSON' :: [ScribeId] -> Value
    toJSON' [sid] = toJSON sid
    toJSON' ss    = toJSON ss
    mapSeverities, mapBackends, mapAggKinds, mapScribes, mapSubtrace, mapMonitors ::
      HM.HashMap Text Value
    mapSeverities = createOption "mapSeverity" toJSON  $ cgMapSeverity  cfg
    mapBackends  = createOption "mapBackends" toJSON  $ cgMapBackend  cfg
    mapAggKinds = createOption "mapAggregatedkinds" toString $ cgMapAggregatedKind cfg
    mapScribes    = createOption "mapScribes"  toJSON' $ cgMapScribe    cfg
    mapSubtrace  = createOption "mapSubtrace" toJSON  $ cgMapSubtrace cfg
    mapMonitors  = createOption "mapMonitors" toObject $ cgMonitors     cfg
  return $
    R.Representation
```

```
    { R.minSeverity     = cgMinSeverity cfg
    , R.rotation        = cgDefRotation cfg
    , R.setupScribes    = cgSetupScribes cfg
    , R.defaultScribes  = map splitScribeId defScribes
    , R.setupBackends   = cgSetupBackends cfg
    , R.defaultBackends = cgDefBackendKs cfg
    , R.hasEKG          = if portEKG ≡ 0 then Nothing else Just portEKG
    , R.hasGraylog      = if portGraylog ≡ 0 then Nothing else Just portGraylog
    , R.hasPrometheus   = cgBindAddrPrometheus cfg
    , R.hasGUI          = if portGUI ≡ 0 then Nothing else Just portGUI
    , R.traceForwardTo  = cgForwardTo cfg
    , R.traceAcceptAt   = cgAcceptAt cfg
    , R.options         = mapSeverities `HM.union`
                          mapBackends `HM.union`
                          mapAggKinds `HM.union`
                          mapSubtrace `HM.union`
                          mapScribes `HM.union`
                          mapMonitors `HM.union`
                          otherOptions
    }
```

Export Configuration into a file

Converts Configuration into the form of *Representation* and writes it to the given file.

```
exportConfiguration :: Configuration → FilePath → IO ()
exportConfiguration cfg file = do
  representation ← toRepresentation cfg
  Yaml.encodeFile file representation
```

Evaluation of FilterTrace

A filter consists of a *DropName* and a list of *UnhideNames*. If the context name matches the *DropName* filter, then at least one of the *UnhideNames* must match the name to have the evaluation of the filters return *True*.

```
findRootSubTrace :: Configuration → LoggerName → IO (Maybe SubTrace)
findRootSubTrace config loggername =
    -- Try to find SubTrace by provided name.
    let find_s :: [Text] → IO (Maybe SubTrace)
        find_s [] = return Nothing
        find_s lnames = findSubTrace config (T.intercalate "." lnames) ≫= λcase
          Just subtrace → return $ Just subtrace
          Nothing → find_s (init lnames)
    in find_s $ T.split (≡ '.') loggername
testSubTrace :: Configuration → LoggerName → LogObject a → IO (Maybe (LogObject a))
testSubTrace config loggername lo = do
    subtrace ← fromMaybe Neutral <$> findRootSubTrace config loggername
    return $ testSubTrace' lo subtrace
  where
    testSubTrace' :: LogObject a → SubTrace → Maybe (LogObject a)
```

```
    testSubTrace' _ NoTrace = Nothing
    testSubTrace' (LogObject _ _ (ObserveOpen _)) DropOpening = Nothing
    testSubTrace' o@(LogObject _ _ (LogValue vname _)) (FilterTrace filters) =
      if evalFilters filters (loggername <> " . " <> vname)
      then Just o
      else Nothing
    testSubTrace' o (FilterTrace filters) =
      if evalFilters filters loggername
      then Just o
      else Nothing
    testSubTrace' o (SetSeverity sev) = Just $ o {loMeta = (loMeta o) {severity = sev}}
    testSubTrace' o _ = Just o-- fallback:  all pass
evalFilters :: [(DropName, UnhideNames)] -> LoggerName -> Bool
evalFilters fs nm =
    all (λ(no, yes) -> if (dropFilter nm no) then (unhideFilter nm yes) else True) fs
  where
    dropFilter :: LoggerName -> DropName -> Bool
    dropFilter name (Drop sel) = (matchName name sel)
    unhideFilter :: LoggerName -> UnhideNames -> Bool
    unhideFilter _ (Unhide [ ]) = False
    unhideFilter name (Unhide us) = any (λsel -> matchName name sel) us
    matchName :: LoggerName -> NameSelector -> Bool
    matchName name (Exact name') = name ≡ name'
    matchName name (StartsWith prefix) = T.isPrefixOf prefix name
    matchName name (EndsWith postfix) = T.isSuffixOf postfix name
    matchName name (Contains name') = T.isInfixOf name' name
```

1.7.27 Cardano.BM.Configuration.Static

Default configuration outputting on *stdout*

```
defaultConfigStdout :: IO CM.Configuration
defaultConfigStdout = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK]
  CM.setDefaultBackends c [KatipBK]
  CM.setSetupScribes c [ScribeDefinition {
        scName = "text"
      ,scFormat = ScText
      ,scKind = StdoutSK
      ,scPrivacy = ScPublic
      ,scRotation = Nothing
      }
      ,ScribeDefinition {
        scName = "json"
      ,scFormat = ScJson
      ,scKind = StdoutSK
      ,scPrivacy = ScPublic
      ,scRotation = Nothing
      }
```

```
    ]
  CM.setDefaultScribes c ["StdoutSK::text"]
  return c
```

Default configuration for testing

```
defaultConfigTesting :: IO CM.Configuration
defaultConfigTesting = do
  c ← CM.empty
  CM.setMinSeverity c Debug
  CM.setSetupBackends c [KatipBK, AggregationBK]
  CM.setDefaultBackends c [KatipBK, AggregationBK]
  CM.setSetupScribes c [ScribeDefinition {
    scName = "nooutput"
    ,scFormat = ScText
    ,scKind = DevNullSK
    ,scPrivacy = ScPublic
    ,scRotation = Nothing
    }
    ]
  CM.setDefaultScribes c ["NullSK::nooutput"]
  return c
```

1.7.28 Cardano.BM.Backend.Switchboard

Switchboard

We are using an *MVar* because we spawn a set of backends that may try to send messages to the switchboard before it is completely setup.

```
type SwitchboardMVar a = MVar (SwitchboardInternal a)
newtype Switchboard a = Switchboard
  {getSB :: SwitchboardMVar a
  }
data SwitchboardInternal a = SwitchboardInternal
  {sbQueue    :: TBQ.TBQueue (LogObject a)
  ,sbDispatch :: Async.Async ()
  ,sbLogBuffer :: !(Cardano.BM.Backend ∘ LogBuffer.LogBuffer a)
  ,sbLogBE    :: !(Cardano.BM.Backend ∘ Log.Log a)
  ,sbBackends :: NamedBackends a
  ,sbRunning :: !SwitchboardStatus
  }
type NamedBackends a = [(BackendKind, Backend a)]
data SwitchboardStatus
  = SwitchboardRunning
  | SwitchboardStopped
  deriving (Eq, Show)
```

Trace that forwards to the Switchboard

Every Trace ends in the Switchboard which then takes care of dispatching the messages to the selected backends.
This Tracer will forward all messages unconditionally to the Switchboard. (currently disabled)

```
mainTrace :: IsEffectuator eff a ⇒ eff a → Tracer IO (LogObject a)
mainTrace = Tracer ∘ effectuate
```

This Tracer will apply to every message the severity filter as defined in the Configuration.

```
mainTraceConditionally :: IsEffectuator eff a ⇒ Configuration → eff a → Trace IO a
mainTraceConditionally config eff = Tracer $ λ(ctxname, item) → do
  mayItem ← Config.testSubTrace config ctxname item
  case mayItem of
    Just itemF@(LogObject _loname meta _) → do
      passSevFilter ← Config.testSeverity config ctxname meta
      when passSevFilter $
        -- pass to backend and insert name
        effectuate eff itemF {loName = ctxname}
    Nothing → pure ()
```

Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher. The switchboard will never block when processing incoming messages ("eager receiver").
The queue is initialized and the message dispatcher launched.

```
instance IsEffectuator Switchboard a where
  effectuate switchboard item = do
    let writequeue :: TBQ.TBQueue (LogObject a) → LogObject a → IO ()
        writequeue q i = do
          nocapacity ← atomically $ TBQ.isFullTBQueue q
          if nocapacity
          then handleOverflow switchboard
          else atomically $ TBQ.writeTBQueue q i
    sb ← readMVar (getSB switchboard)
    if (sbRunning sb) ≡ SwitchboardRunning
      then writequeue (sbQueue sb) item
      else TIO.hPutStrLn stderr "Error: Switchboard is not running, dropping log items!"
  handleOverflow _ = TIO.hPutStrLn stderr "Error: Switchboard's queue full, dropping log ite
```

Switchboard implements Backend functions

Switchboard is an IsBackend

```
instance (FromJSON a, ToJSON a) ⇒ IsBackend Switchboard a where
  bekind _ = SwitchboardBK
  realize cfg = realizeSwitchboard cfg
  unrealize switchboard = unrealizeSwitchboard switchboard
realizeSwitchboard :: (FromJSON a, ToJSON a) ⇒ Configuration → IO (Switchboard a)
```

```
realizeSwitchboard cfg = do
  -- we setup LogBuffer explicitly so we can access it as a Backend and as LogBuffer
  logbuf :: Cardano.BM.Backend ∘ LogBuffer.LogBuffer a ← Cardano.BM.Backend ∘ LogBuffer.realize cfg
  katipBE :: Cardano.BM.Backend ∘ Log.Log a ← Cardano.BM.Backend ∘ Log.realize cfg
  let spawnDispatcher :: Switchboard a → TBQ.TBQueue (LogObject a) → IO (Async.Async ())
      spawnDispatcher switchboard queue =
        let sendMessage nli befilter = do
              let name = case nli of
                           LogObject loname _ (LogValue valueName _) →
                             loname <> " . " <> valueName
                           LogObject loname _ _ → loname
              selectedBackends ← getBackends cfg name
              let selBEs = befilter selectedBackends
              withMVar (getSB switchboard) $ λsb →
                forM_ (sbBackends sb) $ λ(bek, be) →
                  when (bek ∈ selBEs) (bEffectuate be nli)
            qProc = do
              -- read complete queue at once and process items
              nlis ← atomically $ do
                r ← TBQ.flushTBQueue queue
                when (null r) retry
                return r
              let processItem nli@(LogObject loname _ loitem) = do
                    Config.findSubTrace cfg loname ⋙= λcase
                      Just (TeeTrace sndName) →
                        atomically $ TBQ.writeTBQueue queue $ nli {loName = loname <> " . " <> sndName
                      _ → return ()
                    case loitem of
                      KillPill → do
                        -- each of the backends will be terminated sequentially
                        withMVar (getSB switchboard) $ λsb →
                          forM_ (sbBackends sb) (λ(_, be) → bUnrealize be)
                        -- all backends have terminated
                        return False
                      (AggregatedMessage _) → do
                        sendMessage nli (filter (≢ AggregationBK))
                        return True
                      (MonitoringEffect (MonitorAlert _)) → do
                        sendMessage nli (filter (≢ MonitoringBK))
                        return True
                      (MonitoringEffect (MonitorAlterGlobalSeverity sev)) → do
                        setMinSeverity cfg sev
                        return True
                      (MonitoringEffect (MonitorAlterSeverity loggerName sev)) → do
                        setSeverity cfg loggerName (Just sev)
                        return True
                      (Command (DumpBufferedTo bk)) → do
                        msgs ← Cardano.BM.Backend ∘ LogBuffer.readBuffer logbuf
                        forM_ msgs (λ(lonm, lobj) → sendMessage (lobj {loName = lonm}) (const [bk]))
                        return True
```

```
                        _ → do
                          sendMessage nli id
                          return True
                res ← mapM processItem nlis
                when (and res) $ qProc
            in
            Async.async qProc
    # ifdef PERFORMANCE_TEST_QUEUE
      let qSize = 1000000
    # else
      let qSize = 2048
    # endif
      q ← atomically $ TBQ.newTBQueue qSize
      sbref ← newEmptyMVar
      let sb :: Switchboard a = Switchboard sbref
      backends ← getSetupBackends cfg
      bs0 ← setupBackends backends cfg sb
      bs1 ← return (LogBufferBK, MkBackend
                        {bEffectuate = Cardano.BM.Backend ∘ LogBuffer.effectuate logbuf
                        ,bUnrealize = Cardano.BM.Backend ∘ LogBuffer.unrealize logbuf
                        })
      bs2 ← return (KatipBK, MkBackend
                        {bEffectuate = Cardano.BM.Backend ∘ Log.effectuate katipBE
                        ,bUnrealize = Cardano.BM.Backend ∘ Log.unrealize katipBE
                        })
      let bs = bs2 : bs1 : bs0
      dispatcher ← spawnDispatcher sb q
      -- link the given Async to the current thread, such that if the Async
      -- raises an exception, that exception will be re-thrown in the current
      -- thread, wrapped in ExceptionInLinkedThread.
      Async.linkOnly (¬ ∘ isBlockedIndefinitelyOnSTM) dispatcher
      -- Modify the internal state of the switchboard, the switchboard
      -- is now running.
      putMVar sbref $ SwitchboardInternal
        {sbQueue = q
        ,sbDispatch = dispatcher
        ,sbLogBuffer = logbuf
        ,sbLogBE = katipBE
        ,sbBackends = bs
        ,sbRunning = SwitchboardRunning
        }
      return sb
unrealizeSwitchboard :: Switchboard a → IO ()
unrealizeSwitchboard switchboard = do
    -- Here we are doing a modification to send the "kill pill"
    -- to the queue and we are waiting for the dispather to exit.
    -- At the end, either return the result or throw an exception.
    dispatcher ← withMVar (getSB switchboard) $ λsb → do
      let dispatcher = sbDispatch sb
```

```
      let queue      = sbQueue sb
      -- Create terminating item, the "kill pill".
      lo ← LogObject <$> pure "kill.switchboard"
                     <*> (mkLOMeta Warning Confidential)
                     <*> pure KillPill
      -- Send terminating item to the queue.
      atomically $ TBQ.writeTBQueue queue lo
      -- Return the dispatcher.
      return dispatcher
    -- Wait for the dispatcher to exit.
    res ← Async.waitCatch dispatcher
    -- Either raise an exception or return the result.
    either throwM return res
    -- Modify the state in the end so we signal that the switchboard is shut down.
    _ ← withMVar (getSB switchboard) (λsb → return $ sb {sbRunning = SwitchboardStopped})
    pure ()
isBlockedIndefinitelyOnSTM :: SomeException → Bool
isBlockedIndefinitelyOnSTM e =
  isJust (fromException e :: Maybe BlockedIndefinitelyOnSTM)
```

Integrate with external backend

```
addUserDefinedBackend :: Switchboard a → Backend a → Text → IO ()
addUserDefinedBackend switchboard be name =
  modifyMVar_ (getSB switchboard) $ λsb →
    return $ sb {sbBackends = (UserDefinedBK name, be) : sbBackends sb}
```

Integrate with external backend

```
addExternalBackend :: Switchboard a → Backend a → BackendKind → IO ()
addExternalBackend switchboard be bk =
  modifyMVar_ (getSB switchboard) $ λsb →
    return $ sb {sbBackends = (bk, be) : sbBackends sb}
```

Integrate with external *katip* scribe

```
addExternalScribe :: Switchboard a → K.Scribe → Text → IO ()
addExternalScribe switchboard sc name =
  withMVar (getSB switchboard) $ λsb →
    Cardano.BM.Backend ∘ Log.registerScribe (sbLogBE sb) sc name
```

Waiting for the switchboard to terminate

```
waitForTermination :: Switchboard a → IO ()
waitForTermination switchboard =
```

```
    tryReadMVar (getSB switchboard) >>= λcase
      Nothing → return ()
      Just sb → Async.waitCatch (sbDispatch sb) >> return ()
```

Reading the buffered log messages

```
readLogBuffer :: Switchboard a → IO [(LoggerName, LogObject a)]
readLogBuffer switchboard = do
  sb ← readMVar (getSB switchboard)
  Cardano.BM.Backend ∘ LogBuffer.readBuffer (sbLogBuffer sb)
```

Realizing the backends according to configuration

```
setupBackends :: (FromJSON a, ToJSON a)
  ⇒ [BackendKind]
  → Configuration
  → Switchboard a
  → IO [(BackendKind, Backend a)]
setupBackends bes c sb = setupBackendsAcc bes [ ]
  where
    setupBackendsAcc [ ] acc = return acc
    setupBackendsAcc (bk : r) acc = do
      setupBackend' bk c sb >>= λcase
        Nothing → setupBackendsAcc r acc
        Just be → setupBackendsAcc r ((bk, be) : acc)
setupBackend' :: (FromJSON a, ToJSON a) ⇒ BackendKind → Configuration → Switchboard a → IO (Mayb
setupBackend' SwitchboardBK _ _ = fail "cannot instantiate a further Switchboard"
setupBackend' (UserDefinedBK _) _ _ = fail "cannot instantiate an user-defined backend"
setupBackend' MonitoringBK _ _ = return Nothing
setupBackend' AggregationBK _ _ = return Nothing
setupBackend' EditorBK _ _ = return Nothing
setupBackend' GraylogBK _ _ = return Nothing
setupBackend' EKGViewBK _ _ = return Nothing
setupBackend' KatipBK _ _ = return Nothing
setupBackend' LogBufferBK _ _ = return Nothing
setupBackend' TraceAcceptorBK _ _ = return Nothing
setupBackend' TraceForwarderBK _ _ = return Nothing
```

1.7.29 Cardano.BM.Backend.Log

Internal representation

```
type LogMVar = MVar LogInternal
newtype Log a = Log
  {getK :: LogMVar}
data LogInternal = LogInternal
  {kLogEnv :: K.LogEnv
  ,configuration :: Config.Configuration}
```

Log implements *effectuate*

```
instance ToJSON a ⇒ IsEffectuator Log a where
  effectuate katip item = do
      let logMVar = getK katip
      -- TODO cache scribe lists, update every n minutes
      c ← configuration <$> readMVar logMVar
      setupScribes ← getSetupScribes c
      selscribes ← getScribes c (loName item)
      let selscribesFiltered =
            case item of
              LogObject _ (LOMeta _ _ _ _ Confidential) (LogMessage _)
                → removePublicScribes setupScribes selscribes
              _ → selscribes
      forM_ (onlyScribes ScText setupScribes selscribesFiltered) $ λsc → passText sc katip item
      forM_ (onlyScribes ScJson setupScribes selscribesFiltered) $ λsc → passStrx sc katip item
    where
      removePublicScribes allScribes = filter $ λscn →
        let (_,nameD) = T.breakOn "::" scn
            name = T.drop 2 nameD-- drop "::" from the start of name
        in
        case find (λscd → scName scd ≡ name) allScribes of
          Nothing → False
          Just scribe → scPrivacy scribe ≡ ScPrivate
      onlyScribes :: ScribeFormat → [ScribeDefinition] → [Text] → [Text]
      onlyScribes form allScribes = filter $ λscn →
        case find (λscd → (pack $ show $ scKind scd) <> "::" <> (scName scd) ≡ scn) allScribes of
          Nothing → False
          Just scribe → scFormat scribe ≡ form
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Katip's queue full, dropping log items!"
```

Log implements backend functions

```
instance (ToJSON a, FromJSON a) ⇒ IsBackend Log a where
  bekind _ = KatipBK
  realize config = do
    let updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
        updateEnv le timer =
          le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
    ver ← Config.getTextOptionOrDefault config "appversion" "<unknown>"
    commit ← Config.getTextOptionOrDefault config "appcommit" "00000"
    le0 ← K.initLogEnv
      (K.Namespace mempty)
      (fromString $ unpack ver <> ":" <> take 5 (unpack commit))
    -- request a new time 'getCurrentTime' at most 100 times a second
    timer ← mkAutoUpdate defaultUpdateSettings {updateAction = getCurrentTime, updateFreq = 10000}
    let le1 = updateEnv le0 timer
    scribes ← getSetupScribes config
    le ← registerScribes scribes le1
    kref ← newMVar $ LogInternal le config
```

```
    return $ Log kref
  unrealize katip = do
    le ← withMVar (getK katip) $ λk → return (kLogEnv k)
    void $ K.closeScribes le
```

Create and register *katip* scribes

```
registerScribe :: Log a → K.Scribe → ScribeId → IO ()
registerScribe katip scr name =
    modifyMVar_ (getK katip) $ λk → do
      newenv ← K.registerScribe name scr scribeSettings (kLogEnv k)
      return $ k {kLogEnv = newenv}
scribeSettings :: KC.ScribeSettings
scribeSettings =
    let bufferSize = 5000-- size of the queue (in log items)
    in
    KC.ScribeSettings bufferSize
registerScribes :: [ScribeDefinition] → K.LogEnv → IO K.LogEnv
registerScribes defscs le =
    foldM withScribeInEnv le defscs
  where
    withScribeInEnv :: K.LogEnv → ScribeDefinition → IO K.LogEnv
    withScribeInEnv le' defsc = do
      let kind = scKind defsc
          sctype = scFormat defsc
          name = scName defsc
          rotParams = scRotation defsc
          name' = pack (show kind) <> "::" <> name
      scribe ← createScribe kind sctype name rotParams
      case scribe of
        Just scr → K.registerScribe name' scr scribeSettings le'
        Nothing → return le'
    createScribe FileSK ScText name rotParams = Just <$> mkTextFileScribe
      rotParams
      (FileDescription $ unpack name)
      False
    createScribe FileSK ScJson name rotParams = Just <$> mkJsonFileScribe
      rotParams
      (FileDescription $ unpack name)
      False
    createScribe StdoutSK sctype _ _ = Just <$> mkStdoutScribe sctype
    createScribe StderrSK sctype _ _ = Just <$> mkStderrScribe sctype
    createScribe DevNullSK _ _ _ = Just <$> mkDevNullScribe
    createScribe JournalSK _ _ _ = return Nothing
    createScribe UserDefinedSK ty nm rot = createScribe FileSK ty nm rot

example :: IO ()
example = do
  config ← Config.setup "from_some_path.yaml"
```

```
k ← setup config
meta ← mkLOMeta Info Public
passText (pack (show StdoutSK)) k $ LogObject
  {loName = ["test"]
  ,loMeta = meta
  ,loContent = LogMessage "Hello!"
  }
meta' ← mkLOMeta Info Public
passStrx (pack (show StdoutSK)) k $ LogObject
  {loName = ["test"]
  ,loMeta = meta'
  ,loContent = LogValue "cpu-no" 1
  }
```

Needed instances for *katip*:

```
deriving instance ToJSON a ⇒ K.ToObject (LogObject a)
deriving instance K.ToObject Text
deriving instance ToJSON a ⇒ K.ToObject (LOContent a)
deriving instance K.ToObject Value
deriving instance ToJSON a ⇒ K.ToObject (Maybe (LOContent a))
instance (ToJSON a, ToJSON b, K.ToObject a, K.ToObject b) ⇒ K.ToObject (Maybe (Either a b)) where
  toObject Nothing = mempty
  toObject (Just (Left x)) = KC.toObject x
  toObject (Just (Right x)) = KC.toObject x
instance (ToJSON a, ToJSON b, K.ToObject a, K.ToObject b) ⇒ KC.LogItem (Maybe (Either a b)) where
  payloadKeys _ _ = KC.AllKeys
instance ToJSON a ⇒ KC.LogItem (LogObject a) where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem Text where
  payloadKeys _ _ = KC.AllKeys
instance ToJSON a ⇒ KC.LogItem (Maybe (LOContent a)) where
  payloadKeys _ _ = KC.AllKeys
```

Entering structured log item into katip's queue

```
passStrx :: forall a ∘ ToJSON a ⇒ ScribeId → Log a → LogObject a → IO ()
passStrx backend katip (LogObject loname lometa loitem) = do
  env ← kLogEnv <$> readMVar (getK katip)
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName, (KC.ScribeHandle _ shChan)) →
      -- check start of name to match ScribeKind
      when (backend `isPrefixOf` scName) $ do
        let sev = severity lometa
            payload :: Maybe (Either (LOContent a) Value)
            payload = case loitem of
              (LogMessage _) → Just $ Left loitem
              (LogError _) → Just $ Left loitem
              (LogStructured s) → Just $ Right (Object s)
              (LogStructuredText s _t) → Just $ Right (Object s)
              (LogValue _ _) → Just $ Left loitem
```

```
        (ObserveDiff _) → Just $ Left loitem
        (ObserveOpen _) → Just $ Left loitem
        (ObserveClose _) → Just $ Left loitem
        (AggregatedMessage _) → Just $ Left loitem
        (MonitoringEffect _) → Just $ Left loitem
        KillPill → Nothing
        Command _ → Nothing
    unless (isNothing payload) $ do
      let threadIdText = KC.ThreadIdText $ tid lometa
      let itemTime = tstamp lometa
      let localname = [loname]
      let itemKatip = K.Item {
        _itemApp       = env ^. KC.logEnvApp
        , _itemEnv     = env ^. KC.logEnvEnv
        , _itemSeverity = sev2klog sev
        , _itemThread  = threadIdText
        , _itemHost    = unpack $ hostname lometa
        , _itemProcess = env ^. KC.logEnvPid
        , _itemPayload = payload
        , _itemMessage = ""
        , _itemTime    = itemTime
        , _itemNamespace = env ^. KC.logEnvApp <> K.Namespace localname
        , _itemLoc     = Nothing
        }
      void $ atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
```

Entering textual log item into katip's queue

```
passText :: forall a ∘ ToJSON a ⇒ ScribeId → Log a → LogObject a → IO ()
passText backend katip (LogObject loname lometa loitem) = do
  env ← kLogEnv <$> readMVar (getK katip)
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName, (KC.ScribeHandle _ shChan)) →
      -- check start of name to match ScribeKind
      when (backend `isPrefixOf` scName) $ do
        let sev = severity lometa
        msg :: Text
        msg = case loitem of
          (LogMessage logItem) → case toJSON logItem of
            (String m) → m
            m          → TL.toStrict $ encodeToLazyText m
          (LogError m) → m
          (LogStructured o) → TL.toStrict (encodeToLazyText o)
          (LogStructuredText _o m) → m
          (LogValue name value) →
            if name ≡ ""
            then pack (showSI value)
            else name <> "  =  " <> pack (showSI value)
          (ObserveDiff _) → TL.toStrict (encodeToLazyText loitem)
          (ObserveOpen _) → TL.toStrict (encodeToLazyText loitem)
```

```
        (ObserveClose _) → TL.toStrict (encodeToLazyText loitem)
        (AggregatedMessage aggregated) →
          T.concat $ flip map aggregated $ λ(name, agg) →
            "\n" <> name <> ":  " <> pack (show agg)
        (MonitoringEffect _) →
          TL.toStrict (encodeToLazyText loitem)
        KillPill → ""
        Command _ → ""
    unless (msg ≡ "") $ do
      let threadIdText = KC.ThreadIdText $ tid lometa
      let itemTime = tstamp lometa
      let localname = [loname]
      let itemKatip = K.Item {
        _itemApp       = env ^. KC.logEnvApp
        , _itemEnv     = env ^. KC.logEnvEnv
        , _itemSeverity = sev2klog sev
        , _itemThread  = threadIdText
        , _itemHost    = unpack $ hostname lometa
        , _itemProcess = env ^. KC.logEnvPid
        , _itemPayload = ()
        , _itemMessage = K.logStr msg
        , _itemTime    = itemTime
        , _itemNamespace = env ^. KC.logEnvApp <> K.Namespace localname
        , _itemLoc     = Nothing
        }
      void $ atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
```

Scribes

The handles to *stdout* and *stderr* will be duplicated because on exit *katip* will close them otherwise.

```
mkStdoutScribe :: ScribeFormat → IO K.Scribe
mkStdoutScribe ScText = do
    stdout' ← hDuplicate stdout
    mkTextFileScribeH stdout' True
mkStdoutScribe ScJson = do
    stdout' ← hDuplicate stdout
    mkJsonFileScribeH stdout' True

mkStderrScribe :: ScribeFormat → IO K.Scribe
mkStderrScribe ScText = do
    stderr' ← hDuplicate stderr
    mkTextFileScribeH stderr' True
mkStderrScribe ScJson = do
    stderr' ← hDuplicate stderr
    mkJsonFileScribeH stderr' True

mkDevNullScribe :: IO K.Scribe
mkDevNullScribe = do
    let logger _ = pure ()
    pure $ K.Scribe logger (pure ()) (pure ∘ const True)

type Formatter a = K.LogItem a ⇒ Handle → Rendering a → IO Int
```

```
textFormatter, jsonFormatter :: Formatter a
textFormatter h r =
  let (len, msg) = renderTextMsg r
  in (TIO.hPutStrLn h $! msg) >> pure len
jsonFormatter h r =
  let (len, msg) = renderJsonMsg r
  in (TIO.hPutStrLn h $! msg) >> pure len

mkTextFileScribeH, mkJsonFileScribeH :: Handle -> Bool -> IO K.Scribe
mkTextFileScribeH = mkFileScribeH textFormatter
mkJsonFileScribeH = mkFileScribeH jsonFormatter

mkTextFileScribe, mkJsonFileScribe :: Maybe RotationParameters -> FileDescription -> Bool -> IO K.Scribe
mkTextFileScribe = mkFileScribe textFormatter
mkJsonFileScribe = mkFileScribe jsonFormatter

mkFileScribeH
    :: (forall a . Formatter a)
    -> Handle
    -> Bool
    -> IO K.Scribe
mkFileScribeH formatter h colorize = do
    hSetBuffering h LineBuffering
    locklocal <- newMVar ()
    let logger :: forall a . K.LogItem a => K.Item a -> IO ()
        logger item = withMVar locklocal $ \_ ->
                        void $ formatter h (Rendering colorize K.V0 item)
    pure $ K.Scribe logger (hClose h) (pure . const True)

data Rendering a = Rendering {colorize :: Bool
  , verbosity :: K.Verbosity
  , logitem                   :: K.Item a
  }

renderTextMsg :: (K.LogItem a) => Rendering a -> (Int, TL.Text)
renderTextMsg r =
    let li = logitem r
        m = toLazyText $ formatItem (colorize r) (verbosity r) $
          case KC._itemMessage li of
            K.LogStr "" -> li {KC._itemMessage = K.logStr . encode . K.toObject $ KC._itemPayload li}
            _ -> li
    in (fromIntegral $ TL.length m, m)

renderJsonMsg :: (K.LogItem a) => Rendering a -> (Int, TL.Text)
renderJsonMsg r =
    let li = logitem r
        li' = li {KC._itemMessage = ""}
        m' = encodeToLazyText $ trimTime $ K.itemJson (verbosity r) li'
    in (fromIntegral $ TL.length m', m')

-- keep only two digits for the fraction of seconds
trimTime :: Value -> Value
trimTime (Object o) = Object $ HM.adjust
                                 keep2Decimals
                                 "at"
                                 o
  where
```

```
    keep2Decimals :: Value → Value
    keep2Decimals v = case fromJSON v of
                Success (utct :: UTCTime) →
                  String $ pack $ formatTime defaultTimeLocale jformat utct
                _ → v
    jformat :: String
    jformat = "%FT%T%2QZ"
trimTime v = v

mkFileScribe
    :: (forall a ∘ K.LogItem a ⇒ Handle → Rendering a → IO Int)
    → Maybe RotationParameters
    → FileDescription
    → Bool
    → IO K.Scribe
mkFileScribe formatter (Just rotParams) fdesc colorize = do
    let prefixDir = prefixPath fdesc
    createDirectoryIfMissing True prefixDir
      `catchIO` prtoutException ("cannot log prefix directory: " ++ prefixDir)
    let fpath = filePath fdesc
    trp ← initializeRotator rotParams fpath
    scribestate ← newMVar trp-- triple of (handle), (bytes remaining), (rotate time)
    -- sporadically remove old log files - every 10 seconds
    cleanup ← mkAutoUpdate defaultUpdateSettings {
                                updateAction = cleanupRotator rotParams fpath
                        ,updateFreq = 10000000
                        }
    let finalizer :: IO ()
        finalizer = withMVar scribestate $
                                λ(h, _, _) → hClose h
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
        logger item =
          modifyMVar_ scribestate $ λ(h, bytes, rottime) → do
            byteswritten ← formatter h (Rendering colorize K.V0 item)
            -- remove old files
            cleanup
            -- detect log file rotation
            let bytes' = bytes – toInteger byteswritten
            let tdiff' = round $ diffUTCTime rottime (K._itemTime item)
            if bytes' < 0 ∨ tdiff' < (0 :: Integer)
              then do -- log file rotation
                hClose h
                (h2, bytes2, rottime2) ← evalRotator rotParams fpath
                return (h2, bytes2, rottime2)
              else
                return (h, bytes', rottime)
    return $ K.Scribe logger finalizer (pure ∘ const True)
-- log rotation disabled.
mkFileScribe formatter Nothing fdesc colorize = do
    let prefixDir = prefixPath fdesc
    createDirectoryIfMissing True prefixDir
      `catchIO` prtoutException ("cannot create prefix directory: " ++ prefixDir)
```

```
    let fpath = filePath fdesc
    h ← catchIO (openFile fpath WriteMode) $
                λe → do
                  prtoutException ("error while opening log: " ++ fpath) e
                  -- fallback to standard output in case of exception
                  return stdout
    hSetBuffering h LineBuffering
    scribestate ← newMVar h
    let finalizer :: IO ()
        finalizer = withMVar scribestate hClose
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
        logger item =
          withMVar scribestate $ λhandler →
            void $ formatter handler (Rendering colorize K.V0 item)
    return $ K.Scribe logger finalizer (pure ∘ const True)


formatItem :: Bool → K.Verbosity → K.Item a → Builder
formatItem withColor _verb K.Item {..} =
    fromText header <>
    fromText "  " <>
    brackets (fromText timestamp) <>
    fromText "  " <>
    KC.unLogStr _itemMessage
  where
    header = colorBySeverity _itemSeverity $
      "[" <> hostname <> mconcat namedcontext <> ":" <> severity <> ":" <> threadid <> "]"
    hostname | _itemHost ≡ "" = ""
      | otherwise = pack _itemHost <> ":"
    namedcontext = KC.intercalateNs _itemNamespace
    severity = KC.renderSeverity _itemSeverity
    threadid = KC.getThreadIdText _itemThread
    timestamp = pack $ formatTime defaultTimeLocale tsformat _itemTime
    tsformat :: String
    tsformat = "%F %T%2Q %Z"
    colorBySeverity s m = case s of
      K.EmergencyS → red m
      K.AlertS     → red m
      K.CriticalS  → red m
      K.ErrorS     → red m
      K.NoticeS    → magenta m
      K.WarningS → yellow m
      K.InfoS      → blue m
      _ → m
    red = colorize "31"
    yellow = colorize "33"
    magenta = colorize "35"
    blue = colorize "34"
    colorize c m
      | withColor = "\ESC[" <> c <> "m" <> m <> "\ESC[0m"
      | otherwise = m
-- translate Severity to Log.Severity
```

```
sev2klog :: Severity → K.Severity
sev2klog = λcase
    Debug     → K.DebugS
    Info      → K.InfoS
    Notice    → K.NoticeS
    Warning   → K.WarningS
    Error     → K.ErrorS
    Critical  → K.CriticalS
    Alert     → K.AlertS
    Emergency → K.EmergencyS
```

```
newtype FileDescription = FileDescription {filePath :: FilePath}
  deriving (Show)
prefixPath :: FileDescription → FilePath
prefixPath = takeDirectory ∘ filePath
```

1.7.30 Cardano.BM.Backend.LogBuffer

Structure of LogBuffer

```
newtype LogBuffer a = LogBuffer
  {getLogBuf :: LogBufferMVar a}
type LogBufferMVar a = MVar (LogBufferInternal a)
data LogBufferInternal a = LogBufferInternal
  {logBuffer :: !(LogBufferMap a)
  }
```

Relation from log context name to log item

We keep the latest LogObject from a log context in a *HashMap*.

```
type LogBufferMap a = HM.HashMap LoggerName (LogObject a)
```

Read out the latest LogObjects

Returns a list of the maps keys and values. And, resets the map.

```
readBuffer :: LogBuffer a → IO [(LoggerName, LogObject a)]
readBuffer buffer =
  modifyMVar (getLogBuf buffer) $ λcurrentBuffer → do
    let !l = HM.toList $ logBuffer currentBuffer
    return (LogBufferInternal HM.empty, l)
```

LogBuffer is an effectuator

Function *effectuate* is called to pass in a LogObject for log buffering.

```
instance IsEffectuator LogBuffer a where
  effectuate buffer lo@(LogObject loname _lometa (LogValue lvname _lvalue)) =
```

```
    modifyMVar_ (getLogBuf buffer) $ λcurrentBuffer →
      return $! LogBufferInternal $ HM.insert ("#buffered." <> loname <> "." <> lvname) lo $ logBuffer cu
  effectuate buffer lo@(LogObject loname _lometa _logitem) =
    modifyMVar_ (getLogBuf buffer) $ λcurrentBuffer →
      return $! LogBufferInternal $ HM.insert ("#buffered." <> loname) lo $ logBuffer currentBuffer
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: overflow in LogBuffer, dropping log items
```

LogBuffer implements Backend functions

LogBuffer is an IsBackend

```
instance FromJSON a ⇒ IsBackend LogBuffer a where
  bekind _ = LogBufferBK
  realize _ =
    let emptyBuffer = LogBufferInternal HM.empty
    in
    LogBuffer <$> newMVar emptyBuffer
  unrealize _ = return ()
```

1.7.31 Cardano.BM.Backend.Aggregation

Plugin definition

```
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace.Trace IO a → s a → IO (Plugin a)
plugin config trace sb = do
  be :: Cardano.BM.Backend ∘ Aggregation.Aggregation a ← realizefrom config trace sb
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Internal representation

```
type AggregationMVar a = MVar (AggregationInternal a)
newtype Aggregation a = Aggregation
  {getAg :: AggregationMVar a}
data AggregationInternal a = AggregationInternal
  {agQueue :: TBQ.TBQueue (Maybe (LogObject a))
  ,agDispatch :: Async.Async ()
  }
```

Relation from context name to aggregated statistics

We keep the aggregated values (Aggregated) for a named context in a *HashMap*.

```
type AggregationMap = HM.HashMap Text Aggregated
```

Aggregation implements *effectuate*

Aggregation is an IsEffectuator Enter the log item into the Aggregation queue.

```
instance IsEffectuator Aggregation a where
  effectuate agg item = do
    ag ← readMVar (getAg agg)
    nocapacity ← atomically $ TBQ.isFullTBQueue (agQueue ag)
    if nocapacity
    then handleOverflow agg
    else atomically $ TBQ.writeTBQueue (agQueue ag) $! Just item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Aggregation's queue full, dropping log ite
```

Aggregation implements Backend functions

Aggregation is an IsBackend

```
instance FromJSON a ⇒ IsBackend Aggregation a where
  bekind _ = AggregationBK
  realize _ = fail "Aggregation cannot be instantiated by 'realize'"
  realizefrom config trace _ = do
    aggref ← newEmptyMVar
# ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
# else
    let qSize = 2048
# endif
    aggregationQueue ← atomically $ TBQ.newTBQueue qSize
    dispatcher ← spawnDispatcher config HM.empty aggregationQueue trace
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar aggref $ AggregationInternal aggregationQueue dispatcher
    return $ Aggregation aggref
  unrealize aggregation = do
    let clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getAg aggregation) (λag →
      return (agDispatch ag, agQueue ag))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    -- TODO add a timeout to waitCatch in order
    -- to be sure that it will finish
    res ← Async.waitCatch dispatcher
    either throwM return res
    (clearMVar ∘ getAg) aggregation
```

Asynchronously reading log items from the queue and their processing

```
spawnDispatcher :: Configuration
                -> AggregationMap
                -> TBQ.TBQueue (Maybe (LogObject a))
                -> Trace.Trace IO a
                -> IO (Async.Async ())
spawnDispatcher conf aggMap aggregationQueue basetrace =
    let trace = Trace.appendName "#aggregation" basetrace
    in
    Async.async $ qProc trace aggMap
  where
    {- lazy qProc -}
    qProc trace aggregatedMap =
      processQueue
        aggregationQueue
        processAggregated
        (trace, aggregatedMap)
        (\_ -> pure ())
    processAggregated lo@(LogObject loname lm _) (trace, aggregatedMap) = do
      (updatedMap, aggregations) <- update lo aggregatedMap trace
      sendAggregated trace loname (severity lm) aggregations
      return (trace, updatedMap)
    createNupdate :: Text -> Measurable -> LOMeta -> AggregationMap -> IO (Either Text Aggregated)
    createNupdate name value lme agmap = do
      case HM.lookup name agmap of
        Nothing -> do
          -- if Aggregated does not exist; initialize it.
          aggregatedKind <- getAggregatedKind conf name
          case aggregatedKind of
            StatsAK -> return $ Right (singletonStats value)
            EwmaAK aEWMA ->
              return $ AggregatedEWMA <$> ewma (EmptyEWMA aEWMA) value
        Just a -> return $ updateAggregation value a (utc2ns $ tstamp lme)
    update :: LogObject a
      -> AggregationMap
      -> Trace.Trace IO a
      -> IO (AggregationMap, [(Text, Aggregated)])
    update (LogObject loname lme (LogValue iname value)) agmap trace = do
      let fullname = loname <> "." <> iname
      eitherAggregated <- createNupdate fullname value lme agmap
      case eitherAggregated of
        Right aggregated -> do
          sendAggregated trace fullname (severity lme) [(iname, aggregated)]
          let updatedMap = HM.alter (const $ Just $ aggregated) fullname agmap
          return (updatedMap, [])
        Left w -> do
          let trace' = Trace.appendName "update" trace
          Trace.traceNamedObject trace' =<<
            (,) <$> liftIO (mkLOMeta Warning Public)
                <*> pure (LogError w)
```

```
        return (agmap, [])
update (LogObject loname lme (ObserveDiff counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (loname, "diff") agmap [] trace
update (LogObject loname lme (ObserveOpen counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (loname, "open") agmap [] trace
update (LogObject loname lme (ObserveClose counterState)) agmap trace =
  updateCounters (csCounters counterState) lme (loname, "close") agmap [] trace
update (LogObject loname lme (LogMessage _)) agmap trace = do
  let iname = pack $ show (severity lme)
  let fullname = loname <> "." <> iname
  eitherAggregated ← createNupdate fullname (PureI 0) lme agmap
  case eitherAggregated of
    Right aggregated → do
      sendAggregated trace fullname (severity lme) [(iname, aggregated)]
      let updatedMap = HM.alter (const $ Just $ aggregated) fullname agmap
      return (updatedMap, [])
    Left w → do
      let trace' = Trace.appendName "update" trace
      Trace.traceNamedObject trace' =≪
        (,) <$> liftIO (mkLOMeta Warning Public)
            <*> pure (LogError w)
      return (agmap, [])
-- everything else
update _ agmap _ = return (agmap, [])
updateCounters :: [Counter]
               → LOMeta
               → (LoggerName, LoggerName)
               → AggregationMap
               → [(Text, Aggregated)]
               → Trace.Trace IO a
               → IO (AggregationMap, [(Text, Aggregated)])
updateCounters [] _ _ aggrMap aggs _ = return (aggrMap, aggs)
updateCounters (counter : cs) lme (logname, msgname) aggrMap aggs trace = do
  let name = cName counter
    subname = msgname <> "." <> (nameCounter counter) <> "." <> name
    fullname = logname <> "." <> subname
    value = cValue counter
  eitherAggregated ← createNupdate fullname value lme aggrMap
  case eitherAggregated of
    Right aggregated → do
      let namedAggregated = (subname, aggregated)
          updatedMap = HM.alter (const $ Just $ aggregated) fullname aggrMap
      updateCounters cs lme (logname, msgname) updatedMap (namedAggregated : aggs) trace
    Left w → do
      let trace' = Trace.appendName "updateCounters" trace
      Trace.traceNamedObject trace' =≪
        (,) <$> liftIO (mkLOMeta Warning Public)
            <*> pure (LogError w)
      updateCounters cs lme (logname, msgname) aggrMap aggs trace
sendAggregated :: Trace.Trace IO a → Text → Severity → [(Text, Aggregated)] → IO ()
```

```
sendAggregated _trace _loname _sev [ ] = pure ()
sendAggregated trace loname sev v = do
  meta ← mkLOMeta sev Public
  traceWith trace (loname, LogObject mempty meta (AggregatedMessage v))
```

1.7.32 Cardano.BM.Backend.Editor

This simple configuration editor is accessible through a browser on http://127.0.0.1:13789, or whatever port has been set in the configuration.

A number of maps that relate logging context name to behaviour can be changed. And, most importantly, the global minimum severity that defines the filtering of log messages.

links

The GUI is built on top of *Threepenny-GUI* (http://hackage.haskell.org/package/threepenny-gui). The appearance is due to *w3-css* (https://www.w3schools.com/w3css).

Plugin definition

```
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace IO a → s a → IO (Plugin a)
plugin config trace sb = do
  be :: Cardano.BM.Backend ∘ Editor.Editor a ← realizefrom config trace sb
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Structure of Editor

```
type EditorMVar a = MVar (EditorInternal a)
newtype Editor a = Editor
  {getEd :: EditorMVar a}
data EditorInternal a = EditorInternal
  {edSBtrace :: Trace IO a
  ,edThread :: Async.Async ()
  ,edBuffer  :: !(LogBuffer a)
  }
```

Editor implements Backend functions

Editor is an IsBackend

```
instance (ToJSON a, FromJSON a) ⇒ IsBackend Editor a where
  bekind _ = EditorBK
  realize _ = fail "Editor cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = mdo
    gref ← newEmptyMVar
    let gui = Editor gref
    port ← getGUIport config
```

```
    when (port ⩽ 0) $ fail "cannot create GUI"
    -- local LogBuffer
    logbuf :: Cardano.BM.Backend ∘ LogBuffer.LogBuffer a ← Cardano.BM.Backend ∘ LogBuffer.realize co
    thd ← Async.async $ do
      startGUI defaultConfig {jsPort = Just port
        ,jsAddr                      = Just "127.0.0.1"
        ,jsStatic                    = Just "iohk-monitoring/static"
        ,jsCustomHTML = Just "configuration-editor.html"
        } $ prepare gui config
      `catch` nullSetup sbtrace gref
        EditorInternal
          {edSBtrace = nullTracer
          ,edThread = thd
          ,edBuffer = logbuf
          }
    Async.link thd
    putMVar gref $ EditorInternal
          {edSBtrace = sbtrace
          ,edThread = thd
          ,edBuffer = logbuf
          }
    return gui
    where
      nullSetup
        :: Trace IO a
        → EditorMVar a
        → EditorInternal a
        → SomeException
        → IO ()
      nullSetup trace mvar nullEditor e = do
        meta ← mkLOMeta Error Public
        traceWith trace $ ("#editor.realizeFrom", LogObject "#editor.realizeFrom" meta $
          LogError $ "Editor backend disabled due to initialisation error: " <> (pack $ shou
        _ ← swapMVar mvar nullEditor
        pure ()
  unrealize editor =
    withMVar (getEd editor) $ λed →
      Async.cancel $ edThread ed
```

Editor is an effectuator

Function *effectuate* is called to pass in a LogObject for display in the GUI.

```
instance IsEffectuator Editor a where
  effectuate editor item =
    withMVar (getEd editor) $ λed →
      effectuate (edBuffer ed) item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: overflow in Editor!"
```

Prepare the view

```
data Cmd = Backends | Scribes | Severities | SubTrace | Aggregation | Buffer | ExportConfiguration
  deriving (Enum, Eq, Show, Read)
prepare :: ToJSON a ⇒ Editor a → Configuration → Window → UI ()
prepare editor config window = void $ do
  let commands = [Backends..]
  inputKey   ← UI.input #. "w3-input w3-border" # set UI.size "34"
  inputValue ← UI.input #. "w3-input w3-border" # set UI.size "60"
  outputMsg ← UI.input #. "w3-input w3-border"
  currentCmd ← UI.p #. "current-cmd"
  let performActionOnId anId action =
        getElementById window anId ≫= λcase
          Nothing        → return ()
          Just anElement → action anElement
  let turn      anElement toState     = void $ element anElement # set UI.enabled toState
  let setValueOf anElement aValue = void $ element anElement # set UI.value aValue
  let setClasses classes anElement = void $ element anElement # set UI.class_ classes
  let setError m = setValueOf outputMsg ("ERROR: " ++ m)
  let setMessage m = setValueOf outputMsg m
  let enable anElement = turn anElement True
  let disable anElement = turn anElement False
  let clean   anElement = setValueOf anElement ""
  let cleanAndDisable anElement = clean anElement ≫ disable anElement
  let rememberCurrent cmd = setValueOf currentCmd $ show cmd
  let removeItem Backends  k = CM.setBackends  config k Nothing
      removeItem Severities  k = CM.setSeverity  config k Nothing
      removeItem Scribes     k = CM.setScribes     config k Nothing
      removeItem SubTrace  k = CM.setSubTrace config k Nothing
      removeItem Aggregation k = CM.setAggregatedKind config k Nothing
      removeItem _           _ = pure ()
  let updateItem Backends  k v = case (readMay v :: Maybe [BackendKind]) of
                                   Nothing → setError "parse error on backend list"
                                   Just v' → liftIO $ CM.setBackends config k $ Just v'
      updateItem Severities  k v = case (readMay v :: Maybe Severity) of
                                   Nothing → setError "parse error on severity"
                                   Just v' → liftIO $ CM.setSeverity config k $ Just v'
      updateItem Scribes     k v = case (readMay v :: Maybe [ScribeId]) of
                                   Nothing → setError "parse error on scribe list"
                                   Just v' → liftIO $ CM.setScribes config k $ Just v'
      updateItem SubTrace  k v = case (readMay v :: Maybe SubTrace) of
                                   Nothing → setError "parse error on subtrace"
                                   Just v' → liftIO $ CM.setSubTrace config k $ Just v'
      updateItem Aggregation k v = case (readMay v :: Maybe AggregatedKind) of
                                   Nothing → setError "parse error on aggregated kind"
                                   Just v' → liftIO $ CM.setAggregatedKind config k $ Just v'
      updateItem _           _ _ = pure ()
  disable inputKey
  disable inputValue
```

```
    disable outputMsg

    let saveItemButtonId       = "save-item-button"
    let cancelSaveItemButtonId = "cancel-save-item-button"
    let addItemButtonId        = "add-item-button"
    let outputTableId          = "output-table"

    let addItemButton          = performActionOnId addItemButtonId
    let saveItemButton         = performActionOnId saveItemButtonId
    let cancelSaveItemButton   = performActionOnId cancelSaveItemButtonId
    let cleanOutputTable       = performActionOnId outputTableId $ λt → void $ element t # set children [ ]

    let mkLinkToFile :: String → FilePath → UI Element
        mkLinkToFile str file = UI.anchor # set (attr "href") file
                                          # set (attr "target") "_blank"
                                          #+ [string str]
    let mkSimpleRow :: ToJSON a ⇒ LoggerName → LogObject a → UI Element
        mkSimpleRow n lo@(LogObject _lonm _lometa _lov) = UI.tr #. "itemrow" #+
          [UI.td #+ [string (unpack n)]
          ,UI.td #+ [string $ BS8.unpack $ encode lo]
          ]
    let mkTableRow :: Show t ⇒ Cmd → LoggerName → t → UI Element
        mkTableRow cmd n v = UI.tr #. "itemrow" #+
          [UI.td #+ [string (unpack n)]
          ,UI.td #+ [string (show v)]
          ,UI.td #+
            [do
              b ← UI.button #. "w3-small w3-btn w3-ripple w3-orange edit-item-button"
                    #+ [UI.bold #+ [string "Edit"]]
              on UI.click b $ const $ do
                saveItemButton enable
                cancelSaveItemButton enable
                clean outputMsg
                enable inputKey
                enable inputValue
                setValueOf inputKey (unpack n)
                setValueOf inputValue (show v)
                rememberCurrent cmd
              return b
            ,UI.span # set html "   "
            ,do
              b ← UI.button #. "w3-small w3-btn w3-ripple w3-red"
                    #+ [UI.bold #+ [string "Delete"]]
              on UI.click b $ const $ do
                liftIO $ removeItem cmd n
                cleanAndDisable inputKey
                cleanAndDisable inputValue
                -- Initiate a click to current menu to update the items list after deleting
                performActionOnId (show cmd) $ runFunction ∘ ffi "$(%1).click()"
              return b
            ]
          ]
    let showCurrentTab cmd = do
```

```
        let baseClasses = "w3-bar-item w3-button"
              classesForCurrentTab = baseClasses <> " " <> "w3-light-grey"
        performActionOnId (show cmd) $ setClasses classesForCurrentTab
        let otherTabs = delete cmd commands
        forM_ otherTabs $ λtabName →
              performActionOnId (show tabName) $ setClasses baseClasses
    let displayItems cmd sel = do
        showCurrentTab cmd
        rememberCurrent cmd
        saveItemButton disable
        cancelSaveItemButton disable
        addItemButton enable
        cleanOutputTable
        performActionOnId outputTableId $
              λt → void $ element t #+
                [UI.tr #+
                  [UI.th #+ [string "LoggerName"]
                  ,UI.th #+ [string $ show cmd <> " value"]
                  ,UI.th #+ [string ""]
                  ]
                ]
        cg ← liftIO $ readMVar (CM.getCG config)
        forM_ (HM.toList $ sel cg) $
              λ(n, v) → performActionOnId outputTableId $
                λt → void $ element t #+ [mkTableRow cmd n v]
    let displayBuffer :: ToJSON a ⇒ Cmd → [(LoggerName, LogObject a)] → UI ()
      displayBuffer cmd sel = do
        showCurrentTab cmd
        rememberCurrent cmd
        saveItemButton disable
        cancelSaveItemButton disable
        addItemButton disable
        cleanOutputTable
        performActionOnId outputTableId $
              λt → void $ element t #+
                [UI.tr #+
                  [UI.th #+ [string "LoggerName"]
                  ,UI.th #+ [string $ show cmd <> " value"]
                  ,UI.th #+ [string ""]
                  ]
                ]
        forM_ sel $
              λ(n, v) → performActionOnId outputTableId $
                λt → void $ element t #+ [mkSimpleRow n v]
    let accessBufferMap = do
        ed ← liftIO $ readMVar (getEd editor)
        liftIO $ readBuffer $ edBuffer ed
    let exportConfiguration = do
        currentDir ← liftIO getCurrentDirectory
        let dir = currentDir </> "iohk-monitoring/static/conf"
```

```
        liftIO $ createDirectoryIfMissing True dir
        tsnow ← formatTime defaultTimeLocale tsformat <$> liftIO getCurrentTime
        let filename = "config.yaml" ++ "-" ++ tsnow
            filepath = dir </> filename
        res ← liftIO $ catch
            (CM.exportConfiguration config filepath ≫
              return ("Configuration was exported to the file: " ++ filepath))
            (λ(e :: IOException) → return $ show e)
        setMessage res
        performActionOnId outputTableId $
            λt → void $ element t #+ [mkLinkToFile
              "Link to configuration file"
              ("/static/conf" </> filename)
              ]
    let displayExport cmd = do
        showCurrentTab cmd
        rememberCurrent cmd
        saveItemButton disable
        cancelSaveItemButton disable
        addItemButton disable
        cleanOutputTable
        exportConfiguration
    let switchToTab c@Backends      = displayItems c CM.cgMapBackend
      switchToTab c@Severities      = displayItems c CM.cgMapSeverity
      switchToTab c@Scribes         = displayItems c CM.cgMapScribe
      switchToTab c@SubTrace        = displayItems c CM.cgMapSubtrace
      switchToTab c@Aggregation     = displayItems c CM.cgMapAggregatedKind
      switchToTab c@Buffer          = accessBufferMap ≫= displayBuffer c
      switchToTab c@ExportConfiguration = displayExport c
    let mkEditInputs =
        row [element inputKey
            ,UI.span #. "key-value-separator" #+ [string ":"]
            ,element inputValue
            ,UI.span #. "key-value-separator" #+ [string ""]
            ,do
              b ← UI.button #. "w3-btn w3-ripple w3-green save-item-button"
                 # set (UI.attr "id") addItemButtonId
                 # set UI.enabled False
                 #+ [UI.bold #+ [string "New"]]
              on UI.click b $ const $ do
                enable inputKey
                enable inputValue
                saveItemButton enable
                cancelSaveItemButton enable
              return b
            ,UI.span #. "key-value-separator" #+ [string ""]
            ,do
              b ← UI.button #. "w3-btn w3-ripple w3-lime save-item-button"
                 # set (UI.attr "id") saveItemButtonId
                 # set UI.enabled False
                 #+ [UI.bold #+ [string "Save"]]
```

```
        on UI.click b $ const $ do
          k ← inputKey # get UI.value
          v ← inputValue # get UI.value
          m ← currentCmd # get UI.value
          case (readMay m :: Maybe Cmd) of
            Nothing → setError "parse error on cmd"
            Just c → do
              cleanAndDisable inputKey
              cleanAndDisable inputValue
              saveItemButton disable
              cancelSaveItemButton disable
              setMessage $ "Setting '" ++ k ++ "' to '" ++ v ++ "' in " ++ m
              updateItem c (pack k) v
              switchToTab c
        return b
      , UI.span #. "key-value-separator" #+ [string ""]
      , do
        b ← UI.button #. "w3-btn w3-ripple w3-white"
          # set (UI.attr "id") cancelSaveItemButtonId
          # set UI.enabled False
          #+ [UI.bold #+ [string "Cancel"]]
        on UI.click b $ const $ do
          cleanAndDisable inputKey
          cleanAndDisable inputValue
          saveItemButton disable
          cancelSaveItemButton disable
        return b
      ]
  let minimumSeveritySelection = do
    confMinSev ← liftIO $ minSeverity config
    let setMinSev _el Nothing = pure ()
        setMinSev _el (Just sev) = liftIO $
          setMinSeverity config (toEnum sev :: Severity)
        mkSevOption sev = UI.option # set UI.text (show sev)
          # set UI.value (show sev)
          # if confMinSev ≡ sev then set UI.selected True else id
    minsev ← UI.select #. "minsevfield" #+
      map mkSevOption (enumFrom Debug)
    on UI.selectionChange minsev $ setMinSev minsev
    row [string "Set minimum severity to:"
        , UI.span # set html " "
        , UI.span #. "severity-dropdown big" #+ [element minsev]
        ]
  let commandTabs =
    row $ flip map commands $ λcmd → do
      b ← UI.button #. "w3-bar-item w3-button w3-grey"
        # set (UI.attr "id") (show cmd)
        #+ [UI.bold #+ [string (show cmd)]]
      on UI.click b $ const $ do
        cleanAndDisable inputKey
```

```
                cleanAndDisable inputValue
                clean outputMsg
                switchToTab cmd
              return b
  getElementById window "main-section" ≫= λcase
    Nothing → pure ()
    Just mainSection → void $ element mainSection #+
      [UI.div #. "w3-panel" #+
          [UI.div #. "w3-border w3-border-dark-grey" #+
            [UI.div #. "w3-panel" #+ [minimumSeveritySelection]
            ]
          ,UI.div #. "w3-panel" #+ [ ]
          ,UI.div #. "w3-border w3-border-dark-grey" #+
            [UI.div #. "w3-bar w3-grey" #+ [commandTabs]
            ,UI.div #. "w3-panel" #+ [mkEditInputs]
            ,UI.div #. "w3-panel" #+ [element outputMsg]
            ]
          ]
      ]
```

1.7.33 Cardano.BM.Backend.EKGView

Plugin definition

```
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace.Trace IO a → s a → IO (Plugin a)
plugin config trace sb = do
  be :: Cardano.BM.Backend ∘ EKGView.EKGView a ← realizefrom config trace sb
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Structure of EKGView

```
type EKGViewMVar a = MVar (EKGViewInternal a)
newtype EKGView a = EKGView
  {getEV :: EKGViewMVar a}
data EKGViewInternal a = EKGViewInternal
  {evQueue    :: Maybe (TBQ.TBQueue (Maybe (LogObject a)))
  ,evLabels   :: !(EKGViewMap Label.Label)
  ,evGauges   :: !(EKGViewMap Gauge.Gauge)
  ,evServer   :: Maybe Server
  ,evDispatch :: Maybe (Async.Async ())
  ,evPrometheusDispatch :: Maybe (Async.Async ())
  }
```

Relation from variable name to label handler

We keep the label handlers for later update in a *HashMap*.

```
type EKGViewMap a = HM.HashMap Text a
```

Internal Trace

This is an internal Trace, named "#ekgview", which can be used to control the messages that are being displayed by EKG.

```
ekgTrace :: ToJSON a ⇒ EKGView a → Configuration → Trace IO a
ekgTrace ekg _c =
    Trace.appendName "#ekgview" $ ekgTrace' ekg
  where
    ekgTrace' :: ToJSON a ⇒ EKGView a → Trace IO a
    ekgTrace' ekgview = Tracer $ λ(_ctx, lo@(LogObject outerloname _ _)) → do
      let setLabel :: Text → Text → EKGViewInternal a → IO (Maybe (EKGViewInternal a))
          setLabel name label ekg_i@(EKGViewInternal _ labels _ mserver _ _) =
            case (HM.lookup name labels, mserver) of
              (Nothing, Just server) → do
                ekghdl ← getLabel name server
                Label.set ekghdl label
                return $ Just $ ekg_i {evLabels = HM.insert name ekghdl labels}
              (Just ekghdl, _) → do
                Label.set ekghdl label
                return Nothing
              (Nothing, Nothing) →
                pure Nothing
          setGauge :: Text → Int64 → EKGViewInternal a → IO (Maybe (EKGViewInternal a))
          setGauge name value ekg_i@(EKGViewInternal _ _ gauges mserver _ _) =
            case (HM.lookup name gauges, mserver) of
              (Nothing, Just server) → do
                ekghdl ← getGauge name server
                Gauge.set ekghdl value
                return $ Just $ ekg_i {evGauges = HM.insert name ekghdl gauges}
              (Just ekghdl, _) → do
                Gauge.set ekghdl value
                return Nothing
              (Nothing, Nothing) →
                pure Nothing
          update :: ToJSON a ⇒ LogObject a → EKGViewInternal a → IO (Maybe (EKGViewInternal a))
          update (LogObject loname _ (LogMessage logitem)) ekg_i =
            setLabel loname (pack $ show $ encode logitem) ekg_i
          update (LogObject loname _ (LogValue iname value)) ekg_i =
            let logname = loname <> "." <> iname
            in
            case value of
              (Microseconds x) → setGauge (logname <> ".us") (fromIntegral x) ekg_i
              (Nanoseconds x) → setGauge (logname <> ".ns") (fromIntegral x) ekg_i
              (Seconds      x) → setGauge (logname <> ".s") (fromIntegral x) ekg_i
              (Bytes        x) → setGauge (logname <> ".B") (fromIntegral x) ekg_i
```

```
        (PureI      x) → setGauge (logname <> ".int") (fromIntegral x) ekg_i
        (PureD      _) → setLabel (logname <> ".real") (pack $ show value) ekg_i
        (Severity   _) → setLabel (logname <> ".sev") (pack $ show value) ekg_i
    update _ _ = return Nothing
  modifyMVar_ (getEV ekgview) $ λekgup → do
    let -- strip off some prefixes not necessary for display
      loname1 = fromMaybe outerloname $ stripPrefix "#ekgview" outerloname
      loname = fromMaybe loname1 $ stripPrefix "#aggregation" loname1
    upd ← update lo {loName = loname} ekgup
    case upd of
      Nothing → return ekgup
      Just ekgup' → return ekgup'
```

EKG view is an effectuator

Function *effectuate* is called to pass in a LogObject for display in EKG. If the log item is an *AggregatedStats* message, then all its constituents are put into the queue. In case the queue is full, all new items are dropped.

```
instance IsEffectuator EKGView a where
  effectuate ekgview item = do
    ekg ← readMVar (getEV ekgview)
    case evQueue ekg of
      Nothing → pure ()
      Just queue → doEnqueue queue
    where
      doEnqueue :: TBQ.TBQueue (Maybe (LogObject a)) → IO ()
      doEnqueue queue =
        let enqueue a = do
              nocapacity ← atomically $ TBQ.isFullTBQueue queue
              if nocapacity
              then handleOverflow ekgview
              else atomically $ TBQ.writeTBQueue queue (Just a)
        in
        case item of
          (LogObject loname lometa (AggregatedMessage ags)) → liftIO $ do
            let traceAgg :: [(Text, Aggregated)] → IO ()
                traceAgg [] = return ()
                traceAgg ((n, AggregatedEWMA agewma) : r) = do
                  enqueue $ LogObject (loname <> "." <> n) lometa (LogValue "avg" $ avg agewma)
                  traceAgg r
                traceAgg ((n, AggregatedStats stats) : r) = do
                  let statsname = loname <> "." <> n
                      qbasestats s' nm = do
                        enqueue $ LogObject nm lometa (LogValue "mean" (PureD $ meanOfStats s'))
                        enqueue $ LogObject nm lometa (LogValue "min" $ fmin s')
                        enqueue $ LogObject nm lometa (LogValue "max" $ fmax s')
                        enqueue $ LogObject nm lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s
                        enqueue $ LogObject nm lometa (LogValue "stdev" (PureD $ stdevOfStats s'))
                  enqueue $ LogObject statsname lometa (LogValue "last" $ flast stats)
                  qbasestats (fbasic stats) $ statsname <> ".basic"
```

```
                qbasestats (fdelta stats) $ statsname <> ".delta"
                qbasestats (ftimed stats) $ statsname <> ".timed"
                traceAgg r
            traceAgg ags
        (LogObject _ _ (LogMessage _)) → enqueue item
        (LogObject _ _ (LogValue _ _)) → enqueue item
        _ → return ()
    handleOverflow _ = TIO.hPutStrLn stderr "Notice: EKGViews's queue full, dropping log items
```

EKGView implements Backend functions

EKGView is an IsBackend

```
instance (ToJSON a, FromJSON a) ⇒ IsBackend EKGView a where
  type BackendFailure EKGView = EKGBackendFailure
  bekind _ = EKGViewBK
  realize _ = fail "EKGView cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    evref ← newEmptyMVar
    let ekgview = EKGView evref
    evport ← getEKGport config
    ehdl ← (forkServer "127.0.0.1" evport
      -- This unfortunate delay is to catch the async exception.
      <* threadDelay 300000)
      `catch` mkHandler EKGServerStartupError
    ekghdl ← getLabel "iohk-monitoring version" ehdl
    Label.set ekghdl $ pack (showVersion version)
    let ekgtrace = ekgTrace ekgview config
# ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
# else
    let qSize = 5120
# endif
    queue ← atomically $ TBQ.newTBQueue qSize
    dispatcher ← spawnDispatcher config queue sbtrace ekgtrace
      `catch` mkHandler EKGDispatcherStartupError
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    prometheusBindAddr ← getPrometheusBindAddr config
    prometheusDispatcher ←
        case prometheusBindAddr of
          Just (host, port) → do
            pd ← spawnPrometheus ehdl (fromString host) port
              `catch` mkHandler EKGPrometheusStartupError
            Async.link pd
            return (Just pd)
          Nothing →
            return Nothing
```

```
    putMVar evref $ EKGViewInternal
      {evLabels = HM.empty
      ,evGauges = HM.empty
      ,evServer = Just ehdl
      ,evQueue = Just queue
      ,evDispatch = Just dispatcher
      ,evPrometheusDispatch = prometheusDispatcher
      }
    return ekgview
  `catch` -- Try to catch specific errors first.
  nullSetup sbtrace
  `catch` -- ..if that fails, catch everything.
  (nullSetup sbtrace ∘ EKGUnknownStartupError ∘ (show :: SomeException → String))
  where
    mkHandler
      :: (String → EKGBackendFailure)
      → SomeException
      → IO b
    mkHandler ctor = throwIO ∘ ctor ∘ show
    nullSetup
      :: Trace IO a
      → EKGBackendFailure
      → IO (EKGView a)
    nullSetup trace e = do
      meta ← mkLOMeta Error Public
      traceWith trace $ ("#ekgview.realizeFrom", LogObject "#ekgview.realizeFrom" meta $
        LogError $ "EKGView backend disabled due to initialisation error: " <> (pack $ show
      _ ← atomically $ TBQ.newTBQueue 0
      ref ← newEmptyMVar
      putMVar ref $ EKGViewInternal
        {evLabels = HM.empty
        ,evGauges = HM.empty
        ,evServer = Nothing
        ,evQueue = Nothing
        ,evDispatch = Nothing
        ,evPrometheusDispatch = Nothing
        }
      pure $ EKGView ref
  unrealize ekgview = do
    let clearMVar :: MVar b → IO ()
        clearMVar = void ∘ tryTakeMVar
    withMVar (getEV ekgview) $ λev → do
      forM_ (evQueue ev) $
        -- send terminating item to the queue
        λqueue →
            atomically $ TBQ.writeTBQueue queue Nothing
      forM_ (evDispatch ev) $
        -- wait for the dispatcher to exit
        λdispatcher → do
            res ← Async.waitCatch dispatcher
```

```
                either throwM return res
        forM_ (evPrometheusDispatch ev) $
          Async.cancel
      withMVar (getEV ekgview) $ λekg →
        forM_ (evServer ekg) $
          λserver → killThread $ serverThreadId server
      clearMVar $ getEV ekgview
data EKGBackendFailure
  = EKGUnknownStartupError String
  | EKGServerStartupError String
  | EKGDispatcherStartupError String
  | EKGPrometheusStartupError String
  deriving Show
instance Exception EKGBackendFailure
```

Asynchronously reading log items from the queue and their processing

```
spawnDispatcher :: Configuration
                → TBQ.TBQueue (Maybe (LogObject a))
                → Trace.Trace IO a
                → Trace.Trace IO a
                → IO (Async.Async ())
spawnDispatcher config evqueue _sbtrace ekgtrace =
    Async.async $ qProc
  where
    {- lazy qProc -}
    qProc :: IO ()
    qProc =
      processQueue
        evqueue
        processEKGView
        ()
        (\_ → pure ())
    processEKGView obj@(LogObject loname0 _ _) _ = do
      obj' ← testSubTrace config ("#ekgview." <> loname0) obj
      case obj' of
        Just lo →
          let trace = Trace.appendName loname0 ekgtrace
          in
          traceWith trace (loname0, lo)
        Nothing → pure ()
      pure ()
```

1.7.34 Cardano.BM.Backend.Prometheus

Spawn Prometheus client from existing EKG server

```
spawnPrometheus :: EKG.Server → ByteString → Int → IO (Async.Async ())
spawnPrometheus ekg host port = Async.async $
```

```
    simpleHttpServe config site
  where
    config :: Config Snap a
    config = setPort port ∘ setBind host ∘ setAccessLog lg ∘ setErrorLog lg $ defaultConfig
    lg = ConfigNoLog
    site :: Snap ()
    site = route [("/metrics/", webhandler ekg)]
    webhandler :: EKG.Server → Snap ()
    webhandler srv = do
      samples ← liftIO $ sampleAll $ EKG.serverMetricStore srv
      writeLBS ∘ toLazyByteString ∘ renderSamples $ HM.toList samples
      pure ()
    renderSamples :: [(Text, Value)] → Builder
    renderSamples [] = mempty
    renderSamples samples = mconcat
      [case sv of
        Counter c → renderNamedValue sk (int64Dec c)
        Gauge g → renderNamedValue sk (int64Dec g)
        Label l → if isFloat l
          then renderNamedValue sk (byteString $ encodeUtf8 l)
          else mempty
        _ → mempty
      | (sk, sv) ← samples]
    renderNamedValue :: Text → Builder → Builder
    renderNamedValue nm bld =
      (byteString $ prepareName nm)
      <> charUtf8 ' '
      <> bld
      <> charUtf8 '\n'
    prepareName nm = encodeUtf8 $ replace "  " "_" $ replace "-" "_" $ replace "." "_" nm
    isFloat v = case double v of
      Right (_n, "") → True-- only floating point number parsed, no leftover
      _ → False
```

1.7.35 Cardano.BM.Backend.Graylog

Plugin definition

```
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace.Trace IO a → s a → IO (Plugin a)
plugin config trace sb = do
  be :: Cardano.BM.Backend ∘ Graylog.Graylog a ← realizefrom config trace sb
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Structure of Graylog

```
type GraylogMVar a = MVar (GraylogInternal a)
newtype Graylog a = Graylog
```

```
  {getGL :: GraylogMVar a}
data GraylogInternal a = GraylogInternal
  {glQueue :: TBQ.TBQueue (Maybe (LogObject a))
  ,glDispatch :: Async.Async ()
  }
```

Graylog is an effectuator

Function *effectuate* is called to pass in a LogObject to forward to Graylog. In case the queue is full, all new items are dropped.

```
instance IsEffectuator Graylog a where
  effectuate graylog item = do
    gelf ← readMVar (getGL graylog)
    let enqueue a = do
          nocapacity ← atomically $ TBQ.isFullTBQueue (glQueue gelf)
          if nocapacity
          then handleOverflow graylog
          else atomically $ TBQ.writeTBQueue (glQueue gelf) (Just a)
    case item of
      (LogObject logname lometa (AggregatedMessage ags)) → liftIO $ do
        let traceAgg :: [(Text, Aggregated)] → IO ()
            traceAgg [] = return ()
            traceAgg ((n, AggregatedEWMA agewma) : r) = do
              enqueue $ LogObject (logname <> "." <> n) lometa (LogValue "avg" $ avg agewma)
              traceAgg r
            traceAgg ((n, AggregatedStats stats) : r) = do
              let statsname = logname <> "." <> n
                  qbasestats s' nm = do
                    enqueue $ LogObject nm lometa (LogValue "mean" (PureD $ meanOfStats s'))
                    enqueue $ LogObject nm lometa (LogValue "min" $ fmin s')
                    enqueue $ LogObject nm lometa (LogValue "max" $ fmax s')
                    enqueue $ LogObject nm lometa (LogValue "count" $ PureI $ fromIntegral $ fcount s')
                    enqueue $ LogObject nm lometa (LogValue "stdev" (PureD $ stdevOfStats s'))
              enqueue $ LogObject statsname lometa (LogValue "last" $ flast stats)
              qbasestats (fbasic stats) $ statsname <> ".basic"
              qbasestats (fdelta stats) $ statsname <> ".delta"
              qbasestats (ftimed stats) $ statsname <> ".timed"
              traceAgg r
            traceAgg ags
        (LogObject _ _ (LogMessage _)) → enqueue item
        (LogObject _ _ (LogValue _ _)) → enqueue item
        _ → return ()
    handleOverflow _ = TIO.hPutStrLn stderr "Notice: Graylogs's queue full, dropping log items
```

Graylog implements Backend functions

Graylog is an IsBackend

```
instance (ToJSON a, FromJSON a) ⇒ IsBackend Graylog a where
  bekind _ = GraylogBK
```

```
  realize _ = fail "Graylog cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    glref ← newEmptyMVar
    let graylog = Graylog glref
# ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
# else
    let qSize = 1024
# endif
    queue ← atomically $ TBQ.newTBQueue qSize
    dispatcher ← spawnDispatcher config queue sbtrace
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar glref $ GraylogInternal
      {glQueue = queue
      ,glDispatch = dispatcher
      }
    return graylog
  unrealize graylog = do
    let clearMVar :: MVar b → IO ()
      clearMVar = void ∘ tryTakeMVar
    (dispatcher,queue) ← withMVar (getGL graylog) (λgelf →
      return (glDispatch gelf,glQueue gelf))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher
    either throwM return res
    clearMVar $ getGL graylog
```

Asynchronously reading log items from the queue and their processing

```
spawnDispatcher :: forall a ∘ ToJSON a
                ⇒ Configuration
                → TBQ.TBQueue (Maybe (LogObject a))
                → Trace.Trace IO a
                → IO (Async.Async ())
spawnDispatcher config evqueue sbtrace =
    let gltrace = Trace.appendName "#graylog" sbtrace
    in
    Async.async $ Net.withSocketsDo $ qProc gltrace Nothing
  where
    {- lazy qProc -}
    qProc :: Trace.Trace IO a → Maybe Net.Socket → IO ()
    qProc gltrace conn =
      processQueue
        evqueue
        processGraylog
```

```
        (gltrace, conn)
        (λ(_, c) → closeConn c)
    processGraylog :: LogObject a → (Trace.Trace IO a, Maybe Net.Socket)
      → IO (Trace.Trace IO a, Maybe Net.Socket)
    processGraylog item (gltrace, mConn) =
      case mConn of
        (Just conn) → do
          sendLO conn item
            `catch` λ(e :: IOException) → do
              let trace' = Trace.appendName "sending" gltrace
              mle ← mkLOMeta Error Public
              Trace.traceNamedObject trace' (mle, LogError (pack $ show e))
              threadDelay 50000
              void $ processGraylog item (gltrace, mConn)
          return (gltrace, mConn)
        Nothing → do
          mConn' ← tryConnect gltrace
          processGraylog item (gltrace, mConn')
    sendLO :: Net.Socket → LogObject a → IO ()
    sendLO conn obj =
      let msg = BS8.toStrict $ encodeMessage obj
      in sendAll conn msg
    closeConn :: Maybe Net.Socket → IO ()
    closeConn Nothing = return ()
    closeConn (Just conn) = Net.close conn
    tryConnect :: Trace.Trace IO a → IO (Maybe Net.Socket)
    tryConnect gltrace = do
      port ← getGraylogPort config
      let hints = Net.defaultHints {Net.addrSocketType = Net.Datagram}
      (addr: _) ← Net.getAddrInfo (Just hints) (Just "127.0.0.1") (Just $ show port)
      sock ← Net.socket (Net.addrFamily addr) (Net.addrSocketType addr) (Net.addrProtocol addr)
      Net.connect sock (Net.addrAddress addr) ≫ return (Just sock)
        `catch` λ(e :: SomeException) → do
          let trace' = Trace.appendName "connecting" gltrace
          mle ← mkLOMeta Error Public
          Trace.traceNamedObject trace' (mle, LogError (pack $ show e))
          return Nothing
    encodeMessage :: ToJSON a ⇒ LogObject a → BS8.ByteString
    encodeMessage lo = encode $ mkGelfItem lo
```

Gelf data structure

GELF defines a data format of the message payload: https://docs.graylog.org/en/3.0/pages/gelf.html

```
data GelfItem = GelfItem {
    version :: !Text,
    host :: !Text,
    short_message :: !Text,
    full_message :: !Value,
    timestamp :: !Double,
    level :: !Int,
```

```
    _tid :: !Text,
    _privacy :: !Text
  }
mkGelfItem :: ToJSON a ⇒ LogObject a → GelfItem
mkGelfItem (LogObject loname lometa locontent) = GelfItem {
    version = "1.1",
    host = "hostname",
    short_message = loname,
    full_message = toJSON locontent,
    timestamp = (fromInteger ∘ toInteger $ utc2ns (tstamp lometa) :: Double) / 1000000000,
    level = fromEnum (maxBound@Severity) − fromEnum (severity lometa),
    _tid = tid lometa,
    _privacy = pack $ show $ privacy lometa
  }
instance ToJSON GelfItem where
  toJSON gli = object [
    "version" .= version gli,
    "host" .= host gli,
    "short_message" .= short_message gli,
    "full_message" .= full_message gli,
    "timestamp" .= (printf "%0.3f" $ timestamp gli :: String),
    "level" .= level gli,
    "_tid" .= _tid gli,
    "_privacy" .= _privacy gli
    ]
```

1.7.36 Cardano.BM.Backend.Monitoring

Plugin definition

```
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace.Trace IO a → s a → IO (Plugin a)
plugin config trace sb = do
  be :: Cardano.BM.Backend ∘ Monitoring.Monitor a ← realizefrom config trace sb
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Structure of Monitoring

```
type MonitorMVar a = MVar (MonitorInternal a)
newtype Monitor a = Monitor
  {getMon :: MonitorMVar a}
data MonitorInternal a = MonitorInternal
  {monQueue :: TBQ.TBQueue (Maybe (LogObject a))
  ,monDispatch :: Async.Async ()
  ,monBuffer :: !(LogBuffer a)
  }
```

Relation from context name to monitoring state

We remember the state of each monitored context name.

```
data MonitorState = MonitorState {
  _preCondition :: !MEvPreCond
  , _expression    :: !MEvExpr
  , _actions       :: [MEvAction]
  , _environment :: !Environment
  } deriving Show
type MonitorMap = HM.HashMap LoggerName MonitorState
```

Monitor view is an effectuator

Function *effectuate* is called to pass in a LogObject for monitoring.

```
instance IsEffectuator Monitor a where
  effectuate monitor item = do
    mon ← readMVar (getMon monitor)
    effectuate (monBuffer mon) item
    nocapacity ← atomically $ TBQ.isFullTBQueue (monQueue mon)
    if nocapacity
    then handleOverflow monitor
    else atomically $ TBQ.writeTBQueue (monQueue mon) $ Just item
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: Monitor's queue full, dropping log items!"
```

Monitor implements Backend functions

Monitor is an IsBackend

```
instance FromJSON a ⇒ IsBackend Monitor a where
  bekind _ = MonitoringBK
  realize _ = fail "Monitoring cannot be instantiated by 'realize'"
  realizefrom config sbtrace _ = do
    monref ← newEmptyMVar
    let monitor = Monitor monref
# ifdef PERFORMANCE_TEST_QUEUE
    let qSize = 1000000
# else
    let qSize = 512
# endif
    queue ← atomically $ TBQ.newTBQueue qSize
    dispatcher ← spawnDispatcher queue config sbtrace monitor
    monbuf :: Cardano.BM.Backend ∘ LogBuffer.LogBuffer a ← Cardano.BM.Backend ∘ LogBuffer.realize
    -- link the given Async to the current thread, such that if the Async
    -- raises an exception, that exception will be re-thrown in the current
    -- thread, wrapped in ExceptionInLinkedThread.
    Async.link dispatcher
    putMVar monref $ MonitorInternal
      {monQueue = queue
      ,monDispatch = dispatcher
```

```
      , monBuffer = monbuf
      }
    return monitor
  unrealize monitoring = do
    let clearMVar :: MVar b → IO ()
      clearMVar = void ∘ tryTakeMVar
    (dispatcher, queue) ← withMVar (getMon monitoring) (λmon →
      return (monDispatch mon, monQueue mon))
    -- send terminating item to the queue
    atomically $ TBQ.writeTBQueue queue Nothing
    -- wait for the dispatcher to exit
    res ← Async.waitCatch dispatcher
    either throwM return res
    clearMVar $ getMon monitoring
```

Asynchronously reading log items from the queue and their processing

```
spawnDispatcher :: TBQ.TBQueue (Maybe (LogObject a))
                → Configuration
                → Trace.Trace IO a
                → Monitor a
                → IO (Async.Async ())
spawnDispatcher mqueue config sbtrace monitor =
  Async.async (initMap ≫= qProc)
  where
    {- lazy qProc -}
    qProc state =
      processQueue
        mqueue
        processMonitoring
        state
        (\_ → pure ())
    processMonitoring lo@LogObject {} state = do
      let accessBufferMap = do
        mon ← tryReadMVar (getMon monitor)
        case mon of
          Nothing → return [ ]
          Just actualMon → readBuffer $ monBuffer actualMon
      mbuf ← accessBufferMap
      let sbtraceWithMonitoring = Trace.appendName "#monitoring" sbtrace
      valuesForMonitoring ← getVarValuesForMonitoring config mbuf
      state' ← evalMonitoringAction sbtraceWithMonitoring
        state
        lo
        valuesForMonitoring
      return state'
    initMap = do
      ls ← getMonitors config
      return $ HM.fromList $ map (λ(n, (precond, e, as)) → (n, MonitorState precond e as HM.empty))
                           $ HM.toList ls
```

```
getVarValuesForMonitoring :: Configuration
  → [(LoggerName, LogObject a)]
  → IO [(VarName, Measurable)]
getVarValuesForMonitoring config mbuf = do
    -- Here we take all var names for all monitors, just in case.
    monitorsInfo ← HM.elems <$> getMonitors config
    let varNames = concat [extractVarNames mEvExpr | (_, mEvExpr, _) ← monitorsInfo]
    return ∘ catMaybes ∘ concat $ map (getVNnVal varNames) mbuf
  where
    extractVarNames expr = case expr of
        Compare vn _ → [vn]
        AND   e1 e2 → extractVarNames e1 ++ extractVarNames e2
        OR    e1 e2 → extractVarNames e1 ++ extractVarNames e2
        NOT   e     → extractVarNames e
    getVNnVal varNames logObj = case logObj of
        (_, LogObject _ _ (LogValue vn val)) → [Just (vn, val) | vn ∈ varNames]
        (_, LogObject _ _ (AggregatedMessage agg)) → concatMap getMeasurable agg
        (_, _)                                → [ ]
      where
        getMeasurable :: (Text, Aggregated) → [Maybe (VarName, Measurable)]
        getMeasurable agg = case agg of
            (vn, AggregatedEWMA (EWMA _ val)) → [Just (vn <> ".ewma.avg", val) | vn ∈ varNames]
            (vn, AggregatedStats st) → if vn ∈ varNames
                                       then stValues vn st
                                       else [ ]
            _                        → [ ]
          where
            stValues vn st =
              [Just (vn <> ".flast", flast st)
              ,Just (vn <> ".fold", fold st)
              ,Just (vn <> ".fbasic.fmin",  fmin ∘ fbasic $ st)
              ,Just (vn <> ".fbasic.fmax",  fmax ∘ fbasic $ st)
              ,Just (vn <> ".fbasic.mean",  PureD ∘ meanOfStats ∘ fbasic $ st)
              ,Just (vn <> ".fbasic.stdev", PureD ∘ stdevOfStats ∘ fbasic $ st)
              ,Just (vn <> ".fbasic.fcount", PureI ∘ fromIntegral ∘ fcount ∘ fbasic $ st)
              ,Just (vn <> ".fdelta.fmin",  fmin ∘ fdelta $ st)
              ,Just (vn <> ".fdelta.fmax",  fmax ∘ fdelta $ st)
              ,Just (vn <> ".fdelta.mean",  PureD ∘ meanOfStats ∘ fdelta $ st)
              ,Just (vn <> ".fdelta.stdev", PureD ∘ stdevOfStats ∘ fdelta $ st)
              ,Just (vn <> ".fdelta.fcount", PureI ∘ fromIntegral ∘ fcount ∘ fdelta $ st)
              ,Just (vn <> ".ftimed.fmin",  fmin ∘ ftimed $ st)
              ,Just (vn <> ".ftimed.fmax",  fmax ∘ ftimed $ st)
              ,Just (vn <> ".ftimed.mean",  PureD ∘ meanOfStats ∘ ftimed $ st)
              ,Just (vn <> ".ftimed.stdev", PureD ∘ stdevOfStats ∘ ftimed $ st)
              ,Just (vn <> ".ftimed.fcount", PureI ∘ fromIntegral ∘ fcount ∘ ftimed $ st)
              ]
```

Evaluation of monitoring action

Inspect the log message and match it against configured thresholds. If positive, then run the action on the current state and return the updated state.

```
evalMonitoringAction :: Trace.Trace IO a
  → MonitorMap
  → LogObject a
  → [(VarName, Measurable)]
  → IO MonitorMap
evalMonitoringAction sbtrace mmap logObj@(LogObject logname1 _ content) variables = do
  let logname = case content of
        ObserveOpen _ → logname1 <> ".open"
        ObserveDiff  _ → logname1 <> ".diff"
        ObserveClose _ → logname1 <> ".close"
        _ → logname1
  let sbtrace' = Trace.appendName logname sbtrace
  case HM.lookup logname mmap of
    Nothing → return mmap
    Just mon@(MonitorState precond expr acts env0) → do
      let env1 = updateEnv env0 logObj
      let env' = HM.union env1 $ HM.fromList variables
      let doMonitor = case precond of
        -- There's no precondition, do monitor as usual.
        Nothing → True
        -- Precondition is defined, do monitor only if it is True.
        Just preCondExpr → evaluate env' preCondExpr
      -- In this place env' already must contain opvn..
      let thresholdIsReached = evaluate env' expr
      if doMonitor ∧ thresholdIsReached then do
        now ← getMonotonicTimeNSec
        let env'' = HM.insert "lastalert" (Nanoseconds now) env'
        mapM_ (evaluateAction sbtrace' env' expr) acts
        return $ HM.insert logname mon {_environment = env''} mmap
      else return mmap
  where
    updateEnv :: Environment → LogObject a → Environment
    updateEnv env (LogObject loname lometa (ObserveOpen (CounterState counters))) =
        let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lometa))
                                 : countersEnvPairs (loname <> ".open") counters
        in
        HM.union addenv env
    updateEnv env (LogObject loname lometa (ObserveDiff (CounterState counters))) =
        let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lometa))
                                 : countersEnvPairs (loname <> ".diff") counters
        in
        HM.union addenv env
    updateEnv env (LogObject loname lometa (ObserveClose (CounterState counters))) =
        let addenv = HM.fromList $ ("timestamp", Nanoseconds $ utc2ns (tstamp lometa))
                                 : countersEnvPairs (loname <> ".close") counters
        in
        HM.union addenv env
```

```
    updateEnv env (LogObject _ lometa (LogValue vn val)) =
        let addenv = HM.fromList [(vn, val)
                                 ,("timestamp", Nanoseconds $ utc2ns (tstamp lometa))
                                 ]
        in
        HM.union addenv env
    updateEnv env (LogObject _ lometa (LogMessage _logitem)) =
        let addenv = HM.fromList [("severity", Severity (severity lometa))
                                 ,("timestamp", Nanoseconds $ utc2ns (tstamp lometa))
                                 ]
        in
        HM.union addenv env
    updateEnv env (LogObject _ lometa (AggregatedMessage vals)) =
        let addenv = ("timestamp", Nanoseconds $ utc2ns (tstamp lometa)) : aggs2measurables vals [ ]
        in
        HM.union (HM.fromList addenv) env
      where
        aggs2measurables [ ] acc = acc
        aggs2measurables ((n, AggregatedEWMA vewma) : r) acc = aggs2measurables r $ (n <> ".avg", avg v
        aggs2measurables ((n, AggregatedStats s) : r) acc = aggs2measurables r $
          (n <> ".mean", PureD ∘ meanOfStats $ fbasic s)
          : (n <> ".flast", flast s)
          : (n <> ".fcount", PureI ∘ fromIntegral ∘ fcount $ fbasic s)
          : acc
    -- catch all
    updateEnv env _ = env

    countersEnvPairs loggerName = map $ λcounter →
        let name = loggerName <> "." <> nameCounter counter <> "." <> cName counter
            value = cValue counter
        in
          (name, value)

    evaluateAction sbtrace' env expr (CreateMessage sev alertMessage) = do
        lometa ← mkLOMeta sev Public
        let fullMessage = alertMessage
            <> "; environment is: " <> pack (show env)
            <> "; threshold expression is: " <> pack (show expr)
        Trace.traceNamedObject sbtrace' (lometa, MonitoringEffect (MonitorAlert fullMessage))
    evaluateAction sbtrace' _ _ (SetGlobalMinimalSeverity sev) = do
        lometa ← mkLOMeta sev Public
        Trace.traceNamedObject sbtrace' (lometa, MonitoringEffect (MonitorAlterGlobalSeverity sev))
    evaluateAction sbtrace' _ _ (AlterSeverity loggerName sev) = do
        lometa ← mkLOMeta sev Public
        Trace.traceNamedObject sbtrace' (lometa, MonitoringEffect (MonitorAlterSeverity loggerName se
```

1.7.37 Cardano.BM.Backend.TraceAcceptor

TraceAcceptor is a backend responsible for processing LogObjects of an external process captured by a pipe or socket. At the time being it redirects the LogObjects to the *SwitchBoard*.

Plugin definition

```
plugin :: forall s a
  . (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ IOManager → Configuration → Trace.Trace IO a → s a → IO (Plugin a)
plugin iomgr cf basicTrace _ = getAcceptAt cf ≫= λcase
  Just acceptors → do
    socketsNServers ← forM acceptors $ λ(RemoteAddrNamed nodeName addr) → do
      let trace = Trace.appendName nodeName basicTrace
      (serverCleanup, serverThr) ← acceptorForAddress trace iomgr addr
      Async.link serverThr
      return (serverCleanup, serverThr)
    let (cleanups, servers) = unzip socketsNServers
      be :: (Cardano.BM.Backend . TraceAcceptor.TraceAcceptor a)
      be = TraceAcceptor
        {taServers    = servers
        ,taShutdown = sequence_ cleanups
        }
    return $ BackendPlugin
      (MkBackend {bEffectuate = effectuate be
        ,bUnrealize = unrealize be})
      (bekind be)
  Nothing → fail "TraceAcceptor not configured: no traceAcceptAt option"
```

Structure of TraceAcceptor

```
data TraceAcceptor a = TraceAcceptor
    {taServers :: [Async.Async ()]
    ,taShutdown :: IO ()
    }
instance IsEffectuator TraceAcceptor a where
    effectuate _ta _item = pure ()
    handleOverflow _ta = pure ()
instance (ToJSON a, FromJSON a) ⇒ IsBackend TraceAcceptor a where
    type BackendFailure TraceAcceptor = TraceAcceptorBackendFailure
    bekind _ = TraceAcceptorBK
    realize _ = fail "TraceAcceptor cannot be instantiated by 'realize'"
    realizefrom _ _ _ = fail "TraceAcceptor cannot be instantiated by 'realizefrom'"
    unrealize ta = do
      mapM_ Async.cancel $ taServers ta
      taShutdown ta
handleError :: (String → BackendFailure TraceAcceptor) → IO a → IO a
handleError ctor = handle $ λ(e :: IOException) → throwIO . ctor . show $ e
data TraceAcceptorBackendFailure
  = TraceAcceptorPipeError String
  | TraceAcceptorSocketError String
  | TraceAcceptorServerError String
  | TraceAcceptorClientThreadError String
```

```
    deriving (Show, Typeable)
instance Exception TraceAcceptorBackendFailure
acceptorForAddress
  :: FromJSON a
  ⇒ Trace.Trace IO a
  → IOManager
  → RemoteAddr
  → IO (IO (), Async.Async ())
acceptorForAddress trace iomgr (RemotePipe pipePath) =
  handleError TraceAcceptorPipeError $
  acceptorForSnocket
    trace
    Snocket.localFDToHandle
    (Snocket.localSnocket iomgr pipePath)
    (Snocket.localAddressFromPath pipePath)
acceptorForAddress trace iomgr (RemoteSocket host port) = handleError TraceAcceptorSocketError $ do
  let sn = Snocket.socketSnocket iomgr
  ainfos ← Socket.getAddrInfo Nothing (Just host) (Just port)
  case ainfos of
    [ ] → throwIO (TraceAcceptorSocketError ("bad socket address: " <> host <> ":" <> port))
    a: _ → acceptorForSnocket
      trace
      (flip Socket.socketToHandle IO.ReadWriteMode)
      sn
      (Socket.addrAddress a)
acceptorForSnocket
  :: forall a fd addr ∘ (FromJSON a)
  ⇒ Trace.Trace IO a
  → (fd → IO Handle)
  → Snocket.Snocket IO fd addr
  → addr
  → IO (IO (), Async.Async ())
acceptorForSnocket trace toHandle sn addr = do
  sock ← Snocket.mkListeningSocket sn (Just addr) (Snocket.addrFamily sn addr)
  server ← Async.async $
    bracket (pure sock) (Snocket.close sn) $
      λsock → acceptLoop $ Snocket.accept sn sock
  pure (Snocket.close sn sock, server)
where
  acceptLoop :: Snocket.Accept addr fd → IO ()
  acceptLoop (Snocket.Accept accept) = do
      (cfd, _caddr, k) ← accept
      h ← toHandle cfd
      _client ← Async.async $ clientThread trace h
      acceptLoop k
```

Reading log items from the client

```
clientThread
  :: forall a ∘ (FromJSON a)
```

```
    ⇒ Trace.Trace IO a
    → Handle
    → IO ()
clientThread sbtrace h = handleError TraceAcceptorClientThreadError pProc
  where
    {- lazy pProc -}
    pProc :: IO ()
    pProc = do
      hn ← BS.hGetLine h -- hostname
      bs ← BS.hGetLine h -- payload
      unless (BS.null bs) $ do
        let hname = decodeUtf8 hn
        case eitherDecodeStrict bs of
          Right lo →
            traceWith sbtrace (loName lo, lo)
          Left e → do
            lometa0 ← mkLOMeta Warning Public
            let trace :: Trace.Trace IO a
                trace = Trace.appendName "#external" sbtrace
                lometa = lometa0 {hostname = hname}
            Trace.traceNamedObject trace =≪
              (,) <$> pure lometa
                  <*> pure (LogError $ "Could not parse external log objects: " <> pack e)
      pProc
```

1.7.38 Cardano.BM.Backend.TraceForwarder

TraceForwarder is a backend responsible for redirecting logs to a different process (running a TraceAcceptor backend), by means of either a pipe or a socket.

The TraceForwarder is looking up a minimum Severity in the options section of the configuration. This filters out all messages that have not at least the Severity.

Plugin definition

```
plugin :: forall a s ∘ (IsEffectuator s a, ToJSON a, FromJSON a)
  ⇒ Configuration → Trace.Trace IO a → s a → Text → IO (Plugin a)
plugin config _trace _sb tfid = do
  opts ← getTextOption config tfid
  let minsev = case opts of
    Nothing → Debug
    Just sevtext → fromMaybe Debug (readMaybe $ unpack sevtext)
  be :: Cardano.BM.Backend ∘ TraceForwarder.TraceForwarder a ← realize config
  dispatcherThr ← spawnDispatcher (getTF be)
  modifyMVar_ (getTF be) $ λinitialBE →
    return $ initialBE
      {tfFilter = minsev
      ,tfDispatcher = Just dispatcherThr
      }
  return $ BackendPlugin
    (MkBackend {bEffectuate = effectuate be, bUnrealize = unrealize be})
    (bekind be)
```

Structure of TraceForwarder

Contains the handler to the pipe or to the socket.

```
newtype TraceForwarder a = TraceForwarder
  {getTF :: TraceForwarderMVar a}
type TraceForwarderMVar a = MVar (TraceForwarderInternal a)
data TraceForwarderInternal a = TraceForwarderInternal
  {tfQueue  :: TBQ.TBQueue (LogObject a)
  ,tfHandle :: Maybe Handle
  ,tfRemoteAddr :: RemoteAddr
  ,tfFilter     :: Severity
  ,tfDispatcher :: Maybe (Async.Async ())
  }
```

TraceForwarder is an effectuator

Every LogObject before being written to the given handler is converted to *ByteString* through its *JSON* representation.

```
instance (ToJSON a) ⇒ IsEffectuator TraceForwarder a where
  effectuate tf lo = do
    let currentMVar = getTF tf
    currentTF ← readMVar currentMVar
    when (severity (loMeta lo) ⩾ tfFilter currentTF) $ do
      let queue = tfQueue currentTF
      noCapacity ← atomically $ TBQ.isFullTBQueue queue
      if noCapacity
        then handleOverflow tf
        else atomically $ TBQ.writeTBQueue queue lo
  handleOverflow _ = TIO.hPutStrLn stderr "Notice: TraceForwarder's queue is full, dropping
```

TraceForwarder implements Backend functions

TraceForwarder is an IsBackend

```
instance (FromJSON a, ToJSON a) ⇒ IsBackend TraceForwarder a where
    type BackendFailure TraceForwarder = TraceForwarderBackendFailure
    bekind _ = TraceForwarderBK
    realize cfg = getForwardTo cfg ⩥ λcase
      Nothing → fail "Trace forwarder not configured:  option 'forwardTo'"
      Just addr → do
        queue ← atomically $ TBQ.newTBQueue queueMaxSize
        tfMVar ← newMVar $ TraceForwarderInternal
          {tfQueue  = queue
          ,tfHandle = Nothing
          ,tfRemoteAddr = addr
          ,tfFilter     = Debug
          ,tfDispatcher = Nothing
          }
        return $ TraceForwarder tfMVar
```

```
    unrealize tf = do
      currentTF ← readMVar (getTF tf)
      -- Cancel dispatcher thread.
      case tfDispatcher currentTF of
        Nothing → return ()
        Just thr → Async.uninterruptibleCancel thr
      -- If there's a handle - close it.
      closeHandle $ tfHandle currentTF
  closeHandle :: Maybe Handle → IO ()
  closeHandle (Just h) = hClose h
  closeHandle Nothing = return ()
  connectForwarder :: IOManager → RemoteAddr → IO Handle
  connectForwarder iomgr (RemotePipe pipePath) = do
    let sn = Snocket.localSnocket iomgr pipePath
    Snocket.localFDToHandle =≪ doConnect sn (Snocket.localAddressFromPath pipePath)
  connectForwarder iomgr (RemoteSocket host port) = do
    let sn = Snocket.socketSnocket iomgr
    addrs ← Socket.getAddrInfo Nothing (Just host) (Just port)
    case addrs of
      [ ] → throwIO (TraceForwarderSocketError ("bad socket address: " <> host <> ":" <> port))
      a: _ → doConnect sn (Socket.addrAddress a)
        ≫= flip Socket.socketToHandle ReadWriteMode
  doConnect :: Snocket.Snocket IO fd addr → addr → IO fd
  doConnect sn remoteAddr = do
    sd ← Snocket.openToConnect sn remoteAddr
    Snocket.connect sn sd remoteAddr
    pure sd
  data TraceForwarderBackendFailure
    = TraceForwarderConnectionError String
    | TraceForwarderSocketError String
    deriving (Show, Typeable)
  instance Exception TraceForwarderBackendFailure
```

Asynchronously reading log items from the queue and sending them to an acceptor.

```
  spawnDispatcher :: ToJSON a ⇒ TraceForwarderMVar a → IO (Async.Async ())
  spawnDispatcher tfMVar = Async.async $ processQueue
  where
    processQueue :: IO ()
    processQueue = do
      currentTF ← readMVar tfMVar
      -- Read the next log item from the queue.  If the queue is still empty -
      -- blocking and waiting for the next log item.
      nextItem ← atomically $ TBQ.readTBQueue (tfQueue currentTF)
      -- Try to write it to the handle.  If there's a problem with connection,
      -- this thread will initiate reestablishing of the connection and
      -- will wait until it's established.
      sendItem tfMVar nextItem
      -- Continue...
```

```
        processQueue
-- Try to send log item to the handle.
sendItem :: ToJSON a ⇒ TraceForwarderMVar a → LogObject a → IO ()
sendItem tfMVar lo =
  tfHandle <$> readMVar tfMVar >>= λcase
      Nothing → do
        -- There's no handle, initiate the connection.
        establishConnection 1 1 tfMVar
        -- Connection is reestablished, try to send log item.
        sendItem tfMVar lo
      Just h →
        try (BSC.hPutStrLn h $! encodedHostname) >>= λcase
          Right _ →
            -- Hostname was written to the handler successfully,
            -- try to write serialized LogObject.
            try (BSC.hPutStrLn h $! bs) >>= λcase
              Right _ →
                return ()-- Everything is ok, LogObject was written to the handler.
              Left (_e :: IOException) → do
                reConnectIfQueueIsAlmostFull
                threadDelay 10000
                sendItem tfMVar lo
          Left (_e :: IOException) → do
            reConnectIfQueueIsAlmostFull
            threadDelay 10000
            sendItem tfMVar lo
  where
    encodedHostname = encodeUtf8 (hostname ∘ loMeta $ lo)
    (_, bs) = jsonToBS lo
    jsonToBS :: ToJSON b ⇒ b → (Int, BS.ByteString)
    jsonToBS a =
        let bs' = BL.toStrict $ encode a
        in (BS.length bs', bs')
    -- Handle is bad, it looks like the connection is broken.
    -- Check if the queue is almost full.
    reConnectIfQueueIsAlmostFull = do
        currentTF ← readMVar tfMVar
        currentQueueSize ← atomically $ TBQ.lengthTBQueue (tfQueue currentTF)
        when (queueIsAlmostFull currentQueueSize) $ do
          -- The queue is almost full, it means that log items will be dropped soon.
          -- Initiate re-establishing of connection.
          closeHandle $ tfHandle currentTF
          modifyMVar_ tfMVar $ λbe → return $ be {tfHandle = Nothing}
    -- When the queue is almost full (80 percent of its max size)
    -- we initiate re-establishing of connection.
    queueIsAlmostFull queueSize = queueSize ⩾ round almostFullSize
      where
        almostFullSize :: Float
        almostFullSize = 0.8 * fromIntegral queueMaxSize
queueMaxSize :: Natural
```

```
queueMaxSize = 500
establishConnection :: Int → Int → TraceForwarderMVar a → IO ()
establishConnection delayInSec delayInSec' tfMVar = withIOManager $ λiomgr → do
  addr ← tfRemoteAddr <$> readMVar tfMVar
  try (connectForwarder iomgr addr) >>= λcase
      Right h →
        modifyMVar_ tfMVar $ λbe → return $ be {tfHandle = Just h}
      Left (e :: IOException) → do
        -- Cannot establish it, let's try again..
        threadDelay $ 1000000 * delayInSec'
        if delayInSec' < 60
          then
            -- Next attempt to re-establish the connection will be perform after Fibona
            establishConnection delayInSec' (delayInSec + delayInSec') tfMVar
          else
            -- Next attempt to re-establish the connection will be perform after fixed
            establishConnection 1 60 tfMVar
```

1.7.39 Cardano.BM.Scribe.Systemd

This plugin provides a scribe to *katip* to output logged items to systemd's journal on *Linux*.

Plugin definition

```
#ifdef LINUX
plugin :: (IsEffectuator s a, ToJSON a, FromJSON a)
    ⇒ Configuration → Trace IO a → s a → T.Text → IO (Plugin a)
plugin _ _ _ syslogIdent =
  ScribePlugin
    <$> mkJournalScribe syslogIdent
    <*> pure ("JournalSK::" <> syslogIdent)
#endif
```

Scribe definition

```
#ifdef LINUX
mkJournalScribe :: T.Text → IO K.Scribe
mkJournalScribe identifier = return $ journalScribe Nothing (sev2klog Debug) identifier K.V3
-- taken from https://github.com/haskell-service/katip-libsystemd-journal
journalScribe :: Maybe Facility
    → K.Severity
    → T.Text
    → K.Verbosity
    → K.Scribe
journalScribe facility severity identifier verbosity =
  K.Scribe liPush scribeFinalizer (pure ∘ const True)
where
  liPush :: K.LogItem a ⇒ K.Item a → IO ()
  liPush i = do
```

```
      permit ← K.permitItem severity i
      when permit $
        sendJournalFields $ itemToJournalFields facility identifier verbosity i
    scribeFinalizer :: IO ()
    scribeFinalizer = pure ()
# endif
```

Conversion utilities

Converts a *Katip Item* into a libsystemd-journal *JournalFields* map.

```
# ifdef LINUX
itemToJournalFields :: K.LogItem a
    ⇒ Maybe Facility
    → T.Text
    → K.Verbosity
    → K.Item a
    → JournalFields
itemToJournalFields facility identifier verbosity item =
  mconcat [defaultFields item
    ,maybe HM.empty facilityFields facility
    ,maybe HM.empty locFields (K._itemLoc item)
    ]
  where
    defaultFields kItem =
      mconcat [message (TL.toStrict $ toLazyText $ KC.unLogStr (KC._itemMessage kItem))
        ,priority (mapSeverity (KC._itemSeverity kItem))
        ,syslogIdentifier identifier
        ,HM.fromList [(environment, T.encodeUtf8 $ KC.getEnvironment (KC._itemEnv kItem))
          ,(namespace, T.encodeUtf8 $ unNS (KC._itemNamespace kItem))
          ,(payload, BL.toStrict $ encode $ KC.payloadObject verbosity (KC._itemPayload kItem))
          ,(thread, T.encodeUtf8 $ KC.getThreadIdText (KC._itemThread kItem))
          ,(time, T.encodeUtf8 $ formatAsIso8601 (KC._itemTime kItem))
          ]
        ]
    facilityFields = syslogFacility
    locFields Loc {..} = mconcat [codeFile loc_filename
      ,codeLine (fst loc_start)
      ]
    unNS ns = case K.unNamespace ns of
      [] → T.empty
      [p] → p
      parts → T.intercalate "." parts
    environment = mkJournalField "environment"
    namespace = mkJournalField "namespace"
    payload = mkJournalField "payload"
    thread = mkJournalField "thread"
    time = mkJournalField "time"
    mapSeverity s = case s of
      K.DebugS    → J.Debug
      K.InfoS     → J.Info
```

```
        K.NoticeS     → J.Notice
        K.WarningS    → J.Warning
        K.ErrorS      → J.Error
        K.CriticalS   → J.Critical
        K.AlertS      → J.Alert
        K.EmergencyS  → J.Emergency
    # endif
```

Chapter 2

Testing

2.1 Test main entry point

```
{-# LANGUAGE CPP #-}
module Main
  (
    main
  ) where
import Test.Tasty
import qualified Cardano.BM.Test.Aggregated (tests)
import qualified Cardano.BM.Test.STM (tests)
import qualified Cardano.BM.Test.Trace (tests)
import qualified Cardano.BM.Test.Configuration (tests)
import qualified Cardano.BM.Test.LogItem (tests)
import qualified Cardano.BM.Test.Rotator (tests)
import qualified Cardano.BM.Test.Routing (tests)
import qualified Cardano.BM.Test.Structured (tests)
import qualified Cardano.BM.Test.Tracer (tests)
main :: IO ()
main = defaultMain tests
tests :: TestTree
tests =
  testGroup "iohk-monitoring"
  [
    Cardano.BM.Test ∘ Aggregated.tests
  ,Cardano.BM.Test ∘ STM.tests
  ,Cardano.BM.Test ∘ Trace.tests
  ,Cardano.BM.Test ∘ Configuration.tests
  ,Cardano.BM.Test ∘ LogItem.tests
  ,Cardano.BM.Test ∘ Rotator.tests
  ,Cardano.BM.Test ∘ Routing.tests
  ,Cardano.BM.Test ∘ Structured.tests
  ,Cardano.BM.Test ∘ Tracer.tests
  ]
```

2.2 Test case generation

2.2.1 `instance Arbitrary Aggregated`

We define an instance of *Arbitrary* for an Aggregated which lets *QuickCheck* generate arbitrary instances of Aggregated. For this an arbitrary list of *Integer* is generated and this list is aggregated into a structure of Aggregated.

```
instance Arbitrary Aggregated where
  arbitrary = do
    vs' ← arbitrary :: Gen [Integer]
    let vs = 42 : 17 : vs'
        ds = map (λ(a, b) → a − b) $ zip vs (tail vs)
        (m1, s1) = updateMeanVar $ map fromInteger vs
        (m2, s2) = updateMeanVar $ map fromInteger ds
        mkBasicStats = BaseStats
          (PureI (minimum vs))
          (PureI (maximum vs))
          (fromIntegral $ length vs)
          (m1)
          (s1)
        mkDeltaStats = BaseStats
          (PureI (minimum ds))
          (PureI (maximum ds))
          (fromIntegral $ length ds)
          (m2)
          (s2)
        mkTimedStats = BaseStats
          (Nanoseconds 0)
          (Nanoseconds 0)
          (0)
          (0)
          (0)
    return $ AggregatedStats (Stats
      (PureI (last vs))
      (Nanoseconds 0)
      mkBasicStats
      mkDeltaStats
      mkTimedStats)
```

Estimators for mean and variance must be updated the same way as in the code.

```
updateMeanVar :: [Double] → (Double, Double)
updateMeanVar [ ] = (0, 0)
updateMeanVar (val : vals) = updateMeanVar' (val, 0) 1 vals
  where
    updateMeanVar' (m, s) _ [ ] = (m, s)
    updateMeanVar' (m, s) cnt (a : r) =
      let delta = a − m
          newcount = cnt + 1
          m' = m + (delta / newcount)
          s' = s + (delta ∗ (a − m'))
      in
      updateMeanVar' (m', s') newcount r
```

2.3 Tests

2.3.1 Cardano.BM.Test.LogItem

```
tests :: TestTree
tests = testGroup "Testing en/de-coding of LogItem" [
  testCase "en/de-code LogMessage" testLogMessage,
  testCase "en/de-code LogValue" testLogValue,
  testCase "en/de-code LogError" testLogError,
  testCase "en/de-code LogStructured" testLogStructured,
  testCase "en/de-code ObserveOpen" testObserveOpen,
  testCase "en/de-code ObserveDiff" testObserveDiff,
  testCase "en/de-code ObserveClose" testObserveClose,
  testCase "en/de-code AggregatedMessage" testAggregatedMessage,
  testCase "en/de-code MonitoringEffect" testMonitoringEffect,
  testCase "en/de-code Command" testCommand,
  testCase "en/de-code KillPill" testKillPill
  ]
```

En/de-coding tests

```
testLogMessage :: Assertion
testLogMessage = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta (LogMessage "hello")
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded
testLogValue :: Assertion
testLogValue = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta (LogValue "value" (PureI 42))
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded
testLogError :: Assertion
testLogError = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta (LogError "error")
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded
testLogStructured :: Assertion
testLogStructured = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta ∘ LogStructured $
    singleton "foo" (String "bar")
  let encoded = encode m
  let decoded = eitherDecode encoded :: Either String (LogObject Text)
  assertEqual "unequal" (Right m) decoded
```

```
testObserveOpen :: Assertion
testObserveOpen = do
  meta ← mkLOMeta Info Public
  let cs = CounterState [Counter StatInfo "some" (Bytes 789),
    Counter RTSStats "gcn" (PureI 42)]
  let m :: LogObject Text = LogObject "test" meta (ObserveOpen cs)
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded

testObserveDiff :: Assertion
testObserveDiff = do
  meta ← mkLOMeta Info Public
  let cs = CounterState [Counter StatInfo "some" (Bytes 789),
    Counter RTSStats "gcn" (PureI 42)]
  let m :: LogObject Text = LogObject "test" meta (ObserveDiff cs)
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded

testObserveClose :: Assertion
testObserveClose = do
  meta ← mkLOMeta Info Public
  let cs = CounterState [Counter StatInfo "some" (Bytes 789),
    Counter RTSStats "gcn" (PureI 42)]
  let m :: LogObject Text = LogObject "test" meta (ObserveClose cs)
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded

testAggregatedMessage :: Assertion
testAggregatedMessage = do
  meta ← mkLOMeta Info Public
  let as = [("test1",AggregatedEWMA (EWMA 0.8 (PureD 47.32))),
    ("test2",AggregatedStats (Stats 1 4 (BaseStats 0 1 2 0.5 0.5) (BaseStats 1 1 2 1 0) (BaseStats (−1) 3 2 77
  let m :: LogObject Text = LogObject "test" meta (AggregatedMessage as)
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded

testMonitoringEffect :: Assertion
testMonitoringEffect = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta (MonitoringEffect (MonitorAlterGlobalSeverity Notice
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded

testCommand :: Assertion
testCommand = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta (Command (DumpBufferedTo KatipBK))
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded
```

```
testKillPill :: Assertion
testKillPill = do
  meta ← mkLOMeta Info Public
  let m :: LogObject Text = LogObject "test" meta KillPill
  let encoded = encode m
  let decoded = decode encoded :: Maybe (LogObject Text)
  assertEqual "unequal" (Just m) decoded
```

2.3.2 Testing aggregation

```
tests :: TestTree
tests = testGroup "Aggregation measurements" [
  propertyTests
  ,unitTests1
  ,unitTests2
  ]
propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Aggregation_minimal
  ,testProperty "commutative" prop_Aggregation_comm
  ]
unitTests1 :: TestTree
unitTests1 = testGroup "Unit tests for Aggregated" [
  testCase "compare equal >" unitAggregatedEqualGT
  ,testCase "compare equal <" unitAggregatedEqualLT
  ,testCase "compare different >" unitAggregatedDiffGT
  ,testCase "compare different <" unitAggregatedDiffLT
  ]
unitTests2 :: TestTree
unitTests2 = testGroup "Unit tests for Aggregation" [
  testCase "initial -1" unitAggregationInitialMinus1
  ,testCase "initial +1" unitAggregationInitialPlus1
  ,testCase "initial +0" unitAggregationInitialZero
  ,testCase "initial +1, -1" unitAggregationInitialPlus1Minus1
  ,testCase "stepwise" unitAggregationStepwise
  ]
```

Property tests

```
prop_Aggregation_minimal :: Bool
prop_Aggregation_minimal = True
lometa :: LOMeta
lometa = unsafePerformIO $ mkLOMeta Debug Public
prop_Aggregation_comm :: Integer → Integer → Aggregated → Property
prop_Aggregation_comm v1 v2 ag =
  let ns = utc2ns $ tstamp lometa
    Right agg2 = updateAggregation (PureI v2) ag ns
    Right agg1 = updateAggregation (PureI v1) ag ns
```

```
    Right (AggregatedStats stats21) = updateAggregation (PureI v1) agg2 ns
    Right (AggregatedStats stats12) = updateAggregation (PureI v2) agg1 ns
  in
  fbasic stats21 === fbasic stats12 .&&.
  (v1 ≡ v2) `implies` (flast stats21 === flast stats12)
-- implication:  if p1 is true, then return p2; otherwise true
implies :: Bool → Property → Property
implies p1 p2 = property (¬ p1) .||. p2
```

Unit tests for Aggregation

```
unitAggregationInitialMinus1 :: Assertion
unitAggregationInitialMinus1 = do
    let ns = utc2ns $ tstamp lometa
      Right (AggregatedStats stats1) = updateAggregation (−1) firstStateAggregatedStats ns
    flast stats1 @? = (−1)
    (fbasic stats1) @? = BaseStats (−1) 0 2 (−0.5) 0.5
    (fdelta stats1) @? = BaseStats (−1) (−1) 2 (−1) 0
      -- AggregatedStats (Stats (-1) x (BaseStats (-1) 0 2 (-0.5) 0.5) (BaseStats (-1)
unitAggregationInitialPlus1 :: Assertion
unitAggregationInitialPlus1 = do
    let ns = utc2ns $ tstamp lometa
      Right (AggregatedStats stats1) = updateAggregation 1 firstStateAggregatedStats ns
    flast stats1 @? = 1
    (fbasic stats1) @? = BaseStats 0 1 2 0.5 0.5
    (fdelta stats1) @? = BaseStats 1 1 2 1 0
      -- AggregatedStats (Stats 1 x (BaseStats 0 1 2 0.5 0.5) (BaseStats 1 1 2 1 0) (B
unitAggregationInitialZero :: Assertion
unitAggregationInitialZero = do
    let ns = utc2ns $ tstamp lometa
      Right (AggregatedStats stats1) = updateAggregation 0 firstStateAggregatedStats ns
    flast stats1 @? = 0
    (fbasic stats1) @? = BaseStats 0 0 2 0 0
    (fdelta stats1) @? = BaseStats 0 0 2 0 0
      -- AggregatedStats (Stats 0 x (BaseStats 0 0 2 0 0) (BaseStats 0 0 2 0 0) (BaseS
unitAggregationInitialPlus1Minus1 :: Assertion
unitAggregationInitialPlus1Minus1 = do
    let ns = utc2ns $ tstamp lometa
      Right agg1 = updateAggregation (PureI 1) firstStateAggregatedStats ns
      Right (AggregatedStats stats1) = updateAggregation (PureI (−1)) agg1 ns
    (fbasic stats1) @? = BaseStats (PureI (−1)) (PureI 1) 3 0.0 2.0
    (fdelta stats1) @? = BaseStats (PureI (−2)) (PureI 1) 3 (−0.5) 4.5
unitAggregationStepwise :: Assertion
unitAggregationStepwise = do
    stats0 ← pure $ singletonStats (Bytes 3000)
    -- putStrLn (show stats0)
    threadDelay 50000-- 0.05 s
    t1 ← mkLOMeta Debug Public
    Right stats1 ← pure $ updateAggregation (Bytes 5000) stats0 (utc2ns $ tstamp t1)
    -- putStrLn (show stats1)
```

```
    -- showTimedMean stats1
    threadDelay 50000-- 0.05 s
    t2 ← mkLOMeta Debug Public
    Right stats2 ← pure $ updateAggregation (Bytes 1000) stats1 (utc2ns $ tstamp t2)
    -- putStrLn (show stats2)
    -- showTimedMean stats2
    checkTimedMean stats2
    threadDelay 50000-- 0.05 s
    t3 ← mkLOMeta Debug Public
    Right stats3 ← pure $ updateAggregation (Bytes 3000) stats2 (utc2ns $ tstamp t3)
    -- putStrLn (show stats3)
    -- showTimedMean stats3
    checkTimedMean stats3
    threadDelay 50000-- 0.05 s
    t4 ← mkLOMeta Debug Public
    Right stats4 ← pure $ updateAggregation (Bytes 1000) stats3 (utc2ns $ tstamp t4)
    -- putStrLn (show stats4)
    -- showTimedMean stats4
    checkTimedMean stats4
  where
    checkTimedMean (AggregatedEWMA _) = return ()
    checkTimedMean (AggregatedStats s) = do
      let mean = meanOfStats (ftimed s)
      assertBool "the mean should be >= the minimum" (mean ⩾ getDouble (fmin (ftimed s)))
      assertBool "the mean should be =< the maximum" (mean ⩽ getDouble (fmax (ftimed s)))
```

commented out:

```
showTimedMean (AggregatedEWMA _) = return ()
showTimedMean (AggregatedStats s) = putStrLn $ "mean = " ++ show (meanOfStats (ftimed s))
  ++ showUnits (fmin (ftimed s))


firstStateAggregatedStats :: Aggregated
firstStateAggregatedStats = AggregatedStats $
  Stats
    z
    z'
    (BaseStats z z 1 0 0)
    (BaseStats z z 1 0 0)
    (BaseStats z' z' 1 0 0)
  where
    z = PureI 0
    z' = Nanoseconds 0
```

Unit tests for Aggregated

```
unitAggregatedEqualGT :: Assertion
unitAggregatedEqualGT = do
  assertBool "comparing seconds"
    ((Seconds 3) > (Seconds 2))
```

```
  assertBool "comparing microseconds"
    ((Microseconds 3000) > (Microseconds 2000))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) > (Nanoseconds 2000000))
  assertBool "comparing bytes"
    ((Bytes 2048) > (Bytes 1024))
  assertBool "comparing doubles"
    ((PureD 2.34) > (PureD 1.42))
  assertBool "comparing integers"
    ((PureI 2) > (PureI 1))
  assertBool "comparing severities"
    ((Severity Error) > (Severity Warning))
unitAggregatedEqualLT :: Assertion
unitAggregatedEqualLT = do
  assertBool "comparing seconds"
    ((Seconds 2) < (Seconds 3))
  assertBool "comparing microseconds"
    ((Microseconds 2000) < (Microseconds 3000))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 2000000) < (Nanoseconds 3000000))
  assertBool "comparing bytes"
    ((Bytes 1024) < (Bytes 2048))
  assertBool "comparing doubles"
    ((PureD 1.34) < (PureD 2.42))
  assertBool "comparing integers"
    ((PureI 1) < (PureI 2))
  assertBool "comparing severities"
    ((Severity Info) < (Severity Notice))
unitAggregatedDiffGT :: Assertion
unitAggregatedDiffGT = do
  assertBool "comparing time (µs vs. s)"
    ((Microseconds 3000000) > (Seconds 2))
  assertBool "comparing time (µs vs. ns)"
    ((Microseconds 30) > (Nanoseconds 29999))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) > (Microseconds 2900))
  assertBool "comparing bytes"
    ((Bytes 2048) > (PureI 1024))
  assertBool "comparing doubles"
    ((PureD 2.34) > (PureI 1))
  assertBool "comparing integers"
    ((PureI 2) > (PureD 1.42))
unitAggregatedDiffLT :: Assertion
unitAggregatedDiffLT = do
  assertBool "comparing time (µs vs. s)"
    ((Microseconds 2999999) < (Seconds 3))
  assertBool "comparing time (µs vs. ns)"
    ((Microseconds 30) < (Nanoseconds 30001))
  assertBool "comparing nanoseconds"
    ((Nanoseconds 3000000) < (Microseconds 3001))
  assertBool "comparing bytes"
```

```
    ((PureI 1024) < (Bytes 2048))
  assertBool "comparing doubles"
    ((PureD 2.34) < (PureI 3))
  assertBool "comparing integers"
    ((PureI 2) < (PureD 3.42))
```

2.3.3 Cardano.BM.Test.STM

```
module Cardano.BM.Test.STM (
  tests
  ) where
import Test.Tasty
import Test.Tasty.QuickCheck
tests :: TestTree
tests = testGroup "Observing STM actions" [
  testProperty "minimal" prop_STM_observer
    ]
prop_STM_observer :: Bool
prop_STM_observer = True
```

2.3.4 Cardano.BM.Test.Trace

```
tests :: TestTree
tests = testGroup "Testing Trace" [
        unit_tests
    ,testCase "forked traces stress testing" stressTraceInFork
#ifdef ENABLE_OBSERVABLES
    ,testCase "stress testing: ObservableTraceSelf vs. NoTrace" timingObservableVsUntimed
    ,testCase "demonstrate capturing of counters" demoObservableSubtrace
#endif
    ,testCaseInfo "demonstrating logging" simpleDemo
    ,testCaseInfo "demonstrating nested named context logging" exampleWithNamedContexts
    ,testCase "major GC doesn't cause an exception for lost traces" unitShutdown
    ]
unit_tests :: TestTree
unit_tests = testGroup "Unit tests" [
       testCase "forked traces" unitTraceInFork
#ifdef ENABLE_OBSERVABLES
    ,testCase "opening messages should not be traced" unitNoOpeningTrace
#endif
  -- , testCase "hierarchy of traces" unitHierarchy
    ,testCase "hierarchy of traces with NoTrace" $
         unitHierarchy' [Neutral, NoTrace, ObservableTraceSelf observablesSet]
           onlyLevelOneMessage
    ,testCase "hierarchy of traces with DropOpening" $
         unitHierarchy' [Neutral, DropOpening, ObservableTraceSelf observablesSet]
           notObserveOpen
    ,testCase "hierarchy of traces with UntimedTrace" $
```

```
        unitHierarchy' [Neutral, UntimedTrace, UntimedTrace]
          observeNoMeasures
    , testCase "changing the minimum severity of a trace at runtime"
        unitTraceMinSeverity
    , testCase "changing the minimum severity of a named context at runtime"
        unitNamedMinSeverity
    , testCase "appending names" unitAppendName
    , testCase "create subtrace which duplicates messages" unitTraceDuplicate
    , testCase "testing name filtering" unitNameFiltering
    , testCase "testing throwing of exceptions" unitExceptionThrowing
    , testCase "NoTrace: check lazy evaluation" unitTestLazyEvaluation
    , testCase "private messages should not be logged into private files" unitLoggingPrivate
    ]
    where
      observablesSet = [MonotonicClock, MemoryStats]
      notObserveOpen :: [LogObject a] → Bool
      notObserveOpen = all (λcase {LogObject _ _ (ObserveOpen _) → False; _ → True})
      notObserveClose :: [LogObject a] → Bool
      notObserveClose = all (λcase {LogObject _ _ (ObserveClose _) → False; _ → True})
      notObserveDiff :: [LogObject a] → Bool
      notObserveDiff = all (λcase {LogObject _ _ (ObserveDiff _) → False; _ → True})
      onlyLevelOneMessage :: [LogObject Text] → Bool
      onlyLevelOneMessage = λcase
        [LogObject _ _ (LogMessage "Message from level 1.")] → True
        _ → False
      observeNoMeasures :: [LogObject a] → Bool
      observeNoMeasures obs = notObserveOpen obs ∧ notObserveClose obs ∧ notObserveDiff obs
```

Helper routines

```
data TraceConfiguration = TraceConfiguration
  {tcConfig      :: Configuration
  ,tcOutputKind :: MockSwitchboard Text
  ,tcName        :: LoggerName
  ,tcSubTrace    :: SubTrace
  }
setupTrace :: TraceConfiguration → IO (Trace IO Text)
setupTrace (TraceConfiguration cfg mockSB name subTr) = do
  let logTrace = traceMock mockSB cfg
  setSubTrace cfg name (Just subTr)
  return $ appendName name logTrace
```

Simple demo of logging.

```
simpleDemo :: IO String
simpleDemo = do
  cfg ← defaultConfigTesting
  logTrace :: Trace IO String ← Setup.setupTrace (Right cfg) "test"
```

```
  putStrLn "\n"
  logDebug    logTrace "This is how a Debug message looks like."
  logInfo     logTrace "This is how an Info message looks like."
  logNotice   logTrace "This is how a Notice message looks like."
  logWarning logTrace "This is how a Warning message looks like."
  logError    logTrace "This is how an Error message looks like."
  logCritical logTrace "This is how a Critical message looks like."
  logAlert    logTrace "This is how an Alert message looks like."
  logEmergency logTrace "This is how an Emergency message looks like."
  return ""
```

Example of using named contexts with Trace

```
exampleWithNamedContexts :: IO String
exampleWithNamedContexts = do
    cfg ← defaultConfigTesting
    Setup.withTrace cfg "test" $ λ(logTrace :: Trace IO Text) → do
      putStrLn "\n"
      logInfo logTrace "entering"
      let logTrace0 = appendName "simple-work-0" logTrace
      work0 ← complexWork0 cfg logTrace0 "0"
      let logTrace1 = appendName "complex-work-1" logTrace
      work1 ← complexWork1 cfg logTrace1 "42"
      Async.wait work0
      Async.wait work1
      -- the named context will include "complex" in the logged message
      logInfo logTrace "done."
      threadDelay 100000
      -- force garbage collection to allow exceptions to be thrown
      performMajorGC
      threadDelay 100000
    return ""
  where
    complexWork0 _ tr msg = Async.async $ logInfo tr ("let's see (0): " ‘append‘ msg)
    complexWork1 cfg tr msg = Async.async $ do
      logInfo tr ("let's see (1): " ‘append‘ msg)
      let trInner = appendName "inner-work-1" tr
          observablesSet = [MonotonicClock]
      setSubTrace cfg "test.complex-work-1.inner-work-1.STM-action" $
        Just $ ObservableTraceSelf observablesSet
# ifdef ENABLE_OBSERVABLES
      _ ← STMObserver.bracketObserveIO cfg trInner Debug "STM-action" setVar_
# endif
      logInfo trInner "let's see: done."
```

Show effect of turning off observables

```
# ifdef ENABLE_OBSERVABLES
runTimedAction :: Configuration → Trace IO Text → LoggerName → Int → IO Measurable
```

```
runTimedAction cfg logTrace name reps = do
    t0 ← getMonoClock
    forM_ [(1 :: Int)..reps] $ const $ observeAction logTrace
    t1 ← getMonoClock
    return $ diffTimeObserved (CounterState t0) (CounterState t1)
  where
    observeAction trace = do
      _ ← MonadicObserver.bracketObserveIO cfg trace Debug name action
      return ()
    action = return $ forM [1 :: Int..100] $ λx → [x] ++ (init $ reverse [1 :: Int..10000])
    diffTimeObserved :: CounterState → CounterState → Measurable
    diffTimeObserved (CounterState startCounters) (CounterState endCounters) =
      let
        startTime = getMonotonicTime startCounters
        endTime = getMonotonicTime endCounters
      in
        endTime – startTime
    getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
      [(Counter MonotonicClockTime _ mus)] → mus
      _ → error "A time measurement is missing!"
    isMonotonicClockCounter :: Counter → Bool
    isMonotonicClockCounter = (MonotonicClockTime ≡) ∘ cType
timingObservableVsUntimed :: Assertion
timingObservableVsUntimed = do
    cfg1 ← defaultConfigTesting
    msgs1 ← STM.newTVarIO [ ]
    traceObservable ← setupTrace $ TraceConfiguration cfg1
      (MockSB msgs1)
      "observables"
      (ObservableTraceSelf observablesSet)
    cfg2 ← defaultConfigTesting
    msgs2 ← STM.newTVarIO [ ]
    traceUntimed ← setupTrace $ TraceConfiguration cfg2
      (MockSB msgs2)
      "no timing"
      UntimedTrace
    cfg3 ← defaultConfigTesting
    msgs3 ← STM.newTVarIO [ ]
    traceNoTrace ← setupTrace $ TraceConfiguration cfg3
      (MockSB msgs3)
      "no trace"
      NoTrace
    t_observable ← runTimedAction cfg1 traceObservable "observables" 100
    t_untimed ← runTimedAction cfg2 traceUntimed "no timing" 100
    t_notrace  ← runTimedAction cfg3 traceNoTrace  "no trace"  100
    ms ← STM.readTVarIO msgs1
    assertBool
      ("Untimed consumed more time than ObservableTraceSelf " ++ (show [t_untimed,t_observabl
      (t_observable > t_untimed ∧ ¬ (null ms))
    assertBool
```

```
      ("NoTrace consumed more time than ObservableTraceSelf" ++ (show [t_notrace, t_observable])
      (t_observable > t_notrace)
    assertBool
      ("NoTrace consumed more time than Untimed" ++ (show [t_notrace, t_untimed]))
      True
  where
    observablesSet = [MonotonicClock, GhcRtsStats, MemoryStats, IOStats, ProcessStats]
#endif
```

Demonstrate observable subtrace

```
#ifdef ENABLE_OBSERVABLES
demoObservableSubtrace :: Assertion
demoObservableSubtrace = do
    ctrs1 ← readCounters (ObservableTraceSelf observablesSet)
    putStrLn "\n"
    logCounters ctrs1
    putStrLn "\n"
    threadDelay 50000
  where
    observablesSet = [MonotonicClock, GhcRtsStats, SysStats, IOStats, MemoryStats, NetStats, ProcessStat
    logCounters [] = pure ()
    logCounters (c : cs) = do
      putStrLn (show c)
      logCounters cs
#endif
```

Control tracing in a hierarchy of Traces

We can lay out traces in a hierarchical manner, that the children forward traced items to the parent Trace. A NoTrace introduced in this hierarchy will cut off a branch from messaging to the root.

```
_unitHierarchy :: Assertion
_unitHierarchy = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO []
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
  logInfo basetrace "This should have been displayed!"
  -- subtrace of trace which traces nothing
  setSubTrace cfg "test.inner" (Just NoTrace)
  let trace1 = appendName "inner" basetrace
  logInfo trace1 "This should NOT have been displayed!"
  setSubTrace cfg "test.inner.innermost" (Just Neutral)
  let trace2 = appendName "innermost" trace1
  logInfo trace2 "This should NOT have been displayed also due to the trace one level abo
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first message should have been traced
```

```
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res ≡ 1)
```

Change a trace's minimum severity

A trace is configured with a minimum severity and filters out messages that are labelled with a lower severity. This minimum severity of the current trace can be changed.

```
unitTraceMinSeverity :: Assertion
unitTraceMinSeverity = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO [ ]
  trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test min severity" Neutral
  logInfo trace "Message #1"
  -- raise the minimum severity to Warning
  setMinSeverity cfg Warning
  msev ← Cardano.BM.Configuration.minSeverity cfg
  assertBool ("min severity should be Warning, but is " ++ show msev)
    (msev ≡ Warning)
  -- this message will not be traced
  logInfo trace "Message #2"
  -- lower the minimum severity to Info
  setMinSeverity cfg Info
  -- this message is traced
  logInfo trace "Message #3"
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first and last messages should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res ≡ 2)
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all
      (λcase
        LogObject _ (LOMeta _ _ _ Info _) (LogMessage "Message #2") → False
        _ → True)
      res)
```

Define a subtrace's behaviour to duplicate all messages

The SubTrace will duplicate all messages that pass through it. Each message will be in its own named context.

```
unitTraceDuplicate :: Assertion
unitTraceDuplicate = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO [ ]
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test-duplicate" Neutral
```

```
    logInfo basetrace "Message #1"
    -- create a subtrace which duplicates all messages
    setSubTrace cfg "test-duplicate.orig" $ Just (TeeTrace "test-duplicate.dup")
    let trace = appendName "orig" basetrace
    -- this message will be duplicated
    logInfo trace "You will see me twice!"
    -- acquire the traced objects
    res ← STM.readTVarIO msgs
    -- only the first and last messages should have been traced
    assertBool
      ("Found more or less messages than expected: " ++ show res)
      (length res ≡ 3)
```

Change the minimum severity of a named context

A trace of a named context can be configured with a minimum severity, such that the trace will filter out messages that are labelled with a lower severity.

```
unitNamedMinSeverity :: Assertion
unitNamedMinSeverity = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO [ ]
  basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test-named-severity" Neutral
  let trace = appendName "sev-change" basetrace
  logInfo trace "Message #1"
  -- raise the minimum severity to Warning
  setSeverity cfg "test-named-severity.sev-change" (Just Warning)
  msev ← Cardano.BM.Configuration.inspectSeverity cfg "test-named-severity.sev-change"
  assertBool ("min severity should be Warning, but is " ++ (show msev))
    (msev ≡ Just Warning)
  -- this message will not be traced
  logInfo trace "Message #2"
  -- lower the minimum severity to Info
  setSeverity cfg "test-named-severity.sev-change" (Just Info)
  -- this message is traced
  logInfo trace "Message #3"
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first and last messages should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (length res ≡ 2)
  assertBool
    ("Found Info message when Warning was minimum severity: " ++ show res)
    (all
      (λcase
        LogObject _ (LOMeta _ _ _ Info _) (LogMessage "Message #2") → False
        _ → True)
      res)
```

```
unitHierarchy' :: [SubTrace] → ([LogObject Text] → Bool) → Assertion
unitHierarchy' subtraces f = do
  cfg ← liftIO Cardano.BM.Configuration ∘ Model.empty
  let (t1 : t2 : t3 : _) = cycle subtraces
  msgs ← STM.newTVarIO [ ]
  -- create trace of type 1
  trace1 ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" t1
  logInfo trace1 "Message from level 1."
  -- subtrace of type 2
  setSubTrace cfg "test.inner" (Just t2)
  let trace2 = appendName "inner" trace1
  logInfo trace2 "Message from level 2."
  -- subsubtrace of type 3
  setSubTrace cfg "test.inner.innermost" (Just t3)
# ifdef ENABLE_OBSERVABLES
  _ ← STMObserver.bracketObserveIO cfg trace2 Debug "innermost" setVar_
# endif
  logInfo trace2 "Message from level 3."
  -- acquire the traced objects
  res ← STM.readTVarIO msgs
  -- only the first message should have been traced
  assertBool
    ("Found more or less messages than expected: " ++ show res)
    (f res)
```

Logging in parallel

```
unitTraceInFork :: Assertion
unitTraceInFork = do
    cfg ← defaultConfigTesting
    msgs ← STM.newTVarIO [ ]
    trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
    let trace0 = appendName "work0" trace
        trace1 = appendName "work1" trace
    work0 ← work trace0
    threadDelay 5000
    work1 ← work trace1
    Async.wait $ work0
    Async.wait $ work1
    res ← STM.readTVarIO msgs
    let names@(_ : namesTail) = map loName res
    -- each trace should have its own name and log right after the other
    assertBool
      ("Consecutive loggernames are not different: " ++ show names)
      (and $ zipWith (≠) names namesTail)
  where
    work :: Trace IO Text → IO (Async.Async ())
    work trace = Async.async $ do
      logInfoDelay trace "1"
```

```
    logInfoDelay trace "2"
    logInfoDelay trace "3"
logInfoDelay :: Trace IO Text → Text → IO ()
logInfoDelay trace msg =
    logInfo trace msg ≫
    threadDelay 10000
```

Stress testing parallel logging

```
stressTraceInFork :: Assertion
stressTraceInFork = do
    cfg ← defaultConfigTesting
    msgs ← STM.newTVarIO [ ]
    trace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
    let names = map (λa → ("work-" <> pack (show a))) [1..(10 :: Int)]
    ts ← forM names $ λname → do
        let trace' = appendName name trace
        work trace'
    forM_ ts Async.wait
    res ← STM.readTVarIO msgs
    let resNames = map loName res
    let frequencyMap = fromListWith (+) [(x, 1) | x ← resNames]
    -- each trace should have traced totalMessages' messages
    assertBool
        ("Frequencies of logged messages according to loggername: " ++ show frequencyMap)
        (all (λname → (lookup ("test." <> name) frequencyMap) ≡ Just totalMessages) names)
  where
    work :: Trace IO Text → IO (Async.Async ())
    work trace = Async.async $ forM_ [1..totalMessages] $ (logInfo trace) ∘ pack ∘ show
    totalMessages :: Int
    totalMessages = 10
```

Dropping ObserveOpen messages in a subtrace

```
#ifdef ENABLE_OBSERVABLES
unitNoOpeningTrace :: Assertion
unitNoOpeningTrace = do
  cfg ← defaultConfigTesting
  msgs ← STM.newTVarIO [ ]
  logTrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" DropOpening
  _ ← STMObserver.bracketObserveIO cfg logTrace Debug "setTVar" setVar_
  res ← STM.readTVarIO msgs
  assertBool
    ("Found non-expected ObserveOpen message: " ++ show res)
    (all (λcase {LogObject _ _ (ObserveOpen _) → False; _ → True}) res)
#endif
```

Assert maximum length of log context name

The name of the log context cannot grow beyond a maximum number of characters, currently the limit is set to 80.

```
unitAppendName :: Assertion
unitAppendName = do
    cfg ← defaultConfigTesting
    msgs ← STM.newTVarIO [ ]
    basetrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "test" Neutral
    let trace1 = appendName bigName basetrace
        trace2 = appendName bigName trace1
    forM_ [basetrace, trace1, trace2] $ (flip logInfo msg)
    res ← reverse <$> STM.readTVarIO msgs
    let loggernames = map loName res
    assertBool
      ("AppendName did not work properly. The loggernames for the messages are: " ++
        show loggernames)
      (loggernames ≡ ["test"
        , "test." <> bigName
        , "test." <> bigName <> "." <> bigName
        ])
  where
    bigName = T.replicate 30 "abcdefghijklmnopqrstuvwxyz"
    msg = "Hello!"


# ifdef ENABLE_OBSERVABLES
setVar_ :: STM.STM Integer
setVar_ = do
  t ← STM.newTVar 0
  STM.writeTVar t 42
  res ← STM.readTVar t
  return res
# endif
```

Testing log context name filters

```
unitNameFiltering :: Assertion
unitNameFiltering = do
  let contextName = "test.sub.1"
  let loname = "sum"-- would be part of a "LogValue loname 42"
  let filter1 = [(Drop (Exact "test.sub.1"), Unhide [ ])]
  assertBool ("Dropping a specific name should filter it out and thus return False")
    (False ≡ evalFilters filter1 contextName)
  let filter2 = [(Drop (EndsWith ".1"), Unhide [ ])]
  assertBool ("Dropping a name ending with a specific text should filter out the context
    (False ≡ evalFilters filter2 contextName)
  let filter3 = [(Drop (StartsWith "test."), Unhide [ ])]
  assertBool ("Dropping a name starting with a specific text should filter out the contex
    (False ≡ evalFilters filter3 contextName)
```

```
let filter4 = [(Drop (Contains ".sub."), Unhide [])]
assertBool ("Dropping a name starting containing a specific text should filter out the
  (False ≡ evalFilters filter4 contextName)
let filter5 = [(Drop (StartsWith "test."),
    Unhide [(Exact "test.sub.1")])]
assertBool ("Dropping all and unhiding a specific name should the context name allow pa
  (True ≡ evalFilters filter5 contextName)
let filter6a = [(Drop (StartsWith "test."),
  Unhide [(EndsWith ".sum"),
    (EndsWith ".other")])]
assertBool ("Dropping all and unhiding some names, the LogObject should pass the filter
  (True ≡ evalFilters filter6a (contextName <> "." <> loname))
assertBool ("Dropping all and unhiding some names, another LogObject should not pass th
  (False ≡ evalFilters filter6a (contextName <> ".value"))
let filter6b = [(Drop (Contains "test."),
  Unhide [(Contains ".sum"),
    (Contains ".other")])]
assertBool ("Dropping all and unhiding some names, the LogObject should pass the filter
  (True ≡ evalFilters filter6b (contextName <> "." <> loname))
assertBool ("Dropping all and unhiding some names, another LogObject should not pass th
  (False ≡ evalFilters filter6b (contextName <> ".value"))
assertBool ("Dropping others and unhiding some names, something different should still
  (True ≡ evalFilters filter6b "some.other.value")
let filter7 = [(Drop (StartsWith "test."),
    Unhide [(EndsWith ".product")])]
assertBool ("Dropping all and unhiding an inexistant named value, the LogObject should
  (False ≡ evalFilters filter7 (contextName <> "." <> loname))
let filter8 = [(Drop (StartsWith "test."),
    Unhide [(Exact "test.sub.1")]),
  (Drop (StartsWith "something.else."),
    Unhide [(EndsWith ".this")])]
assertBool ("Disjunction of filters that should pass")
  (True ≡ evalFilters filter8 contextName)
let filter9 = [(Drop (StartsWith "test."),
    Unhide [(Exact ".that")]),
  (Drop (StartsWith "something.else."),
    Unhide [(EndsWith ".this")])]
assertBool ("Disjunction of filters that should not pass")
  (False ≡ evalFilters filter9 contextName)
```

Exception throwing

Exceptions encountered should be thrown. Lazy evaluation is really happening! This test fails if run with a configuration *defaultConfigTesting*, because this one will ignore all traced messages.

```
unitExceptionThrowing :: Assertion
unitExceptionThrowing = do
    action ← work msg
    res ← Async.waitCatch action
    assertBool
```

```
      ("Exception should have been rethrown")
      (isLeft res)
  where
    msg :: Text
    msg = error "faulty message"
    work :: Text → IO (Async.Async ())
    work message = Async.async $ do
      cfg ← defaultConfigStdout
      trace ← Setup.setupTrace (Right cfg) "test"
      logInfo trace message
      threadDelay 10000
```

Check lazy evaluation of trace

Exception should not be thrown when type of Trace is NoTrace.

```
unitTestLazyEvaluation :: Assertion
unitTestLazyEvaluation = do
    action ← work msg
    res ← Async.waitCatch action
    assertBool
      ("Exception should not have been rethrown when type of Trace is NoTrace")
      (isRight res)
  where
    msg :: Text
    msg = error "faulty message"
    work :: Text → IO (Async.Async ())
    work message = Async.async $ do
      cfg ← defaultConfigTesting
      basetrace ← Setup.setupTrace (Right cfg) "test"
      setSubTrace cfg "test.work" (Just NoTrace)
      let trace = appendName "work" basetrace
      logInfo trace message
```

Check that private messages do not end up in public log files.

```
unitLoggingPrivate :: Assertion
unitLoggingPrivate = do
    tmpDir ← getTemporaryDirectory
    let privateFile = tmpDir </> "private.log"
      publicFile = tmpDir </> "public.log"
    conf ← empty
    setDefaultBackends conf [KatipBK]
    setSetupBackends conf [KatipBK]
    setDefaultScribes conf ["FileSK::" <> pack privateFile
      ,"FileSK::" <> pack publicFile
      ]
    setSetupScribes conf [ScribeDefinition
```

```
        { scKind     = FileSK
        , scFormat   = ScText
        , scName     = pack privateFile
        , scPrivacy  = ScPrivate
        , scRotation = Nothing
        }
      , ScribeDefinition
        { scKind     = FileSK
        , scFormat   = ScText
        , scName     = pack publicFile
        , scPrivacy  = ScPublic
        , scRotation = Nothing
        }
      ]
    Setup.withTrace conf "test" $ λtrace → do
      -- should log in both files
      logInfo  trace message
      -- should only log in private file
      logInfoS trace message
    countPublic ← length ∘ lines <$> readFile publicFile
    countPrivate ← length ∘ lines <$> readFile privateFile
    -- delete files
    forM_ [privateFile, publicFile] removeFile
    assertBool
      ("Confidential file should contain 2 lines and it contains " ++ show countPrivate ++ "
        "Public file should contain 1 line and it contains " ++ show countPublic ++ ".\n"
      )
      (countPublic ≡ 1 ∧ countPrivate ≡ 2)
  where
    message :: Text
    message = "Just a message"
```

Verify that the shutdown-free sequence survives a major GC.

```
unitShutdown :: Assertion
unitShutdown = do
  _ :: (Trace IO Text, Switchboard Text)
    ← flip Setup.setupTrace_ "" =≪ empty
  threadDelay 1000
  performMajorGC
  threadDelay 1000
  assertBool "Win!" True
```

2.3.5 Testing configuration

Test declarations

```
tests :: TestTree
tests = testGroup "config tests" [
```

```
  propertyTests
  ,unitTests
  ]
propertyTests :: TestTree
propertyTests = testGroup "Properties" [
  testProperty "minimal" prop_Configuration_minimal
  ]
unitTests :: TestTree
unitTests = testGroup "Unit tests" [
  testCase "static representation" unitConfigurationStaticRepresentation
  ,testCase "parsed representation" unitConfigurationParsedRepresentation
  ,testCase "parsed configuration" unitConfigurationParsed
  ,testCase "export configuration: from file" unitConfigurationExport
  ,testCase "export configuration: defaultConfigStdout" unitConfigurationExportStdout
  ,testCase "check scribe caching" unitConfigurationCheckScribeCache
  ,testCase "test ops on Configuration" unitConfigurationOps
  ]
```

Property tests

```
prop_Configuration_minimal :: Bool
prop_Configuration_minimal = True
```

Unit tests

```
unitConfigurationStaticRepresentation :: Assertion
unitConfigurationStaticRepresentation =
  let r = Representation
        {minSeverity = Info
        ,rotation = Just $ RotationParameters
                             {rpLogLimitBytes = 5000000
                             ,rpMaxAgeHours  = 24
                             ,rpKeepFilesNum = 10
                             }
        ,setupScribes =
          [ScribeDefinition {scName = "stdout"
            ,scKind = StdoutSK
            ,scFormat           = ScText
            ,scPrivacy = ScPublic
            ,scRotation = Nothing}
          ]
        ,defaultScribes = [(StdoutSK,"stdout")]
        ,setupBackends = [EKGViewBK,KatipBK]
        ,defaultBackends = [KatipBK]
        ,hasGUI = Just 12789
        ,hasGraylog = Just 12788
        ,hasEKG = Just 18321
        ,hasPrometheus = Just ("localhost",12799)
        ,traceForwardTo = Just (RemotePipe "to")
```

```
      ,traceAcceptAt = Just [RemoteAddrNamed "a" (RemotePipe "at")]
      ,options =
        HM.fromList [("test1",Object (HM.singleton "value" "object1"))
          ,("test2",Object (HM.singleton "value" "object2"))]
      }
  in
  encode r @?=
    (intercalate "\n"
      ["rotation:"
      ,"  rpLogLimitBytes: 5000000"
      ,"  rpKeepFilesNum: 10"
      ,"  rpMaxAgeHours: 24"
      ,"defaultBackends:"
      ,"- KatipBK"
      ,"setupBackends:"
      ,"- EKGViewBK"
      ,"- KatipBK"
      ,"hasPrometheus:"
      ,"- localhost"
      ,"- 12799"
      ,"hasGraylog: 12788"
      ,"hasGUI: 12789"
      ,"traceForwardTo:"
      ,"  tag: RemotePipe"
      ,"  contents: to"
      ,"traceAcceptAt:"
      ,"- remoteAddr:"
      ,"    tag: RemotePipe"
      ,"    contents: at"
      ,"  nodeName: a"
      ,"defaultScribes:"
      ,"- - StdoutSK"
      ,"  - stdout"
      ,"options:"
      ,"  test2:"
      ,"    value: object2"
      ,"  test1:"
      ,"    value: object1"
      ,"setupScribes:"
      ,"- scName: stdout"
      ,"  scRotation: null"
      ,"  scKind: StdoutSK"
      ,"  scFormat: ScText"
      ,"  scPrivacy: ScPublic"
      ,"hasEKG: 18321"
      ,"minSeverity: Info"
      ,""-- to force a line feed at the end of the file
      ]
    )
unitConfigurationParsedRepresentation :: Assertion
unitConfigurationParsedRepresentation = do
```

```
repr ← readRepresentation "test/config.yaml"
encode repr @?=
  (intercalate "\n"
    ["rotation:"
    ,"  rpLogLimitBytes: 5000000"
    ,"  rpKeepFilesNum: 10"
    ,"  rpMaxAgeHours: 24"
    ,"defaultBackends:"
    ,"- KatipBK"
    ,"setupBackends:"
    ,"- AggregationBK"
    ,"- EKGViewBK"
    ,"- KatipBK"
    ,"hasPrometheus: null"
    ,"hasGraylog: 12788"
    ,"hasGUI: null"
    ,"traceForwardTo:"
    ,"  tag: RemotePipe"
    ,"  contents: to"
    ,"traceAcceptAt:"
    ,"- remoteAddr:"
    ,"    tag: RemotePipe"
    ,"    contents: at"
    ,"  nodeName: a"
    ,"defaultScribes:"
    ,"- - StdoutSK"
    ,"  - stdout"
    ,"options:"
    ,"  mapSubtrace:"
    ,"    iohk.benchmarking:"
    ,"      contents:"
    ,"      - GhcRtsStats"
    ,"      - MonotonicClock"
    ,"      subtrace: ObservableTraceSelf"
    ,"    iohk.deadend:"
    ,"      subtrace: NoTrace"
    ,"  mapSeverity:"
    ,"    iohk.startup: Debug"
    ,"    iohk.background.process: Error"
    ,"    iohk.testing.uncritical: Warning"
    ,"  mapAggregatedkinds:"
    ,"    iohk.interesting.value: EwmaAK {alpha = 0.75}"
    ,"    iohk.background.process: StatsAK"
    ,"  cfokey:"
    ,"    value: Release-1.0.0"
    ,"  mapMonitors:"
    ,"    chain.creation.block:"
    ,"      actions:"
    ,"      - CreateMessage Warning \"chain.creation\""
    ,"      - AlterSeverity \"chain.creation\" Debug"
    ,"      monitor: ((time > (23 s)) Or (time < (17 s)))"
```

```
        ,"    '#aggregation.critproc.observable':"
        ,"      actions:"
        ,"      - CreateMessage Warning \"the observable has been too long too high!\""
        ,"      - SetGlobalMinimalSeverity Info"
        ,"      monitor: (mean >= (42))"
        ,"  mapScribes:"
        ,"    iohk.interesting.value:"
        ,"    - StdoutSK::stdout"
        ,"    - FileSK::testlog"
        ,"    iohk.background.process: FileSK::testlog"
        ,"  mapBackends:"
        ,"    iohk.user.defined:"
        ,"    - kind: UserDefinedBK"
        ,"      name: MyBackend"
        ,"    - KatipBK"
        ,"    iohk.interesting.value:"
        ,"    - EKGViewBK"
        ,"    - AggregationBK"
        ,"setupScribes:"
        ,"- scName: testlog"
        ,"  scRotation:"
        ,"    rpLogLimitBytes: 25000000"
        ,"    rpKeepFilesNum: 3"
        ,"    rpMaxAgeHours: 24"
        ,"  scKind: FileSK"
        ,"  scFormat: ScText"
        ,"  scPrivacy: ScPrivate"
        ,"- scName: stdout"
        ,"  scRotation: null"
        ,"  scKind: StdoutSK"
        ,"  scFormat: ScText"
        ,"  scPrivacy: ScPublic"
        ,"hasEKG: 12789"
        ,"minSeverity: Info"
        ,""-- to force a line feed at the end of the file
        ]
    )
unitConfigurationParsed :: Assertion
unitConfigurationParsed = do
  cfg ← setup "test/config.yaml"
  cfgInternal ← readMVar $ getCG cfg
  cfgInternal @? = ConfigurationInternal
    {cgMinSeverity     = Info
    ,cgDefRotation     = Just $ RotationParameters
                         {rpLogLimitBytes = 5000000
                         ,rpMaxAgeHours = 24
                         ,rpKeepFilesNum = 10
                         }
    ,cgMapSeverity     = HM.fromList [("iohk.startup",Debug)
                         ,("iohk.background.process",Error)
                         ,("iohk.testing.uncritical",Warning)
```

```
                        ]
  ,cgMapSubtrace     = HM.fromList [("iohk.benchmarking",
                                    ObservableTraceSelf [GhcRtsStats,MonotonicClock])
                                   ,("iohk.deadend",NoTrace)
                                   ]
  ,cgOptions         = HM.fromList
    [("mapSubtrace",
      Object $
      HM.fromList [("iohk.benchmarking",
                    Object (HM.fromList [("subtrace",String "ObservableTraceSelf")
                                        ,("contents",Array $ V.fromList
                                                    [String "GhcRtsStats"
                                                    ,String "MonotonicClock"])]))
                   ,("iohk.deadend",
                    Object (HM.fromList [("subtrace",String "NoTrace")]))])
    ,("mapMonitors",Object $
                    HM.fromList [("chain.creation.block",Object (HM.fromList
                        [("monitor",String "((time > (23 s)) Or (time < (17 s)))")
                        ,("actions",Array $ V.fromList
                            [String "CreateMessage Warning \"chain.creation\""
                            ,String "AlterSeverity \"chain.creation\" Debug"
                            ])]))
                      ,("#aggregation.critproc.observable",Object (HM.fromList
                        [("monitor",String "(mean >= (42))")
                        ,("actions",Array $ V.fromList
                            [String "CreateMessage Warning \"the observable has been too
                            ,String "SetGlobalMinimalSeverity Info"
                            ])]))])
    ,("mapSeverity",Object $
                    HM.fromList [("iohk.startup",String "Debug")
                      ,("iohk.background.process",String "Error")
                      ,("iohk.testing.uncritical",String "Warning")])
    ,("mapAggregatedkinds",Object $
      HM.fromList [("iohk.interesting.value",
                                    String "EwmaAK {alpha = 0.75}")
                                   ,("iohk.background.process",
                                    String "StatsAK")])
    ,("cfokey",Object $ HM.fromList [("value",String "Release-1.0.0")])
    ,("mapScribes",Object $ HM.fromList [("iohk.interesting.value",
                        Array $ V.fromList [String "StdoutSK::stdout"
                                                                ,String "FileSK::testlog"])
      ,("iohk.background.process",String "FileSK::testlog")])
    ,("mapBackends",Object $
                    HM.fromList [("iohk.user.defined",
                            Array $ V.fromList [Object (HM.fromList [("kind",String "UserDefinedB
                              ,("name",String "MyBackend")])
                              ,String "KatipBK"
                              ])
                            ,("iohk.interesting.value",
                            Array $ V.fromList [String "EKGViewBK"
                              ,String "AggregationBK"
```

```
                                ])])
    ]
  ,cgMapBackend     = HM.fromList [("iohk.user.defined"
                                   ,[UserDefinedBK "MyBackend"
                                     ,KatipBK
                                     ]
                                   )
                                  ,("iohk.interesting.value"
                                   ,[EKGViewBK
                                     ,AggregationBK
                                     ]
                                   )
                                  ]
  ,cgDefBackendKs   = [KatipBK]
  ,cgSetupBackends  = [AggregationBK
                      ,EKGViewBK
                      ,KatipBK
                      ]
  ,cgMapScribe      = HM.fromList [("iohk.interesting.value",
                                    ["StdoutSK::stdout","FileSK::testlog"])
                                   ,("iohk.background.process",["FileSK::testlog"])
                                   ]
  ,cgMapScribeCache = HM.fromList [("iohk.interesting.value",
                                    ["StdoutSK::stdout","FileSK::testlog"])
                                   ,("iohk.background.process",["FileSK::testlog"])
                                   ]
  ,cgDefScribes     = ["StdoutSK::stdout"]
  ,cgSetupScribes   = [ScribeDefinition
                       {scKind     = FileSK
                       ,scFormat   = ScText
                       ,scName     = "testlog"
                       ,scPrivacy  = ScPrivate
                       ,scRotation = Just $ RotationParameters
                         {rpLogLimitBytes = 25000000
                         ,rpMaxAgeHours = 24
                         ,rpKeepFilesNum = 3
                         }
                       }
                      ,ScribeDefinition
                       {scKind = StdoutSK
                       ,scFormat = ScText
                       ,scName = "stdout"
                       ,scPrivacy = ScPublic
                       ,scRotation = Nothing
                       }
                      ]
  ,cgMapAggregatedKind = HM.fromList [("iohk.interesting.value",EwmaAK {alpha = 0.75})
                                     ,("iohk.background.process",StatsAK)
                                     ]
  ,cgDefAggregatedKind = StatsAK
  ,cgMonitors       = HM.fromList [("chain.creation.block"
```

```
                                    ,(Nothing
                                      ,(OR (Compare "time" (GT,(OpMeasurable (Agg.Seconds 23)))) (Compare
                                      ,[CreateMessage Warning "chain.creation"
                                        ,AlterSeverity "chain.creation" Debug
                                        ]
                                      )
                                    )
                                   ,("#aggregation.critproc.observable"
                                    ,(Nothing
                                      ,Compare "mean" (GE,(OpMeasurable (Agg.PureI 42)))
                                      ,[CreateMessage Warning "the observable has been too long too
                                        ,SetGlobalMinimalSeverity Info
                                        ]
                                      )
                                    )
                                   ]
      ,cgPortEKG          = 12789
      ,cgPortGraylog      = 12788
      ,cgBindAddrPrometheus = Nothing
      ,cgPortGUI          = 0
      ,cgForwardTo        = Just (RemotePipe "to")
      ,cgAcceptAt         = Just [RemoteAddrNamed "a" (RemotePipe "at")]
      }
unitConfigurationExport :: Assertion
unitConfigurationExport = do
  cfg ← setup "test/config.yaml"
  cfg' ← withSystemTempFile "config.yaml-1213" $ λfile0 _ → do
      let file = file0 <> "-copy"
      exportConfiguration cfg file
      setup file
  cfgInternal ← readMVar $ getCG cfg
  cfgInternal' ← readMVar $ getCG cfg'
  cfgInternal' @? = cfgInternal
unitConfigurationExportStdout :: Assertion
unitConfigurationExportStdout = do
  cfg ← defaultConfigStdout
  cfg' ← withSystemTempFile "config.yaml-1213" $ λfile0 _ → do
      let file = file0 <> "-copy"
      exportConfiguration cfg file
      setup file
  cfgInternal ← readMVar $ getCG cfg
  cfgInternal' ← readMVar $ getCG cfg'
  cgMinSeverity       cfgInternal' @? = cgMinSeverity       cfgInternal
  cgDefRotation       cfgInternal' @? = cgDefRotation       cfgInternal
  cgMapSeverity       cfgInternal' @? = cgMapSeverity       cfgInternal
  cgMapSubtrace       cfgInternal' @? = cgMapSubtrace       cfgInternal
  cgOptions           cfgInternal' @? = cgOptions           cfgInternal
  cgMapBackend        cfgInternal' @? = cgMapBackend        cfgInternal
  cgDefBackendKs      cfgInternal' @? = cgDefBackendKs      cfgInternal
```

```
    cgSetupBackends      cfgInternal' @? = cgSetupBackends      cfgInternal
    cgMapScribe          cfgInternal' @? = cgMapScribe          cfgInternal
    cgMapScribeCache     cfgInternal' @? = cgMapScribeCache     cfgInternal
    cgDefScribes         cfgInternal' @? = cgDefScribes         cfgInternal
    cgSetupScribes       cfgInternal' @? = cgSetupScribes       cfgInternal
    cgMapAggregatedKind cfgInternal' @? = cgMapAggregatedKind cfgInternal
    cgDefAggregatedKind  cfgInternal' @? = cgDefAggregatedKind  cfgInternal
    cgMonitors           cfgInternal' @? = cgMonitors           cfgInternal
    cgPortEKG            cfgInternal' @? = cgPortEKG            cfgInternal
    cgPortGraylog        cfgInternal' @? = cgPortGraylog        cfgInternal
    cgBindAddrPrometheus cfgInternal' @? = cgBindAddrPrometheus cfgInternal
    cgPortGUI            cfgInternal' @? = cgPortGUI            cfgInternal
    cfgInternal' @? = cfgInternal
```

Test caching and inheritance of Scribes.

```
unitConfigurationCheckScribeCache :: Assertion
unitConfigurationCheckScribeCache = do
  configuration ← empty
  let defScribes = ["FileSK::node.log"]
  setDefaultScribes configuration defScribes
  let scribes12 = ["StdoutSK::stdout","FileSK::out.txt"]
  setScribes configuration "name1.name2" $ Just scribes12
  scribes1234 ← getScribes configuration "name1.name2.name3.name4"
  scribes1 ← getScribes configuration "name1"
  scribes1234cached ← getCachedScribes configuration "name1.name2.name3.name4"
  scribesXcached ← getCachedScribes configuration "nameX"
  assertBool "Scribes for name1.name2.name3.name4 must be the same as name1.name2" $
    scribes1234 ≡ scribes12
  assertBool "Scribes for name1 must be the default ones" $
    scribes1 ≡ defScribes
  assertBool "Scribes for name1.name2.name3.name4 must have been cached" $
    scribes1234cached ≡ Just scribes1234
  assertBool "Scribes for nameX must not have been cached since getScribes was not calle
    scribesXcached ≡ Nothing
```

Test operations on Configuration.

```
unitConfigurationOps :: Assertion
unitConfigurationOps = do
  configuration ← defaultConfigStdout
  defBackends ← getDefaultBackends configuration
  setDefaultAggregatedKind configuration $ EwmaAK 0.01
  -- since loggername does not exist the default must be inherited
  defAggregatedKind ← getAggregatedKind configuration "non-existent loggername"
  setAggregatedKind configuration "name1" $ Just StatsAK
  name1AggregatedKind ← getAggregatedKind configuration "name1"
  setEKGport configuration 11223
  ekgPort ← getEKGport configuration
  setGUIport configuration 1080
```

```
  guiPort ← getGUIport configuration
  assertBool "Default backends" $
    defBackends ≡ [KatipBK]
  assertBool "Default aggregated kind" $
    defAggregatedKind ≡ EwmaAK 0.01
  assertBool "Specific name aggregated kind" $
    name1AggregatedKind ≡ StatsAK
  assertBool "Set EKG port" $
    ekgPort ≡ 11223
  assertBool "Set GUI port" $
    guiPort ≡ 1080
```

2.3.6 Rotator

```
tests :: TestTree
tests = testGroup "testing Trace" [
  property_tests
  ]
property_tests :: TestTree
property_tests = testGroup "Property tests" [
  testProperty "rotator: file naming" propNaming
#ifdef POSIX
  ,testProperty "rotator: cleanup" $ propCleanup $ rot n
#endif
  ]
#ifdef POSIX
  where
    n = 5
    rot num = RotationParameters
      {rpLogLimitBytes = 10000000-- 10 MB
      ,rpMaxAgeHours = 24
      ,rpKeepFilesNum = num
      }
#endif
```

Check that the generated file name has only 15 digits added to the base name.

```
propNaming :: FilePath → Property
propNaming name = ioProperty $ do
  filename ← nameLogFile name
  return $ length filename === length name + 15
```

Test cleanup of rotator.

This test creates a random number of files with the same name but with different dates and afterwards it calls the *cleanupRotator* function which removes old log files keeping only rpKeepFilesNum files and deleting the others.

```
# ifdef POSIX
data LocalFilePath = Dir FilePath
  deriving (Show)
instance Arbitrary LocalFilePath where
  arbitrary = do
      start ← QC.sized $ λn → replicateM (n + 1) (QC.elements $ [ 'a' .. 'z' ])
      x ← QC.sized $ λn → replicateM n (QC.elements $ [ 'a' .. 'd' ] ++ "/")
      pure $ Dir $ start ++ removeAdjacentAndLastSlashes x
  shrink (Dir path) = map (Dir ∘ removeAdjacentAndLastSlashes ∘ (intercalate "/")) $
      product' $ map (filter (≢ "")) $ map QC.shrink (splitOn "/" path)
    where
      product' :: [[a]] → [[a]]
      product' = mapM (λx → x ≫= return)
removeAdjacentAndLastSlashes :: FilePath → FilePath
removeAdjacentAndLastSlashes = concat ∘ filter (≢ "/") ∘ groupBy (\_b → b ≢ '/')
data SmallAndLargeInt = SL Int
  deriving (Show)
instance Arbitrary SmallAndLargeInt where
  arbitrary = do
      QC.oneof [smallGen
        ,largeGen
        ]
    where
      smallGen :: QC.Gen SmallAndLargeInt
      smallGen = do
        QC.Small x ← (QC.arbitrary :: QC.Gen (QC.Small Int))
        pure $ SL $ abs x
      largeGen :: QC.Gen SmallAndLargeInt
      largeGen = do
        let maxBoundary = 00100000000000-- 10 years for the format which is used
            minBoundary = 00000000010000-- 1 hour for the format which is used
        x ← QC.choose (minBoundary,maxBoundary)
        pure $ SL x
  shrink _ = [ ]
data NumFiles = NF Int deriving (Show)
instance Arbitrary NumFiles where
  arbitrary = QC.oneof [return (NF 0),return (NF 1),return (NF 5),return (NF 7)]
propCleanup :: RotationParameters → LocalFilePath → NumFiles → SmallAndLargeInt → Property
propCleanup rotationParams (Dir filename) (NF nFiles) (SL maxDev) = QC.withMaxSuccess 20 $ ioProperty
  tmpDir0 ← getTemporaryDirectory
  let tmpDir = tmpDir0 </> "rotatorTest.base"
  let path = tmpDir </> filename
  -- generate nFiles different dates
  now ← getCurrentTime
  let tsnow = formatTime defaultTimeLocale tsformat now
  deviations ← replicateM nFiles $ QC.generate $ QC.choose (1,maxDev + 1)
  -- TODO if generated within the same sec we have a problem
  let dates = map show $ scanl (+) (read tsnow) deviations
      files = map (λa → path ++ ('-' : a)) dates
```

```
        sortedFiles = reverse $ sort files
        keepFilesNum = fromIntegral $ rpKeepFilesNum rotationParams
        toBeKept = reverse $ take keepFilesNum sortedFiles
    createDirectoryIfMissing True $ takeDirectory path
    forM_ (files) $ λf → openFile f WriteMode
    cleanupRotator rotationParams path
    filesRemained ← listLogFiles path
    let kept = case filesRemained of
          Nothing → [ ]
          Just l → NE.toList l
    removeDirectoryRecursive tmpDir
    return $ kept === toBeKept
# endif
```

2.3.7 Cardano.BM.Test.Structured

```
tests :: TestTree
tests = testGroup "Testing Structured Logging" [
  testCase "logging simple text" logSimpleText
  ,testCase "logging data structures" logStructured
  ,testCase "logging with filtering" logFiltered
  ,testCase "logging data structures (stdout)" logStructuredStdout
  ]
```

Simple logging of text

Trace textual messages. This is not structured logging and only here for reference.

```
logSimpleText :: Assertion
logSimpleText = do
  cfg ← defaultConfigTesting
  baseTrace :: Trace IO Text ← Setup.setupTrace (Right cfg) "logSimpleText"
  traceWith (toLogObject baseTrace) ("This is a simple message." :: Text)
  traceWith (toLogObject baseTrace) (".. and another!" :: String)
  assertBool "OK" True
```

Structured logging

This test shows how a user-defined structure *Pet* can be traced. The trTransformer by default is the nullTracer. Therefore, an instance of *Transformable Text IO Pet* uses the transformer trStructured to create a structured log item using the ToObject instance. The function toObject depends on the verbosity level and in case of MinimalVerbosity will return an emptyObject and not output the structure at all. The output in NormalVerbosity level will be a shortened structure with just its type. Only in MaximalVerbosity level will the complete structure be output.

```
data Pet = Pet {name :: Text, age :: Int}
  deriving (Show)
```

```
instance ToObject Pet where
  toObject MinimalVerbosity _ = emptyObject-- do not log
  toObject NormalVerbosity (Pet _ _) =
    mkObject ["kind" .= String "Pet"]
  toObject MaximalVerbosity (Pet n a) =
    mkObject ["kind" .= String "Pet"
      ,"name" .= toJSON n
      ,"age" .= toJSON a]
instance Transformable Text IO Pet where
  -- transform to JSON Object
  trTransformer MaximalVerbosity tr = trStructured MaximalVerbosity tr
  trTransformer MinimalVerbosity _tr = nullTracer
  -- transform to textual representation using show
  trTransformer _v tr = Tracer $ λpet → do
    meta ← mkLOMeta Info Public
    traceWith tr $ ("pet",LogObject "pet" meta $ (LogMessage ∘ pack ∘ show) pet)
-- default privacy annotation:  Public
instance HasPrivacyAnnotation Pet
-- default severity:  Debug
instance HasSeverityAnnotation Pet
logStructured :: Assertion
logStructured = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO []
  baseTrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "logStructured" Neutral
  let noticeTracer = severityNotice baseTrace
  let confidentialTracer = annotateConfidential baseTrace
  let pet = Pet "bella" 8
  traceWith (toLogObject noticeTracer) (42 :: Integer)
  traceWith (toLogObject confidentialTracer) pet
  traceWith (toLogObjectMinimal confidentialTracer) pet
  ms ← STM.readTVarIO msgs
  assertBool
    ("assert number of messages traced == 2: " ++ (show $ length ms))
    (2 ≡ length ms)
  assertBool
    ("verify traced integer with severity Notice: " ++ (show ms))
    (Notice ≡ severity (loMeta (ms !! 1)))
  assertBool
    ("verify traced structure with privacy annotation Confidential: " ++ (show ms))
    (Confidential ≡ privacy (loMeta (ms !! 0)))
logStructuredStdout :: Assertion
logStructuredStdout = do
  cfg ← defaultConfigStdout
  baseTrace :: Trace IO Text ← Setup.setupTrace (Right cfg) "logStructured"
  let noticeTracer = severityNotice baseTrace
  let confidentialTracer = annotateConfidential baseTrace
  let pet = (Pet "bella" 8)
  traceWith (toLogObject noticeTracer) (42 :: Integer)
```

```
  traceWith (toLogObject confidentialTracer) pet
  traceWith (toLogObjectVerbose confidentialTracer) pet
  traceWith (toLogObjectMinimal confidentialTracer) pet
  traceWith (toLogObject MinimalVerbosity noticeTracer) (42 :: Integer)
  traceWith (toLogObject MinimalVerbosity confidentialTracer) pet
  traceWith (toLogObject MaximalVerbosity noticeTracer) pet
  assertBool "OK" True
```

Structured logging with filtering

```
data Material = Material {description :: Text, weight :: Int}
  deriving (Show)
instance ToObject Material where
  toObject MinimalVerbosity _ = emptyObject-- do not log
  toObject NormalVerbosity (Material d _) =
    mkObject ["kind" .= String "Material"
      ,"description" .= toJSON d]
  toObject MaximalVerbosity (Material d w) =
    mkObject ["kind" .= String "Material"
      ,"description" .= toJSON d
      ,"weight" .= toJSON w]
instance Transformable Text IO Material where
  -- transform to JSON Object
  trTransformer MaximalVerbosity tr = trStructured MaximalVerbosity tr
  trTransformer MinimalVerbosity _tr = nullTracer
  -- transform to textual representation using show
  trTransformer _v tr = Tracer $ λmat → do
    meta ← mkLOMeta Info Public
    traceWith tr $ ("material",LogObject "material" meta $ (LogMessage ∘ pack ∘ show) mat)
instance HasPrivacyAnnotation Material where
  getPrivacyAnnotation _ = Confidential
instance HasSeverityAnnotation Material where
  getSeverityAnnotation (Material _d w) =
    if w < 100
    then Debug
    else Info
logFiltered :: Assertion
logFiltered = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO []
  baseTrace ← setupTrace $ TraceConfiguration cfg (MockSB msgs) "logStructured" Neutral
  let stone = Material "stone" 1400
    water = Material "H20" 1000
    dust = Material "dust" 13
    confidentialTracer = annotatePrivacyAnnotation
      $ filterPrivacyAnnotation (pure ∘ const Confidential)
      $ toLogObject $ baseTrace
    infoTracer = annotateSeverity
      $ filterSeverity (pure ∘ const Info)
```

```
      $ toLogObject $ baseTrace
  traceWith confidentialTracer stone
  traceWith infoTracer water
  traceWith infoTracer dust-- does not pass severity filter
  ms ← STM.readTVarIO msgs
  assertBool
    ("assert number of messages traced == 2: " ++ (show $ length ms))
    (2 ≡ length ms)
```

2.3.8 Cardano.BM.Test.Tracer

```
tests :: TestTree
tests = testGroup "Testing Extensions to Tracer" [
  testCase "simple tracing of messages in a named context" tracingInNamedContext,
  testCase "tracing with privacy and severity annotation" tracingWithPrivacyAndSeverityAnnota
  testCase "tracing with a predicate filter" tracingWithPredicateFilter,
  testCase "tracing with a filter that is evaluated in a monad" tracingWithMonadicFilter,
  testCase "tracing with filtering for both severity and privacy" tracingWithComplexFilterin
  testCase "eliding equivalent messages on tracer" tracingElidedMessages,
  testCase "eliding equivalent messages only one" tracingElidedMessages1,
  testCase "eliding equivalent messages only two" tracingElidedMessages2,
  testCase "eliding equivalent messages from three" tracingElidedMessages3,
  testCase "eliding messages, output after n repeats" tracingElidedMessagesRepeat
  ]
```

Helper routines

```
data TraceConfiguration = TraceConfiguration
  { _tcConfig       :: Configuration
  , _tcOutputKind   :: MockSwitchboard Text
  , _tcName         :: LoggerName
  , _tcSubTrace     :: SubTrace
  }
setupMockTrace :: TraceConfiguration → IO (Trace IO Text)
setupMockTrace (TraceConfiguration cfg mockSB name subTr) = do
  let logTrace = traceMock mockSB cfg
  setSubTrace cfg name (Just subTr)
  return $ appendName name logTrace


renderNamedItemTracing' :: Show a ⇒ Tracer m String → Trace m a
renderNamedItemTracing' = contramap $ λ(ctx, item) →
  unpack ctx ++ ": " ++ show (loContent item) ++ ", (meta): " ++ show (loMeta item)
```

Tracing messages in a named context

```
tracingInNamedContext :: Assertion
tracingInNamedContext = do
```

```
  let logTrace = appendName "named" $ renderNamedItemTracing' $ stdoutTracer
  void $ callFun2 logTrace
  assertBool "OK" True
callFun2 :: Trace IO Text → IO Int
callFun2 logTrace = do
  let logTrace' = appendName "fun2" logTrace
  traceWith (toLogObject logTrace') ("in function 2" :: Text)
  callFun3 logTrace'
callFun3 :: Trace IO Text → IO Int
callFun3 logTrace = do
  traceWith (toLogObject $ appendName "fun3" $ logTrace) ("in function 3" :: Text)
  return 42
```

Tracing messages with pricacy and severity annotation

A Tracer transformer creating a LogObject from *PrivacyAndSeverityAnnotated*.

```
logObjectFromAnnotated :: Show a
  ⇒ Trace IO a
  → Tracer IO (PrivacyAndSeverityAnnotated a)
logObjectFromAnnotated tr = Tracer $ λ(PSA sev priv a) → do
  lometa ← mkLOMeta sev priv
  traceWith tr $ (mempty, LogObject mempty lometa (LogMessage a))
```

```
tracingWithPrivacyAndSeverityAnnotation :: Assertion
tracingWithPrivacyAndSeverityAnnotation = do
  let logTrace =
    logObjectFromAnnotated $ appendName "example3" $ renderNamedItemTracing' stdoutTracer
  traceWith logTrace $ PSA Info Confidential ("Hello" :: String)
  traceWith logTrace $ PSA Warning Public "World"
  assertBool "OK" True
```

Filter Tracer

```
filterAppendNameTracing :: Monad m
    ⇒ m ((LoggerName, LogObject a) → Bool)
    → LoggerName
    → Trace m a
    → Trace m a
filterAppendNameTracing test name = (appendName name) ∘ (condTracingM test)
tracingWithPredicateFilter :: Assertion
tracingWithPredicateFilter = do
    let appendF = filterAppendNameTracing oracle
      logTrace :: Trace IO Text = appendF "example4" (renderNamedItemTracing' stdoutTracer)
    traceWith (toLogObject logTrace) ("Hello" :: String)
    let logTrace' = appendF "inner" logTrace
    traceWith (toLogObject logTrace') ("World" :: String)
```

```
    let logTrace'' = appendF "innest" logTrace'
    traceWith (toLogObject logTrace'') ("!!" :: String)
    assertBool "OK" True
  where
    oracle :: Monad m => m ((LoggerName, LogObject a) -> Bool)
    oracle = return $ λ(ctx, _lo) -> ctx /= "example4.inner"


-- severity anotated
tracingWithMonadicFilter :: Assertion
tracingWithMonadicFilter = do
    let logTrace =
      condTracingM oracle $
        logObjectFromAnnotated $
          appendName "test5" $ renderNamedItemTracing' stdoutTracer
    traceWith logTrace $ PSA Debug Confidential ("Hello" :: String)
    traceWith logTrace $ PSA Warning Public "World"
    assertBool "OK" True
  where
    oracle :: Monad m => m (PrivacyAndSeverityAnnotated a -> Bool)
    oracle = return $ λ(PSA sev _priv _) -> (sev > Debug)
```

tracing with combined filtering for name and severity

```
tracingWithComplexFiltering :: Assertion
tracingWithComplexFiltering = do
    let logTrace0 = -- the basis, will output using the local renderer to stdout
          appendName "test6" $ renderNamedItemTracing' stdoutTracer
        logTrace1 = -- the trace from Privacy...Annotated to LogObject
          condTracingM oracleSev $ logObjectFromAnnotated $ logTrace0
        logTrace2 =
          appendName "row" $ condTracingM oracleName $ logTrace0
        logTrace3 = -- oracle should eliminate messages from this trace
          appendName "raw" $ condTracingM oracleName $ logTrace0
    traceWith logTrace1 $ PSA Debug Confidential ("Hello" :: String)
    traceWith logTrace1 $ PSA Warning Public "World"
    lometa <- mkLOMeta Info Public
    traceWith logTrace2 $ (mempty, LogObject mempty lometa (LogMessage ", RoW!"))
    traceWith logTrace3 $ (mempty, LogObject mempty lometa (LogMessage ", RoW!"))
    assertBool "OK" True
  where
    oracleSev :: Monad m => m (PrivacyAndSeverityAnnotated a -> Bool)
    oracleSev = return $ λ(PSA sev _priv _) -> (sev > Debug)
    oracleName :: Monad m => m ((LoggerName, LogObject a) -> Bool)
    oracleName = return $ λ(ctx, _lo) -> (ctx == "row")
```

Tracer transformer for eliding repeated messages

```
data MsgTy = Item1 Int
  | Elided1 Int
  | Elided2 Int
  deriving (Show)
```

```
instance HasSeverityAnnotation MsgTy
instance HasPrivacyAnnotation MsgTy
instance Transformable Text IO MsgTy where
  trTransformer _verb tr = Tracer $ λs → do
    meta ← mkLOMeta (getSeverityAnnotation s) (getPrivacyAnnotation s)
    traceWith tr ("",LogObject mempty
      meta
      (LogMessage $ pack $ show s))
instance ElidingTracer (WithSeverity MsgTy) where
  -- only Elided1 and Elided2 can be elided
  doelide (WithSeverity _s (Elided1 _)) = True
  doelide (WithSeverity _s (Elided2 _)) = True
  doelide _ = False
  -- any Elided1 is equivalent to another Elided1
  isEquivalent (WithSeverity _ (Elided1 _)) (WithSeverity _ (Elided1 _)) = True
  -- instances of Elided2 are equivalent if they are equal
  isEquivalent (WithSeverity _ (Elided2 n1)) (WithSeverity _ (Elided2 n2)) = n1 ≡ n2
  isEquivalent _ _ = False
  conteliding _tverb _tr _ (Nothing, _count) = return (Nothing, 0)
  conteliding _tverb tr ev (_old,count) = do
    when (count > 0 ∧ count `mod` 100 ≡ 0) $ do -- report every 100th elided messages
      meta ← mkLOMeta (getSeverityAnnotation ev) (getPrivacyAnnotation ev)
      traceNamedObject tr (meta, LogValue "messages elided" (PureI $ toInteger count))
    return (Just ev,count + 1)
tracingElidedMessages :: Assertion
tracingElidedMessages = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO [ ]
  baseTrace ← setupMockTrace $ TraceConfiguration cfg (MockSB msgs) "eliding" Neutral
  s_elide ← newstate
  let msg11 = Elided1 1400
    msg12 = Elided1 1000
    msg21 = Elided2 999
    msg22 = Elided2 998
    msg23 = Elided2 998
    msg31 = Item1 42
    msg32 = Item1 42
    infoTracer = annotateSeverity
      $ elideToLogObject NormalVerbosity s_elide $ baseTrace
  traceWith infoTracer msg11
  traceWith infoTracer msg12
  traceWith infoTracer msg31
  traceWith infoTracer msg11
  traceWith infoTracer msg12-- elided
  traceWith infoTracer msg12-- elided
  traceWith infoTracer msg11
  traceWith infoTracer msg31
  traceWith infoTracer msg21
  traceWith infoTracer msg22
  traceWith infoTracer msg23
```

```
    traceWith infoTracer msg31
    traceWith infoTracer msg32
    traceWith infoTracer msg31
    traceWith infoTracer msg32
    traceWith infoTracer msg31
    ms ← STM.readTVarIO msgs
    assertBool
      ("assert number of messages traced == 15: " ++ show (reverse $ map loContent ms) ++ " len =
      (15 ≡ length ms)
```

The first elided message is output and the internal counter of elided messages is set to zero. When the non-equivalent message is traced, the last elided message is not output since this is the same as the first one.

```
tracingElidedMessages1 :: Assertion
tracingElidedMessages1 = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO [ ]
  baseTrace ← setupMockTrace $ TraceConfiguration cfg (MockSB msgs) "eliding2" Neutral
  s_elide ← newstate
  let msg11 = Elided1 1400
      msg31 = Item1 42
      tracer = annotateSeverity
        $ elideToLogObject NormalVerbosity s_elide $ baseTrace
  traceWith tracer msg11
  traceWith tracer msg31
  ms ← STM.readTVarIO msgs
  assertBool
    ("assert number of messages traced == 2: " ++ (show $ reverse $ map loContent ms))
    (2 ≡ length ms)
```

The first message is output. When the non-equivalent message is traced, the last message is output. Since the first and last messages are output, no count of elided messages is reported.

```
tracingElidedMessages2 :: Assertion
tracingElidedMessages2 = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO [ ]
  baseTrace ← setupMockTrace $ TraceConfiguration cfg (MockSB msgs) "eliding1" Neutral
  s_elide ← newstate
  let msg11 = Elided1 1400
      msg12 = Elided1 1000
      msg31 = Item1 42
      tracer = annotateSeverity
        $ elideToLogObject NormalVerbosity s_elide $ baseTrace
  traceWith tracer msg11
  traceWith tracer msg12
  traceWith tracer msg31
  ms ← STM.readTVarIO msgs
  assertBool
```

```
    ("assert number of messages traced == 3: " ++ (show $ reverse $ map loContent ms))
    (3 ≡ length ms)
```

The second tracing of *msg12* increases the internal counter of elided messages to two. One (2 - 1) elided message is reported, and the last message is output.

```
tracingElidedMessages3 :: Assertion
tracingElidedMessages3 = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO [ ]
  baseTrace ← setupMockTrace $ TraceConfiguration cfg (MockSB msgs) "eliding3" Neutral
  s_elide ← newstate
  let msg11 = Elided1 1400
      msg12 = Elided1 1000
      msg31 = Item1 42
      tracer = annotateSeverity
        $ elideToLogObject NormalVerbosity s_elide $ baseTrace
  traceWith tracer msg11
  traceWith tracer msg12
  traceWith tracer msg12-- elided
  traceWith tracer msg31
  ms ← STM.readTVarIO msgs
  assertBool
    ("assert number of messages traced == 4: " ++ (show $ reverse $ map loContent ms))
    (4 ≡ length ms)
```

An elided message is output every *n* occurences.

```
tracingElidedMessagesRepeat :: Assertion
tracingElidedMessagesRepeat = do
  cfg ← defaultConfigStdout
  msgs ← STM.newTVarIO [ ]
  baseTrace ← setupMockTrace $ TraceConfiguration cfg (MockSB msgs) "eliding3" Neutral
  s_elide ← newstate
  let msg11 = Elided1 1400
      msg12 = Elided1 1000
      msg31 = Item1 42
      tracer = annotateSeverity
        $ elideToLogObject NormalVerbosity s_elide $ baseTrace
  traceWith tracer msg11
  traceWith tracer msg12
  let mlist = map Elided1 [1..320]
  forM_ mlist $ λm → traceWith tracer m
  traceWith tracer msg31
  ms ← STM.readTVarIO msgs
  assertBool
    ("assert number of messages traced == 7: " ++ (show $ reverse $ map loContent ms))
    (7 ≡ length ms)
```

Index