0.1 Collecting information

In the Data.Graph library, a graph is represented as $Array\ Vertex\ [Vertex]$, mapping each vertex to a list of adjacent vertices. A $Vertex$ is simply encoded by an Int. So to test whether an edge (x, y) belongs to g we can evaluate $y \in g!x$

For more efficiency, we use Maps instead of lists. Sets would also have done, but we also want to each edge to have a path as a witness.

Moreover, as we will mostly be adding edges to the graph, we use a mutable array. If we want to use any of the library functions, we can convert our representation by $fmap\ Map.keys \circ freeze$.

```
type Graph    = Array Vertex [Vertex]
type MGraph = Array Vertex (Map.Map Vertex Path)
type MMGraph s = STArray s Vertex (Map.Map Vertex Path)
singleStep :: (Vertex → Vertex → PathStep) → Edge → EdgePath
singleStep f e@(s, t) = (e, [f s t])
```

We can add an edge to a graph, or remove it. These functions return whether they did something (resp. addition or removal) or not. hasEdge only checks whether a graph contains an edge or not.

```
addEdge :: MMGraph s → EdgePath → ST s Bool
addEdge graph ((s, t), p)
 = do m ← readArray graph s
   let b = ¬ (Map.member t m)
   when b (writeArray graph s (Map.insert t p m))
   return b
hasEdge :: MMGraph s → EdgePath → ST s Bool
hasEdge graph ((s, t), _)
 = do m ← readArray graph s
   return (Map.member t m)
```

The first step is to assign a number to all attributes, and a different one to all attribute occurrences. We create an array mapping the numbers to the information about the attribute occurrences ($ruleTable$), so we can look up this information in $O(1)$ time. We also build mappings from attributes to their occurrences ($tdsToTdp$) and vice versa ($tdpToTds$). LMH indicates the division of the attributes - an element $(l, m, h) \in LMH$ means that vertices $i, l \leqslant i \leqslant h$ are attributes of the same nonterminal, with vertices $j, l \leqslant j < m$ being inherited and $k, m \leqslant k \leqslant h$ being synthesized attributes.

See the $SequentialTypes.Info$ and $SequentialTypes.LMH$

Then we collect the direct dependencies, using the integer representations. This list of tuples (edges in the dependency graph) all information that is collected is passed to a function that will compute the interfaces and visit subsequences. We cannot do this computation in AG, because mutable arrays require the ST monad, which cannot be used inside AG.

Now we can build a graph for attributes, and a graph for ao's, and add the direct dependencies to the ao graph. Like Pennings we will call the attribte graph Tds (transitive dependencies of symbols), and the ao-graph Tdp (transitive dependencies of productions). Unlike him, we will have only one Tds and one Tdp graph. In *STGraph*, we can lookup outgoing edges in $O\ (1)$ time, but looking up incoming edges will take $O\ (e)$ time, where e is the number of edges in the graph. As we will be doing this quite often it is worthwhile to keep both Tdp and its transposed version. The computation will involve both Tds and Tdp. It treats specially. TODO elaborate on that.

```
type Tdp s = (MMGraph s, MMGraph s)
type Tds s = MMGraph s
type Comp s = (Tds s, Tdp s)
```

0.2 Generating IDS

As we insert edges into Tdp we keep it transitively closed, so every time we add the edge (s, t) to V, we also add the edges $\{(r, t) | (r, s) \in V\}$ and $\{(s, u) | (t, u) \in V\}$.

```
insertTdp :: Info → Comp s → EdgePath → ST s ()
insertTdp info comp@(_, (tdpN, tdpT)) e@((s, t), ee)   -- how to insert an edge (s,t):
   = do b ← hasEdge tdpN e                              -- if it's not yet present
     unless b
       (do rs ← readArray tdpT s                         -- find all sources r for an edge to s
           us ← readArray tdpN t                         -- find all targets u for an edge from t
           let edges = e : [((r, t), er ++ ee) | (r, er) ← Map.toList rs]
                 ++ [((s, u), ee ++ eu) | (u, eu) ← Map.toList us]
                 ++ [((r, u), er ++ ee ++ eu) | (r, er) ← Map.toList rs, (u, eu) ← Map.toList us]
           mapM_ (addTdpEdge info comp) edges           -- and add all of them, without having to bot
       )
```

Edges in *Tdp* can induce edges in *Tds*, so whenever we add an edge, we also add the induced edge if necessary

```
addTdpEdge :: Info → Comp s → EdgePath → ST s ()   -- how to add an edge (s,t) when not having
addTdpEdge info comp@(_, (tdpN, tdpT)) e@((s, t), ee)
   = do b ← addEdge tdpN e                             -- add it to the normal graph
     when b                                            -- if it was a new edge
       (do addEdge tdpT ((t, s), ee)                    -- also add it to the transposed graph
           let u = tdpToTds info ! s                    -- find the corresponding attributes...
               v = tdpToTds info ! t
               nonlocal = u ≢ −1 ∧ v ≢ −1
               equalfield = isEqualField (ruleTable info ! s) (ruleTable info ! t)
           when (nonlocal ∧ equalfield)                -- ...and when necessary...
```

```
            (insertTds info comp ((u, v), ee))            -- ...insert it to the Tds graph
        )
```

Inserting edges into *Tds* will insert edges between the occurrences of the attributes into *Tdp*.

```
insertTds :: Info → Comp s → EdgePath → ST s ()
insertTds info comp@(tds, _) e@((u, v), ee)
   = do b ← addEdge tds e
        when b
          (mapM_ (insertTdp info comp) [((s, t), [AttrStep u v])
              | s ← tdsToTdp info ! u
              , ¬ (getIsIn (ruleTable info ! s))   -- inherited at LHS, or synthesized at RHS
              , t ← tdsToTdp info ! v
              , getIsIn (ruleTable info ! t)        -- synthesized at LHS, or inherited at RHS
              , isEqualField (ruleTable info ! s) (ruleTable info ! t)
              ]
          )
```

If we add the direct dependencies to the Tdp graph in the way above, the Tds graph is filled with IDS. Below is a way to only build up the Tdp graph, without reflect the changes in the Tds graph.

```
simpleInsert :: Tdp s → EdgePath → ST s ()
simpleInsert tdp@(tdpN, tdpT) e@((s, t), ee)
   = do b ← hasEdge tdpT ((t, s), ⊥)
      unless b (do rs ← readArray tdpT s
                   us ← readArray tdpN t
                   let edges = e : [((r, t), er ++ ee) | (r, er) ← Map.toList rs]
                       ++ [((s, u), ee ++ eu) | (u, eu) ← Map.toList us]
                       ++ [((r, u), er ++ ee ++ eu) | (r, er) ← Map.toList rs, (u, eu) ← Map.toList us]
                   mapM_ (addSimpleEdge tdp) edges
        )
addSimpleEdge :: Tdp s → EdgePath → ST s ()
addSimpleEdge (tdpN, tdpT) e@((s, t), ee)
   = do b ← addEdge tdpN e
      when b (do addEdge tdpT ((t, s), ee)
        return ()
        )
```

0.3 Interfaces

In absence of cycles we can find the interfaces. We only take attributes that are used.

When an attribute has no incoming edges it can be computed. As the emphasis is on incoming edges, we will work with the transposed Tds graph. The funtion *used* indicates which vertices are included in the interfaces.

See modules Interfaces and InterfacesRules for more information.

```
makeInterfaces :: Info → Graph → T_IRoot
makeInterfaces info tds
  = let interslist = reverse ∘ makeInterface tds [ ]
        mkSegments = foldr (:_Segments ∘uncurry sem_Segment_Segment) []_Segments ∘ interslist
        mkInter ((nt, cons), lmh) = sem_Interface nt cons (mkSegments lmh)
        inters = foldr (:_Interfaces ∘mkInter) []_Interfaces (zip (nonts info) (lmh info))
    in sem_IRoot inters
```

The sinks of a graph are those vertices that have no outgoing edges. We define a function that determines whether a vertex is a sink if a set *del* of vertices had been removed from the graph. This means that the attribute can be computed if all attributes in *del* have been computed.

```
isSink :: Graph → [ Vertex ] → Vertex → Bool
isSink graph del v = null (graph ! v \\ del)
```

Now we can make interfaces by taking inherited sinks and synthesized sinks alternatively. If there are no synthesized attributes at all, generate an interface with one visit computing nothing.

```
makeInterface :: Graph → [ Vertex ] → LMH → [([ Vertex ], [ Vertex ])]
makeInterface tds del (l, m, h)
  | m > h = [([ ], [ ])]
  | otherwise = let syn = filter (isSink tds del) ([m . . h] \\ del)
                    del′ = del ++ syn
                    inh = filter (isSink tds del′) ([l . . (m − 1)] \\ del′)
                    del′′ = del′ ++ inh
                    rest = makeInterface tds del′′ (l, m, h)
    in if null inh ∧ null syn
         then [ ]
         else (inh, syn) : rest
```

0.4 Detecting of cycles

We only want to return s2i edges.

```
findCycles :: Info → MGraph → [EdgePaths]
findCycles info tds
  = [((u, v), p1, p2)
    | (l, m, h) ← lmh info          -- for every nonterminal: [l..m-1] are inherited, [m..h] are s
   , v ← [m . . h]                  -- for every synthesized attribute
```

```
    , (u, p1) ← Map.toList (tds ! v)          -- find dependent attributes...
    , l ⩽ u, u < m                            -- ...that are inherited...
    , let mbp2 = Map.lookup v (tds ! u)       -- ...and have a cycle back
    , isJust mbp2
    , let p2 = fromJust mbp2
    ]
findLocCycles :: MGraph → [EdgePath]
findLocCycles tdp
  = let (low, high) = bounds tdp
    in [((u, u), p)
       | u ← [low .. high]
       , (v, p) ← Map.toList (tdp ! u)
       , v ≡ u
       ]
findInstCycles :: [Edge] → MGraph → [EdgePath]
findInstCycles instToSynEdges tdp
  = [((i, s), fromJust mbp)
    | (i, s) ← instToSynEdges
    , let mbp = Map.lookup i (tdp ! s)
    , isJust mbp
    ]
```

0.5 Tying it together

```
generateVisits :: Info → MGraph → MGraph → [Edge] → (CInterfaceMap, CVisitsMap, [Edge])
generateVisits info tds tdp dpr
  = let inters = makeInterfaces info (fmap Map.keys tds)
        inhs = Inh_IRoot { info_Inh_IRoot = info
                         , tdp_Inh_IRoot  = fmap Map.keys tdp
                         , dpr_Inh_IRoot  = dpr
                         }
        iroot = wrap_IRoot inters inhs
    in (inters_Syn_IRoot iroot, visits_Syn_IRoot iroot, edp_Syn_IRoot iroot)
reportLocalCycle :: MGraph → [EdgePath] → [[Vertex]]
reportLocalCycle tds cyc
  = fst (foldr f ([], Set.empty) (map (edgePathToEdgeRoute tds) cyc))
    where f ((x, _), p) res@(paths, syms) | Set.member x syms = res   -- don't report a cyclic verte
      | otherwise = (p : paths, Set.union syms (Set.fromList p))
reportCycle :: Info → MGraph → [EdgePaths] → [EdgeRoutes]
reportCycle info tds cyc
  = fst (foldr f ([], Set.empty) (map (edgePathsToEdgeRoutes tds) cyc))
    where f epp@((x, y), p1, p2) res@(paths, syms) | Set.member x syms ∧
      Set.member y syms = res   -- don't report mutually dependent vertices if both appear on pat
```

```
      | otherwise           = (epp : paths, Set.union syms (Set.fromList (map tdp2tds (p1 ++ p2))))
      tdp2tds (−2) = −2
      tdp2tds v = tdpToTds info ! v
edgePathsToEdgeRoutes :: MGraph → EdgePaths → EdgeRoutes
edgePathsToEdgeRoutes tds (e, p1, p2) = (e, pathToRoute tds p1, pathToRoute tds p2)
edgePathToEdgeRoute :: MGraph → EdgePath → EdgeRoute
edgePathToEdgeRoute tds (e, p) = (e, pathToRoute tds p)
pathToRoute :: MGraph → Path → Route
pathToRoute tds p = convertPath (expandAll p)
where expandAll :: Path → Path
  expandAll p | hasAttrStep p = expandAll (expandOne p)
    | otherwise                = p
  expandOne :: Path → Path
  expandOne p = shortcut (concatMap expandStep p)
  expandStep :: PathStep → Path
  expandStep (AttrStep u v) = fromJust (Map.lookup v (tds ! u))
  expandStep x = [x]
  convertPath :: Path → Route
  convertPath p = concatMap convertStep p
  convertStep :: PathStep → Route
  convertStep (AtOcStep s t) = [s, t]
  convertStep (AttrIndu s t) = [−2, −2]
hasAttrStep :: Path → Bool
hasAttrStep [] = False
hasAttrStep (AttrStep _ _ : _) = True
hasAttrStep (_ : xs) = hasAttrStep xs
shortcut :: Eq a ⇒ [a] → [a]
shortcut [] = []
shortcut (x : xs) = x : shortcut (removeBefore x xs)
removeBefore :: Eq a ⇒ a → [a] → [a]
removeBefore x ys = reverse (takeWhile (≢ x) (reverse ys))
isLocLoc :: Table CRule → EdgePath → Bool
isLocLoc rt ((s, t), _) = isLocal (rt ! s) ∧ isLocal (rt ! t)
                                    -- — (isInst (rt ! s)  isInst (rt ! t))
computeSequential :: Info → [Edge] → [Edge] → CycleStatus
computeSequential info dpr instToSynEdges
  = runST
    (do let bigBounds = bounds (tdpToTds info)
            smallBounds = bounds (tdsToTdp info)
            (ll, es) = partition (isLocLoc (ruleTable info)) (map (singleStep AtOcStep) (dpr ++ instToSy
        tds ← newArray smallBounds Map.empty
        tdpN ← newArray bigBounds Map.empty
        tdpT ← newArray bigBounds Map.empty
        let tdp = (tdpN, tdpT)
```

```
    comp = (tds, tdp)
  mapM_ (simpleInsert tdp) ll                                              -- insert the local de
  tdp1 ← freeze tdpN
  let cyc1 = findLocCycles tdp1
  if ¬ (null cyc1)                                                         -- are they cyclic?
    then do return (LocalCycle (reportLocalCycle ⊥ cyc1))                  -- then report an er
    else do mapM_ (insertTdp info comp) es                                 -- insert the other d
            tds2 ← freeze tds
            let cyc2 = findCycles info tds2
            if ¬ (null cyc2)                                               -- are they cyclic?
              then do return (DirectCycle (reportCycle info tds2 cyc2))    -- then report an er
              else do  tdp2 ← freeze tdpN
                       let cyc4 = findInstCycles instToSynEdges tdp2
                       if ¬ (null cyc4)
                         then do return (InstCycle (reportLocalCycle tds2 cyc4))   -- then rep
                         else do let (cim, cvm, edp) = generateVisits info tds2 tdp2 dpr
                           mapM_ (insertTds info comp) (map (singleStep AttrIndu) edp)   --
                           tds3 ← freeze tds
                           let cyc3 = findCycles info tds3
                           if ¬ (null cyc3)   -- are they cyclic?
                             then return (InducedCycle cim (reportCycle info tds3 cyc3))   -
                             else do tdp3 ← freeze tdpN
                               let cyc5 = findInstCycles instToSynEdges tdp3
                               if ¬ (null cyc5)
                                 then do return (InstCycle (reportLocalCycle tds3 cyc5))   -
                                 else do return (CycleFree cim cvm)   -- otherwise we succe
)
```