```
  -- Authors: Bas van Gijzel and Henrik Nilsson
  -- See: http://www.cs.nott.ac.uk/ bmv/CarneadesDSL/index.html
module CarneadesDSLWithCycleChecking where

import Prelude hiding (negate)
import Data.Graph.Inductive
import Data.Map (Map)
import Data.List (nub)
import qualified Data.Set as Set
import qualified Data.Map as Map
import Data.Maybe (fromJust)


  -- Based on initial code provided by Stefan Sabev
cyclic :: (DynGraph g) => g a b -> Bool
cyclic g | ¬ (null leafs) = cyclic (delNodes leafs g)
  | otherwise = ¬ (isEmpty g)
  where leafs = filter (isLeaf g) (nodes g)

isLeaf :: (DynGraph g) => g a b -> Node -> Bool
isLeaf g n = n ∉ map fst (edges g)
```

0.1 Arguments

As our ultimate goal is a DSL for argumentation theory, we strive for a realisation in Haskell that mirrors the mathematical model of Carneades argumentation framework as closely as possible. Ideally, there would be little more to a realisation than a transliteration. We will thus proceed by stating the central definitions of Carneades along with our realisation of them in Haskell.

Definition 1 (Arguments) *Let $\mathcal{L}$ be a propositional language. An* argument *is a tuple $\langle P, E, c\rangle$ where $P \subset \mathcal{L}$ are its* premises*, $E \subset \mathcal{L}$ with $P \cap E = \emptyset$ are its* exceptions *and $c \in \mathcal{L}$ is its* conclusion*. For simplicity, all members of $\mathcal{L}$ must be literals, i.e. either an atomic proposition or a negated atomic proposition. An argument is said to be* pro *its conclusion c (which may be a negative atomic proposition) and* con *the negation of c.*

In Carneades all logical formulae are literals in propositional logic; i.e., all propositions are either positive or negative atoms. Taking atoms to be strings suffice in the following, and propositional literals can then be formed by pairing this atom with a Boolean to denote whether it is negated or not:

```
type PropLiteral = (Bool, String)
```

We write $\overline{p}$ for the negation of a literal p. The realisation is immediate:

```
negate :: PropLiteral → PropLiteral
negate (b, x) = (¬ b, x)
```

We chose to realise an *argument* as a newtype (to allow a manual Eq instance) containing a tuple of two lists of propositions, its *premises* and its *exceptions*, and a proposition that denotes the *conclusion*:

```
newtype Argument = Arg ([PropLiteral], [PropLiteral], PropLiteral)
deriving Ord
```

Arguments are considered equal if their premises, exceptions and conclusion are equal; thus arguments are identified by their logical content. The equality instance for *Argument* (omitted for brevity) takes this into account by comparing the lists as sets.

```
  -- Manual Eq instance for set equality on premises and exceptions
instance Eq Argument where
(Arg (prems, excs, c)) ≡ (Arg (prems', excs', c'))
   = Set.fromList prems ≡ Set.fromList prems' ∧
     Set.fromList excs ≡ Set.fromList excs'   ∧
     c ≡ c'

showProp :: PropLiteral → String
showProp (True, c) = c
showProp (_, c)    = '-' : c

instance Show Argument where
show (Arg (prems, excs, (True, c))) = show (map showProp prems) ++ ' ' : '~' : show (map showProp excs) ++
show (Arg (prems, excs, (_, c)))    = show (map showProp prems) ++ ' ' : '~' : show (map showProp excs) ++
```

A set of arguments determines how propositions depend on each other. Carneades requires that there are no cycles among these dependencies. Following Brewka and Gordon [**?**], we use a dependency graph to determine acyclicity of a set of arguments.

Definition 2 (Acyclic set of arguments) *A set of* arguments *is* acyclic *iff its corresponding dependency graph is acyclic. The corresponding dependency graph has a node for every literal appearing in the set of arguments. A node p has a link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its set of premises or exceptions.*

Our realisation of a set of arguments is considered abstract for DSL purposes, only providing a check for acyclicity and a function to retrieve arguments pro a proposition. We use FGL [**?**] to implement the dependency graph, forming nodes for propositions and edges for the dependencies. For simplicity, we opt to keep the graph also as the representation of a set of arguments.

```
  -- Note that for practical purposes we do not need to know the following
  -- implementation but can use the abstraction further below
type ArgSet = Gr (PropLiteral, [Argument]) ()


  -- abstraction of ArgSet and the operating functions on it
type ArgSet = ...
getArgs :: PropLiteral → ArgSet → [Argument]
  -- checkCycle :: ArgSet -¿ Bool
```

0.2 Carneades Argument Evaluation Structure

The main structure of the argumentation model is called a Carneades Argument Evaluation Structure (CAES):

Definition 3 (Carneades Argument Evaluation Structure (CAES)) *A* Carneades Argument Evaluation Structure *(CAES) is a triple*

$$\langle arguments, audience, standard \rangle$$

where arguments is an acyclic set of arguments, audience is an audience as defined below (Def. 4), and standard *is a total function mapping each proposition to to its specific proof standard.*

Note that propositions may be associated with *different* proof standards. This is considered a particular strength of the Carneades framework. The transliteration into Haskell is almost immediate[1]:

```
newtype CAES = CAES (ArgSet, Audience, PropStandard)
```

[1]Note that we use a newtype to prevent a cycle in the type definitions.

Definition 4 (Audience) *Let $\mathcal{L}$ be a propositional language. An* audience *is a tuple* ⟨assumptions, weight⟩*, where assumptions $\subset \mathcal{L}$ is a propositionally consistent set of literals (i.e., not containing both a literal and its negation) assumed to be acceptable by the audience and* weight *is a function mapping arguments to a real-valued weight in the range* $[0, 1]$.

This definition is captured by the following Haskell definitions:

```
type Audience = (Assumptions, ArgWeight)
type Assumptions = [PropLiteral]
type ArgWeight = Argument → Weight
type Weight = Double
```

Further, as each proposition is associated with a specific proof standard, we need a mapping from propositions to proof standards:

```
type PropStandard = PropLiteral → PSName

data PSName = Scintilla
   | Preponderance | ClearAndConvincing
   | BeyondReasonableDoubt | DialecticalValidity
deriving Eq


psMap :: PSName → ProofStandard
```

A proof standard is a function that given a proposition p, aggregates arguments pro and con p and decides whether it is acceptable or not:

```
type ProofStandard = PropLiteral → CAES → Bool

newtype ProofStandardNamed = P (String, PropLiteral → CAES → Bool)
instance Eq ProofStandardNamed where
P (l1, _) ≡ P (l2, _) = l1 ≡ l2
```

This aggregation process will be defined in detail in the next section, but note that it is done relative to a specific CAES, and note the cyclic dependence at the type level between *CAES* and *ProofStandard*.

The above definition of proof standard also demonstrates that implementation in a typed language such as Haskell is a useful way of verifying definitions from argumentation theoretic models. Our implementation effort revealed that the original definition as given in [**?**] could not be realised as stated, because proof standards in general not only depend on a set of arguments and the audience, but may need the whole CAES.

0.3 Evaluation

Two concepts central to the evaluation of a CAES are *applicability of arguments*, which arguments should be taken into account, and *acceptability of propositions*, which conclusions can be reached under the relevant proof standards, given the beliefs of a specific audience.

Definition 5 (Applicability of arguments) *Given a set of arguments and a set of assumptions (in an audience) in a CAES C, then an argument $a = \langle P, E, c\rangle$ is* applicable *iff*

- *$p \in P$ implies p is an assumption or [$\overline{p}$ is not an assumption and p is acceptable in C] and*
- *$e \in E$ implies e is not an assumption and [$\overline{e}$ is an assumption or e is not acceptable in C].*

Definition 6 (Acceptability of propositions) *Given a CAES C, a proposition p is* acceptable *in C iff $(s\ p\ C)$ is true, where s is the proof standard for p.*

Note that these two definitions in general are mutually dependent because acceptability depends on proof standards, and most sensible proof standards depend on the applicability of arguments. This is the reason that Carneades restricts the set of arguments to be acyclic. (Specific proof standards are considered in the next section.) The realisation of applicability and acceptability in Haskell is straightforward:

```
applicable :: Argument → CAES → Bool
applicable  (Arg (prems, excns, _))
            caes@(CAES (_, (assumptions, _), _))
  = and $ [p ∈ assumptions ∨
             (negate p ∉ assumptions ∧
              p `acceptable` caes) | p ← prems]
           ++
           [(e ∉ assumptions) ∧
             (negate e ∈ assumptions ∨
              ¬ (e `acceptable` caes)) | e ← excns]
acceptable :: PropLiteral → CAES → Bool
acceptable c caes@(CAES (_, _, standard))
  = c `s` caes
  where s = psMap $ standard c
```

0.4 Proof standards

Carneades predefines five proof standards, originating from the work of Freeman and Farley [**?**, **?**]: *scintilla of evidence*, *preponderance of the evidence*, *clear and convincing evidence*, *beyond reasonable doubt* and *dialectical validity*. Some proof standards depend on constants such as α, β, γ; these are assumed to be defined once and globally. This time, we proceed to give the definitions directly in Haskell, as they really only are translitarations of the original definitions.

For a proposition p to satisfy the weakest proof standard, scintilla of evidence, there should be at least one applicable argument pro p in the CAES:

```
scintilla :: ProofStandard
scintilla p caes@(CAES (g, _, _))
 = any (`applicable`caes) (getArgs p g)
```

Preponderance of the evidence additionally requires the maximum weight of the applicable arguments pro p to be greater than the maximum weight of the applicable arguments con p. The weight of zero arguments is taken to be 0. As the maximal weight of applicable arguments pro and con is a recurring theme in the definitions of several of the proof standards, we start by defining those notions:

```
maxWeightApplicable :: [Argument] → CAES → Weight
maxWeightApplicable as caes@(CAES (_, (_, argWeight), _))
 = foldl max 0 [argWeight a | a ← as, a `applicable` caes]
maxWeightPro :: PropLiteral → CAES → Weight
maxWeightPro p caes@(CAES (g, _, _))
 = maxWeightApplicable (getArgs p g) caes
maxWeightCon :: PropLiteral → CAES → Weight
maxWeightCon p caes@(CAES (g, _, _))
 = maxWeightApplicable (getArgs (negate p) g) caes
```

We can then define the proof standard preponderance:

```
preponderance :: ProofStandard
preponderance p caes = maxWeightPro p caes > maxWeightCon p caes
```

Clear and convincing evidence strengthen the preponderance constraints by insisting that the difference between the maximal weights of the pro and con arguments must be greater than a given positive constant β, and there should furthermore be at least one applicable argument pro p that is stronger than a given positive constant α:

$$
\begin{aligned}
&\mathit{clear\_and\_convincing} :: \mathit{ProofStandard}\\
&\mathit{clear\_and\_convincing}\ p\ \mathit{caes}\\
&=\quad (\mathit{mwp} > \mathit{alpha}) \wedge (\mathit{mwp} - \mathit{mwc} > \mathit{beta})\\
&\quad \textbf{where}\\
&\qquad \mathit{mwp} = \mathit{maxWeightPro}\ p\ \mathit{caes}\\
&\qquad \mathit{mwc} = \mathit{maxWeightCon}\ p\ \mathit{caes}
\end{aligned}
$$

Beyond reasonable doubt has one further requirement: the maximal strength of an argument con p must be less than a given positive constant γ; i.e., there must be no reasonable doubt:

$$
\begin{aligned}
&\mathit{beyond\_reasonable\_doubt} :: \mathit{ProofStandard}\\
&\mathit{beyond\_reasonable\_doubt}\ p\ \mathit{caes}\\
&= \mathit{clear\_and\_convincing}\ p\ \mathit{caes} \wedge (\mathit{maxWeightCon}\ p\ \mathit{caes} < \mathit{gamma})
\end{aligned}
$$

Finally dialectical validity requires at least one applicable argument pro p and no applicable arguments con p:

$$
\begin{aligned}
&\mathit{dialectical\_validity} :: \mathit{ProofStandard}\\
&\mathit{dialectical\_validity}\ p\ \mathit{caes}\\
&\quad = \mathit{scintilla}\ p\ \mathit{caes} \wedge \neg\ (\mathit{scintilla}\ (\mathit{negate}\ p)\ \mathit{caes})
\end{aligned}
$$

0.5 Convenience functions

We provide a set of functions to facilitate construction of propositions, arguments, argument sets and sets of assumptions. Together with the definitions covered so far, this constitute our DSL for constructing Carneades argumentation models.

$$
\begin{aligned}
&\mathit{mkProp} && :: \mathit{String} \to \mathit{PropLiteral}\\
&\mathit{mkArg} && :: [\mathit{String}] \to [\mathit{String}] \to \mathit{String} \to \mathit{Argument}\\
&\mathit{mkArgSet} && :: [\mathit{Argument}] \to \mathit{ArgSet}\\
&\mathit{mkAssumptions} && :: [\mathit{String}] \to [\mathit{PropLiteral}]
\end{aligned}
$$

A string starting with a `'-'` is taken to denote a negative atomic proposition.

To construct an audience, native Haskell tupling is used to combine a set of assumptions and a weight function, exactly as it would be done in the Carneades model:

$$
\begin{aligned}
&\mathit{audience} :: \mathit{Audience}\\
&\mathit{audience} = (\mathit{assumptions}, \mathit{weight})
\end{aligned}
$$

Carneades Argument Evaluation Structures and weight functions are defined in a similar way, as will be shown in the next subsection.

Finally, we provide a function for retrieving the arguments for a specific proposition from an argument set, a couple of functions to retrieve all arguments and propositions respectively from an argument set, and functions to retrieve the (not) applicable arguments or (not) acceptable propositions from a CAES:

$$
\begin{aligned}
&\mathit{getArgs} && :: \mathit{PropLiteral} \to \mathit{ArgSet} \to [\mathit{Argument}]\\
&\mathit{getAllArgs} && :: \mathit{ArgSet} \to [\mathit{Argument}]\\
&\mathit{getProps} && :: \mathit{ArgSet} \to [\mathit{PropLiteral}]\\
&\mathit{applicableArgs} && :: \mathit{CAES} \to [\mathit{Argument}]
\end{aligned}
$$

```
nonApplicableArgs   :: CAES → [Argument]
acceptableProps     :: CAES → [PropLiteral]
nonAcceptableProps  :: CAES → [PropLiteral]

getAllArgs :: ArgSet → [Argument]
getAllArgs g = nub $ concatMap (snd ∘ snd) (labNodes g)
getProps :: ArgSet → [PropLiteral]
getProps g = map (fst ∘ snd) (labNodes g)
applicableArgs :: CAES → [Argument]
applicableArgs c@(CAES (argSet, _, _)) = filter (`applicable` c) (getAllArgs argSet)
nonApplicableArgs :: CAES → [Argument]
nonApplicableArgs c@(CAES (argSet, _, _)) = filter (¬ ∘ (`applicable` c)) (getAllArgs argSet)
acceptableProps :: CAES → [PropLiteral]
acceptableProps c@(CAES (argSet, _, _)) = filter (`acceptable` c) (getProps argSet)
nonAcceptableProps :: CAES → [PropLiteral]
nonAcceptableProps c@(CAES (argSet, _, _)) = filter (¬ ∘ (`acceptable` c)) (getProps argSet)

contextP :: PropLiteral → AGraph → [Context (PropLiteral, [Argument]) ()]
contextP p = gsel (λ(_, _, a, _) → fst a ≡ p)
getArgs :: PropLiteral → AGraph → [Argument]
getArgs p g
  = case contextP p g of
      []                          → []
      ((_, _, (_, args), _) : _) → args
```

0.6 Graph construction

We associate a graph along with a *Map* that stores the node number for every *PropLiteral* to make construction of the *AGraph* easier.

```
type AGraph = ArgSet
type PropNode = LNode (PropLiteral, [Argument])
type AssociatedGraph = (AGraph, Map PropLiteral Node)
```

An argument graph is then constructed as following:

```
mkArgSet :: [Argument] → ArgSet
mkArgSet = mkArgGraph
mkArgGraph :: [Argument] → AGraph
mkArgGraph = fst ∘ foldr addArgument (empty, Map.empty)
```

Carneades uses the following definition for acyclicity:

Definition 7 (Acyclic set of arguments) *A list of arguments is acyclic iff its corresponding dependency graph is acyclic. The corresponding dependency graph has nodes for every literal appearing in the list of arguments. A node p has a directed link to node q whenever p depends on q in the sense that there is an argument pro or con p that has q or $\overline{q}$ in its list of premises or exceptions.*

So when we add an argument (*Arg premises exceptions conclusion*) to our graph, we need to add both the conclusion and its negate to the graph, adding edges for both to all premises and exceptions while adding the argument to the list of arguments for *conclusion* as well.

```
addArgument :: Argument → AssociatedGraph → AssociatedGraph
addArgument arg@(Arg (prem, exc, c)) gr =
  let deps             = prem ++ exc
      (gr', nodeNr)    = addArgument' arg gr
      (gr'', nodeNr')  = addNode (negate c) gr'
  in addEdges nodeNr' deps $ addEdges nodeNr deps gr''


addToContext :: Argument → (Context (PropLiteral, [Argument]) (), AGraph) → AGraph
addToContext arg ((adjb, n, (p, args), adja), g') = (adjb, n, (p, arg : args), adja) & g'
unsafeMatch :: Graph gr ⇒ Node → gr a b → (Context a b, gr a b)
unsafeMatch n g = mapFst fromJust $ match n g
```

Add an argument to the graph. If there is no node present yet for the conclusion insert it, in both cases add the argument to the context of the conclusion.

```
addArgument' :: Argument → AssociatedGraph → (AssociatedGraph, Node)
addArgument' arg@(Arg (_, _, c)) (g, m)
= case Map.lookup c m of
  Nothing → ((insNode (nodeNr, (c, [arg])) g,
                Map.insert c nodeNr m),
                  nodeNr)
  Just n  → ((addToContext arg (unsafeMatch n g),
                m),
                  n)
  where nodeNr = Map.size m + 1
```

Add a proposition to the graph.

```
addNode :: PropLiteral → AssociatedGraph → (AssociatedGraph, Node)
addNode p gr@(g, m)
= case Map.lookup p m of
    Nothing → ((insNode (nodeNr, (p, [])) g, Map.insert p nodeNr m), nodeNr)
    Just n → (gr, n)
   where nodeNr = Map.size m + 1
```

For a specific node, add an edge for every *PropLiteral* in the list for the given graph.

```
addEdges :: Node → [PropLiteral] → AssociatedGraph → AssociatedGraph
addEdges n ps (g, m) = addEdges' n (map (fromJust ∘ flip Map.lookup m') ps) (g', m')   -- addEdges' c n ps (g
where nodeNr   = Map.size m + 1
      newProps = filter ((≡ Nothing) ∘ flip Map.lookup m) ps
      g'       = insNodes (propsToNodes newProps nodeNr) g
      m'       = Map.union m ∘ Map.fromList $ zip newProps [nodeNr ..]
```

Generate unlabelled edges from a *Node* to a list of *Node*s and add it to the graph.

```
addEdges' :: Node → [Node] → AssociatedGraph → AssociatedGraph
addEdges' c ps (g, m) = (insEdges edges' g, m)
where edges' = map (genEdge c) ps
      genEdge k l = (k, l, ())
```

Some useful functions.

```
propsToNodes :: [PropLiteral] → Node → [PropNode]
propsToNodes ps n = zip [n ..] (map (λp → (p, [])) ps)
```

```
checkCycle :: AGraph → Bool
checkCycle = cyclic
  -- checkCycle = not . null . cyclesIn
mkProp :: String → PropLiteral
mkProp ('-' : s) = mapFst ¬ (mkProp s)
mkProp s         = (True, s)
mkAssumptions :: [String] → [PropLiteral]
mkAssumptions = map mkProp
mkArg :: [String] → [String] → String → Argument
mkArg ps es c = Arg ((map mkProp ps), (map mkProp es), (mkProp c))
```

0.7 Implementing a CAES

This subsection shows how an argumentation theorist given the Carneades DSL developed in this section quickly and at a high level of abstraction can implement a Carneades argument evaluation structure and evaluate it as well. We revisit the arguments from Section **??** and assume the following:

$$\begin{aligned} arguments &= \{arg1, arg2, arg3\}, \\ assumptions &= \{kill, witness, witness2, unreliable2\}, \\ standard(intent) &= beyond\text{-}reasonable\text{-}doubt, \\ standard(x) &= scintilla, \text{ for any other proposition x}, \\ \alpha = 0.4,\ \beta &= 0.3,\ \gamma = 0.2. \end{aligned}$$

Globally predefined alpha, beta and gamma.

```
alpha, beta, gamma :: Double
alpha = 0.4
beta  = 0.3
gamma = 0.2
```


```
alpha, beta, gamma :: Double
alpha = 0.4
beta  = 0.3
gamma = 0.2
```


Arguments and the argument graph are constructed by calling *mkArg* and *mkArgSet* respectively:

```
arg1, arg2, arg3 :: Argument
arg1 = mkArg ["kill", "intent"] [] "murder"
arg2 = mkArg ["witness"] ["unreliable"] "intent"
arg3 = mkArg ["witness2"] ["unreliable2"] "-intent"
argSet :: ArgSet
argSet = mkArgSet [arg1, arg2, arg3]
```

The audience is implemented by defining the *weight* function and calling *mkAssumptions* on the propositions which are to be assumed. The audience is just a pair of these:

```
weight :: ArgWeight
weight arg | arg ≡ arg1 = 0.8
```

```
weight arg | arg ≡ arg2 = 0.3
weight arg | arg ≡ arg3 = 0.8
weight _                = error "no weight assigned"

assumptions :: [PropLiteral]
assumptions = mkAssumptions ["kill","witness","witness2","unreliable2"]

audience :: Audience
audience = (assumptions, weight)
```

Finally, after assigning proof standards in the *standard* function, we form the CAES from the argument graph, audience and function *standard*:

```
standard :: PropStandard
standard (_,"intent") = BeyondReasonableDoubt
standard _            = Scintilla

caes :: CAES
caes = CAES (argSet, audience, standard)
```

We can now try out the argumentation structure.

```
getAllArgs argSet
> [["witness2"]       ~["unreliable2"] ⇒ "-intent",
   ["witness"]        ~["unreliable"]  ⇒ "intent",
   ["kill","intent"]~[]                ⇒ "murder"]
```

Then, as expected, there are no applicable arguments for $-intent$, since *unreliable2* is an exception, but there is an applicable argument for *intent*, namely *arg2*:

```
filter (`applicable` caes) $ getArgs (mkProp "intent") argSet
> [["witness"] ⇒ "intent"]
filter (`applicable` caes) $ getArgs (mkProp "-intent") argSet
> []
```

Despite the applicable argument *arg2* for *intent*, *murder* should not be acceptable, because the weight of $arg2 < \alpha$. However, note that we can't reach the opposite conclusion either:

```
acceptable (mkProp "murder") caes
> False
acceptable (mkProp "-murder") caes
> False
```

As a further extension, one could for example imagine giving an argumentation theorist the means to see a trace of the derivation of acceptability. It would be straightforward to add further primitives to the DSL and keeping track of intermediate results for acceptability and applicability to achieve this.

```
testAppIntent :: [Argument]
testAppIntent = filter (`applicable` caes) $ getArgs (mkProp "intent") argSet

testAppNotIntent :: [Argument]
testAppNotIntent = filter (`applicable` caes) $ getArgs (mkProp "-intent") argSet

testAppMurder :: [Argument]
testAppMurder = filter (`applicable` caes) $ getArgs (mkProp "murder") argSet

testAppNotMurder :: [Argument]
testAppNotMurder = filter (`applicable` caes) $ getArgs (mkProp "-murder") argSet

testMurder :: Bool
```

```
testMurder = acceptable (mkProp "murder") caes
testNotMurder :: Bool
testNotMurder = acceptable (mkProp "-murder") caes
```