UCSP: Implementation

Dmitry K.

July 8, 2016

Abstract

This article proposes a system for generating possible *University Classes Schedules*. It uses multi-agent negotiation to find satisfactory solutions to the problem, while trying to consider *personal preferences* of the represented people and institutions.

Contents

1 Implementation **1**
1.1 University Classes 1
1.2 Negotiating Agents 4
1.2.1 Common Goal 5
1.3 Coherence 5
1.3.1 Information 5
1.3.2 Relations 7
1.3.3 Information graph 8
1.4 Contexts 8
1.4.1 Capabilities 11
1.4.2 Beliefs 13
1.4.3 Obligations 16
1.4.4 Preferences 17
1.4.5 External 18
1.5 Agent 20
1.5.1 Behavior definition 20
1.5.2 Role-depending behavior 21
1.5.3 Referencing agents 22
1.5.4 Agent control 22
1.5.5 Agent extended referencing 23
1.5.6 Agent Creation 24
1.5.7 Agent Management 24

1 Implementation

1.1 University Classes

A class is an en event, that brings together a *group of students*, and a *professor* in certain *classroom* in order to learn/teach the specified *discipline*. It happens periodically, usually weekly, at the established *day of week* and *time*.

A *discipline* should describe an atomic (not dividable) educational activity. For example, if the students are required to take a normal class and also do some specific laboratory practice, then two disciplines should be created, one of them describing the required lab equipment.

```haskell
data Discipline = Discipline { disciplineId                :: String
                             , disciplineMinutesPerWeek    :: Int
                             , disciplineRequirements      :: Set Requirement
                             }
                deriving (Typeable, Show, Eq, Ord)

newtype Requirement = Requirement String deriving (Show, Eq, Ord)
```

For inner usage, the classes are divided into

- *abstract* — without day and time;
- *concrete* — with full time information.

```haskell
class (Ord c, Show c, Typeable c) =>
    AbstractClass c where classDiscipline :: c -> Discipline
                          classGroup      :: c -> GroupRef
                          classProfessor  :: c -> ProfessorRef
                          classRoom       :: c -> ClassroomRef
                          classNumber     :: c -> Word

class (AbstractClass c, DiscreteTime time) =>
  ConcreteClass c time | c -> time
    where classDay    :: c -> Day
          classBegins :: c -> time
          classEnds   :: c -> time

data Class     = forall c time. ConcreteClass c time => Class c
data SomeClass = forall c     . AbstractClass c      => SomeClass c
```

The "System.Time.Day" is redefined, dropping the "Sunday".

```haskell
data Day = Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday
  deriving (Eq, Ord, Enum, Bounded, Ix, Read, Show)
```

The classes are negotiated by the interested parties: 1) students / groups, 2) professors, 3) classrooms. Each negotiation participant has a *timetable*, holding a schedule for one week, that repeats throughout the academic period. The *timetable* is actually a table: the columns represent days of week; the rows — discrete time intervals. Actual timetable structure may vary, as can be seen in figure 1.

```haskell
class (Ord t, Bounded t, Show t, Typeable t) => DiscreteTime t where
  toMinutes   :: t -> Int
  fromMinutes :: Int -> t

data SomeTime = forall t.(DiscreteTime t) => SomeTime t

someTimeMinutes (SomeTime t) = toMinutes t
```

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:00						
09:00 – 09:30						
09:30 – 10:00						
10:00 – 10:30						
10:30 – 11:00						
11:00 – 11:30						
11:30 – 12:00						
⋮ ⋮						

(a) Timetable without recesses.

	Mon	Tue	Wed	Thu	Fri	Sat
08:30 – 09:10						
09:15 – 09:55						
10:05 – 10:45						
10:50 – 11:30						
11:40 – 12:20						
12:25 – 13:05						
13:15 – 13:55						
⋮ ⋮						

(b) Timetable with recesses.

Figure 1: Possible *timetable* structures.

```haskell
instance Eq  SomeTime where (≡)     = (≡)     `on` someTimeMinutes
instance Ord SomeTime where compare = compare `on` someTimeMinutes

class (DiscreteTime time) ⇒ Timetable tt e time | tt → time
                                                 , tt → e
                                                 , e  → time
  where listEvents :: tt → [e]
        eventsOn   :: tt → Day  → [e]
        eventsAt   :: tt → time → [(Day, e)]
        eventAt    :: tt → Day  → time → Maybe e
```

One should distinguish the resulting timetables, shown in figure 1 and the timetable, held an agent during the negotiation. The first one is immutable and is the result of agent's participation in the negotiation. The set of such timetables, produced by every the participant, is the **university schedule** for given academic period.

During the negotiation, an agent's inner timetable gets changed on the fly, in order to record agreements made. This means that we are dealing with *side effects*, that need to be explicitly denoted in Haskell. The following definition leaves it free to choose the monad abstraction for those effects.

```
class (DiscreteTime time, Monad m) ⇒
  TimetableM tt m e time | tt → time
                         , tt → e
                         , e  → time
  where putEvent   :: tt → e → m tt
        delEvent   :: tt → e → m tt
        ttSnapshot :: (Timetable ts x time) ⇒ tt → m ts
```

1.2 Negotiating Agents

As it was mentioned before, the schedule is formed in a negotiation between *professors*, *groups* and *classrooms*. To distinguish those three types of participants, agent's role is introduced. The role: 1) identifies the kind of person/entity, represented by the agent; 2) defines agent's reaction on the messages received; 3) defines agent's goal.

A *representing agent* is a computational entity, that represents a *real person or object* in it's virtual environment. In current case, it represents one's interests in a *negotiation*. Such an agent must

(1) pursue the *common goal* — it must consider the common benefits, while being egoistic enough to achieve it's own goal;

(2) respond to the messages received in correspondence with (1);

(3) initiate conversations (send messages, that are not responses), driven by (1);

(4) become more susceptible (less egoistic) with passage of time.

```
data NegotiationRole = GroupRole
                     | FullTimeProfRole
                     | PartTimeProfRole
                     | ClassroomRole
  deriving (Show, Typeable)

data Role' (r :: NegotiationRole) = Role'

instance Show (Role' GroupRole) where
  show _ = "Role: Group"
instance Show (Role' FullTimeProfRole) where
  show _ = "Role: Professor (full time)"
instance Show (Role' PartTimeProfRole) where
  show _ = "Role: Professor (part time)"
instance Show (Role' ClassroomRole) where
  show _ = "Role: Classroom"
  -- ----------------------------------------

class RoleIx r where roleIx :: Role' r → Int
```

```
-- ---------------------------------------------
data AnyRole = ∀r.(Show (Role' r), RoleIx r) ⇒
       AnyRole (Role' r)
roleIx' (AnyRole r) = roleIx r
instance Show AnyRole where show (AnyRole r) = show r
instance Eq   AnyRole where (≡) = (≡) `on` roleIx'
instance Ord  AnyRole where compare = compare `on` roleIx'
```

1.2.1 Common Goal

Agent's own *goal* represents its egoistical interests. They may (and will) contradict another agent's interests, thus creating *incoherence*. The general rule in this case is to strive for solutions, benefiting the whole schedule. Because the schedule doesn't yet exist as a whole during the negotiation, an agent should consider instead the benefits, obtained by itself and the rest of the agents.

The *common goal* is incorporated in the *contexts* mechanism, and is discussed in Section 1.4.5.

1.3 Coherence

The coherence mechanism is based on [**?**]. It uses the *contexts* as means of separating (and further prioritizing) different *cognitive aspects*. The contexts used are based on *BDI* agent architecture.

The *combined coherence* is used as a measure of goal achievement. It's combined of coherence values, calculated by agent's contexts.

The *binary relations* connect some information pieces, assigning to the edge some value. The *whole graph relations*, on the other side, are applied to the graph as a whole and produce a single value.

The relations used, as well as the information in the graph, depend on the *context*.

The coherence is calculated over an *information graph*, that represents some aspect of agent's knowledge. The nodes of the graph are some *pieces of information* and the edges represent some *relations* between theese pieces.

1.3.1 Information

The proposed system makes use of the following information:

1. **Personal knowledge**, known only by one actor.

   (a) **Capabilites**: information about what an agent can do, what kind of arrangments it can make.

   (b) **Obligations**: information about *strong restrictions*, imposed over the agent.

   (c) **Preferences**: information about *weak restrictions*.

2. **Shared knowledge**, obtained in the negotiation.
   (a) **Others' capabilities** — information about the counterpart agents, that are known to be (un-) capable of doing something.
   (b) **Classes proposals**:
      i. **Abstract** — has no specific time assigned.
         Concrete — has a specific time defined.
      ii. **Complete** — references all three representing agents: a *group*, a *professor* and a *classroom*.
          Partial — references less then three representing agents.
   (c) **Classes decisions**:
      i. **Class acceptance** — a mark for *accepted classes proposals*. Only *complete* proposals can be accepted; all the three mentioned agents must accept it, or none.
      ii. **Class rejection** — a mark for *ignored classes proposals*, a result of *yield* decision, discussed in Section **??**.

data *InformationScope* = *Personal* | *Shared*

-- "Ord" instance is mainly needed to create "Set"s.
class $(Typeable\ i, Eq\ i, Ord\ i) \Rightarrow InformationPiece\ i$
    where type $IScope\ i :: InformationScope$

class $(InformationPiece\ i, Personal{\sim}IScope\ i) \Rightarrow PersonalInformation\ i$
class $(InformationPiece\ i, Shared{\sim}IScope\ i) \quad \Rightarrow SharedInformation\ i$
    where $sharedBetween :: i \rightarrow Set\ AgentRef$

-- ————————————————

instance *Eq SomeClass* **where**
    $(SomeClass\ a) \equiv (SomeClass\ b) = cast\ a \equiv Just\ b$

        -- TODO
instance *Ord SomeClass*
instance *InformationPiece SomeClass* **where**
    type $IScope\ SomeClass = Shared$

instance *SharedInformation SomeClass*

-- ————————————————

instance *Eq Class*
instance *Ord Class*
instance *InformationPiece Class* **where type** $IScope\ Class = Shared$
instance *SharedInformation Class*

-- ————————————————

data $Information = \forall i.InformationPiece\ i \Rightarrow Information\ i$

$collectInf :: (Typeable\ a) \Rightarrow Information \rightarrow Maybe\ a$
$collectInf\ (Information\ i) = cast\ i$

```
instance Eq Information where
  (Information i1) ≡ (Information i2) =
    case cast i1 of Just x      → x ≡ i2
                    _           → False
instance Ord Information where
  (Information i1) ≦≧ (Information i2) = ⊥
  -- ────────────────────────────
newtype Needs = Needs (Set Discipline)
    deriving (Eq, Ord, Show, Typeable)
newtype CanTeach = CanTeach (Set Discipline)
    deriving (Eq, Ord, Show, Typeable)
instance InformationPiece Needs
instance InformationPiece CanTeach
```

1.3.2 Relations

```
class InformationRelation r where
  relationName   :: r a → String
  coerceRelation :: (Coercible a b) ⇒ r a → r b
class InformationRelation r ⇒
  BinaryRelation r where
    binRelValue :: (Num a) ⇒ r a → Information → Information → Maybe a
class InformationRelation r ⇒
  WholeRelation r where
    wholeRelValue :: r a → IGraph → a
class InformationRelation r ⇒
  BinaryIORelation r where
    binRelIOValue :: (Num a, Typeable a, Show a) ⇒ r a → Information → Information → IOMaybe a
type IOMaybe a = IO (Maybe a)
  -- ────────────────────────────
data RelValBetween a = RelValBetween {
                relBetween    :: (Information, Information)
        ,       relValBetween :: a
        }
type RelValsBetween a = Map (IRelation a) [RelValBetween a]
newtype RelValWhole a = RelValWhole a
unwrapRelValWhole (RelValWhole a) = a
type RelValsWhole a = Map (IRelation a) (RelValWhole a)
  -- ────────────────────────────
data IRelation a = ∀r.BinaryRelation r   ⇒ RelBin (r a)
```

```
                     | ∀r.BinaryIORelation r ⇒ RelBinIO (r a)
                     | ∀r.WholeRelation r    ⇒ RelWhole (r a)

relName (RelBin a)   = relationName a
relName (RelWhole a) = relationName a

instance Eq (IRelation a) where (≡) = (≡) ‘on‘ relName

instance Ord (IRelation a) where compare = compare ‘on‘ relName

coerceIRelation :: (Coercible a b) ⇒ IRelation a → IRelation b
coerceIRelation (RelBin r)   = RelBin (coerceRelation r)
coerceIRelation (RelWhole r) = RelWhole (coerceRelation r)

  -- ------------------------------------

type RelValue a = Either [RelValBetween a] (RelValWhole a)
```

1.3.3 Information graph

```
newtype IGraph = IGraph (Set Information)

graphNodes :: IGraph → [Information]
graphNodes (IGraph inf) = Set.toList inf

graphJoin :: IGraph → [Information] → IGraph
graphJoin (IGraph inf) new = IGraph (inf ∪ Set.fromList new)

fromNodes :: [Information] → IGraph
fromNodes = IGraph ∘ Set.fromList

relationOn :: (Num a, Typeable a, Show a) ⇒ IRelation a → IGraph → IO (RelValue a)
relationOn rel iGraph = case rel of
  RelBin r → return ∘ Left $ do i1 ← graphNodes iGraph
                                i2 ← graphNodes iGraph
                                if i1 ≡ i2 then []
                                else maybeToList $
                                  RelValBetween (i1, i2) <$>
                                  binRelValue r i1 i2
  RelBinIO r → fmap (Left ∘ concat) ∘ sequence $ do
    i1 ← graphNodes iGraph
    i2 ← graphNodes iGraph
    return $ if i1 ≡ i2 then return []
                        else fmap (maybeToList
                                  ∘ fmap (RelValBetween (i1, i2)))
                                  (binRelIOValue r i1 i2)
  RelWhole r → return ∘ Right ∘ RelValWhole $ wholeRelValue r iGraph
```

1.4 Contexts

In order to use contexts for information *coherence assessment*, the concepts of *context-specific information graph* and *assessed information* are introduced.

The context-specific graph holds the information, already known/accepted by the agent, and is relevant for the context in question. The assessed one is *assumed* during the evaluation process.

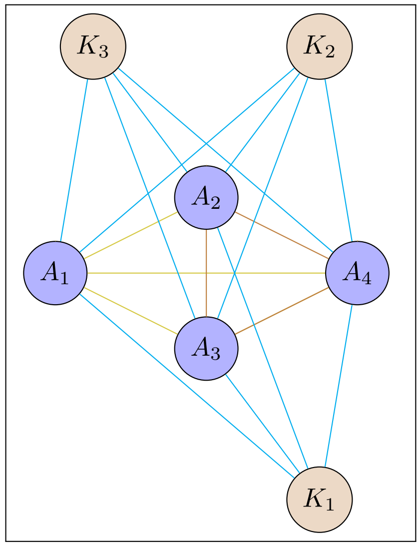

Figure 2: Binary relations within an information graph. One can distinguish the relations between the assessed information pieces and the relations between assessed and the known ones.

To assess some information, it's propagated through the contexts, in the *specified order*, that stands for contexts priority. Each context should have a *coherence threshold* specified; after the assessed information's coherence has been estimated, it's compared against the threshold and either `Success` or `Failure` is returned, along with the evaluated coherence value. The information, that has successfully passed a context, is propagated further; otherwise the failure is returned.

```
class Context (c :: * -> *) a where
  contextName        :: c a -> String
  contextInformation :: c a -> IO IGraph
  contextRelations   :: c a -> IO [IRelation a]
  contextThreshold   :: c a -> IO a

  combineBinRels     :: c a -> RelValsBetween a   -> Maybe (CBin a)
  combineWholeRels   :: c a -> RelValsWhole a     -> Maybe (CWhole a)
  combineRels        :: c a -> CBin a -> CWhole a -> a

newtype CBin a   = CBin a
newtype CWhole a = CWhole a

getCBin   (CBin a)   = a
getCWhole (CWhole a) = a
```

```
data AssessmentDetails a   -- TODO
data SomeContext a = ∀c.Context c a ⇒ SomeContext (c a)
  -- ------------------------------------------
type AnyFunc₁ res = ∀a.a → res a
mapEither :: AnyFunc₁ r → Either a b → Either (r a) (r b)
mapEither f (Left a)   = Left $ f a
mapEither f (Right a)  = Right $ f a
assessWithin' ::      (Context c a, Num a, Typeable a, Show a) ⇒
                      [Information]
   →                  c a
   →                  IO (Maybe a, AssessmentDetails a)
assessWithin' inf c = do
  contextInf  ← contextInformation c
  contextRels ← contextRelations c
  let assumed = contextInf `graphJoin` inf
      relsIO  = sequence $ (λr → mapEither ((, ) r) <$> r `relationOn` assumed)
                        <$> contextRels
  (bins, whole) ← partitionEithers <$> relsIO
  let assessed = case (bins, whole) of
        ([], []) → Nothing
        (_, [])  → getCBin   <$> c `combineBinRels`   Map.fromList bins
        ([], _)  → getCWhole <$> c `combineWholeRels` Map.fromList whole
        _        → do rBin   ← c `combineBinRels`     Map.fromList bins
                      rWhole ← c `combineWholeRels`   Map.fromList whole
                      return $ combineRels c rBin rWhole
  return (assessed, ⊥)
  -- ------------------------------------------
data AssessedCandidate a = AssessedCandidate {
      assessedAt      :: SomeContext a
  ,   assessedVal     :: Maybe a
  ,   assessedDelails :: AssessmentDetails a
  }
data Candidate a =  Success { assessHistory :: [AssessedCandidate a]
                            , candidate     :: [Information]
                            }
                 |  Failure { assessHistory :: [AssessedCandidate a]
                            , candidate     :: [Information]
                            }
  -- ------------------------------------------
assessWithin :: (Context c a, Num a, Ord a, Typeable a, Show a) ⇒
                Candidate a → c a → IO (Candidate a)
assessWithin f@Failure { } _ = return f
```

```
assessWithin (Success hist c) cxt = do
  (mbA, details)    ← c `assessWithin'` cxt
  threshold         ← contextThreshold cxt
  let ac = AssessedCandidate (SomeContext cxt) mbA details
  return $ if mbA > Just threshold
           then  Success (ac : hist) c
           else  Failure (ac : hist) c
```

Some contexts might also be capable of *splitting* information graphs into *valid candidates* – the sub-graphs, that are *valid* at the context. The candidates can be assessed by the rest of the contexts.

```
class (Context c a) ⇒ SplittingContext c a where
  splitGraph :: c a → IGraph → IO [Candidate a]
```

1.4.1 Capabilities

The capabilities context handles question "Am I able to do it?". It's main purpose is to discard immediately any proposal that would never be accepted.

- *Group*: "Am I interested in the discipline?"
- *Professor*: "Am I qualified to teach the disciple?"
- *Classroom*: "Do I suit the disciple?", "Do I have the capacity required?"

An agent should mark any other agent, that has declined some proposal for *capabilities* reasons, describing the reason. It should further avoid making same kind of proposals to the uncapable agent.

```
data family Capabilities (r :: NegotiationRole) :: ∗ → ∗
data instance Capabilities GroupRole a = GroupCapabilities {
  needsDisciplines :: [Discipline]
  }
data instance Capabilities FullTimeProfRole a = FullTimeProfCapabilities {
  canTeachFullTime :: [Discipline]
  }
-- ---------------------------------
data CanTeachRel a = CanTeachRel
instance InformationRelation CanTeachRel where
  relationName _ = "CanTeach"
  coerceRelation = coerce
instance BinaryRelation CanTeachRel where
  binRelValue _ a b =
    let v ds c = if classDiscipline c ∈ ds then 1 else 0
```

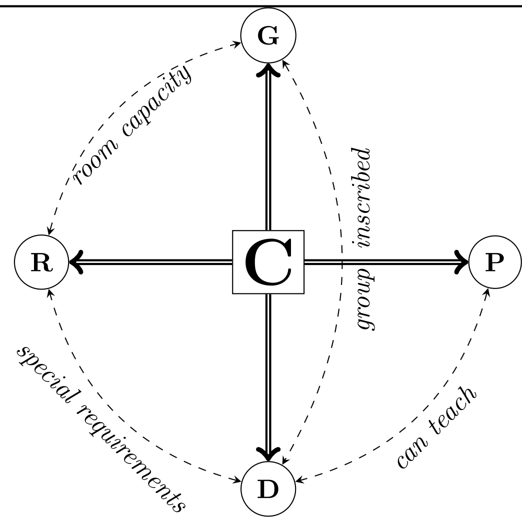

Figure 3: Capabilities required to form a *class*.

```
    in case collectInf a of
      Just (CanTeach ds) → let
        r1 = case collectInf b of Just (SomeClass c) → Just $ v ds c
        r2 = case collectInf b of Just (Class c)     → Just $ v ds c
        in r1 <|> r2
      _                   → Nothing
  -- ──────────────────────────────
data NeedsDisciplineRel a = NeedsDisciplineRel
instance InformationRelation NeedsDisciplineRel where   -- TODO
instance BinaryRelation NeedsDisciplineRel where        -- TODO
  -- ──────────────────────────────
  -- product X
combineBinRelsStrict _ bRels | null bRels = Nothing
combineBinRelsStrict _ bRels = Just ∘ CBin ∘ product
                             ∘ concatMap (map relValBetween)
                             $ Map.elems bRels
combineWholeRelsStrict _ wRels | null wRels = Nothing
combineWholeRelsStrict _ wRels = Just ∘ CWhole ∘ product
                               ∘ map unwrapRelValWhole
                               $ Map.elems wRels
combineRelsStrict _ (CBin b) (CWhole w) = b * w
  -- ──────────────────────────────
instance (Num a) ⇒ Context (Capabilities GroupRole) a where
```

$$
\begin{array}{ll}
\quad contextName\ \_ & = \texttt{"Capabilities"} \\
\quad contextInformation & = return \circ fromNodes \circ (:[\,]) \\
 & \quad \circ\ Information \circ Needs \\
 & \quad \circ\ Set.fromList \circ needsDisciplines \\
\quad contextRelations\ \_ & = return\ [RelBin\ NeedsDisciplineRel] \\
\quad contextThreshold\ \_ & = return\ 0 \\
\quad combineWholeRels & = combineWholeRelsStrict \\
\quad combineBinRels & = combineBinRelsStrict \\
\quad combineRels & = combineRelsStrict
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathbf{instance}\ (Num\ a) \Rightarrow Context\ (Capabilities\ FullTimeProfRole)\ a\ \mathbf{where}} \\
\quad contextName\ \_ & = \texttt{"Capabilities"} \\
\quad contextInformation & = return \circ fromNodes \circ (:[\,]) \\
 & \quad \circ\ Information \circ CanTeach \\
 & \quad \circ\ Set.fromList \circ canTeachFullTime \\
\quad contextRelations\ \_ & = return\ [RelBin\ CanTeachRel] \\
\quad contextThreshold\ \_ & = return\ 0 \\
\quad combineWholeRels & = combineWholeRelsStrict \\
\quad combineBinRels & = combineBinRelsStrict \\
\quad combineRels & = combineRelsStrict
\end{array}
$$

1.4.2 Beliefs

The beliefs is a *splitting* context, that uses as it's internal knowledge: 1) *state of the timetable*, that represents *best* candidate, generated until now; 2) *interesting* proposals, both generated by agent itself and received from the others, that are preserved throughout agent's lifetime.

Assessing yields one of three values

$$
\begin{cases}
-1 & \text{if two proposals intersect in time} \\
0 & \text{if both proposals have the same } \textit{abstract} \text{ part} \\
1 & \text{otherwise}
\end{cases}
$$

should be written in another place, not in this context

The assessment of *concrete proposals* (containing concrete classes) in the graph consists in

1. *assuming* the proposal information;
2. *splitting* the assumed information graph into valid candidates;
3. *propagating* of the candidates through the rest of the contexts;
4. comparing the *best candidate* with the previous *best*.

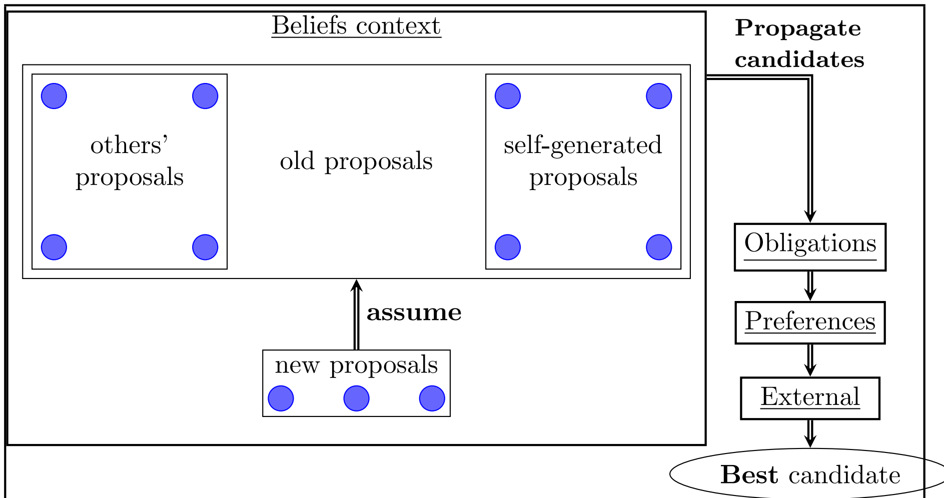

Figure 4: Assessing proposal coherence, starting from *Beliefs* context.

? The proposal is called *interesting* and is accepted (and the assumed graph becomes the new information graph of *beliefs* context) if it's assumption causes better candidate generation. It's rejected otherwise (and the assumed graph is discarded).

Splitting is a process of extraction of *acceptable* sub-graphs, that compares the coherence values at graph's edges against a threshold. The splitting can be achieved with one of two following strategies:

1. *Joining* proposals while validness is preserved.
2. *Partitioning* of proposals until validness is achieved.

First strategy is used in this project, due to less memory consumption (it doesn't have to generate or store big invalid graphs, that would be present at the first steps of the second strategy).

The splitting is implemented as follows:

Let $C = \{c\}$ be a set of *class proposals*.

$A_i = \{a_i\}$ be a set of *acceptable candidates*, composed of i proposals.

$A = \bigcup_i A_i$ be a set of *acceptable candidates*.

1. Each single candidate is acceptable: $A_1 = \{[c] \mid \forall\ c \in C\}$.
2. Form A_2 by extending each candidate $[c'] = a_1 \in A_1$ with $c \in C$, if and only if c' and c do not intersect. If $A_1 \neq \emptyset$, then try to form A_2.

⋮

i. Form A_i by extending each candidate $[c'_1, \ldots, c'_{i-1}] = a_{i-1} \in A_{i-1}$ with $c \in C$, if and only if $\forall c' \in a_{i-1},\ c'$ and c do not intersect. If $A_i \neq \emptyset$, then try to form A_{i+1}.

⋮

n. $A_n = \emptyset \implies$ all the *acceptable candidates* were generated. Done.

```
data Beliefs a = Beliefs { knownProposals :: IORef IGraph }
data TimeConsistency a = TimeConsistency
instance InformationRelation TimeConsistency where
  relationName _ = "TimeConsistency"
  coerceRelation = coerce
instance BinaryRelation TimeConsistency where
  binRelValue _ i1 i2 = do
    Class c1 ← collectInf i1
    Class c2 ← collectInf i2
    let sameParticipant =    classGroup c1     ≡ classGroup c2
                          ∨  classProfessor c1 ≡ classProfessor c2
                          ∨  classRoom c1      ≡ classRoom c2
        sameDay = classDay c1 ≡ classDay c2
        timeIntersects x y     = classBegins x ⩽ classBegins y
                               ∧ classEnds x   ⩾ classBegins y
        sameAbstract = classDiscipline c1 ≡ classDiscipline c2
                     ∧ classGroup c1      ≡ classGroup c2
                     ∧ classProfessor c1  ≡ classProfessor c2
                     ∧ classRoom c1       ≡ classRoom c2
                     ∧ classNumber c1     ≡ classNumber c2
        intersect = sameParticipant
                  ∧ sameDay
                  ∧ (timeIntersects c1 c2 ∨ timeIntersects c2 c1)
    return $ if sameAbstract then 0
                             else if intersect then − 1 else 1
instance (Num a) ⇒ Context Beliefs a where
  contextName _       = "Beliefs"
  contextInformation = readIORef ∘ knownProposals
  contextRelations _  = return [RelBin TimeConsistency]
  contextThreshold _  = return 0
  combineWholeRels   = combineWholeRelsStrict
```

```
    combineBinRels      = combineBinRelsStrict
    combineRels         = combineRelsStrict
  instance (Num a) ⇒ SplittingContext Beliefs a where
    splitGraph b gr = do
      iGraph ← readIORef $ knownProposals b
      let cNodes = catMaybes $ collectInf <$> graphNodes gr
          consistent x y = binRelValue TimeConsistency x y ≡ Just 1
          extendCandidate Failure { } = [ ]
          extendCandidate Success {candidate = inf } = do
            c ← cNodes
            [Success {assessHistory = [ ], candidate = graphNodes gr ++ [c]}
              | all (consistent c) inf ]
          a1 = Success [ ] ∘ (:[ ]) <$> cNodes
      return $ fix (λf acc last → let ext = concatMap extendCandidate last
                                  in if null ext
                                       then acc
                                       else f (acc ++ ext) ext
            ) a1 a1
```

1.4.3 Obligations

Obligations determine the rest *strong restrictions* over the classes. Possible obligations might depend on agent's role and are usually determined by the institution. For example: maximum classes per day, lunch recess, lower/upper class time limit, two classes must/cannot follow etc.

The expected values are

0 if the obligation is broken;

1 otherwise.

All the obligations must comply over a candidate.

```
  data Obligations a    = Obligations {
    obligationsInfo :: [Information],
    obligationsRels :: [IRelation (ZeroOrOne a)]
    }
  instance (Num a) ⇒ Context Obligations a where
    contextName _ = "Obligations"
    contextInformation = return ∘ fromNodes ∘ obligationsInfo
    contextRelations    = return ∘ map coerceIRelation ∘ obligationsRels
    contextThreshold _ = return 0
    combineBinRels     = combineBinRelsStrict
    combineWholeRels = combineWholeRelsStrict
    combineRels         = combineRelsStrict
```

```
  -- This constructor should be hidden.
newtype ZeroOrOne a = ZeroOrOne a
complies = ZeroOrOne 0
fails    = ZeroOrOne 1
```

1.4.4 Preferences

Preferences determine *weak restrictions*, that are intended to be set by the represented person (the institution in case of the classroom).

The expected value must be inside $[0,1]$ (unit) interval. They are combined as follows:

Binary:

$$\forall \text{ binary preference relation } \mathrm{pref_i} \implies$$

$$P^i_{\text{bin}} = \frac{\sum_{\langle n_1,n_2\rangle} \mathrm{pref}_i(n_1,n_2)}{|\{\langle n_1,n_2\rangle\}|}$$

$$P_{\text{bin}} = \prod_i P^i_{\text{bin}}; \qquad P^i_{\text{bin}} \in [0,1]; \quad P_{\text{bin}} \in [0,1].$$

Whole:

$$\forall \text{ whole graph relation } \mathrm{pref}_i$$

$$P_{\text{whole}} = \prod_i \mathrm{pref}_i(\text{graph})$$

Combined: $P = P_{\text{whole}} \times P_{\text{bin}}$

The context should diminish its influence over time to avoid possible over-restrictions due to conflicting personal interests.

```
data Preferences a = Preferences {
  preferencesInfo      :: [Information],
  preferencesRels      :: [IRelation (InUnitInterval a)],
  preferencesThreshold :: IORef a
  }
instance (Fractional a) => Context Preferences a where
  contextName _ = "Preferences"
  contextInformation  = return . fromNodes . preferencesInfo
  contextRelations    = return . map coerceIRelation . preferencesRels
  contextThreshold    = readIORef . preferencesThreshold
```

$$
\begin{aligned}
&\quad \mathit{combineBinRels}\ \_ \quad\ \, = \mathit{fmap}\ \mathit{CBin} \circ \mathit{combineBinRelsMeansProd'} \\
&\quad \mathit{combineWholeRels}\ \_ = \mathit{fmap}\ \mathit{CWhole} \circ \mathit{combineWholeRelsProd'} \\
&\quad \mathit{combineRels} \qquad\quad\ = \mathit{combineRelsProd} \\
&\quad \text{-- } \rule{10em}{0.4pt}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{maybeMean}\ [\,] = \mathit{Nothing} \\
&\mathit{maybeMean}\ \mathit{xs} = \mathit{Just}\ \$\ \mathit{sum}\ \mathit{xs}\ /\ \mathit{fromIntegral}\ (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{combineBinRelsMeansProd'} :: (\mathit{Fractional}\ a) \Rightarrow \mathit{RelValsBetween}\ a \to \mathit{Maybe}\ a \\
&\mathit{combineBinRelsMeansProd'} = \mathit{foldr}\ f\ \mathit{Nothing} \\
&\quad \mathbf{where}\ \mathit{mean'} = \mathit{maybeMean} \circ \mathit{map}\ \mathit{relValBetween} \\
&\qquad\qquad f\ \mathit{xs}\ \mathit{acc}@(\mathit{Just}\ \_) = ((*)\ \mathord{<}\$\mathord{>}\ \mathit{acc}\ \mathord{<}{*}\mathord{>}\ \mathit{mean'}\ \mathit{xs})\ \mathord{<}|\mathord{>}\ \mathit{acc} \\
&\qquad\qquad f\ \mathit{xs}\ \_ \qquad\qquad\quad\ = \mathit{mean'}\ \mathit{xs}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{combineWholeRelsProd'}\ \mathit{mp}\ |\ \mathit{null}\ \mathit{mp} = \mathit{Nothing} \\
&\mathit{combineWholeRelsProd'}\ \mathit{mp} = \mathit{Just} \circ \mathit{product} \circ \mathit{map}\ \mathit{unwrapRelValWhole} \\
&\qquad\qquad\qquad\qquad\qquad\quad \$\ \mathit{Map.elems}\ \mathit{mp}
\end{aligned}
$$

$$
\mathit{combineRelsProd}\ \_\ (\mathit{CBin}\ \mathit{bin})\ (\mathit{CWhole}\ \mathit{whole}) = \mathit{bin} * \mathit{whole}
$$

$$
\quad \text{-- } \rule{10em}{0.4pt}
$$

$$
\mathbf{newtype}\ \mathit{InUnitInterval}\ a = \mathit{InUnitInterval}\ a
$$

$$
\begin{aligned}
&\mathit{inUnitInterval} :: (\mathit{Fractional}\ a, \mathit{Ord}\ a) \Rightarrow a \to \mathit{Maybe}\ (\mathit{InUnitInterval}\ a) \\
&\mathit{inUnitInterval}\ x\ | \qquad x \geqslant 0 \\
&\qquad\qquad\qquad\quad \wedge \qquad x \leqslant 1 = \mathit{Just}\ \$\ \mathit{InUnitInterval}\ x \\
&\mathit{inUnitInterval}\ \_ = \mathit{Nothing}
\end{aligned}
$$

$$
\mathit{fromUnitInterval}\ (\mathit{InUnitInterval}\ x) = x
$$

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Eq}\ (\mathit{InUnitInterval}\ a)\ \mathbf{where} \\
&\mathbf{instance}\ \mathit{Ord}\ (\mathit{InUnitInterval}\ a)\ \mathbf{where}
\end{aligned}
$$

1.4.5 External

External contexts take into account the *opinions* of the agents that are referenced by the solution candidate. It is responsible for *common goal* assessment. The assessment must be *objective* — it must give no preference to agent's own interests.

The *context-specific information* consists of references to the known agents with cached information about their capabilities.

There is a single binary relation in this context — *opinion* of agent $\mathrm{ag}_i^{\mathrm{role}}$ on class c_i, of which consists the proposal in question p_k. They are combined using operation.

$$
\begin{aligned}
&\mathbf{data}\ \mathit{KnownAgent}\ a = \forall r :: \mathit{NegotiationRole.KnownAgent}\ \{ \\
&\quad \mathit{knownAgentRef} \qquad\quad\ :: \mathit{AgentRef}, \\
&\quad \mathit{knownAgentRole} \qquad\quad :: \mathit{Role'}\ r, \\
&\quad \mathit{knownAgentCapabilities} :: [\mathit{Capabilities}\ r\ a] \\
&\quad \} \\
&\quad \mathbf{deriving}\ \mathit{Typeable}
\end{aligned}
$$

```
askKnownAgent :: (MessageT msg a) ⇒ KnownAgent a
                                   → msg a
                                   → IOMaybe (ExpectedResponse1 msg a)
askKnownAgent knownAg message =
    case knownAgentRef knownAg of
      AgentRef comm → do resp ← askT comm message
                         return $ gcast resp

instance Eq (KnownAgent a) where
  (≡) = (≡) `on` knownAgentRef

instance Ord (KnownAgent a) where
  compare = compare `on` knownAgentRef

instance (Typeable a) ⇒ InformationPiece (KnownAgent a)

  -- ------------------------------------------------

data External a = External {
    knownAgents       :: IORef [KnownAgent a]
  , externalThreshold :: IORef a
  }

instance (Typeable a, Num a) ⇒ Context External a where
  contextName _       = "External"
  contextInformation  = fmap (fromNodes ∘ map Information)
                        ∘ readIORef ∘ knownAgents
  contextRelations r  = return [RelBinIO OpinionRel]
  contextThreshold    = readIORef ∘ externalThreshold
  combineBinRels      = combineBinRelsStrict
  combineWholeRels    = ⊥
  combineRels         = ⊥

  -- ------------------------------------------------

data OpinionRel a = OpinionRel

newtype OpinionAbout a = OpinionAbout (Class, a) deriving (Typeable, Show)

data MyOpinion a = MyOpinion (Maybe (InUnitInterval a)) deriving (Typeable, Show)

type instance ExpectedResponse1 OpinionAbout = MyOpinion

extractMyOpinion (MyOpinion mbOpinion) = mbOpinion

  -- ------------------------------------------------

instance InformationRelation OpinionRel where
  relationName _ = "Opinion"
  coerceRelation = coerce

instance BinaryIORelation OpinionRel where
  binRelIOValue rel a b = fromMaybe (return Nothing)
     $ do knownAg ← collectInf a
          class'  ← collectInf b
          return $ do resp ← askKnownAgent knownAg (OpinionAbout class')
                      return ∘ fmap fromUnitInterval $ extractMyOpinion =≪ resp
```

1.5 Agent

$$-- \rule{10em}{0.4pt}$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{GroupRef}\quad\ \ = \mathit{GroupRef}\quad\ \ \mathit{String}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})\\
\textbf{data}\ \mathit{ProfessorRef}\ \ = \mathit{ProfessorRef}\ \ \mathit{String}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})\\
\textbf{data}\ \mathit{ClassroomRef} = \mathit{ClassroomRef}\ \mathit{String}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})
\end{array}
$$

$$-- \rule{10em}{0.4pt}$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{AgentComm}\ \mathit{GroupRef}\ \textbf{where}\quad \text{-- TODO}\\
\textbf{instance}\ \mathit{AgentComm}\ \mathit{ProfessorRef}\ \textbf{where}\quad \text{-- TODO}\\
\textbf{instance}\ \mathit{AgentComm}\ \mathit{ClassroomRef}\ \textbf{where}\quad \text{-- TODO}
\end{array}
$$

Misc code:

$$\textbf{instance}\ \mathit{Show}\ \mathit{Class}\ \textbf{where}\quad \text{-- TODO}$$

$$\textbf{instance}\ (\mathit{Show}\ a) \Rightarrow \mathit{Show}\ (\mathit{InUnitInterval}\ a)\ \textbf{where}\ \mathit{show}\ (\mathit{InUnitInterval}\ x) = \mathit{show}\ x$$

1.5.1 Behavior definition

Agent's behavior is defined by its *action loop* and incoming *messages handling*.

$$
\begin{array}{l}
\textbf{data}\ \mathit{AgentBehavior}\ \mathit{states} = \mathit{AgentBehavior}\ \{\\
\quad \mathit{act} :: \forall i.(\mathit{AgentInnerInterface}\ i) \Rightarrow i \to \mathit{states} \to \mathit{IO}\ (),\\
\quad \mathit{handleMessages} :: \mathit{AgentHandleMessages}\ \mathit{states}\\
\quad \}
\end{array}
$$

Messages can be just *sent* to any agent or a specific *response* may be *asked*.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Typeable}\ \mathit{ref}, \mathit{Ord}\ \mathit{ref}) \Rightarrow \mathit{AgentComm}\ \mathit{ref}\ \textbf{where}\\
\quad \mathit{agentId} :: \mathit{ref} \to \mathit{AgentId}\\
\quad \mathit{send}\quad :: (\mathit{Message}\ \mathit{msg})\qquad \Rightarrow \mathit{ref} \to \mathit{msg}\quad \to \mathit{IO}\ ()\\
\quad \mathit{ask}\quad\ \ :: (\mathit{Message}\ \mathit{msg})\qquad \Rightarrow \mathit{ref} \to \mathit{msg}\quad \to \mathit{IO}\ (\mathit{ExpectedResponse}\ \mathit{msg})\\
\quad \mathit{askT}\quad :: (\mathit{MessageT}\ \mathit{msg}\ t) \Rightarrow \mathit{ref} \to \mathit{msg}\ t \to \mathit{IO}\ (\mathit{ExpectedResponse1}\ \mathit{msg}\ t)
\end{array}
$$

These messages are handled by the corresponding agent's functions.

$$\textbf{data}\ \mathit{AgentHandleMessages}\ \mathit{states} = \mathit{AgentHandleMessages}\ \{$$

- react to sent messages (sent with `send`):

$$
\begin{array}{l}
\mathit{handleMessage} :: \forall \mathit{msg}\ i.(\mathit{Message}\ \mathit{msg}\\
\qquad\qquad\qquad\qquad\quad , \mathit{AgentInnerInterface}\ i) \Rightarrow\\
\quad i \to \mathit{states} \to \mathit{msg} \to \mathit{IO}\ (),
\end{array}
$$

- respond un-typed messages (responding to `ask`):

$$\begin{aligned}
&\mathit{respondMessage} :: \forall msg\ resp\ i.(\mathit{Message}\ msg\\
&\qquad\qquad , \mathit{ExpectedResponse}\ msg{\sim}resp\\
&\qquad\qquad , \mathit{AgentInnerInterface}\ i) \Rightarrow\\
&\quad i \rightarrow states \rightarrow msg \rightarrow \mathit{IO}\ resp,
\end{aligned}$$

- respond typed messages (responding to `askT`):

$$\begin{aligned}
&\mathit{respondTypedMessage} :: \forall msg\ resp\ t\ i.(\mathit{MessageT}\ msg\ t\\
&\qquad\qquad , \mathit{ExpectedResponse1}\ msg\ t{\sim}resp\ t\\
&\qquad\qquad , \mathit{AgentInnerInterface}\ i) \Rightarrow\\
&\quad i \rightarrow states \rightarrow msg\ t \rightarrow \mathit{IO}\ (resp\ t)\\
&\}
\end{aligned}$$

The expected response type should be defined for every message that is intended to get responses.

$$\begin{aligned}
&\textbf{type family}\ \mathit{ExpectedResponse}\ \ (msg :: *) \qquad :: *\\
&\textbf{type family}\ \mathit{ExpectedResponse1}\ (msg :: * \rightarrow *) :: * \rightarrow *
\end{aligned}$$

Restriction for messages is having instances of `Typeable` and `Show`.

$$\begin{aligned}
&\textbf{type}\ \mathit{Message}\ msg \quad\ = (\mathit{Typeable}\ msg, \mathit{Show}\ msg)\\
&\textbf{type}\ \mathit{MessageT}\ msg\ t = (\mathit{Typeable}\ t, \mathit{Typeable}\ msg, \mathit{Show}\ (msg\ t))\\
&\textbf{data}\ \mathit{StartMessage} = \mathit{StartMessage}\ \textbf{deriving}\ (\mathit{Typeable}, \mathit{Show})\\
&\textbf{data}\ \mathit{StopMessage}\ = \mathit{StopMessage}\ \ \textbf{deriving}\ (\mathit{Typeable}, \mathit{Show})
\end{aligned}$$

Agent interface is used to reference agent-self within behavior definitions.

$$\begin{aligned}
&\textbf{class}\ \mathit{AgentInnerInterface}\ i\ \textbf{where}\ \mathit{selfRef}\ \ :: i \rightarrow \mathit{AgentRef}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{selfStop} :: i \rightarrow \mathit{IO}\ ()
\end{aligned}$$

1.5.2 Role-depending behavior

The expected response may depend on agent's *role*.

$$\begin{aligned}
&\textbf{class}\ (\mathit{AgentComm}\ ref) \Rightarrow \mathit{AgentCommRole}\ ref\ \textbf{where}\\
&\quad \textbf{type}\ \mathit{AgentRole}\ ref :: *\\
&\quad \mathit{askR} :: (\mathit{Message}\ msg) \qquad \Rightarrow ref\\
&\qquad\qquad\qquad\qquad\qquad\quad \rightarrow msg\\
&\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \mathit{IO}\ (\mathit{ExpectedResponseForRole}\ (\mathit{AgentRole}\ ref)\ msg)\\
&\quad \mathit{askRT} :: (\mathit{MessageT}\ msg\ t) \Rightarrow ref\\
&\qquad\qquad\qquad\qquad\qquad\quad \rightarrow msg\ t\\
&\qquad\qquad\qquad\qquad\qquad\quad \rightarrow \mathit{IO}\ (\mathit{ExpectedResponseForRole1}\ (\mathit{AgentRole}\ ref)\ msg\ t)
\end{aligned}$$

$$\textbf{type family}\ \mathit{ExpectedResponseForRole}\ \ r\ (\mathit{msg} :: *) \quad :: *$$
$$\textbf{type family}\ \mathit{ExpectedResponseForRole1}\ r\ (\mathit{msg} :: * \to *) :: * \to *$$

-- System role.

$$\textbf{data}\ \mathit{System} = \mathit{System}$$

-- Generic role.

$$\textbf{data}\ \mathit{Generic} = \mathit{Generic}$$

1.5.3 Referencing agents

Agents are identified (also compared and searched) by its `AgentId`, that must contain a ***unique*** string, for example an UUID.

$$\textbf{data}\ \mathit{AgentId} = \mathit{AgentId}\ \mathit{String}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}, \mathit{Ord})$$

Normal agent reference is a container for types of class `AgentComm`.

$$\textbf{data}\ \mathit{AgentRef} = \forall \mathit{ref}.(\mathit{AgentComm}\ \mathit{ref}) \Rightarrow \mathit{AgentRef}\ \mathit{ref}$$

A reference itself provides `AgentComm` interface for the underlying agent.

$$\begin{array}{l}
\textbf{instance}\ \mathit{AgentComm}\ \mathit{AgentRef}\ \textbf{where} \\
\quad \mathit{agentId}\ (\mathit{AgentRef}\ \mathit{ref}) = \mathit{agentId}\ \mathit{ref} \\
\quad \mathit{send}\ \ \ \ (\mathit{AgentRef}\ \mathit{ref}) = \mathit{send}\ \mathit{ref} \\
\quad \mathit{ask}\ \ \ \ \ (\mathit{AgentRef}\ \mathit{ref}) = \mathit{ask}\ \mathit{ref} \\
\quad \mathit{askT}\ \ \ \ (\mathit{AgentRef}\ \mathit{ref}) = \mathit{askT}\ \mathit{ref}
\end{array}$$

It is used the referenced agent's id for establishing `Eq` and `Ord` relations over it.

$$\textbf{instance}\ \mathit{Eq}\ \mathit{AgentRef}\ \textbf{where}\ \mathit{AgentRef}\ a \equiv \mathit{AgentRef}\ b = \mathit{agentId}\ a \equiv \mathit{agentId}\ b$$
$$\textbf{instance}\ \mathit{Ord}\ \mathit{AgentRef}\ \textbf{where}\ \mathit{AgentRef}\ a \lesseqqgtr \mathit{AgentRef}\ b = \mathit{agentId}\ a \lesseqqgtr \mathit{agentId}\ b$$

1.5.4 Agent control

Agents should support *priority messages*, that are processed before any normal message.

$$\begin{array}{l}
\textbf{class}\ (\mathit{AgentComm}\ \mathit{ref}) \Rightarrow \mathit{AgentCommPriority}\ \mathit{ref}\ \textbf{where} \\
\quad \mathit{sendPriority} :: (\mathit{Message}\ \mathit{msg}) \quad\ \ \Rightarrow \mathit{ref} \to \mathit{msg} \quad \to \mathit{IO}\ () \\
\quad \mathit{askPriority}\ \ :: (\mathit{Message}\ \mathit{msg}) \quad\ \ \Rightarrow \mathit{ref} \to \mathit{msg} \quad \to \mathit{IO}\ (\mathit{ExpectedResponse}\ \mathit{msg}) \\
\quad \mathit{askTPriority} :: (\mathit{MessageT}\ \mathit{msg}\ t) \Rightarrow \mathit{ref} \to \mathit{msg}\ t \to \mathit{IO}\ (\mathit{ExpectedResponse1}\ \mathit{msg}\ t)
\end{array}$$

A *control interface* should be based on the priority messages.

$$\begin{array}{l}
\textbf{class}\ (\mathit{AgentCommPriority}\ \mathit{ag}) \Rightarrow \mathit{AgentControl}\ \mathit{ag}\ \textbf{where} \\
\quad \mathit{startAgent} :: \mathit{ag} \to \mathit{IO}\ () \\
\quad \mathit{stopAgent}\ \ :: \mathit{ag} \to \mathit{IO}\ () \\
\quad \mathit{stopAgentNow} :: \mathit{ag} \to \mathit{IO}\ ()
\end{array}$$

1.5.5 Agent extended referencing

A `AgentFullRef` is used for agent control and status monitoring. It contains some instance of `AgentControl` and the information about agent's threads.

$$\textbf{data}\ AgentFullRef = \forall ref.(AgentControl\ ref) \Rightarrow AgentFullRef\ ref\ AgentThreads$$

Each agent is expected to be composed of two execution threads: *message handling* and *actions*.

$$\begin{aligned}&\textbf{data}\ AgentThreads = AgentThreads\ \{\_actThread :: AgentThread\\&\qquad , \_messageThread :: AgentThread\\&\qquad \}\\&fromFullRef\ (AgentFullRef\ ref\ \_) = AgentRef\ ref\\&extractThreads\ (AgentFullRef\ \_\ (AgentThreads\ act\ msg)) = (act, msg)\end{aligned}$$

The information about agent's thread permits checking on its status, waiting for it to finish or killing it, using the provided `ThreadId`.

$$\begin{aligned}&\textbf{data}\ AgentThread = AgentThread\ \{\_threadId :: ThreadId\\&\quad ,\ \_threadFinished :: IO\ Bool\\&\quad ,\ \_waitThread :: IO\ ()\\&\quad \}\\&forceStopAgent :: AgentFullRef \rightarrow IO\ ()\\&forceStopAgent\ fref = \textbf{do}\ \_killThread\ act\\&\qquad\qquad \_killThread\ msg\\&\quad \textbf{where}\ (act, msg) = extractThreads\ fref\\&\qquad\quad \_killThread = killThread \circ \_threadId\\&waitAgent :: AgentFullRef \rightarrow IO\ ()\\&waitAgent\ fref = \textbf{do}\ \_waitThread\ act\\&\quad \_waitThread\ msg\\&\quad \textbf{where}\ (act, msg) = extractThreads\ fref\end{aligned}$$

Just like a normal reference, the full one is compared and tested by the `AgentId`.

$$\begin{aligned}&\textbf{instance}\ Eq\ AgentFullRef\ \textbf{where}\\&\quad AgentFullRef\ a\ \_ \equiv AgentFullRef\ b\ \_ = agentId\ a \equiv agentId\ b\\&\textbf{instance}\ Ord\ AgentFullRef\ \textbf{where}\\&\quad AgentFullRef\ a\ \_ \lesseqqgtr AgentFullRef\ b\ \_ = agentId\ a \lesseqqgtr agentId\ b\end{aligned}$$

It also provides instances of `AgentComm`, `AgentCommPriority` and `AgentControl`.

$$\begin{aligned}&\textbf{instance}\ AgentComm\ AgentFullRef\ \textbf{where}\\&\quad agentId\ (AgentFullRef\ ref\ \_) = agentId\ ref\\&\quad send\ \ (AgentFullRef\ ref\ \_) = send\ ref\end{aligned}$$

```
    ask       (AgentFullRef ref _) = ask ref
    askT      (AgentFullRef ref _) = askT ref
  instance AgentCommPriority AgentFullRef where
    sendPriority  (AgentFullRef ref _) = sendPriority ref
    askPriority   (AgentFullRef ref _) = askPriority ref
    askTPriority  (AgentFullRef ref _) = askTPriority ref
  instance AgentControl AgentFullRef where
    startAgent (AgentFullRef ref _) = startAgent ref
    stopAgent  (AgentFullRef ref _) = stopAgent ref
    stopAgentNow (AgentFullRef ref _) = stopAgentNow ref
```

1.5.6 Agent Creation

Generic creation is defined in for types from and ag.

```
  class (AgentControl ag) ⇒ AgentCreate from ag where
    createAgent :: from → IO (ag, AgentFullRef)
```

A simple *agent descriptor* that can be used for agent creation.

```
  data AgentDescriptor states = AgentDescriptor {
    agentBehaviour :: AgentBehavior states,
    newAgentStates :: IO states,
      nextAgentId   :: IO AgentId
           }
```

1.5.7 Agent Management

A manager registers/unregisters agent references and provides agent-related operations over them.

```
  class AgentsManager m where
    newAgentsManager :: IO m
    listAgents        :: m → IO [AgentFullRef]
    registerAgent     :: m → AgentFullRef → IO ()
    unregisterAgent   :: m → AgentFullRef → IO ()
    mapAgents         :: (AgentFullRef → IO a)  → m → IO [a]
    mapAgents_        :: (AgentFullRef → IO ()) → m → IO ()
    foreachAgent      :: m → (AgentFullRef → IO a)  → IO [a]
    foreachAgent_     :: m → (AgentFullRef → IO ()) → IO ()
    foreachAgent  = flip mapAgents
    foreachAgent_ = flip mapAgents_
  class AgentsManagerOps m where
    agentsStopped         :: m → IO Bool
```

$$
\begin{array}{ll}
\mathit{waitAllAgents} & :: m \rightarrow \mathit{IO}\ () \\
\mathit{sendEachAgent} & :: (\mathit{Message}\ \mathit{msg}) \Rightarrow m \rightarrow \mathit{msg} \rightarrow \mathit{IO}\ () \\
\mathit{orderEachAgent} & :: (\mathit{Message}\ \mathit{msg}) \Rightarrow m \rightarrow \mathit{msg} \rightarrow \mathit{IO}\ () \\
\mathit{createWithManager} & :: (\mathit{AgentCreate}\ \mathit{from}\ \mathit{ag}) \Rightarrow m \rightarrow \mathit{from} \rightarrow \mathit{IO}\ (\mathit{ag}, \mathit{AgentFullRef})
\end{array}
$$