# A Concurrent Interaction Net Implementation in Haskell
# HINet-0.1

Wolfram Kahl

February 14, 2015

## Abstract

Due to their “inherent parallelism”, interaction nets have since their introduction by Lafont in 1990 been considered as an attractive implementation mechanism for functional programming. The HINet package provides a simple highly-concurrent implementation in Haskell, which already can achieve promising speed-ups on multiple cores.

The HINet package also includes an interpreter RunInets for a restricted sublanguage of the .inet files of the Inets project.

The project home page is at `http://www.cas.mcmaster.ca/~kahl/Haskell/HINet/`.

## HINet Licensing

# Contents

# Chapter 1

# Interaction Net Descriptions

## 1.1 Polarity

```
module INet.Polarity where
```

Lafont Lafont (1990) and Banach and Papadopoulos Banach and Papadopoulos (1997) use typed connections in their interaction nets, where the two ports incident in a connection have the same type, but different polarity. Since we design our interaction net implementation as a run-time system, types are currently not important, and will be assumed to have been taken care of before net generation. Polarity, however, drives several run-time decisions; for the sake of readability, we define a special-purpose data-type for it (and let Haskell's "**deriving**" mechanism provide us with the default implementation of equality and ordering tests, and of conversion to strings):

```
data Polarity = Neg | Pos
  deriving (Eq, Ord, Show)
opposite :: Polarity → Polarity
opposite Neg = Pos
opposite Pos = Neg
```

We will follow Lafont's convention of letting "constructors" have positive polarity, and "functions" negative polarity.

## 1.2 Net Descriptions

```
module INet.Description where
import INet.Polarity
import INet.Utils.Vector (Vector)
```

Whereas in Sect. 2.1, we introduced types for nets considered as run-time states, here we introduce *net description* for static representation of, in particular, rule right-hand sides.

The following types are dictated by our current choice of array implementation (Data.Vector from the vector package, for efficiency), but aliased for readability:

```
type PI = Int   -- "port index"
type NI = Int   -- "node index"
```

The port index type PI will be used also in actual nets, while the node index type NI is needed only for right-hand side nodes in descriptions and during creation. We arbitrarily call the two nodes engaged in an interaction "source" and "target"; the "source" interface consists of the auxiliary ports of the node with the "function" label with negative principal port, and the "target" interface consists of the auxiliary ports of the "constructor" node with positive principal port. The following data type serves to identify all ports in a rule's right-hand side (the "!" specifies strict constructor argument positions for efficiency):

```
data PortTargetDescription
   = SourcePort  ! PI
   | InternalPort ! NI ! PI   -- node, port
   | TargetPort  ! PI      deriving (Eq, Ord, Show)
```

Therefore, each RHS node is described by its label and by the connections of *all* its ports:

```
data NodeDescription nLab = NodeDescription
  {nLab             ::                 ! nLab
  , portDescriptions :: {-# UNPACK #-} ! (Vector PortTargetDescription)
  }
```

A NetDescription is intended as description of the RHS of interaction rules:

```
data NetDescription nLab = NetDescription
  {source :: {-# UNPACK #-} ! (Vector PortTargetDescription)
  , target :: {-# UNPACK #-} ! (Vector PortTargetDescription)
  , nodes  :: {-# UNPACK #-} ! (Vector (NodeDescription nLab))
  }
```

A *language* for interaction nets consists of a type of node labels together with arity and polarity information defining *all* ports for each node label, and for any "function" node label f and any "constructor" node label c that can occur as "argument" to f a rule, specified by a right-hand side ruleRHS f c, which needs to be a net description having a source compatible with the *auxiliary* ports of f, and a target compatible with the auxiliary ports of c.

```
data INetLang nLab = INetLang {polarity :: !(nLab → Vector Polarity)
                              , ruleRHS :: !(nLab → nLab → NetDescription nLab)
                              }
```

## 1.3 INet.Description.Utils

```
module INet.Description.Utils where
import INet.Description
import qualified INet.Utils.Vector as V
```

```
isSourcePTD :: PortTargetDescription → Bool
isSourcePTD (SourcePort _) = True
isSourcePTD _ = False

isTargetPTD :: PortTargetDescription → Bool
isTargetPTD (TargetPort _) = True
isTargetPTD _ = False
```

```
instance Functor NodeDescription where
  fmap f (NodeDescription nl ptds) = NodeDescription (f nl) ptds

instance Functor NetDescription where
  fmap f (NetDescription src trg ns) = NetDescription src trg $ fmap (fmap f) ns
```

For extracting arity information:

```
collectArityFromNetDescription :: (nLab → n) → NetDescription nLab → [(n, Int)] → [(n, Int)]
collectArityFromNetDescription f nd r
   = V.foldr (λ(NodeDescription nl ptds) → ((f nl, V.length ptds):)) r (nodes nd)
```

## 1.4 INet.PTerm — "Principality Terms"

```
{-# LANGUAGE ScopedTypeVariables, FlexibleContexts, PatternGuards #-}
module INet.PTerm where
import INet.Description
import INet.Description.Utils
import INet.Utils.List (removeDuplicates)
import INet.Utils.Vector (Vector)
import qualified INet.Utils.Vector as V
import Data.Map (Map)
import qualified Data.Map as Map
import Data.List (intersperse, sortBy)
import Data.Function (on)
import Control.Monad.State
import Control.Applicative
  -- import Debug.Trace
```

PTerm is the type of Lafont-style terms denoting parts of interaction nets, where the orientation of the term constructors is "principal port up".

```
data PTerm nlab var
   = ConnVar var
   | PNode nlab [PTerm nlab var]
```

An "equation" (pt1, pt2) denotes the net created by connecting the two trees pt1 and pt2 at the principal ports of their roots.

```
type PTermEq nLab var = (PTerm nLab var, PTerm nLab var)
```

```
instance Functor (PTerm nlab) where
  fmap f (ConnVar v) = ConnVar $ f v
  fmap f (PNode nL pts) = PNode nL $ map (fmap f) pts
```

```
mapPTermNLab :: (a → b) → PTerm a var → PTerm b var
mapPTermNLab f (ConnVar v) = ConnVar v
mapPTermNLab f (PNode nL pts) = PNode (f nL) $ map (mapPTermNLab f) pts
```

```
mapPTermEqNLab :: (a → b) → PTermEq a var → PTermEq b var
mapPTermEqNLab f (t1, t2) = (mapPTermNLab f t1, mapPTermNLab f t2)
```

```
showsPTerm :: (nlab → ShowS) → (var → ShowS) → PTerm nlab var → ShowS
showsPTerm showsNLab showsVar (ConnVar v) = showsVar v
showsPTerm showsNLab showsVar (PNode nL [ ]) = showsNLab nL
showsPTerm showsNLab showsVar (PNode nL ts) = showsNLab nL ◦ ('(':) ◦
  (foldr ($) `flip` intersperse (',':) (map (showsPTerm showsNLab showsVar) ts))
   ◦ (')':)
instance (Show var, Show nlab) ⇒ Show (PTerm nlab var) where
  showsPrec _ = showsPTerm shows shows
```

```
connVars :: PTerm nlab var → [var]
connVars t = h t [ ]
  where
    h (ConnVar v) = (v:)
    h (PNode _ pts) = foldr h `flip` pts
```

```
connVarsEq :: PTermEq nlab var → [var]
connVarsEq (t1, t2) = connVars t1 ++ connVars t2
```

```
connVarsEqs :: [PTermEq nlab var] → [var]
connVarsEqs = concatMap connVarsEq
```

[ **WK:** *Raw substitution is actually never needed — see* closedSubstPTermEqs *below. Commented out:*

```
substPTerm :: (Ord var) ⇒ Map var (PTerm nLab var) → PTerm nLab var → PTerm nLab var
substPTerm m t@(ConnVar v) = case Map.lookup v m of
  Nothing → t
  Just t′ → t′
substPTerm m (PNode nL pts) = PNode nL $ map (substPTerm m) pts

substPTermEq :: (Ord var) ⇒ Map var (PTerm nLab var) → PTermEq nLab var → PTermEq nLab var
substPTermEq m (t1, t2) = (substPTerm m t1, substPTerm m t2)
```

]

mkVarRenaming newVar avoid vs returns a mapping of all vs occurring in avoid to different new variables.

```
mkVarRenaming :: (Ord var) ⇒ ([var] → var → var) → [var] → [var] → Map var var
mkVarRenaming newVar avoid = h avoid
  where
    h av [ ] = Map.empty
    h av (v : vs) = let v′ = newVar av v
      in (if v′ ≡ v then id else Map.insert v v′) (h (v′ : avoid) vs)

varRenaming :: (Ord var) ⇒ ([var] → var → var) → [var] → [var] → var → var
varRenaming newVar avoid vs v = Map.findWithDefault v v m
  where
    m = mkVarRenaming newVar avoid vs

newStringVar :: [String] → String → String
newStringVar avoid v = if v ∉ avoid then v
  else head $ filter (∉ avoid) $ map ((v++) ∘ ('_':) ∘ show) [1 . .]
```

When expanding named nets, their local variables need to be renamed to avoid clashes, with a different renaming for each occurrence.

```
closedSubstPTermEqs :: forall var nLab ∘ (Ord var)
                    ⇒ ([var] → var → var)
                    → Map var (PTerm nLab var)
                    → [PTermEq nLab var]
                    → [PTermEq nLab var]
closedSubstPTermEqs newVar m eqs = let
  avoid = connVarsEqs eqs

  substPTerm :: PTerm nLab var → State [var] (PTerm nLab var)
  substPTerm t@(ConnVar v) = case Map.lookup v m of
    Nothing → return t
    Just t1 → do
      avoid ← get
      let ren = varRenaming newVar avoid $ connVars t1
          t1′ = fmap ren t1
      modify (connVars t1′++)
      return t1′
  substPTerm (PNode nL pts) = fmap (PNode nL) $ mapM substPTerm pts

  substPTermEq :: PTermEq nLab var → State [var] (PTermEq nLab var)
  substPTermEq (t1, t2) = (, ) < $ > substPTerm t1 < ∗ > substPTerm t2

  substPTermEqs :: [PTermEq nLab var] → State [var] [PTermEq nLab var]
  substPTermEqs = mapM substPTermEq

  in evalState (substPTermEqs eqs) avoid
```

In the generalised portDescriptions for each node, we keep the principal port separate from the auxiliary ports, since target information for the principal port may become available only much later.

```
data BuildState nLab var = BuildState
  { nodeCount :: Int
  , nodeInfo :: Map NI (nLab, (Maybe (Either var PortTargetDescription), [Either var PortTargetDescription]))
  , varMap :: Map var PortTargetDescription
  }
```

```
buildPTerm :: (MonadState (BuildState nLab var) m, Ord var)
           ⇒ PTerm nLab var → m (Either var PortTargetDescription)
buildPTerm pt = buildPTermInto pt Nothing
buildPTermInto :: (MonadState (BuildState nLab var) m, Ord var)
               ⇒ PTerm nLab var
               → Maybe (Either var PortTargetDescription)
               → m (Either var PortTargetDescription)
buildPTermInto (ConnVar v) mvptd = do
  vm ← liftM varMap get
  case mvptd of
    Just (Right ptd) → modify (λs → s {varMap = Map.insert v ptd vm})
    _ → return ()
  case Map.lookup v vm of    -- in state before!
    Nothing → return $ Left v
    Just ptd′ → return $ Right ptd′
buildPTermInto (PNode nlab succs) mvptd = do
  ni ← liftM nodeCount get
  modify (λs → s {nodeCount = succ ni})
  succInfo ← sequence $ zipWith f succs $ map (InternalPort ni) [1 ..]
  modify (λs → s {nodeInfo = Map.insert ni (nlab, (mvptd, succInfo)) (nodeInfo s)})
  let result = Right $ InternalPort ni 0
  return result
  where f pt iptd = buildPTermInto pt (Just $ Right iptd)
```

```
buildEquation :: (MonadState (BuildState nLab var) m, Ord var, Show var, Show nLab)
              ⇒ PTermEq nLab var
              → m ()
buildEquation (ptL@(PNode _ _), ptR@(ConnVar _))
   = buildEquation (ptR, ptL)   -- [ WK: This avoids the current problem with this pattern. ]
buildEquation (ptL, ptR) = do
  vptdL ← buildPTerm ptL
  vptdR ← buildPTermInto ptR (Just vptdL)
  case (ptL, vptdL, vptdR) of
    (ConnVar v, _, Right ptdR) → do
      vm ← liftM varMap get
      modify (λs → s {varMap = Map.insert v ptdR vm})
    (_, Right (InternalPort ni′ 0), Right ptdR) → do    -- can only be the principal port
      nodeInfo1 ← liftM nodeInfo get
      let (nl′, (ppInfo′, auxInfo′)) = nodeInfo1Map. ! ni′
      modify (λs → s {nodeInfo = Map.insert ni′ (nl′, (Just vptdR, auxInfo′)) nodeInfo1})
    _ → return ()
    -- trace (unwords ["\nbuildEquation", show ptL, " ~ ", show ptR, " --> ", show [vptdL, vptdR]])$
  return ()
```

```
buildOrdinaryRHS :: (Ord var, Show var, Show nLab)
                 ⇒ Map var PortTargetDescription
```

```
                   → [PTermEq nLab var]
                   → NetDescription nLab
buildOrdinaryRHS ifaceMap eqs = let
  endState = execState (mapM_ buildEquation eqs) $ BuildState
    {nodeCount = 0
    , nodeInfo = Map.empty
    , varMap = ifaceMap
    }
  varMap′ = varMap endState
  mkIface msg isIfacePTD = let
    ps = sortBy (compare ‘on‘ snd) $ filter (isIfacePTD ∘ snd) $ Map.toList ifaceMap
    f (v, ptd0) = case Map.lookup v varMap′ of
      Just ptd′ → if ptd′ ≡ ptd0
        then error $ unlines $
          unwords ["\nbuildOrdinaryRHS: Unconnected", msg, "variable:", show v, show ptd0]
           : show ifaceMap : map show eqs
        else ptd′
      Nothing → error $ unlines $
          unwords ["\nbuildOrdinaryRHS: Unknown", msg, "variable:", show v, show ptd0]
           : show ifaceMap : map show eqs
    in V.fromList $ map f ps
  in NetDescription
    {source = mkIface "source" isSourcePTD
    , target = mkIface "target" isTargetPTD
    , nodes = buildStateExtractNodeDescrs endState
    }

buildStateExtractNodeDescrs :: (Ord var, Show var, Show nLab)
                              ⇒ BuildState nLab var → Vector (NodeDescription nLab)
buildStateExtractNodeDescrs endState = let
  varMap′ = varMap endState
  mkNode ni = case Map.lookup ni $ nodeInfo endState of
    Nothing → error $ unwords ["buildStateExtractNodeDescrs: Undefined node", show ni]
    Just (nlab, (Nothing, auxInfo)) → error $
      unwords ["buildStateExtractNodeDescrs: Undefined principal port for node ", show ni, show nlab, show auxInf
        -- [ WK: This shows up in ArithExpression1 with equations of shape Add (s, s2)∼s1. ]
    Just (nlab, (Just vptd, auxInfo)) → let
        mkPortDescr (Left v) = case Map.lookup v varMap′ of
          Nothing → error $
            unwords ["buildOrdinaryRHS: Undefined variable", show v, " in node ", show ni, show nlab, show (vptd : aux
          Just ptd → ptd
        mkPortDescr (Right ptd) = ptd
      in NodeDescription
        {nLab = nlab
        , portDescriptions = V.fromList $ map mkPortDescr (vptd : auxInfo)
        }
  in V.fromList $ map mkNode [0 . . nodeCount endState − 1]

buildOrdinaryRHS′ :: (Ord var, Show var, Show nLab)
                  ⇒ [var] → [var] → [PTermEq nLab var] → NetDescription nLab
buildOrdinaryRHS′ sourceVars targetVars eqs = buildOrdinaryRHS vm eqs
  where
    vm = Map.fromList $ f SourcePort sourceVars ++ f TargetPort targetVars
    f mkPTD = zipWith (λi v → (v, mkPTD i)) [1 . .]

buildNet :: (Ord var, Show var, Show nLab)
```

```
          ⇒ [PTermEq nLab var] → ([var], NetDescription nLab)
buildNet eqs = (,) sourceVars $ buildOrdinaryRHS′ sourceVars [] eqs
  where
    sourceVars = removeDuplicates $ foldr (λ(t1, t2) vs → connVars t1 ⧺ connVars t2 ⧺ vs) [] eqs
```

Lafont-style presentation of interaction net rules uses a single pair of non-variable PTerms, has their roots as LHS, and the arguments of the roots connect to the corresponding interface ports of the redex.

```
buildLafontRule :: (Ord var, Show var, Show nLab)
                ⇒ PTermEq nLab var → ((nLab, nLab), NetDescription nLab)
buildLafontRule (PNode nlab1 pts1, PNode nlab2 pts2) = (,) (nlab1, nlab2) $ NetDescription
      {source = mkIFace src
      , target = mkIFace trg
      , nodes = buildStateExtractNodeDescrs endState
      }
  where
    f mkPTD = zipWith (λi t → buildPTermInto t (Just $ Right $ mkPTD i)) [1 ..]
    mkIFace = V.fromList ∘ map (either (varMap′Map.!) id)
    varMap′ = varMap endState
    ((src, trg), endState) = runState (do
      src0 ← sequence $ f SourcePort pts1
      trg0 ← sequence $ f TargetPort pts2
      return (src0, trg0)
      ) $ BuildState
      {nodeCount = 0
      , nodeInfo = Map.empty
      , varMap = Map.empty
      }
buildLafontRule eq = error $ "buildLafontRule: Illegal argument: " ⧺ show eq
```

For extracting a natural number from a net producing unary natural numbers created from S and Z constructors, returning Int should normally be sufficient. However, it is at least theoretically possible to be successfully extracting results from nets that can never have been completely in memory at any one time. For example, one might have long-running nets on 32-bit platforms where natFromPTerm consumes more than $2^{32}$ top-level S constructors before the Z is finally even created. Therefore we still return Integer here. (We use pattern guards to ensure that the node label is inspected before the successor list.)

```
natFromPTerm :: (nLab → String) → (nLab → Bool) → (nLab → Bool) → PTerm nLab var → Either String Integer
natFromPTerm showNLab isZ isS = h 0 where
  err i s = Left $ "natFromPTerm " ⧺ shows i (": " ⧺ s)
  h i (PNode nL ts) | isZ nL, [] ← ts  = Right i
  h i (PNode nL ts) | isS nL, [t] ← ts = h (succ i) t
  h i (PNode nL ts) = err i $ "encountered " ⧺ showNLab nL ⧺
                              " with " ⧺ shows (length ts) " successors"
  h i (ConnVar _) = err i $ "encountered ConnVar!"
```

## 1.5 JSON Representation of Net Descriptions

```
 {-# LANGUAGE PatternGuards #-}
module INet.Description.JSON where

import INet.Description
import INet.Rule

import INet.Utils.Vector (Vector)
import qualified INet.Utils.Vector as V
import Text.JSON

showVector :: (JSON a) ⇒ Vector a → JSValue
showVector = showJSON ∘ V.toList
readVector :: (JSON a) ⇒ JSValue → Result (Vector a)
readVector = fmap V.fromList ∘ readJSON

instance (JSON a) ⇒ JSON (Vector a) where
  readJSON = readVector
  showJSON = showVector

showPortTargetDescription :: PortTargetDescription → JSValue
showPortTargetDescription ptd = JSObject ∘ toJSObject $ case ptd of
  SourcePort pi1 → [("SourcePort", showJSON pi1)]
  TargetPort pi2 → [("TargetPort", showJSON pi2)]
  InternalPort ni pi3 → [("InternalPort", JSArray [showJSON ni, showJSON pi3])]
readPortTargetDescription :: JSValue → Result PortTargetDescription
readPortTargetDescription (JSObject obj) = case fromJSObject obj of
  [(s, v)]
     | s ∈ ["SourcePort", "src"] → case readJSON v of
       Ok pi1 → Ok (SourcePort pi1)
       Error e → Error $ "PortTargetDescription: SourcePort: " ⧺ e
     | s ∈ ["TargetPort", "trg"] → case readJSON v of
       Ok pi2 → Ok (TargetPort pi2)
       Error e → Error $ "PortTargetDescription: TargetPort: " ⧺ e
     | s ∈ ["InternalPort", "int"] → case v of
       JSArray [vni, vpi3] → case readJSON vni of
         Ok ni → case readJSON vpi3 of
           Ok pi3 → Ok (InternalPort ni pi3)
           Error e → Error $ "PortTargetDescription: InternalPort2: " ⧺ e
         Error e → Error $ "PortTargetDescription: InternalPort1: " ⧺ e
       _ → Error "PortTargetDescription: InternalPort: 2-element array expected"
  _ → Error "PortTargetDescription: one-element object expected"
readPortTargetDescription _ = Error "PortTargetDescription expected"

instance JSON PortTargetDescription where
  readJSON = readPortTargetDescription
  showJSON = showPortTargetDescription

showNetDescription :: (JSON nLab) ⇒ NetDescription nLab → JSValue
showNetDescription nd = JSObject $ toJSObject
  [("source", showJSON $ source nd)
  , ("target", showJSON $ target nd)
  , ("nodes", showJSON $ nodes nd)
  ]
readNetDescription :: (JSON nLab) ⇒ JSValue → Result (NetDescription nLab)
readNetDescription (JSObject nd)
   | [("source", vsi), ("target", vti), ("nodes", vns)] ← fromJSObject nd
  , Ok si ← readJSON vsi
  , Ok ti ← readJSON vti
  , Ok ns ← readJSON vns
   = Ok $ NetDescription
     {source = si
```

```
      , target = ti
      , nodes = ns
      }
readNetDescription _ = Error "readNetDescription"

instance (JSON nLab) ⇒ JSON (NetDescription nLab) where
  readJSON = readNetDescription
  showJSON = showNetDescription

showNodeDescription :: (JSON nLab) ⇒ NodeDescription nLab → JSValue
showNodeDescription nd = JSArray
  [showJSON (nLab nd)
  , showJSON $ portDescriptions nd
  ]
readNodeDescription :: (JSON nLab) ⇒ JSValue → Result (NodeDescription nLab)
readNodeDescription (JSArray [vnd, vpds]) = case readJSON vnd of
  Ok label → case readJSON vpds of
    Ok pds → Ok $ NodeDescription
      {nLab = label
      , portDescriptions = pds
      }
    Error e → Error $ "readNodeDescription: Port descriptions expected: " ⧺ e
  Error e → Error $ "readNodeDescription: Label expected: " ⧺ e
readNodeDescription _ = Error "readNodeDescription: 2-element array expected"

instance (JSON nLab) ⇒ JSON (NodeDescription nLab) where
  readJSON = readNodeDescription
  showJSON = showNodeDescription

showRule :: (JSON nLab) ⇒ Rule nLab → JSValue
showRule = showJSON ◦ fromRule
readRule :: (JSON nLab) ⇒ JSValue → Result (Rule nLab)
readRule = fmap (uncurry Rule) ◦ readJSON

instance (JSON nLab) ⇒ JSON (Rule nLab) where
  readJSON = readRule
  showJSON = showRule
```

## 1.6 GraphViz Representation of Net Descriptions

```
 {-# LANGUAGE PatternGuards, FlexibleInstances #-}
module INet.Description.Dot where
import INet.Description
import INet.Description.Check (ptdRel)
import INet.Utils.Dot
import qualified INet.Utils.Vector as V

srcName = "Source"
trgName = "Target"
```

```
nodeName ni = 'N' : show ni
portName1 i = 'p' : show i
portName2 i = 'p' : show i
portName3 i = 'p' : show i


ptdString :: Char → (NI → String) → PortTargetDescription → String
ptdString sep nodeName ptd = case ptd of
   SourcePort pi1 → srcName ⧺ sep : portName1 pi1
   TargetPort pi2 → trgName ⧺ sep : portName2 pi2
   InternalPort ni pi3 → nodeName ni ⧺ sep : portName3 pi3


ptdName :: PortTargetDescription → String
ptdName = ptdString ':' (show ∘ nodeName)


shapeRecord = ("shape", "record")
shapeConn = ("shape", "circle")
showsPortId portName i s = '<' : portName i ⧺ '>' : shows i (' ' : s)
showsPortIds portName [ ] s = s
showsPortIds portName [i] s = showsPortId portName i s
showsPortIds portName (i : is) s = showsPortId portName i $ foldr (λj r → '|' : showsPortId portName j r) s is
showsPortIds′ portName [i] = showsPortId portName i
showsPortIds′ portName is = brace $ showsPortIds portName is
brace ss = ('{':) ∘ ss ∘ ('}':)
defaultDotLabel portName label (i, j) = defaultDotLabel1 portName label [i . . j]
defaultDotLabel1 portName label (i : is) = defaultDotLabel2 portName label [i] is
defaultDotLabel1 portName label [ ] = error $ "defaultDotLabel1 " ⧺ label ⧺ " []"
defaultDotLabel2 portName label is1 is2 = brace
   ((if null is1 then id else showsPortIds′ portName is1 ∘ ('|':))
    ∘ (label⧺) ∘
      (if null is2 then id else ('|':) ∘ showsPortIds′ portName is2)
   ) ""
dotNode :: String → String → Attrs → Stmt
dotNode name label attrs
    = Node name
    $ shapeRecord
    : ("height", "0.1")
    : ("label", label)
    : attrs
dotConnection :: (PortTargetDescription, PortTargetDescription) → [Stmt] → [Stmt]
dotConnection (ptd1, ptd2) s
    = Node connNode [("shape", "circle"), ("height", "0.1"), ("width", "0.1"), ("label", ""), ("fixedsize", "true")]
    : UndirEdge (ptdName ptd1) connNode [ ]
    : UndirEdge connNode (ptdName ptd2) [ ]
    : s
   where connNode = (ptdString '_' nodeName ptd1 ⧺ "__" ⧺ ptdString '_' nodeName ptd2)


dotNetDescription
    :: (nLab → String)
    → (nLab → Maybe String)
    → String
    → NetDescription nLab
    → DotGraph
```

```
dotNetDescription showNLab dotNLab name nd = DotGraph Graph name
  $ dotNode srcName (defaultDotLabel2 portName1 srcName [ ] [0 . . pred (V.length $ source nd)]) [ ]
  : dotNode trgName (defaultDotLabel2 portName2 trgName [0 . . pred (V.length $ target nd)] [ ]) [ ]
  : flip (foldr (λ(ni, d) → (:) $ dotNode (nodeName ni)
     (case dotNLab $ nLab d of
        Nothing → defaultDotLabel portName3 (showNLab $ nLab d) (V.bounds $ portDescriptions d)
        Just s → s
     ) [ ])) (V.assocs $ nodes nd)
  (foldr dotConnection [ ] $ ptdRel nd)
```

## 1.7 Show Instances for Net Descriptions

```
module INet.Description.Show where
import INet.Description
import qualified INet.Utils.Vector as V
```

```
instance Show nLab ⇒ Show (NodeDescription nLab) where
  show (NodeDescription nLab ptds) = unwords ["NodeDescr", show nLab, show (V.toList ptds)]
```

```
instance Show nLab ⇒ Show (NetDescription nLab) where
  show (NetDescription src trg ns) = unlines $
     "NetDescription" : map ("   "⧺)
       (["source = " ⧺ show (V.toList src)
         , "target = " ⧺ show (V.toList trg)
         , "ns = "
         ]
         ⧺ zipWith (λi n → ("   " ⧺ shows i ("   " ⧺ show n))) [0 . .] (V.toList ns)
       )
```

## 1.8 Rules

For the actual execution of interaction net reduction, the rules are "contained" in the function constituents of an INetLang record (defined in INet.Description, Sect. 1.2).

Here we present a datatype for rules together with a number of utility functions for checking rules, and for converting rule sets into the ruleRHS function required for INetLang.

```
module INet.Rule where

import INet.Description
import INet.Description.Check
import INet.Description.Flip
import INet.Utils.List (groupByOnFst)
import INet.Utils.Map (mkLookup2)
import qualified INet.Utils.Vector as V

import Data.List
import Data.Function (on)

import Data.Map.Strict (Map)
import qualified Data.Map.Strict as Map

import Control.Arrow (second)
import Control.Monad (when)

   -- import Debug.Trace
```

```
data Rule nLab = Rule
  {lhs ::  {-# UNPACK #-}  ! (nLab, nLab)
  , rhs ::  {-# UNPACK #-}  ! (NetDescription nLab)
  }
```

```
showLHS :: (Show nLab) ⇒ Rule nLab → String
showLHS r = unwords [show ∘ fst $ lhs r, show ∘ snd $ lhs r]
```

[ **WK:** *No way to check that* source *and* target *are the tails?* ]

```
checkRule :: (Show nLab) ⇒ (nLab → (PI, PI)) → Rule nLab → IO ()
checkRule arity r = let
      header = showLHS r
      ptdMismatches = checkNetDescription $ rhs r
   in do
      putStrLn header
      let srcArity = second pred ∘ arity ∘ fst $ lhs r
            srcBounds = V.bounds ∘ source $ rhs r
         in when (srcArity ≢ srcBounds)
            $ putStrLn $ "source arity: " ⧺ shows srcArity "\t     source bounds: " ⧺ show srcBounds
      let trgArity = second pred ∘ arity ∘ snd $ lhs r
            trgBounds = V.bounds ∘ target $ rhs r
         in when (trgArity ≢ trgBounds)
            $ putStrLn $ "target arity: " ⧺ shows trgArity "\t     target bounds: " ⧺ show trgBounds
      case ptdMismatches of
         [ ] → return ()
         _ → putStrLn ∘ unlines $ map show ptdMismatches
```

```
fromRule :: Rule nLab → ((nLab, nLab), NetDescription nLab)
fromRule r = (lhs r, rhs r)

fromRule′ :: Rule nLab → (nLab, (nLab, NetDescription nLab))
fromRule′ r = case lhs r of
   (f, c) → (f, (c, rhs r))

flipRule :: Rule nLab → Rule nLab
flipRule (Rule (lSrc, lTrg) r) = Rule (lTrg, lSrc) $ flipNetDescription r

type RuleMap nLab = Map (nLab, nLab) (NetDescription nLab)

type Rules nLab = (nLab, nLab) → NetDescription nLab
```

```
ruleLHS :: Rule nLab → NetDescription nLab
ruleLHS (Rule (labL, labR) (NetDescription src trg nodes)) = NetDescription
   {source = V.generate (V.length src) (InternalPort 0 ∘ succ)
   , target  = V.generate (V.length trg) (InternalPort 1 ∘ succ)
   , nodes   = V.fromList
      [NodeDescription labL $ V.cons (InternalPort 1 0)
          $ V.generate (V.length src) (SourcePort ∘ succ)
      , NodeDescription labR $ V.cons (InternalPort 0 0)
          $ V.generate (V.length trg) (TargetPort ∘ succ)
      ]
   }
```

```
mkRulesFC :: (Ord nLab, Show nLab) ⇒ [Rule nLab] → nLab → nLab → NetDescription nLab
mkRulesFC rs = mkLookup2 (map fromRule rs)
```

```
mkRulesFC′ :: (Ord nLab, Show nLab) ⇒ [Rule nLab] → nLab → nLab → NetDescription nLab
mkRulesFC′ rs = let
      rs′ = groupByOnFst (≡) ∘ sortBy (compare ‘on‘ second fst) $ map fromRule′ rs
      h (f, ps) = (f
                  , let m = Map.fromList ps
                  in (Map.!) m   -- findRule m (λc → show (f, c))
                 )
    m = Map.fromList $ map h rs′
  in (Map.!) m   -- findRule m show
  where
    findRule m descr label = case Map.lookup label m of
                               Nothing → error $ "No rule for " ⧺ descr label
                               Just d → d


mkRulesM :: (Ord nLab, Show nLab) ⇒ [Rule nLab] → (nLab, nLab) → Maybe (NetDescription nLab)
mkRulesM rs = let
  rs′ = map flipRule rs
  ruleMap = Map.fromList $ map fromRule (rs′ ⧺ rs)
  inλp → Map.lookup p ruleMap

mkRules :: (Ord nLab, Show nLab) ⇒ [Rule nLab] → Rules nLab
mkRules rs p = case mkRulesM rs p of
  Nothing → error $ "No rule for " ⧺ show p
  Just rhs → rhs
```

# Chapter 2

# Polarity-Based Concurrent Implementation

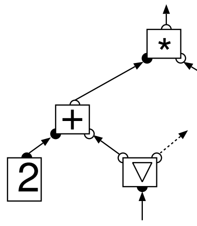
This chapter contains the core of our implementation. Its design essentially follows the main ideas of Banach and Papadopoulos (1997):

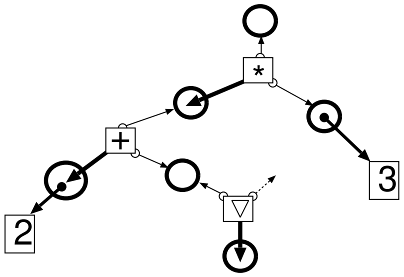
- Two-way connections, which easily introduce opportunities for deadlock and race conditions, can be avoided by using polarities to direct the connections between ports (which, in a large part of the literature, are treated as undirected, and implemented as two-way connections).
- These *directed* connections hold mutable state.
- The connection with the principal port of a constructor does not need to be known to the constructor node if the connection state refers to the node.

The following main decisions then determine most of our implementation details:

- Connections (drawn below as thick circles) are initially "empty", and each node has references to the connections attached to its auxiliary ports.
- Attaching the principal port of a constructor to a connection deposits a reference to the constructor node in the connection (which is then "full"). (This reference is drawn below with a thick arrow with a bullet tail.)
- Attaching the principal port of a function to a connection starts a concurrent thread that waits for a constructor reference in that connection, and if/when it finds one, starts the corresponding rule application. (This is drawn below with an even thicker arrow ending inside the connection.)

The following shows a net fragment to the left in more conventional style, with principal ports drawn as filled bullets, auxiliary ports has hollow circles, and positive polarity ports facing upwards, and to the right with implementation details added.





## 2.1 Mutable Net Representation

```
module INet.Polar.INet where

import Control.Concurrent.MVar
import INet.Utils.Vector (Vector)
import INet.Polarity (Polarity, opposite)
```

A connection between two ports is implemented as a single MVar that is either empty, or contains the constructor node for which the connection is at the principal port. (To allow different node label types to be used, we use the type variable nLab throughout.)

```
type Conn nLab = MVar (Node nLab)
```

For an auxiliary port of a node, besides its connection we also record the port's polarity to make it available efficiently at run-time. (In Haskell, data constructors for simple record types habitually are given the same name as the type constructor; the fields pol and conn here are declared strict using "!", and the "UNPACK" pragma declares an "unpacking" optimisation as desired to the compiler.)

```
data Port nLab = Port
  {pol  ::                     ! Polarity
  ,conn ::  {-# UNPACK #-}  ! (Conn nLab)
  }
```

We introduce the type synonym Ports to abbreviate the type of port arrays.

```
type Ports nLab = Vector (Port nLab)
```

Given a port p, the port at the other end of its connection is obtained as opPort p by flipping the polarity:

```
opPort :: Port nLab → Port nLab
opPort p = p {pol = opposite $ pol p}
```

A node contains a label, and the array of its *non-principal* ports. We do not include the principal port in ports since

- the prinicipal port of a constructor is connected to the MVar pointing back to the constructor, and
- the prinicipal port of a function is connected to the MVar the function's thread is waiting on.

```
data Node nLab = Node
  {label :: nLab
  ,ports :: Ports nLab
  }
```

## 2.2 Interaction Net Reduction

```
 {-# LANGUAGE ScopedTypeVariables, RecursiveDo #-}
module INet.Polar.Reduce where

import INet.Polarity (Polarity (. .))
import INet.Polar.INet
import INet.Description
import INet.Utils.MVar

import INet.Utils.Vector (atErr)
import qualified INet.Utils.Vector as V

import Control.Concurrent
```

The main purpose of the function replaceNet is to implement the instantiation part of the rule application step. It is a separate function because it also serves the secondary purpose of constructing the start net.

The function replaceNet takes as arguments a NetDescription (defined in Sect. 1.2) for the rule's RHS, and arrays src and trg containing the non-principal connections of the two nodes of the image of rule's LHS in the mutable net representation (Sect. 2.1) of the run-time state.

The mdo is a "recursive do" as introduced by Erkök and Launchbury (2002), and the use here essentially corresponds to the imperative programming pattern of allocating an array of uninitialised cells, and creating references to the array cells possibly before initialising them. (Functions prefix with "V." operate on Vectors.)

```
replaceNet :: forall nLab ∘ INetLang nLab → NetDescription nLab
           → Ports nLab → Ports nLab → IO ()
```

```
replaceNet lang descr src trg = mdo
  nps ← let mkNode (NodeDescription lab pds) = do
                  ps ← V.zipWithM mkPort (polarity lang lab) pds
                  return (Node {label = lab, ports = V.tail ps}
                         , V.head ps
                         )
              where mkPort Pos (InternalPort _ _) = fmap (Port Pos) newEmptyMVar
                    mkPort _   ptd                = return (portTarget ptd)
        in V.mapM mkNode (nodes descr)
```

The first step above creates descr image nodes, taking over interface ports from src and trg, creating new internal connections at positive ports, and lazily connecting negative ports with internal connections located via the function portTarget defined below.

Note that the prose explanations here are interspersed within the scope of the mdo above, since all code before the definition of reduce below remains indented below the mdo.

```
  let portTarget :: PortTargetDescription → Port nLab
      portTarget (SourcePort i) = atErr "portTarget: SourcePort S" src (pred i)
      portTarget (TargetPort i) = atErr "portTarget: TargetPort S" trg (pred i)
      portTarget (InternalPort n i) = let e = "portTarget: InternalPort "
                                          (n', pp) = atErr e nps n
        in opPort (if i ≡ 0 then pp else atErr (e ++ shows n " S") (ports n') (pred i))
```

We traverse the newly created nodes and “connect” their principal ports.

```
  let doNode (n@(Node lab prts), Port pl c) = case pl of
        Neg → forkIO (reduce lang (ruleRHS lang lab) c prts) ≫ return ()
        Pos → putMVar c n
    in V.mapM_ doNode nps
```

For source and target ports, we only need to take care of short-circuits:

```
  let doIfacePort (Port Pos c) ptd = return ()  -- will be done from the other side if necessary
      doIfacePort (Port Neg c) ptd = let        -- original port of the LHS node
        Port _pl' c' = portTarget ptd           -- connecting port in image of RHS
        in if c ≡ c'    then return ()          -- empty cycle
                        else case ptd of
                          InternalPort n i' → return ()  -- already dealt with
                          _                 → do forkIO (moveMVar c c')
                                                 return ()
    in do V.zipWithM_ doIfacePort src $ source descr
          V.zipWithM_ doIfacePort trg $ target descr
```

Whenever a function node is created, i.e., a node with positive principal port, a reduce thread is started (via forkIO). This thread waits on the connection (pconn) between the principal ports of the rule until this contains the constructor node (the principal port of which has positive polarity). The array src contains the auxiliary ports of the function node (the principal port of which has negative polarity).

```
reduce :: INetLang nLab → (nLab → NetDescription nLab) → Conn nLab → Ports nLab → IO ()
reduce lang rules pconn src = do
  Node clab trg ← takeMVar pconn
  replaceNet lang (rules clab) src trg
```

## 2.3 Creation from Descriptions

```
{-# LANGUAGE ScopedTypeVariables, RecursiveDo #-}
module INet.Polar.Create where
```

```
import INet.Polar.INet
import INet.Polar.Reduce
import INet.Description
import INet.Description.InterfacePolarity (interfacePolarity)
import qualified INet.Utils.Vector as V
import Control.Concurrent
```

```
createNet :: (Show nLab, Eq nLab   -- for debugging
  ) ⇒ INetLang nLab
     → NetDescription nLab → IO (Ports nLab, Ports nLab)
createNet lang descr = let
    mkConns = V.mapM (λpl → fmap (Port pl) newEmptyMVar)
  in case interfacePolarity lang descr of
    Nothing → fail "createNet: insufficient polarity information"
    Just (srcPol, trgPol) → do
      src ← mkConns srcPol
      trg ← mkConns trgPol
      replaceNet lang descr src trg
      return (src, trg)
```

## 2.4 Reading the End State of Polar Nets

```
module INet.Polar.Read where
import INet.Polar.INet
import INet.PTerm
import INet.Polarity
import qualified INet.Utils.Vector as V
import Control.Concurrent.MVar (takeMVar)
```

```
getPTerm :: (Show nLab) ⇒ Port nLab → IO (PTerm nLab (Conn nLab))
getPTerm (Port pl c) = h Nothing (Port (opposite pl) c) where
  err mNLab s = fail $ "getPTerm: " ⧺ s ⧺ case mNLab of
    Nothing → ""
    Just nL → " below " ⧺ show nL
  h mNLab (Port Pos c) = return $ ConnVar c
      -- err mNLab $ "encountered positive polarity"
  h mNLab (Port pl c) = do
      -- putStrLn $ unwords ["getPTerm", show mNLab, show pl]
    Node nL ps ← takeMVar c
    succs ← mapM (h $ Just nL) $ V.toList ps
    return $ PNode nL succs
```

# Chapter 3

# Interaction Nets with Plain Strings as Node Labels

## 3.1 Simple Nets

```
module INet.Simple where
import INet.Description
import INet.Rule
import Text.JSON
import INet.Description.JSON

newtype NLab = NLab {unNLab :: String} deriving (Eq, Ord)
instance Show NLab where show (NLab s) = s
instance JSON NLab where
  readJSON = fmap NLab ∘ readJSON
  showJSON (NLab s) = showJSON s

readRules :: FilePath → IO [Rule NLab]
readRules fp = do
  s ← readFile fp
  case readJSONs =≪ decodeStrict s of
    Ok rs → return rs
    Error e → fail e

readNet :: FilePath → IO (NetDescription NLab)
readNet fp = do
  s ← readFile fp
  case readNetDescription =≪ decodeStrict s of
    Ok nd → return nd
    Error e → fail e

showNet :: NetDescription NLab → String
showNet = encodeStrict
printNet :: NetDescription NLab → IO ()
printNet = putStrLn ∘ showNet
writeNet :: FilePath → NetDescription NLab → IO ()
writeNet fp = writeFile fp ∘ showNet
```

```
showRules :: [Rule NLab] → String
showRules = encodeStrict

printRules :: [Rule NLab] → IO ()
printRules = putStrLn ∘ showRules

writeRules :: FilePath → [Rule NLab] → IO ()
writeRules fp = writeFile fp ∘ showRules
```

## 3.2 Example Rules for Simple Nets

```
 {-# LANGUAGE TypeSynonymInstances, FlexibleInstances #-}
module INet.Simple.Rules where

import INet.Polarity (Polarity (..))
import INet.Description
import INet.Description.Dot
import INet.Rule
import INet.Simple

import INet.Utils.Vector (Vector, (!))
import qualified INet.Utils.Vector as V

import Data.Map (Map)
import qualified Data.Map as Map
  -- import Data.Set (Set)
import qualified Data.Set as Set
import Control.Arrow (first)

  -- import Debug.Trace


proj1_pair_redex :: NetDescription NLab
proj1_pair_redex = NetDescription
  {source = V.fromList [InternalPort 0 1]
  , target = V.fromList [InternalPort 1 1, InternalPort 1 2]
  , nodes = V.fromList
     [NodeDescription (NLab "proj") (V.fromList [InternalPort 1 0, SourcePort 0])
     , NodeDescription (NLab "pair") (V.fromList [InternalPort 0 0, TargetPort 0, TargetPort 1])
     ]
  }


lastPorts :: [(NLab, PI)]
lastPorts = map (first NLab)
  [("!", 0)   -- needs to be first for -rules
  , ("", 2)
  , ("pair", 2)
  , ("proj", 1)
  , ("proj", 1)
  , ("Z", 0)
  , ("S", 1)
  , ("+", 2)
  , ("*", 2)
  , ("fact", 1)
  , ("fib", 1)
  , ("fib'", 1)
  , ("ack", 2)
  , ("ack'", 2)
  , ("c", 2)
  , ("", 3)
  , ("b", 3)
  , ("@", 2)
  , ("b", 3)
  , ("b", 3)
  , ("v", 0)
  , ("d", 0)
  , ("", 2)
  ]
lastPortMap :: Map NLab PI
lastPortMap = Map.fromList lastPorts
lastPort :: NLab → PI
lastPort = (lastPortMapMap.!)
arity nLab = (0, lastPort nLab)
nLabs = map fst lastPorts


data PortLayout = PortLayout
  {resPorts :: ![PI]
  , argPorts :: ![PI]
  }
plLength (PortLayout xs ys) = length xs + length ys
plC0 = PortLayout [0] [ ]
plC1 = PortLayout [0] [1]
plC2 = PortLayout [0] [1, 2]
plC n = PortLayout [0] [1 . . n]
plF1 = PortLayout [1] [0]
plF2a = PortLayout [2] [0, 1]
plF2b = PortLayout [2] [1, 0]
plPolarity pl i = if i ∈ resPorts pl then Pos
  else if i ∈ argPorts pl then Neg
  else error $ "plPolarity " ++ show i
plPolV :: PortLayout → Vector Polarity
plPolV pl = V.generate (plLength pl) (plPolarity pl)


plNLabMap :: Map NLab PortLayout
plNLabMap = Map.fromList $ map (first NLab)
  [("!", PortLayout [] [0])
  , ("", PortLayout [1, 2] [0])
  , ("pair", plC2)
  , ("proj", plF1)
  , ("proj", plF1)
  , ("S", plC1)
  , ("Z", plC0)
  , ("fact", plF1)
  , ("fib", plF1)
  , ("fib'", plF1)
  , ("+", plF2a)
  , ("*", plF2a)
  , ("ack", plF2a)
  , ("ack'", plF2b)
  , ("c", PortLayout [2, 0] [1])      -- 2:BV is Pos
  , ("", PortLayout [2, 0] [3, 1])   -- 3:fv-link is Neg, 2:BV is Pos
  , ("b", PortLayout [2, 3] [0, 1])   -- 0:fv-link is Neg, 2:BV is Pos, 3:result
  , ("@", plF2a)
  , ("b", PortLayout [2, 1] [0, 3])   -- fv passes from right (3:Neg) to left (2:Pos)
  , ("b", PortLayout [2, 1] [0, 3])   -- like b
  , ("v", plC0)
  , ("d", PortLayout [] [0])
  , ("", PortLayout [0] [1, 2])
  ]
plNLab :: NLab → PortLayout
plNLab = (plNLabMapMap.!)

polMap :: Map NLab (Vector Polarity)
polMap = Map.map plPolV plNLabMap

portPolarity :: NLab → Vector Polarity
portPolarity = (polMapMap.!)


portName i = 'p' : show i

defaultDotLabel′ nl = defaultDotLabel portName nl (0, lastPort nl′)
  where nl′ = NLab nl
```

[ **WK:** dotNLabMap *duplicates* plNLabMap *information!* ]

```
dotNLabMap :: Map NLab String
dotNLabMap = Map.fromList $ map (first NLab)
  [("!", defaultDotLabel2 portName "!" [] [0])
  , ("", defaultDotLabel2 portName "dup" [1, 2] [0])
  , ("pair", defaultDotLabel′ "pair")
  , ("proj", defaultDotLabel1 portName "proj1" [1, 0])
  , ("proj", defaultDotLabel1 portName "proj2" [1, 0])
  , ("S", defaultDotLabel′ "S")
  , ("Z", defaultDotLabel′ "Z")
  , ("fact", defaultDotLabel1 portName "fact" [1, 0])
  , ("fib", defaultDotLabel1 portName "fib" [1, 0])
  , ("fib'", defaultDotLabel1 portName "fib'" [1, 0])
  , ("+", defaultDotLabel1 portName "+" [2, 0, 1])
  , ("*", defaultDotLabel1 portName "*" [2, 0, 1])
  , ("ack", defaultDotLabel1 portName "ack" [2, 0, 1])
  , ("ack'", defaultDotLabel1 portName "ack'" [2, 1, 0])
  , ("c", defaultDotLabel2 portName "lambdaC" [2, 0] [1])
  , ("", defaultDotLabel2 portName "lambda" [2, 0] [3, 1])
  , ("b", defaultDotLabel2 portName "lambdaB" [2, 3] [0, 1])
  , ("@", defaultDotLabel1 portName "@" [2, 0, 1])
  , ("b", defaultDotLabel2 portName "b" [2, 1] [0, 3])
  , ("b", defaultDotLabel2 portName "b" [2, 1] [0, 3])
  , ("v", defaultDotLabel′ "v")
  , ("d", defaultDotLabel2 portName "d" [] [0])
  , ("", defaultDotLabel2 portName "join" [0] [1, 2])
  ]
dotNLab :: NLab → Maybe String
dotNLab = flip Map.lookup dotNLabMap


  -- instance HasDot (NetDescription NLab) where
  -- dotGraph = dotNetDescription show dotNLab


mkNodeDescr :: NLab → [PortTargetDescription] → NodeDescription NLab
mkNodeDescr nLab ptds = NodeDescription nLab $ V.fromList ptds
```

```
mkIFace :: [PortTargetDescription] → Vector (PortTargetDescription)
mkIFace ptds = V.fromList ptds   -- previously started from 1!

mkNodes :: [NodeDescription NLab] → Vector (NodeDescription NLab)
mkNodes nds = V.fromList nds

mkNat :: PortTargetDescription → Int → Int → [NodeDescription NLab]
mkNat caller start 0 = [mkNodeDescr (NLab "Z") [caller]]
mkNat caller start k = mkNodeDescr (NLab "S") [caller, InternalPort start′ 0] : mkNat (InternalPort start 1) start′ k′
  where k′ = pred k
        start′ = succ start

fact :: Int → NetDescription NLab
fact n = NetDescription
  {source = mkIFace [InternalPort 0 1]
  , target = mkIFace []
  , nodes = mkNodes $ mkNodeDescr (NLab "fact") [InternalPort 1 0, SourcePort 1] : mkNat (InternalPort 0 0) 1 n
  }

fibND :: Int → NetDescription NLab
fibND n = NetDescription
  {source = mkIFace [InternalPort 0 1]
  , target = mkIFace []
  , nodes = mkNodes $ mkNodeDescr (NLab "fib") [InternalPort 1 0, SourcePort 1] : mkNat (InternalPort 0 0) 1 n
  }

ackND :: Int → Int → NetDescription NLab
ackND m n = NetDescription
  {source = mkIFace [InternalPort 0 2]
  , target = mkIFace []
  , nodes = mkNodes $ mkNodeDescr (NLab "ack") [InternalPort 1 0, InternalPort (2 + m) 0, SourcePort 1] : mkNat (InternalPort
  }

rules :: [Rule NLab]
rules =
  [proj1_pair, proj2_pair
  , plus_Z, plus_S
  , mult_Z, mult_S
  , fact_Z, fact_S
  , fib_Z, fib_S
  , fib′_Z, fib′_S
  , ack_Z, ack_S, ack′_Z, ack′_S
  , apply_lambdaC, apply_lambda, d_v, b_lambdaC, b_lambda
  , dup_join, dup_lambdaC, dup_lambda, deltaB_v, lambdaB_v
  ]
  ⧺ map bang (  {-drop 2 -}  nLabs)
  ⧺ map nabla (filter ((∉ ["c", "", ""]) ∘ unNLab) $ drop 2 nLabs)
```

**[ WK:** nabla *needs to be careful with polarity!* **]**

```
simpleLang = INetLang
  {polarity = portPolarity
  , ruleRHS = mkRulesFC rules
  }
```

() = !p0

```
bang :: NLab → Rule NLab
bang nl = Rule
  {lhs = (NLab "!", nl)
  , rhs = NetDescription
    {source = mkIFace []
    , target = mkIFace $ map (λn → InternalPort n 0) is
    , nodes = V.fromList   -- previously started from 1!
      $ map (λi → mkNodeDescr (NLab "!") [TargetPort $ succ i]) is
    }
  }
  where
    k = pred $ lastPort nl
    is = [0 . . k]   -- previously started from 1!
```

p1 p2 = p0

```
nabla :: NLab → Rule NLab
nabla nl = Rule
  {lhs = (NLab "", nl)
  , rhs = NetDescription
    {source = mkIFace $ [InternalPort k1 0, InternalPort k2 0]
    , target = mkIFace $ map (λn → InternalPort (pred n) 0) is
    , nodes = V.fromList   -- previously started from 1!
      $ map (λi → mkNodeDescr (NLab "") [TargetPort i, InternalPort k1 i, InternalPort k2 i]) is
      ⧺ [mkNodeDescr nl $ SourcePort 1 : map (λn → InternalPort (pred n) 1) is
        , mkNodeDescr nl $ SourcePort 2 : map (λn → InternalPort (pred n) 2) is
        ]
    }
  }
  where
    k = lastPort nl
    is = [1 . . k]   -- previously started from 1!
    k1 = k
    k2 = succ k
```

p0 = pair p1 p2

p1 = proj p0

```
proj1_pair :: Rule NLab
proj1_pair = Rule
  {lhs = (NLab "proj", NLab "pair")
  , rhs = NetDescription
    {source = mkIFace [TargetPort 1]
    , target = mkIFace [SourcePort 1, InternalPort 0 0]
    , nodes = mkNodes [mkNodeDescr (NLab "!") [TargetPort 2]]
    }
  }
```

p1 = proj p0

```
proj2_pair :: Rule NLab
proj2_pair = Rule
  {lhs = (NLab "proj", NLab "pair")
  , rhs = NetDescription
    {source = mkIFace [TargetPort 2]
```

```
        , target = mkIFace [InternalPort 0 0, SourcePort 1]
        , nodes = mkNodes [mkNodeDescr (NLab "!") [TargetPort 1]]
        }
      }
```

p2 = p0 + p1

```
    plus_Z :: Rule NLab
    plus_Z = Rule
      {lhs = (NLab "+", NLab "Z")
      , rhs = NetDescription
        {source = mkIFace [SourcePort 2, SourcePort 1]
        , target = mkIFace [ ]
        , nodes = mkNodes [ ]
        }
      }
```

p0 = S p1

```
    plus_S :: Rule NLab
    plus_S = Rule
      {lhs = (NLab "+", NLab "S")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 1 1, InternalPort 0 0]
        , target = mkIFace [InternalPort 1 0]
        , nodes = mkNodes [mkNodeDescr (NLab "S") [SourcePort 2, InternalPort 1 2]
                          , mkNodeDescr (NLab "+") [TargetPort 1, SourcePort 1, InternalPort 0 1]
                          ]
        }
      }
```

p2 = p0 * p1

```
    mult_Z :: Rule NLab
    mult_Z = Rule
      {lhs = (NLab "*", NLab "Z")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 1 0, InternalPort 0 0]
        , target = mkIFace [ ]
        , nodes = mkNodes [mkNodeDescr (NLab "Z") [SourcePort 2]
                          , mkNodeDescr (NLab "!") [SourcePort 1]
                          ]
        }
      }


    mult_S :: Rule NLab
    mult_S = Rule
      {lhs = (NLab "*", NLab "S")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 2 0, InternalPort 0 2]
        , target = mkIFace  [InternalPort 1 0]
        , nodes = mkNodes [mkNodeDescr (NLab "+") [InternalPort 2 1, InternalPort 1 2, SourcePort 2]
                          , mkNodeDescr (NLab "*") [TargetPort 1, InternalPort 2 2, InternalPort 0 1]
                          , mkNodeDescr (NLab "") [SourcePort 1, InternalPort 0 0, InternalPort 1 1]
                          ]
        }
      }
```

```
p1 = fact p0
```

```
fact_Z :: Rule NLab
fact_Z = Rule
  {lhs = (NLab "fact", NLab "Z")
  , rhs = NetDescription
    {source = mkIFace [InternalPort 0 0]
    , target = mkIFace  []
    , nodes = mkNodes [mkNodeDescr (NLab "S") [SourcePort 1, InternalPort 1 0]
                      , mkNodeDescr (NLab "Z") [InternalPort 0 1]
                      ]
    }
  }
```

```
fact_S :: Rule NLab
fact_S = Rule
  {lhs = (NLab "fact", NLab "S")
  , rhs = NetDescription
    {source = mkIFace [InternalPort 0 2]
    , target = mkIFace  [InternalPort 3 0]
    , nodes = mkNodes [mkNodeDescr (NLab "*") [InternalPort 1 0, InternalPort 2 1, SourcePort 1]
                      , mkNodeDescr (NLab "S") [InternalPort 0 0, InternalPort 3 1]
                      , mkNodeDescr (NLab "fact") [InternalPort 3 2, InternalPort 0 1]
                      , mkNodeDescr (NLab "") [TargetPort 1, InternalPort 1 1, InternalPort 2 0]
                      ]
    }
  }
```

```
fib 0 = 0
fib m = fib' n
  where n = pred m
fib' 0 = 1
fib' m = fib n + fib (n + 1)   -- = fib n + fib' n
  where n = pred m
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
ffib 0 = 0
ffib m = ffib' n
  where n = pred m
ffib' 0 = 1
ffib' m = ffib'' n 0 1
  where n = pred m
ffib'' 0 k l = k + l
ffib'' m k l = ffib'' n l (k + l)
  where n = pred m
```

```
fib_Z :: Rule NLab
fib_Z = Rule
  {lhs = (NLab "fib", NLab "Z")
  , rhs = NetDescription
    {source = mkIFace [InternalPort 0 0]
    , target = mkIFace  []
    , nodes = mkNodes [mkNodeDescr (NLab "Z") [SourcePort 1]
                      ]
    }
  }
```

```
    fib_S :: Rule NLab
    fib_S = Rule
      {lhs = (NLab "fib", NLab "S")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 0 1]
        , target = mkIFace  [InternalPort 0 0]
        , nodes = mkNodes [mkNodeDescr (NLab "fib'") [TargetPort 1, SourcePort 1]]
        }
      }


    fib′_Z :: Rule NLab
    fib′_Z = Rule
      {lhs = (NLab "fib'", NLab "Z")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 0 0]
        , target = mkIFace  []
        , nodes = mkNodes [mkNodeDescr (NLab "S") [SourcePort 1, InternalPort 1 0]
                          , mkNodeDescr (NLab "Z") [InternalPort 0 1]
                          ]
        }
      }


    fib′_S :: Rule NLab
    fib′_S = Rule
      {lhs = (NLab "fib'", NLab "S")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 0 2]
        , target = mkIFace  [InternalPort 3 0]
        , nodes = mkNodes [mkNodeDescr (NLab "+") [InternalPort 1 1, InternalPort 2 1, SourcePort 1]
                          , mkNodeDescr (NLab "fib") [InternalPort 3 1, InternalPort 0 0]
                          , mkNodeDescr (NLab "fib'") [InternalPort 3 2, InternalPort 0 1]
                          , mkNodeDescr (NLab "") [TargetPort 1, InternalPort 1 0, InternalPort 2 0]
                          ]
        }
      }

p2 = ack p0 p1

    ack 0 n = succ n
    ack m n = ack′ (pred m) n                   -- ack (S m) n = ack′ m n
    ack′ m 0 = ack m 1
    ack′ m n = ack m (ack (succ m) (pred n))    -- ack′ m (S n) = ack m (ack (S m) n)


    ack_Z :: Rule NLab
    ack_Z = Rule
      {lhs = (NLab "ack", NLab "Z")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 0 1, InternalPort 0 0]
        , target = mkIFace  []
        , nodes = mkNodes [mkNodeDescr (NLab "S") [SourcePort 2, SourcePort 1]]
        }
      }


    ack_S :: Rule NLab
    ack_S = Rule
      {lhs = (NLab "ack", NLab "S")
      , rhs = NetDescription
        {source = mkIFace [InternalPort 0 0, InternalPort 0 2]
        , target = mkIFace  [InternalPort 0 1]
        , nodes = mkNodes [mkNodeDescr (NLab "ack'") [SourcePort 1, TargetPort 1, SourcePort 2]]
        }
      }
```

```
p2 = ack′ p1 p0
```

```
  ack′_Z :: Rule NLab
  ack′_Z = Rule
    {lhs = (NLab "ack'", NLab "Z")
    , rhs = NetDescription
      {source = mkIFace [InternalPort 0 0, InternalPort 0 2]
      , target = mkIFace  []
      , nodes = mkNodes  $ mkNodeDescr (NLab "ack") [SourcePort 1, InternalPort 1 0, SourcePort 2]
                         : mkNat (InternalPort 0 1) 1 1
      }
    }

  ack′_S :: Rule NLab
  ack′_S = Rule
    {lhs = (NLab "ack'", NLab "S")
    , rhs = NetDescription
      {source = mkIFace [InternalPort 3 0, InternalPort 0 2]   -- m, result
      , target = mkIFace  [InternalPort 1 1]                  -- n
      , nodes = mkNodes [mkNodeDescr (NLab "ack") [InternalPort 3 1, InternalPort 1 2, SourcePort 2]
                        , mkNodeDescr (NLab "ack") [InternalPort 2 0, TargetPort 1, InternalPort 0 1]
                        , mkNodeDescr (NLab "S") [InternalPort 1 0, InternalPort 3 2]
                        , mkNodeDescr (NLab "") [SourcePort 1, InternalPort 0 0, InternalPort 2 1]
                        ]
      }
    }
```

### 3.2.1 Result Extraction

```
  checkNat :: NetDescription NLab → Either String Int
  checkNat nd = case V.toList $ source nd of
    [InternalPort n 0] → let
      h vis m = if Set.member m vis
        then Left $ "Cycle at " ++ show m
        else case nodes nd ! m of
          NodeDescription (NLab "Z") _ → Right 0
          NodeDescription (NLab "S") sna → case V.toList sna of
            [_, InternalPort m′ 0] → fmap succ $ h (Set.insert m vis) m′
            ptds → Left $ "node " ++ shows m " interface: " ++ show ptds
          NodeDescription nl _ → Left $ "node " ++ shows m ": " ++ show nl
      in h Set.empty n
    ptds → Left $ "source interface: " ++ show ptds
```

### 3.2.2 KCLE according to Mackie (2004)

```
  apply_lambdaC :: Rule NLab
  apply_lambdaC = Rule
  {lhs = (NLab "@", NLab "c")
  , rhs = NetDescription
     {source = mkIFace [TargetPort 2, TargetPort 1]
     , target = mkIFace [SourcePort 2, SourcePort 1]
     , nodes = mkNodes []
     }
  }

apply_lambda :: Rule NLab
apply_lambda = Rule
  {lhs = (NLab "@", NLab "")
  , rhs = NetDescription
     {source = mkIFace [TargetPort 2, TargetPort 1]
     , target = mkIFace [SourcePort 2, SourcePort 1, InternalPort 0 0]
     , nodes = mkNodes [mkNodeDescr (NLab "d") [TargetPort 3]]
     }
  }

d_v :: Rule NLab
d_v = Rule
  {lhs = (NLab "d", NLab "v")
  , rhs = NetDescription
     {source = mkIFace []
     , target = mkIFace []
     , nodes = mkNodes []
     }
  }

b_lambdaC :: Rule NLab
b_lambdaC = Rule
  {lhs = (NLab "b", NLab "c")
  , rhs = NetDescription
     {source = mkIFace [InternalPort 0 0, SourcePort 3, SourcePort 2]
     , target = mkIFace [InternalPort 0 1, InternalPort 0 2]
     , nodes = mkNodes [mkNodeDescr (NLab "c") [SourcePort 1, TargetPort 1, TargetPort 2]]
     }
  }

b_lambda :: Rule NLab
b_lambda = Rule
  {lhs = (NLab "b", NLab "")
  , rhs = NetDescription
     {source = mkIFace [InternalPort 0 0, InternalPort 1 2, InternalPort 1 3]
     , target = mkIFace [InternalPort 0 2, InternalPort 0 1, InternalPort 1 0]
     , nodes = mkNodes [mkNodeDescr (NLab "") [SourcePort 1, TargetPort 2, TargetPort 1, InternalPort 1 1]
        , mkNodeDescr (NLab "b") [TargetPort 3, InternalPort 0 3, SourcePort 2, SourcePort 3]
        ]
     }
  }

dup_join :: Rule NLab
dup_join = Rule
  {lhs = (NLab "", NLab "")
  , rhs = NetDescription
```

```
      {source = mkIFace [TargetPort 1, TargetPort 2]
      , target = mkIFace [SourcePort 1, SourcePort 2]
      , nodes = mkNodes [ ]
      }
   }


dup_lambdaC :: Rule NLab
dup_lambdaC = Rule
   {lhs = (NLab "", NLab "c")
   , rhs = NetDescription
      {source = mkIFace [InternalPort 0 0, InternalPort 1 0]
      , target = mkIFace [InternalPort 3 0, InternalPort 2 0]
      , nodes = mkNodes [mkNodeDescr (NLab "c") [SourcePort 1, InternalPort 3 1, InternalPort 2 1]
         , mkNodeDescr (NLab "c") [SourcePort 2, InternalPort 3 2, InternalPort 2 2]
         , mkNodeDescr (NLab "") [TargetPort 2, InternalPort 0 2, InternalPort 1 2]
         , mkNodeDescr (NLab "") [TargetPort 1, InternalPort 0 1, InternalPort 1 1]
         ]
      }
   }


dup_lambda :: Rule NLab
dup_lambda = Rule
   {lhs = (NLab "", NLab "")
   , rhs = NetDescription
      {source = mkIFace [InternalPort 0 1, InternalPort 0 2]
      , target = mkIFace [InternalPort 1 1, InternalPort 1 2, InternalPort 1 0]
      , nodes = mkNodes [mkNodeDescr (NLab "") [InternalPort 1 3, SourcePort 1, SourcePort 2]
         , mkNodeDescr (NLab "b") [TargetPort 3, TargetPort 1, TargetPort 2, InternalPort 0 0]
         ]
      }
   }


deltaB_v :: Rule NLab
deltaB_v = Rule
   {lhs = (NLab "b", NLab "v")
   , rhs = NetDescription
      {source = mkIFace [InternalPort 0 0, SourcePort 3, SourcePort 2]
      , target = mkIFace [ ]
      , nodes = mkNodes [mkNodeDescr (NLab "v") [SourcePort 1]]
      }
   }


lambdaB_v :: Rule NLab
lambdaB_v = Rule
   {lhs = (NLab "b", NLab "v")
   , rhs = NetDescription
      {source = mkIFace [InternalPort 0 1, InternalPort 0 2, InternalPort 0 0]
      , target = mkIFace [ ]
      , nodes = mkNodes [mkNodeDescr (NLab "c") [SourcePort 3, SourcePort 1, SourcePort 2]
         ]
      }
   }
```

## 3.3 Top-level Testing Programs for Simple Nets using the Polar Implementation

```
module INet.Simple.PolarTest where
import INet.Rule
import INet.PTerm
import INet.Polar.INet
import INet.Polar.Create
import INet.Polar.Read
import INet.Simple
import INet.Simple.Rules
import INet.Utils.Usage
import INet.Utils.Vector ((!))
import System.Environment
import System.IO (hPutStrLn, stderr)


checkRules = mapM_ (checkRule arity) rules


mainCheckRules = do
  checkRules
  putStrLn "=== checkRules DONE ==="
mainAck = do
  args ← getArgs
  case map read args of
    [m, n] → do
      (src, _) ← createNet simpleLang $ ackND m n
      i ← getNat $ src ! 0
      putStrLn $ unwords ["getNat result: ackND", show m, show n, "=", show i]
    _ → hPutStrLn stderr $ usage "ackND" "<nat> <nat>"
mainAck′ = do
  [m, n] ← fmap (map read) getArgs
  print $ unwords ["Haskell result: ack", show m, show n, "=", show $ ack m n]
mainFib = do
  args ← getArgs
  case map read args of
    [m] → do
      (src, _) ← createNet simpleLang $ fibND m
      i ← getNat $ src ! 0
      putStrLn $ unwords ["getNat result: fibND", show m, "=", show i]
    _ → hPutStrLn stderr $ usage "fibND" "<nat>"


getNat :: Port NLab → IO Integer
getNat p = getPTerm p ≫= either fail return
        ∘ natFromPTerm show (("Z" ≡) ∘ unNLab) (("S" ≡) ∘ unNLab)
```

# Chapter 4

# Handling `.inet` Files

The Inets project led by Ian Mackie has implemented the only publicly available general implementation of interaction nets, the compiler Hassan and Jiresch (2012) for the interaction net programming language "Inets". This language was introduced by Mackie (2005), with the core of the Inets implementation described by Hassan et al. (2009).

We implemented a front-end to our interaction net reduction system for the core sublanguage of Inets, leaving out in particular the extension of nested pattern matching described by Hassan et al. (2010), and generic rules and variadic agents.

Since our system depends on polarity for its directed implementation of connections, but Inets has no concept of polarity, we adopted the convention that the first-mentioned agent of each rule has negative principal port (that is, is considered as a function), and the second agent has positive principal port (constructor). This convention is adopted in most of the Inets examples anyways; only two rules in fibonacci.inet had been written the other way around. From this starting point we attempt to deduce the polarities of all other ports; for the examples accessible to us so far, we only needed to add a single additional heuristic: A function for which all other ports except one are known to have negative polarity is assumed to have positive polarity on the last port.

Inets supports "parameters", that is, agent attributes of the primitive types int, bool, float, char, and String. The description by Hassan et al. (2009) suggests that only a single parameter is allowed per agent; our implementation allows arbitrary numbers of such attributes, but expects the number and types of attributes to be determined by the agent label. We also interpret the type int as Haskell's arbitrary-precision Integer type. Our current interpreting implementation uses a parameterised agent label type:

```
data NLab arg = NLab {nLabName :: Name, nLabAttrs :: [arg]}
```

When reading a .inet file, the nets on the rule RHSs are translated into NetDescription (NLab Expression) and stored in a finite map for lookup by the rule LHS agent label pair; in the run-time net, agent labels of type NLab Value are used, and the variable bindings induced by the attributes of the interacting nodes are used at the time of rule application to evaluate the expressions in the RHSs (and the condition expressions for the conditional structure of Inets RHSs).

Inets modules can contain global variables, which are used in the examples to implement reduction counts etc.; since in a parallel implementation such global variables would require synchronisation (and thus would destroy the independence of parallel reduction), we did not implement any feature related to global variables.

## 4.1 INet.InetsFile.Abstract — Abstract Syntax of `.inet` Files

The Inets project (source code at `https://gna.org/svn/?group=inets`) uses a `.inet` file format as input.

(Variadic ranges / arrays seem to be almost unused in the `.inet` files available there.)

The following datatypes have been quickly assembled from reading src / inets / syntax / Parser.jjt.

```
module INet.InetsFile.Abstract where
import qualified Data.Char as Char
```

File output of this format will, if needed, be done using a pretty-printing library; we add "**deriving** (Show)" to the data type definitions here only for testing and debugging purposes.

```
type CompilationUnit = Either [Module] [ModuleComponent]

data Module = Module
  {moduleName :: String
  , moduleComponents :: [ModuleComponent]
  }
  deriving (Show)

cuComponents :: CompilationUnit → [ModuleComponent]
cuComponents (Left ms) = concatMap moduleComponents ms
cuComponents (Right mcs) = mcs

type Name = String

data ModuleComponent
   = MCImportStmt [Name]   -- dot-separated non-empty file path
   | MCRule Rule
   | MCNet Net
   | MCStatements [StatementOrDec]
  deriving (Show)

mcImports :: ModuleComponent → [[Name]]
mcImports (MCImportStmt p) = [p]
mcImports _ = [ ]

mcRules :: ModuleComponent → [Rule]
mcRules (MCRule rule) = [rule]
mcRules _ = [ ]

cuRules :: CompilationUnit → [Rule]
cuRules = concatMap mcRules ◦ cuComponents

data Rule
   = AgentRule
     {ruleDef :: RuleDef
     }
   | NamedRule
     {ruleName :: Name
     , ruleMoreNames :: [Name]   -- [ WK: What are these for? ]
     , ruleDef :: RuleDef
     }
  deriving (Show)

cuRuleDefs :: CompilationUnit → [RuleDef]
cuRuleDefs = map ruleDef ◦ cuRules

data RuleDef = RuleDef
  {ruleLHS :: Agent
  , ruleMatchStmts :: [StatementOrDec]
  , ruleRHSs :: [RHS]
  }
  deriving (Show)
ruleFuncName :: RuleDef → Name
ruleFuncName (RuleDef (Agent name _) _ _) = name
```

In a RuleDef, the principal port of the agent before the >< is taken to have negative polarity, while the principal ports of agents after >< but before ==>, which include nested patterns, are taken to be constructors, that is, have positive polarity. The agents of the replacing net, that is, after ==>, need their polarity derived from the incident connections.

```
ruleDefAgents :: RuleDef → (Agent, [(Agent, [Agent])])
ruleDefAgents (RuleDef lhs matchStmts rhss) = (lhs, map (rhsAgents ∘ ruleRHS) rhss)
  where
    rhsAgents (RuleRHS pat stmts body) = (pat, bodyAgents body)
    bodyAgents (RBeq eqs) = concatMap eqAgents eqs
    bodyAgents (RBif ifstmt) = concatMap eqAgents $ ifEqs ifstmt
    bodyAgents RBskip = [ ]
    ifEqs (IF b t e) = ifBlockEqs t ++ either ifBlockEqs ifEqs e
    ifBlockEqs (IfBlock pre eqs post) = eqs


data RHS = RhsBlock
  {rhsPre :: [StatementOrDec]
  , ruleRHS :: RuleRHS
  , rhsPost :: [StatementOrDec]
  }
  deriving (Show)
      -- don't emit braces for empty statements both before and after.


data RuleRHS = RuleRHS
  {rhsPattern :: Agent
  , rhsStmts :: [StatementOrDec]
  , rhsBody :: RuleBody
  }
  deriving (Show)
data RuleBody = RBeq [Equation]   -- non-empty
   | RBif IfStatement
   | RBskip
  deriving (Show)


data IfStatement = IF Expression IfBlock ElsePart
  deriving (Show)
type ElsePart = Either IfBlock IfStatement
data IfBlock = IfBlock [StatementOrDec] [Equation] [StatementOrDec]
  deriving (Show)


data Net
 = UnNamedNet [NetDef]
 | NamedNet Name ParamOrArgs [Either StatementOrDec [NetDef]]
  deriving (Show)
type NetDef = Either Equation NetInst
data NetInst = NetInst Name ParamOrArgs
  deriving (Show)


data Equation = Equation Term Term
  deriving (Show)
```

```
eqAgents :: Equation → [Agent]
eqAgents (Equation t1 t2) = [termAgent t1, termAgent t2]
data Term
    = TermAgent   {termAgent :: Agent}   -- uppercase AGENTNAME
    | TermVar     {termAgent :: Agent}   -- lowercase VARNAME
    | TermVariadic {termAgent :: Agent}  -- VARIADICNAME starting with "'" (not part of Agent name)
  deriving (Show)
data Agent = Agent Name ParamOrArgs
  deriving (Show)
```

Only uppercase names make "real" agents; lowercase names are connection variables, called "varAgent" in the Inets source.

```
isVarName :: Name → Bool
isVarName (c : _) = Char.isLower c
isVarName [] = error "isVarName: unexpected empty name"
isVarAgent :: Agent → Bool
isVarAgent (Agent name _) = isVarName name


type ParamOrArgs = [ParamOrArg]
data ParamOrArg
    = Param PrimitiveType Agent
    | ParamArray Name Int
    | Arg ExpressionOrTerm
  deriving (Show)


data ExpressionOrTerm
    = EAgent Agent
    | EVar Agent
    | Evariadic Agent
    | Eexpr Expression
  deriving (Show)


data StatementOrDec
    = Return (Maybe Term)
    | StatementExpression ETerm Expression   -- LhsExp () "=" RhsExp ()
    | Declaration PrimitiveType [Assignment]   -- non-empty
    | Print [ExpOrString]   -- non-empty
    | PrintNet [Term]   -- non-empty
  deriving (Show)
data Assignment = Assignment Name (Maybe Expression)
  deriving (Show)
type ExpOrString = Either Expression String


data PrimitiveType = Tint | Tfloat | Tchar | Tbool | TString | TAgent | TAgents
  deriving (Show)


data Expression
    = Atom ETerm [(Pred, ETerm)]
    | Or Expression Expression
    | And Expression Expression
  deriving (Show)


data Pred = Peq | Pneq | Plt | Pgt | Pleq | Pgeq
  deriving (Show)
```

```
interpPred :: Ord a ⇒ Pred → a → a → Bool
interpPred Peq = (≡)
interpPred Pneq = (≢)
interpPred Plt = (<)
interpPred Pgt = (>)
interpPred Pleq = (⩽)
interpPred Pgeq = (⩾)
```

```
data BinOp = Bplus | Bminus | Bmult | Bdiv | Bmod
  deriving (Show)
data UnOp = Uminus | Unot
  deriving (Show)
```

```
data ETerm
   = Bin BinOp ETerm ETerm
   | Un UnOp ETerm
   | Var Name
   | BoolLit Bool
   | IntLit Integer
   | StringLit String
  deriving (Show)
```

The "PrimaryPrefix ()" readc occurs only in test / examples / hanoi ∘ inet.

## 4.2 INet.InetsFile.Parser — Parsing .inet Files

[ **WK:** *For testing:*

```
either print print $ runParser (pStatementExpression < ∗ eof) [] "@" "counter = counter +1"
do bs ← mapM (λn → parseInetsFile′ (exampleDir ++ n ++ ".inet")) exampleNames; mapM_ (putStrLn ∘ show) (zip exampleName
do Right cu ← parseInetsFile (exampleDir ++ "examples/Ackerman.inet"); mapM_ print (map ruleDefAgents (cuRuleDefs cu))
```

]

```
 {-# LANGUAGE FlexibleContexts, NoMonomorphismRestriction #-}
module INet.InetsFile.Parser where
import INet.InetsFile.Abstract
import Text.Parsec
import Text.Parsec.Expr
import Text.Parsec.Token (GenLanguageDef (. .))
import qualified Text.Parsec.Token as P
import Data.Map (Map)
import qualified Data.Map as Map
import qualified Data.Char as Char
import Control.Applicative ((< $ >), (< ∗ >), (∗ >), (< ∗))
import Control.Monad (guard, mplus)
```

```
parseInetsFile :: FilePath → IO (Either ParseError CompilationUnit)
parseInetsFile path = do
  s ← readFile path
  return $ runParser pCompilationUnit [Map.empty] path s
```

```
parseInetsFile′ :: FilePath → IO Bool
parseInetsFile′ path = do
  putStrLn $ "Parsing " ++ path
  e ← parseInetsFile path
  case e of
```

```
    Right r → putStrLn "... success" ≫ return True   -- putStrLn (show r)
    Left e → print e ≫ return False

exampleDir = "../../../svn/inets/test/"
exampleNames = map ("examples/"++)
  ["Ackerman"
  ,"ArithExpression"
  ,"factorial"
  ,"fibonacci"
  ,"hanoi"
  ,"nesttest1"
  ,"nesttest2"
  ,"nonTerminate"
  ,"quicktest"
  ,"simple_generic"
  ,"sort"
  ,"triple"
  ,"yale"
  ,"monad/list"
  ,"monad/list_generic"
  ,"monad/maybe"
  ,"monad/maybe_generic"
  ,"monad/writer"
  ,"monad/writer_generic"
  ] ++ map ("lib/"++)
  ["Copier"
  ,"Eraser"
  ,"List"
  ,"Num"
  ]

primTypes :: [(String, PrimitiveType)]
primTypes =
  [("int", Tint)
  ,("bool", Tbool)
  ,("float", Tfloat)
  ,("String", TString)
  ,("char", Tchar)
  ,("Agent", TAgent)
  ,("Agents", TAgents)
  ]
primTypeNames :: [String]
primTypeNames = map fst primTypes

inetLanguageDef :: Stream s m Char ⇒ GenLanguageDef s u m
inetLanguageDef = LanguageDef
  {commentStart = "/*"
  ,commentEnd = "*/"
  ,commentLine = "//"   -- [ WK: Not recognised as first line??? ]
  ,nestedComments = True
  ,identStart = letter <|> char '_'
  ,identLetter = alphaNum <|> char '_'
  ,opStart = opChar
  ,opLetter = opChar
  , reservedNames = ["module", "import", "open", "as", "export", "print", "printNet"
                    , "end", "if", "else", "read", "readc", "readln", "return"
                    , "true", "false", "Net"] ⧺ primTypeNames
  , reservedOpNames = ["><", "~", "=", "=>"
                      , "||", "&&", "!=", "==", "<", ">", "<=", ">="
                      , "+", "-", "*", "/", "%", "!"]
  , caseSensitive = True
  }
  where
    opChar = oneOf ":!#$%&*+./<=>?@\\^|-~"

lexer :: Stream s m Char ⇒ P.GenTokenParser s u m
lexer = P.makeTokenParser inetLanguageDef

reserved = P.reserved lexer
reservedOp = P.reservedOp lexer
parens = P.parens lexer
braces = P.braces lexer
comma = P.comma lexer
semi = P.semi lexer

parens′ :: (Stream s m Char) ⇒ ParsecT s u m [a] → ParsecT s u m [a]
parens′ p = try (parens p) < | > return []

(< + >) :: ParsecT s u m a → ParsecT s u m b → ParsecT s u m (Either a b)
pa < + > pb = fmap Left pa < | > fmap Right pb

pCompilationUnit = (many1 pModule < + > pModuleComponents) < ∗ eof
pModuleComponents = many pModuleComponent

pName = P.identifier lexer

pModule :: Stream s m Char ⇒ ParsecT s Scopes m Module
pModule = do
  reserved "module"
  name ← pName
  body ← braces pModuleComponents
  return $ Module name body

pModuleComponent
    =   fmap MCImportStmt (reserved "import" ∗ > sepBy1 pName (P.dot lexer) < ∗ semi)
  < | > try (fmap MCRule pRule)
  < | > try (fmap MCNet pNet)
  < | > fmap MCStatements pStatements

type Scope = Map String PrimitiveType
type Scopes = [Scope]

withLocalScope :: (Monad m) ⇒ Scope → ParsecT s Scopes m a → ParsecT s Scopes m a
withLocalScope init p = do
```

```
   modifyState (init:)
   a ← p
   modifyState tail
   return a
addToLocalScope :: (Monad m) ⇒ String → PrimitiveType → ParsecT s Scopes m ()
addToLocalScope name ty = modifyState (λ(sc : scs) → Map.insert name ty sc : scs)
addParamOrArg (Param ty (Agent name params)) = do
   addToLocalScope name ty
   mapM_ addParamOrArg params
addParamOrArg (ParamArray name k) = return ()   -- arrays not yet supported
addParamOrArg (Arg (EVar (Agent name _))) = addToLocalScope name TAgent
addParamOrArg (Arg (EAgent (Agent name _))) = addToLocalScope name TAgent
addParamOrArg (Arg exprOrTerm) = fail $ "addParamOrArg " ++ show exprOrTerm
scopeLookup :: (Monad m) ⇒ String → ParsecT s Scopes m (Maybe PrimitiveType)
scopeLookup name = do
   scs ← getState
   return (foldr mplus Nothing $ map (Map.lookup name) scs)


pRule :: Stream s m Char ⇒ ParsecT s Scopes m Rule
pRule                    =                    fmap AgentRule pRuleDef
        -- "or" fmap NamedRule pNamedRuleDef
pRuleDef :: Stream s m Char ⇒ ParsecT s Scopes m RuleDef
pRuleDef = withLocalScope Map.empty $ do
   a@(Agent name params) ← pAgent
   addToLocalScope name TAgent
   mapM_ addParamOrArg params
   reservedOp "><"
   pre ← many pStatementOrDec
   rhss ← many1 (try pRHS)
   return $ RuleDef a pre rhss


pBlock :: Stream s m Char
        ⇒ ([StatementOrDec] → a → [StatementOrDec] → b)
        → ParsecT s Scopes m a → ParsecT s Scopes m b
pBlock wrap p = braces (do
      pre ← many pStatementOrDec
      r ← p
      post ← many pStatementOrDec
      return $ wrap pre r post
   ) < | > do
      r ← p
      return $ wrap [] r []


pRHS :: Stream s m Char ⇒ ParsecT s Scopes m RHS
pRHS = pBlock RhsBlock pRuleRHS


pRuleBody :: Stream s m Char ⇒ ParsecT s Scopes m RuleBody
pRuleBody = fmap RBeq pEquationList
    < | > fmap RBif pIfStatement
    < | > (semi ∗ > return RBskip)


pRuleRHS :: Stream s m Char ⇒ ParsecT s Scopes m RuleRHS
pRuleRHS = RuleRHS < $ > (pAgent < ∗ (optional $ reservedOp "=>"))
    < ∗ > many pStatementOrDec < ∗ > pRuleBody
```

```
pIfStatement :: Stream s m Char ⇒ ParsecT s Scopes m IfStatement
pIfStatement = do
  reserved "if"
  b ← parens pExpression
  t ← pIfBlock
  reserved "else"
  e ← fmap Left pIfBlock < | > fmap Right pIfStatement
  return $ IF b t e


pIfBlock :: Stream s m Char ⇒ ParsecT s Scopes m IfBlock
pIfBlock = pBlock IfBlock pEquationList


commaSemiList :: Stream s m Char ⇒ ParsecT s u m a → ParsecT s u m [a]
commaSemiList p = sepBy1 p comma < ∗ semi


pEquationList :: Stream s m Char ⇒ ParsecT s Scopes m [Equation]
pEquationList = commaSemiList pEquation


pNet :: Stream s m Char ⇒ ParsecT s Scopes m Net
pNet = try (withLocalScope Map.empty
  (NamedNet < $ > pName
             < ∗ > parens pParamOrArgs
             < ∗ > braces (many1 pNetBody)
  )) < | > fmap UnNamedNet pUnNamedNet


pNetBody :: Stream s m Char ⇒ ParsecT s Scopes m (Either StatementOrDec [NetDef])
pNetBody = try pStatementOrDec < + > commaSemiList pNetDef


pNetDef :: Stream s m Char ⇒ ParsecT s Scopes m NetDef
pNetDef = try pEquation < + > pNetInst


pNetInst :: Stream s m Char ⇒ ParsecT s Scopes m NetInst
pNetInst = NetInst < $ > pName < ∗ > parens pParamOrArgs


pUnNamedNet :: Stream s m Char ⇒ ParsecT s Scopes m [NetDef]
pUnNamedNet = commaSemiList pNetDef


pEquation :: Stream s m Char ⇒ ParsecT s Scopes m Equation
pEquation = do
  t1 ← pTerm
  reservedOp "~"
  t2 ← pTerm
  return $ Equation t1 t2


pTerm :: Stream s m Char ⇒ ParsecT s Scopes m Term
pTerm = try (fmap TermAgent pAgent)
   < | > fmap TermVar pVarAgent


pAgent :: Stream s m Char ⇒ ParsecT s Scopes m Agent
pAgent = Agent < $ > pAgentName < ∗ > parens′ pParamOrArgs
```

```
pAgentName :: Stream s m Char ⇒ ParsecT s Scopes m Name
pAgentName = do
  s@(c : _) ← pName
  guard (Char.isUpper c)
  return s
pVarAgentName :: Stream s m Char ⇒ ParsecT s Scopes m Name
pVarAgentName = do
  s@(c : _) ← pName
  guard (Char.isLower c)
  return s


pVarAgent :: Stream s m Char ⇒ ParsecT s Scopes m Agent
pVarAgent = Agent <$> pVarAgentName <*> parens′ pParamOrArgs


pParamOrArgs :: Stream s m Char ⇒ ParsecT s Scopes m ParamOrArgs
pParamOrArgs = sepBy pParamOrArg comma


pParamOrArg :: Stream s m Char ⇒ ParsecT s Scopes m ParamOrArg
pParamOrArg = try pParam <|> fmap Arg pExpressionOrTerm


pParam :: Stream s m Char ⇒ ParsecT s Scopes m ParamOrArg
pParam = do
  ty ← pPrimitiveType
  v@(Agent name params) ← pVarAgent
  addToLocalScope name ty
  mapM_ addParamOrArg params
  return $ Param ty v   -- other alternatives in Param () left out


inScope :: Stream s m Char ⇒ ParsecT s Scopes m Agent → ParsecT s Scopes m Agent
inScope p = do
  a@(Agent name params) ← p
  mty ← scopeLookup name
  case mty of
    Nothing → return a
    Just TAgent → return a
    _ → fail $ "Scope |- name : " ++ show mty
```

[ **WK:** ] pExpressionOrTerm *needs to use the* Scopes*, apparently to distinguish variable agent names from expression variables!* ]

```
pExpressionOrTerm :: Stream s m Char ⇒ ParsecT s Scopes m ExpressionOrTerm
pExpressionOrTerm = try (fmap EAgent $ inScope pAgent)
   <|> try (fmap EVar $ inScope pVarAgent)
   <|> fmap Eexpr pExpression


pConstants :: Stream s m Char ⇒ (String → ParsecT s u m x) → [(String, a)] → ParsecT s u m a
pConstants mkP ps = foldr1 (<|>) $ map (λ(s, t) → mkP s *> return t) ps


pPrimitiveType :: Stream s m Char ⇒ ParsecT s Scopes m PrimitiveType
pPrimitiveType = pConstants reserved primTypes


pStatements :: Stream s m Char ⇒ ParsecT s Scopes m [StatementOrDec]
pStatements = many1 pStatementOrDec
```

```
pStatementOrDec :: Stream s m Char ⇒ ParsecT s Scopes m StatementOrDec
pStatementOrDec = (try pReturnStatement
    < | > try pStatementExpression
    < | > try pDeclaration
    < | > pPrintStatement
   ) < ∗ semi


pReturnStatement :: Stream s m Char ⇒ ParsecT s Scopes m StatementOrDec
pReturnStatement = reserved "return" ∗ >
   (Return < $ > (fmap Just pTerm < | > return Nothing))


pStatementExpression :: Stream s m Char ⇒ ParsecT s Scopes m StatementOrDec
pStatementExpression = StatementExpression < $ > (pETerm < ∗ reservedOp "=") < ∗ > pExpression


pDeclaration :: Stream s m Char ⇒ ParsecT s Scopes m StatementOrDec
pDeclaration = do
   ty ← pPrimitiveType
   assigns ← sepBy1 pAssignment comma
   mapM_ (λ(Assignment name _) → addToLocalScope name ty) assigns
   return $ Declaration ty assigns


pAssignment :: Stream s m Char ⇒ ParsecT s Scopes m Assignment
pAssignment = Assignment < $ > pName < ∗ > (fmap Just pExpression < | > return Nothing)


pPrintStatement :: Stream s m Char ⇒ ParsecT s Scopes m StatementOrDec
pPrintStatement = (reserved "print" ∗ > (Print < $ > sepBy1 pExpOrString comma))
    < | > (reserved "printNet" ∗ > (PrintNet < $ > sepBy1 pTerm comma))


pExpOrString :: Stream s m Char ⇒ ParsecT s Scopes m ExpOrString
pExpOrString = try pExpression < + > P.stringLiteral lexer


pExpression :: Stream s m Char ⇒ ParsecT s Scopes m Expression
pExpression = buildExpressionParser table pETermChain
   where
      table = [[Infix (reservedOp "&&" ∗ > return And) AssocRight]
         , [Infix (reservedOp "||" ∗ > return Or)          AssocRight]]


pETermChain :: Stream s m Char ⇒ ParsecT s Scopes m Expression
pETermChain = try (Atom < $ > pETerm < ∗ > many ((, ) < $ > pPred < ∗ > pETerm))


pPred :: Stream s m Char ⇒ ParsecT s Scopes m Pred
pPred = pConstants reservedOp infixPreds
infixPreds =
   [("==", Peq)
   , ("!=", Pneq)
   , ("<", Plt)
   , (">", Pgt)
   , ("<=", Pleq)
   , (">=", Pgeq)
   ]
```

```
pLit :: Stream s m Char ⇒ ParsecT s Scopes m ETerm
pLit = fmap BoolLit (pConstants reserved [("false", False), ("true", True)])
   < | > fmap IntLit (P.integer lexer)
   < | > fmap StringLit (P.stringLiteral lexer)
```

```
pETerm :: Stream s m Char ⇒ ParsecT s Scopes m ETerm
pETerm = buildExpressionParser table (pLit < | > fmap Var pName < | > parens pETerm)
  where
    table =
      [[  Prefix (reservedOp "!" ∗ > return (Un Unot))
        , Prefix (reservedOp "-" ∗ > return (Un Uminus))
        , Prefix (reservedOp "+" ∗ > return id)
        ]
      , [  Infix (reservedOp "*" ∗ > return (Bin Bmult)) AssocLeft
        , Infix (reservedOp "/" ∗ > return (Bin Bdiv)) AssocLeft
        , Infix (reservedOp "%" ∗ > return (Bin Bmod)) AssocLeft
        ]
      , [  Infix (reservedOp "+" ∗ > return (Bin Bplus)) AssocLeft
        , Infix (reservedOp "-" ∗ > return (Bin Bminus)) AssocLeft
        ]
      ]
```

## 4.3 INet.InetsFile.ToDescription — Conversion to NetDescriptions

[ **WK:** *For testing:*

```
:l "INet/InetsFile/ToDescription.lhs" "INet/InetsFile/Parser.lhs"
:m + INet.InetsFile.Parser
do Right cu ← parseInetsFile (exampleDir ++ "examples/Ackerman.inet"); print (snd $ fst $ cuINetLang cu)
do Right cu ← parseInetsFile (exampleDir ++ "examples/factorial.inet"); print (fst $ fst $ cuINetLang cu)
```

]

```
 {-# LANGUAGE Rank2Types #-}
module INet.InetsFile.ToDescription where

import INet.InetsFile.Abstract

import INet.Description hiding (ruleRHS)
import INet.Description.Check (checkNetDescription)
import INet.Description.Utils (collectArityFromNetDescription)
import INet.Description.Show ()   -- instances only

import INet.Polarity

import INet.PTerm

import INet.Utils.Vector (Vector)   -- , (!?), atErr, bounds, (!))
import qualified INet.Utils.Vector as V

import Data.Map (Map)
import qualified Data.Map as Map
import Data.List (nub, intersperse)

import Control.Monad (mplus)
import Control.Arrow (first, second)

  -- import Debug.Trace
```

```
data Value
   = ValInt Integer
   | ValFloat Double
   | ValBool Bool
   | ValChar Char
   | ValString String
  deriving (Eq, Ord, Show)
valTy :: Value → PrimitiveType
valTy (ValInt _) = Tint
valTy (ValFloat _) = Tfloat
valTy (ValBool _) = Tbool
valTy (ValChar _) = Tchar
valTy (ValString _) = TString

valBinOp :: BinOp → Value → Value → Either String Value
valBinOp Bplus (ValInt i1) (ValInt i2) = Right $ ValInt $ i1 + i2
valBinOp Bplus (ValFloat d1) (ValFloat d2) = Right $ ValFloat $ d1 + d2
valBinOp Bplus (ValInt i1) (ValFloat d2) = Right $ ValFloat $ fromInteger i1 + d2
valBinOp Bplus (ValFloat d1) (ValInt i2) = Right $ ValFloat $ d1 + fromInteger i2
valBinOp Bplus (ValString s1) (ValString s2) = Right $ ValString $ s1 ⧺ s2
valBinOp Bplus (ValChar c1) (ValString s2) = Right $ ValString $ c1 : s2
valBinOp Bplus (ValString s1) (ValChar c2) = Right $ ValString $ s1 ⧺ [c2]
valBinOp Bplus (ValInt i1) (ValString s2) = Right $ ValString $ show i1 ⧺ s2
valBinOp Bplus (ValString s1) (ValInt i2) = Right $ ValString $ s1 ⧺ show i2
valBinOp Bplus (ValFloat d1) (ValString s2) = Right $ ValString $ show d1 ⧺ s2
valBinOp Bplus (ValString s1) (ValFloat d2) = Right $ ValString $ s1 ⧺ show d2
valBinOp Bplus (ValBool b1) (ValString s2) = Right $ ValString $ show b1 ⧺ s2
valBinOp Bplus (ValString s1) (ValBool b2) = Right $ ValString $ s1 ⧺ show b2
valBinOp Bminus (ValInt i1) (ValInt i2) = Right $ ValInt $ i1 − i2
valBinOp Bminus (ValFloat d1) (ValFloat d2) = Right $ ValFloat $ d1 − d2
valBinOp Bminus (ValInt i1) (ValFloat d2) = Right $ ValFloat $ fromInteger i1 − d2
valBinOp Bminus (ValFloat d1) (ValInt i2) = Right $ ValFloat $ d1 − fromInteger i2
valBinOp Bmult (ValInt i1) (ValInt i2) = Right $ ValInt $ i1 ∗ i2
valBinOp Bmult (ValFloat d1) (ValFloat d2) = Right $ ValFloat $ d1 ∗ d2
valBinOp Bmult (ValInt i1) (ValFloat d2) = Right $ ValFloat $ fromInteger i1 ∗ d2
valBinOp Bmult (ValFloat d1) (ValInt i2) = Right $ ValFloat $ d1 ∗ fromInteger i2
valBinOp Bdiv (ValInt i1) (ValInt i2) = Right $ ValInt $ i1 ‘div‘ i2
valBinOp Bdiv (ValFloat d1) (ValFloat d2) = Right $ ValFloat $ d1 / d2
valBinOp Bdiv (ValInt i1) (ValFloat d2) = Right $ ValFloat $ fromInteger i1 / d2
valBinOp Bdiv (ValFloat d1) (ValInt i2) = Right $ ValFloat $ d1 / fromInteger i2
valBinOp Bmod (ValInt i1) (ValInt i2) = Right $ ValInt $ i1 ‘mod‘ i2
valBinOp op v1 v2 = Left $ unwords ["valBinOp undefined: ", show v1, show op, show v2]

valUnOp :: UnOp → Value → Either String Value
valUnOp Uminus (ValInt i) = Right $ ValInt $ negate i
valUnOp Uminus (ValFloat d) = Right $ ValFloat $ negate d
valUnOp Unot (ValBool b) = Right $ ValBool $ ¬ b
valUnOp op v = Left $ unwords ["valUnOp undefined: ", show op, show v]

valPred :: Pred → Value → Value → Either String Bool
valPred p (ValInt i1)    (ValInt i2)    = Right $ interpPred p i1 i2
valPred p (ValFloat d1)  (ValFloat d2)  = Right $ interpPred p d1 d2
valPred p (ValBool b1)   (ValBool b2)   = Right $ interpPred p b1 b2
valPred p (ValChar c1)   (ValChar c2)   = Right $ interpPred p c1 c2
valPred p (ValString s1) (ValString s2) = Right $ interpPred p s1 s2
valPred p v1 v2 = Left $ unwords ["valPred undefined: ", show v1, show p, show v2]

evalExpression :: Map Name Value → Expression → Either String Value
evalExpression m (Or e1 e2) = do
  v1 ← evalExpression m e1
  case v1 of
```

```
    ValBool True → Right $ ValBool True
    _ → evalExpression m e2
evalExpression m e@(And e1 e2) = do
  v1 ← evalExpression m e1
  case v1 of
    ValBool False → Right $ ValBool False
    ValBool True → evalExpression m e2
    v2 → Left $ unwords ["evalExpression", show e, ": ill-typed conjunction of", show v1, "and", show v2]
evalExpression m (Atom e [ ]) = evalETerm m e
evalExpression m (Atom e ps) = do
    v ← evalETerm m e
    h v ps
  where
    h v1 ((pred1, e2) : ps) = do
      v2 ← evalETerm m e2
      r1 ← valPred pred1 v1 v2
      if r1
        then if null ps
          then Right $ ValBool True
          else h v2 ps
        else Right $ ValBool False
    h v1 [ ] = error "evalExpression: IMPOSSIBLE"
evalETerm :: Map Name Value → ETerm → Either String Value
evalETerm m (Bin op e1 e2) = do
  v1 ← evalETerm m e1
  v2 ← evalETerm m e2
  valBinOp op v1 v2
evalETerm m (Un op e) = do
  v ← evalETerm m e
  valUnOp op v
evalETerm m (Var n) = case Map.lookup n m of
  Nothing → Left $ unwords ["evalETerm: Variable", n, "undefined in", show m]
  Just v → Right v
evalETerm m (BoolLit b) = Right $ ValBool b
evalETerm m (IntLit i) = Right $ ValInt i
evalETerm m (StringLit s) = Right $ ValString s


data NLab arg = NLab
  {nLabName :: !Name
  , nLabAttrs :: ![arg]
  }
  deriving (Eq, Ord)
instance Show arg ⇒ Show (NLab arg) where
  showsPrec _ (NLab name [ ]) = (name++)
  showsPrec _ (NLab name args) = (name++) ∘ ('(':) ∘
    (foldr ($) `flip` intersperse (',':) (map shows args)) ∘ (')':)
instance Functor NLab where
  fmap f (NLab n as) = NLab n $ map f as


eVars :: [(PrimitiveType, Name)] → [Value] → Either String [(Name, Value)]
eVars tyns attrs = sequence $ zipWith f tyns attrs
    -- [ WK: Does not yet capture different lengths! ]
  where
    f (Tint, n)     v@(ValInt _)     = Right (n, v)
    f (Tfloat, n)   v@(ValFloat _)   = Right (n, v)
    f (Tfloat, n)   (ValInt i)         = Right (n, ValFloat $ fromInteger i)
    f (Tbool, n)    v@(ValBool _)      = Right (n, v)
    f (Tchar, n)    v@(ValChar _)      = Right (n, v)
    f (TString, n)  v@(ValString _)    = Right (n, v)
    f (ty, n) v = Left $ unwords ["eVars: Variable", n, "expected:", show ty, "found:", show v]
```

```
type SplitAttribs = [ParamOrArg] → ([ParamOrArg], [ParamOrArg])
    -- forall a ○ [a] → ([a], [a])
type SplitAttribsMap = Map Name SplitAttribs
```

```
mkSplitAttribs :: [ParamOrArg] → SplitAttribs
mkSplitAttribs (Param TAgent ag : pas) (x : xs) = second (x:) $ mkSplitAttribs pas xs
mkSplitAttribs (Param ty (Agent name []) : pas) (x : xs) = first (x:) $ mkSplitAttribs pas xs
mkSplitAttribs (_ : pas) (x : xs) = second (x:) $ mkSplitAttribs pas xs
mkSplitAttribs [] [] = ([], [])
mkSplitAttribs (_ : pas) [] = error $ "mkSplitAttribs: Argument list too short"
mkSplitAttribs [] _ = error $ "mkSplitAttribs: Argument list too long"
```

```
getSplitAttribs :: [RuleDef] → Map Name SplitAttribs
getSplitAttribs = foldr enter Map.empty ○ concatMap fromRuleDef
  where
    fromRuleDef (RuleDef f@(Agent fName fPAs) _ rhss) = (fName, fPAs)
         : map (fromRHS ○ ruleRHS) rhss
    fromRHS (RuleRHS (Agent cName cPAs) _ _) = (cName, cPAs)
    enter (name, pas) m = case Map.lookup name m of
      Nothing → Map.insert name (mkSplitAttribs pas) m
      Just _ → m
```

```
data Conditional expr r
   = CondLeaf r
   | CondBranch expr (Conditional expr r) (Conditional expr r)
  deriving (Show)
instance Functor (Conditional expr) where
  fmap f (CondLeaf r) = CondLeaf (f r)
  fmap f (CondBranch b t e) = CondBranch b (fmap f t) (fmap f e)
```

```
listFromConditional :: Conditional expr r → [r]
listFromConditional cond = h cond []
  where
    h (CondLeaf r) = (r:)
    h (CondBranch b t e) = h t ○ h e
```

```
evalConditional :: (expr → Bool) → Conditional expr r → r
evalConditional eval (CondLeaf r) = r
evalConditional eval (CondBranch b t e) = if eval b then evalConditional eval t
                                                   else evalConditional eval e
```

```
extractConditional :: RuleBody → Conditional Expression [Equation]
extractConditional (RBeq eqs) = CondLeaf eqs
extractConditional RBskip = CondLeaf []
extractConditional (RBif s) = fromIfStatement s
  where
    fromIfStatement (IF b t e) = CondBranch b (fromIfBlock t)
    (either fromIfBlock fromIfStatement e)
  fromIfBlock (IfBlock _ eqs _) = CondLeaf eqs
```

[ **WK:** *The* String *arguments in the following are only for debugging during development, while I am not fully understanding the Inets grammar and language definition.* ]

```
getAttrib :: String → ParamOrArg → Expression
getAttrib _ (Arg (Eexpr e)) = e
  -- getAttrib (Arg (EAgent ag)) =
getAttrib caller pa = error $ unwords ["getAttrib: unexpected", show pa, "\n     in", caller]
getArg :: String → ParamOrArg → Term
getArg _ (Arg (EAgent ag)) = TermAgent ag
getArg _ (Arg (EVar ag)) = TermVar ag
getArg _ (Arg (Evariadic ag)) = TermVariadic ag
getArg _ (Param TAgent ag@(Agent name [ ])) = TermVar ag
getArg caller pa = error $ unwords ["getArg: unexpected", show pa, "\n     in", caller]
getDecl :: String → ParamOrArg → (PrimitiveType, Name)
getDecl _ (Param ty (Agent name [ ])) = (ty, name)
getDecl caller pa = error $ unwords ["getArg: unexpected", show pa, "\n     in", caller]
mkSplitAttribsL :: String → Map Name SplitAttribs → Name → [ParamOrArg] → ([(PrimitiveType, Name)], [Term])
mkSplitAttribsL _ _ name [ ] = ([ ], [ ])
mkSplitAttribsL caller splitAttribsMap name pas = case Map.lookup name splitAttribsMap of
  Nothing → error $ unwords ["splitAttribsL:", name, "not found in splitAttribsMap"]
  Just split → case split pas of
    (attribs, args) → (map (getDecl caller) attribs, map (getArg caller) args)
mkSplitAttribsR :: String → Map Name SplitAttribs → Name → [ParamOrArg] → ([Expression], [Term])
mkSplitAttribsR _ _ name [ ] = ([ ], [ ])
mkSplitAttribsR caller splitAttribsMap name pas = case Map.lookup name splitAttribsMap of
  Nothing → error $ unwords ["splitAttribsR:", name, "not found in splitAttribsMap"]
  Just split → case split pas of
    (attribs, args) → (map (getAttrib caller) attribs, map (getArg caller) args)
argVars :: (PI → PortTargetDescription) → [Term] → [(Name, PortTargetDescription)]
argVars mkPTD = h 1
  where
    h pi [ ] = [ ]
    h pi (TermAgent ag : ts) = h (succ pi) ts
    h pi (TermVar (Agent name [ ]) : ts) = (name, mkPTD pi) : h (succ pi) ts
    h pi (_ : ts) = h (succ pi) ts
toPTerm :: String → Map Name SplitAttribs → Term → PTerm (NLab Expression) Name
toPTerm _ _ (TermVar (Agent name [ ])) = ConnVar name
toPTerm caller splitAttribsMap (TermAgent (Agent name pas))
   = case mkSplitAttribsR caller splitAttribsMap name pas of
    (attribs, args) → PNode (NLab name attribs) (map (toPTerm caller splitAttribsMap) args)
toPTerm _ _ t = error $ "toPTerm: unexpected argument: " ++ show t
toPTermEq :: String → Map Name SplitAttribs → Equation → PTermEq (NLab Expression) Name
toPTermEq caller splitAttribsMap (Equation t1 t2)
   = (toPTerm caller splitAttribsMap t1
     , toPTerm caller splitAttribsMap t2)
```

In the Inets source, src / inets / transform / NestedToOrd.java documents the following decisions:

1. All rules with more than 2 agents on the LHS *and* also having an RHS net will be considered as rules with nested patterns.
2. All rules that have more than 2 agents on the LHS *and* do not have an RHS net will be considered as ordinary rules in Lafont style.

3. All rules with exactly two agents on the LHS and an equation list as the RHS net will be considered as ordinary rules.

A twist is the following rule in lib / List.inet:

App (b, b) >< Nil;

This can be recognised as being in Lafont-style only from the fact that its variable occurs twice.

```
convertRuleDef :: Map Name SplitAttribs
               → RuleDef
               → (NLab (PrimitiveType, Name)
                 , [ (NLab (PrimitiveType, Name)
                   , Conditional Expression (NetDescription (NLab Expression)))])
convertRuleDef splitAttribsMap rd@(RuleDef f@(Agent fName fPAs) _ rhss) = let
  (fParams, fArgs0) = splitAttribsL fName fPAs
  fArgVars = argVars SourcePort fArgs0
  splitAttribsL :: Name → [ParamOrArg] → ([(PrimitiveType, Name)], [Term])
  splitAttribsL = mkSplitAttribsL (show rd) splitAttribsMap
  mkPTerm :: Term → PTerm (NLab Expression) Name
  mkPTerm = toPTerm (show rd) splitAttribsMap
  mkPTermEq :: Equation → PTermEq (NLab Expression) Name
  mkPTermEq = toPTermEq (show rd) splitAttribsMap
  convertRHS :: RuleRHS → (NLab (PrimitiveType, Name)
                          , Conditional Expression (NetDescription (NLab Expression)))
  convertRHS rhs@(RuleRHS c@(Agent cName cPAs) _ RBskip)
    | length fArgVars < length fArgs0 ∨ length cArgVars < length cArgs0
      ∨ length (nub vNames) < length vNames
    = let   -- this is a Lafont-style rule
        f′, c′ :: PTerm (NLab Expression) Name
        f′ = PNode (NLab fName $ error "fParams") $ map mkPTerm fArgs0
        c′ = PNode (NLab cName $ error "cParams") $ map mkPTerm cArgs0
        (_, nd) = buildLafontRule (f′, c′)
      in   -- trace (unwords ["\nconvertRuleDef: Lafont:", show f, "><", show c])$
        case checkNetDescription nd of
              [] → (, ) (NLab cName cParams) $ CondLeaf nd
              clashes → error $ unlines $
                ("\nconvertRHS: Inconsistent net description in Lafont-style rule:" ⧺ fName ⧺ " >< " ⧺ cN
                : unwords ["     ", show f, "><", show c]
                : show nd : "Clashes:" : map (("    "⧺) ∘ show) clashes
      where
        (cParams, cArgs0) = splitAttribsL cName cPAs
        cArgVars = argVars TargetPort cArgs0
        vNames = map fst $ fArgVars ⧺ cArgVars
  convertRHS rhs@(RuleRHS c@(Agent cName cPAs) _ body)   -- RBskip is now an empty list of equations
    = (, ) (NLab cName cParams) $ let
        patArgVars :: Map Name PortTargetDescription
        patArgVars = Map.fromList $ fArgVars ⧺ cArgVars
      in if length fArgVars < length fArgs0 ∨ length cArgVars < length cArgs0
        then error $ unwords ["Nesting not yet implemented:", show f, show rhs]
        else let
          mkNet :: [Equation] → NetDescription (NLab Expression)
          mkNet eqs = let nd = buildOrdinaryRHS patArgVars $ map mkPTermEq eqs
            in case checkNetDescription nd of
              [] → nd
              clashes → error $ unlines $
                ("\nconvertRHS: Inconsistent net description: in " ⧺ fName ⧺ " >< " ⧺ cName)
                : "Equations:" : map (("    "⧺) ∘ show) eqs
```

```
                  ++ show nd : "Clashes:" : map (("    "++) ∘ show) clashes
            in fmap mkNet $ extractConditional body
      where
        (cParams, cArgs0) = splitAttribsL cName cPAs
        cArgVars = argVars TargetPort cArgs0
  in (NLab fName fParams, map (convertRHS ∘ ruleRHS) rhss)


type SymbLangSubMap = Map Name (([(PrimitiveType, Name)], [(PrimitiveType, Name)])
                               , Conditional Expression (NetDescription (NLab Expression)))
type SymbLangMap = Map Name SymbLangSubMap


arityFromSymbLangMap :: SymbLangMap → Map Name Int
arityFromSymbLangMap = let
  fromConditional :: Conditional Expression (NetDescription (NLab Expression))
     → [(Name, Int)] → [(Name, Int)]
  fromConditional (CondLeaf nd) = collectArityFromNetDescription nLabName nd
  fromConditional (CondBranch b t e) = fromConditional t ∘ fromConditional e
  condIfaceLengths cond = let nd = head $ listFromConditional cond
                          in (V.length (source nd), V.length (target nd))
  fromSubMap :: SymbLangSubMap → (Int, [(Name, Int)] → [(Name, Int)])
  fromSubMap sm = case Map.toList sm of
      ps@(~((cName1, (_, cond1)) : _)) → case condIfaceLengths cond1 of
        (fSize, cSize) → (, ) (succ fSize) $ foldr h `flip` ps
    where
      h (cName, (_, cond)) = ((cName, succ $ snd $ condIfaceLengths cond):) ∘ fromConditional cond
  g fName sm = case fromSubMap sm of
      (fSize, collect_sm) → ((fName, fSize):) ∘ collect_sm
  in Map.fromList ∘ Map.foldrWithKey g []


cuINetLang :: CompilationUnit
   → (          (SymbLangMap, Map Name (Vector Polarity))
              , (SplitAttribsMap, INetLang (NLab Value))
              )
cuINetLang cu = let
  rds = cuRuleDefs cu
  splitAttribsMap = getSplitAttribs rds
  addToFMap :: (NLab (PrimitiveType, Name)
                , [(NLab (PrimitiveType, Name)
                   , Conditional Expression (NetDescription (NLab Expression)))]
               ) → SymbLangMap → SymbLangMap
  addToFMap (NLab fName fParams, ps)
     = Map.insertWith Map.union    -- (λnewsub subm → foldr (addToCMap fParams) subm ps)
               fName   (foldr (addToCMap fParams) Map.empty ps)
  addToCMap :: [(PrimitiveType, Name)]
               → (NLab (PrimitiveType, Name)
                 , Conditional Expression (NetDescription (NLab Expression)))
               → SymbLangSubMap → SymbLangSubMap
  addToCMap fParams (NLab cName cParams, cond) = Map.insert cName ((fParams, cParams), cond)
  symbLangMap :: SymbLangMap
  symbLangMap = foldr addToFMap Map.empty $ map (convertRuleDef splitAttribsMap) rds
  find :: NLab Value → NLab Value → NetDescription (NLab Value)
  find (NLab fName fAttribs) = case Map.lookup fName symbLangMap of
    Nothing → error $ "cuINetLang: Undefined function label " ++ fName
    Just cMap → λ(NLab cName cAttribs) → case Map.lookup cName cMap of
      Nothing → error $ "uINetLang: Undefined constructor label " ++ cName
```

```
          Just ((fParams, cParams), cond) → case eVars fParams fAttribs of
            Left e → error $ "cuINetLang: Function parameters mismatch: " ++ e
            Right fVal → case eVars cParams cAttribs of
              Left e → error $ "cuINetLang: Constructor parameters mismatch: " ++ e
              Right cVal → let
                val :: Map Name Value
                val = Map.fromList $ fVal ++ cVal
                evalExpr :: Expression → Value
                evalExpr e = case evalExpression val e of
                  Left err → error $ unwords ["cuINetLang: evaluating", show e, ":", err]
                  Right v → v
                evalBool :: Expression → Bool
                evalBool e = case evalExpr e of
                  ValBool b → b
                  v → error $ unwords ["cuINetLang: condition", show e, "evaluates to", show v]
                ndE :: NetDescription (NLab Expression)
                ndE = evalConditional evalBool cond
                in fmap (fmap evalExpr) ndE
  polMap = extractPolarity symbLangMap
  pol :: NLab Value → Vector Polarity
  pol (NLab name _) = polMapMap. ! name
  in ((symbLangMap, polMap), (splitAttribsMap, INetLang pol find))


mmInsert :: (Ord a, Ord b, Eq c, Show a, Show b, Show c)
   ⇒ String → a → b → c → Map a (Map b c) → Map a (Map b c)
mmInsert msg a b c m =    -- trace (unwords ["\nTrace mmInsert:\n   ", msg, "\n   ", show a, show b, "->", show c, "\n"])$
  Map.insertWith (Map.unionWith h) a (Map.singleton b c) m
  where
    h c2 c1 = if c1 ≡ c2 then c1
      else error $ unwords ["mmInsert", show a, show b, ":", show c1, "- ->", show c2
        , "\n     ", msg
        , "\n", unlines $ map show $ Map.toList m]
mmLookup :: (Ord a, Ord b) ⇒ a → b → Map a (Map b c) → Maybe c
mmLookup a b m = Map.lookup a m ≫= Map.lookup b


vectorFromMap :: Map PI a → Vector a
vectorFromMap = V.fromList ∘ h 0 ∘ Map.toAscList
  where
    h i [] = []
    h i ((j, x) : ps) = if j > i
      then error $ unwords ["vectorFromMap: Gap from", show i, "to below", show j]
      else x : h (succ i) ps


extractPolarity :: SymbLangMap → Map Name (Vector Polarity)
extractPolarity symbLangMap = let
  pports :: Map Name (Map PI Polarity)
  pports = Map.foldrWithKey (λfName subMap → h subMap ∘ mmInsert "fName PP" fName 0 Neg) Map.empty symbLangMap
    where
      h sm m = Map.foldrWithKey (λcName (_, cond) → mmInsert "cName PP" cName 0 Pos) m sm
  subrules :: [((Name, Name), NetDescription (NLab Expression))]
  subrules = concatMap flattenSubMap $ Map.toList symbLangMap
    where
      flattenSubMap (fName, cMap) = concatMap (flattenEntry fName) $ Map.toList cMap
      flattenEntry fName (cName, (_, cond)) = map ((, ) (fName, cName)) $ listFromConditional cond
  addSubRule    :: ((Name, Name), NetDescription (NLab Expression))
```

```
            → Map Name (Map PI Polarity) → Map Name (Map PI Polarity)
addSubRule ((fName, cName), nd) = (V.ifoldr addNode ‘flip‘ (nodes nd))
  ∘ addIface fName source
  ∘ addIface cName target
  where
      -- (fAuxSize, cAuxSize) = (V.length (source nd), V.length (target nd))
    mkMsg s = fName ++ " >< " ++ cName ++ " : " ++ s
    addIface name getPTDs m = let
        pols = Map.findWithDefault Map.empty name m
        ptds = getPTDs nd
        addPort′ i ptd = case Map.lookup i pols of
          Nothing → id
          Just pol → addPort (unwords ["f:", show i, show ptd]) ptd (opposite pol)
      in foldr ($) m $ zipWith addPort′ [1 . .] (V.toList ptds)
    addPort :: String → PortTargetDescription → Polarity
            → Map Name (Map PI Polarity) → Map Name (Map PI Polarity)
    addPort msg0 ptd pol m = case ptd of
      SourcePort pi → mmInsert (mkMsg "addPort: SourcePort") fName pi pol m
      TargetPort pi → mmInsert (mkMsg "addPort: TargetPort") cName pi pol m
      InternalPort ni pi → let
          msg2 = unwords [msg1, "addPort: InternalPort", show ni, nL′, show pi]
          nL′ = nLabName ∘ nLab $ nodes ndV. ! ni
        in mmInsert (mkMsg msg2) (nLabName ∘ nLab $ nodes ndV. ! ni) pi (opposite pol) m
      where
        msg1 = unwords ["\n", msg0, show ptd, show pol, "\n   ", show m, "\n   "]
    addNode ni0 (NodeDescription nL pds) m = case Map.lookup (nLabName nL) m of
        Nothing → m
        Just pols → Map.foldrWithKey addPol m pols
      where
        addPol :: PI → Polarity
              → Map Name (Map PI Polarity) → Map Name (Map PI Polarity)
        addPol i pol = addPort msg0 (V.atErr "addPol" pds i) pol
          where
            msg0 = unwords [show nL, show ni0, show i, show pds, "\n   "]
arityMap = arityFromSymbLangMap symbLangMap
arityCount = Map.size arityMap
inComplete :: Map Name (Map PI Polarity) → Map Name (Map PI Polarity)
inComplete = Map.filterWithKey h
  where
    h name sm = Map.size sm ≢ arityMapMap. ! name
  -- The following material is heuristic in nature, for filling in gaps in the
  -- polarity information that are not covered by addSubRule.
  -- singleGapAllNeg nL sm returns Just pi if pi is the only port index of nL
  -- without polarity in sm, and all polarities in sm are Neg.
singleGapAllNeg :: Name → Map PI Polarity → Maybe (Name, PI)
singleGapAllNeg name sm = if succ (Map.size sm) ≢ arity
    then Nothing
    else  -- exactly one gap
      h 0 $ Map.toAscList sm  -- find that gap
  where
    arity = arityMapMap. ! name
    h i [ ] = Nothing  -- don’t consider zero-ary agents to avoid terminators.
    h i ((i′, pol) : ps) = case pol of
      Pos → Nothing
      Neg → if i ≡ i′
```

```
              then h′ (succ i) ps
              else if all (Neg ≡) $ map snd ps
                   then Just (name, i)
                   else Nothing   -- not all negative
        h′ i [] = Just (name, i)   -- no earlier gap found
        h′ i ps = h i ps
  add i m = let m′ = foldr addSubRule m subrules
    in if m ≢ m′
      then add (succ i) m′
      else let
             mIncomplete = inComplete m
             mIncCount = Map.size mIncomplete
          in   -- trace (unlines
               -- (("\nextractPolarity: " ++ shows i " iterations.\n")
               -- :map show (Map.toList m)
               -- ++[unwords [show (Map.size m), "entries;"
               -- , show mIncCount, "incomplete;"
               -- , show arityCount, "needed."]]
               -- ))$
             if Map.size m ≡ arityCount   -- ∧ mIncCount ≡ 0
               then m
               else case Map.foldrWithKey (λn sm r → singleGapAllNeg n sm ‘mplus‘ r) Nothing m of
                 Just (name, pi) →    -- trace (unwords ["singleGapAllNeg:", name, show pi, "-> Pos"])$
                   add 0 $ mmInsert "singleGapAllNeg" name pi Pos m
                 Nothing → error $ unlines $
                   "Don’t know how to complete polarity.\n"
                    : map show (Map.toList arityMap)
                    ++ [unwords [show (Map.size m), "entries;"
                      , show mIncCount, "incomplete;"
                      , show arityCount, "needed."]]
  in Map.map vectorFromMap $ add 1 pports
```

## 4.4 INet.InetsFile.Interpret — Interpreter for .inet Files

```
module INet.InetsFile.Interpret where

import INet.InetsFile.Abstract
import INet.InetsFile.Parser
import INet.InetsFile.ToDescription

import INet.Description
import INet.Description.Check (checkNetDescription)
import INet.PTerm

import INet.Polar.INet
import INet.Polar.Create
import INet.Polar.Read

import INet.Utils.Usage
import qualified INet.Utils.Vector as V

import Data.Maybe (mapMaybe)
import Data.Map (Map)
import qualified Data.Map as Map

import Control.Monad.State

import System.Environment (getArgs)
import System.IO
import System.Exit
import System.FilePath
import System.Directory
```

```
runInets = do
  args ← getArgs
  case args of
    [s] → runFile s
    _ → hPutStrLn stderr $ usage "RunInets" "<file>.inet"
```

```
locateImport :: FilePath → FilePath → IO (Maybe FilePath)
locateImport baseFile importFile = do
    b ← doesFileExist importFile
    if b
      then return (Just importFile)
      else do
        cwd ← getCurrentDirectory
        findFile [cwd, baseDir, baseDir2] importFile
  where
    baseDir = takeDirectory baseFile
    baseDir2 = takeDirectory baseDir
```

```
readInetsFile :: FilePath → IO CompilationUnit
readInetsFile path = do
  e ← parseInetsFile path
  case e of
    Left err → do hPutStrLn stderr $ "RunInets: Parse error: " ++ show err
                  exitFailure
    Right cu → return cu
```

```
expandImportsCU :: FilePath → CompilationUnit → IO CompilationUnit
expandImportsCU baseFile (Left ms) = liftM Left $ mapM (expandImportsM baseFile) ms
expandImportsCU baseFile (Right mcs) = liftM Right $ expandImportsMCs baseFile mcs

expandImportsMCs :: FilePath → [ModuleComponent] → IO [ModuleComponent]
expandImportsMCs baseFile = liftM concat ∘ mapM (expandImportsMC baseFile)

expandImportsM :: FilePath → Module → IO Module
expandImportsM baseFile m = do
  mcs′ ← expandImportsMCs baseFile $ moduleComponents m
  return $ m {moduleComponents = mcs′}
```

We implement naïve and restricted import chasing: Besides from the current directory, we also start from the directory containing the importing file, and its parent directory (if the importing file had a more-than-two-component file path). This is sufficient for the examples in the Inets repository as of r65 (2012-03-20, still current on 2015-02-10).

```
expandImportsMC :: FilePath → ModuleComponent → IO [ModuleComponent]
expandImportsMC baseFile (MCImportStmt ss) = do
  let importPath0 = joinPath ss < . > "inet"
  importPathM ← locateImport baseFile importPath0
  case importPathM of
    Nothing → do
      hPutStrLn stderr $ unwords [baseFile ++ ": Import", importPath0, "not found"]
      exitFailure
    Just importPath → do
      cu ← readInetsFile importPath
      case cu of
        Left ms → do
          ms′ ← mapM (expandImportsM baseFile) ms
          return $ concatMap moduleComponents ms   -- have to flatten
        Right mcs → expandImportsMCs baseFile mcs
expandImportsMC baseFile mc = return [mc]
```

```
runFile path = do
  cu ← readInetsFile path
  expandImportsCU path cu ≫= runCompilationUnit
```

```
filterCU :: CompilationUnit → [Either Net [Term]]
filterCU (Left ms) = filterMs ms
filterCU (Right mcs) = filterMCs mcs
filterMCs :: [ModuleComponent] → [Either Net [Term]]
filterMCs (MCNet net : mcs) = Left net : filterMCs mcs
filterMCs (MCStatements stmts : mcs) = map Right (mapMaybe filterStmt stmts) ++ filterMCs mcs
filterMCs (_ : mcs) = filterMCs mcs
filterMCs [] = []
filterMs = concatMap (filterMCs ∘ moduleComponents)
filterStmt :: StatementOrDec → Maybe [Term]
  -- only PrintNet is interesting:
filterStmt (PrintNet ts) = Just ts
filterStmt _ = Nothing
```

```
filterNamedNetBody :: [Either StatementOrDec [NetDef]] → [Either Net [Term]]
filterNamedNetBody (Left stmt : es) = case filterStmt stmt of
  Nothing → filterNamedNetBody es
  Just ts → Right ts : filterNamedNetBody es
filterNamedNetBody (Right nds : es) = Left (UnNamedNet nds) : filterNamedNetBody es
filterNamedNetBody [] = []
```

```
filterNetDefs :: [NetDef] → Either String [Equation]
filterNetDefs (Left eq : nds) = fmap (eq:) $ filterNetDefs nds
filterNetDefs (Right (NetInst name pas) : nds) = Left $
  "Net instances not handled yet --- found: " ++ name ++ show pas
filterNetDefs [] = Right []
```

```
data InterpState = InterpState
  {iLang :: INetLang (NLab Value)
  , iSplitAttribsMap :: SplitAttribsMap
  , iPortVars :: Map Name (Port (NLab Value))
  , iNamedNets :: Map Name (PTerm (NLab Value) Name)
  }
modifyPortVars f = modify (λs → s {iPortVars = f $ iPortVars s})
```

```
evalExpr :: String → Expression → Value
evalExpr msg e = case evalExpression val e of
    Left err → error $ unwords [msg, show e, ":", err]
    Right v → v
  where
    val :: Map Name Value
    val = Map.empty   -- we are ignoring any global declarations for now.
```

```
interpret :: [Either Net [Term]] → StateT InterpState IO ()
interpret [] = return ()
interpret (Left (NamedNet name@"main" [] body) : qs) = do
  lift $ putStrLn $ "Entering " ++ name
  interpret $ filterNamedNetBody body
  lift $ putStrLn $ "Leaving " ++ name
  interpret qs
interpret (Left (NamedNet name [] [Left (Return (Just t))]) : qs) = do
  splitAttribsMap ← liftM iSplitAttribsMap get
  let pt :: PTerm (NLab Value) Name
```

```
      pt = mapPTermNLab (fmap (evalExpr $ "interpret: NamedNet " ⧺ name ⧺ ":"))
               $ toPTerm ("NamedNet: " ⧺ name ⧺ " = " ⧺ show t) splitAttribsMap t
  modify (λs → s {iNamedNets = Map.insert name pt $ iNamedNets s})
  interpret qs
interpret (Left (NamedNet name pas body) : _) = do    -- [ WK: See ArithExpression.inet. ]
  lift $ do   hPutStrLn stderr $ "General named nets not handled yet --- found: " ⧺ name ⧺ show pas
              exitFailure
interpret (Left (UnNamedNet nds) : qs) = case filterNetDefs nds of
  Left err → lift $ hPutStrLn stderr err
  Right eqs → do
    lang ← liftM iLang get
    splitAttribsMap ← liftM iSplitAttribsMap get
    namedNets ← liftM iNamedNets get
    let ptEqs0 = map (toPTermEq (show eqs) splitAttribsMap) eqs
        ptEqs1 = map (mapPTermEqNLab (fmap $ evalExpr "interpret: UnNamedNet")) ptEqs0
        ptEqs = closedSubstPTermEqs newStringVar namedNets ptEqs1
      -- lift $ putStrLn $ unlines $ "UnNamedNet equations:" : map (("   "⧺) ∘ show) ptEqs
    case buildNet ptEqs of
      (sourceVars, ndV) → case checkNetDescription ndV of
        [ ] → do
            -- lift $ putStrLn "No clashes"
            -- lift $ putStrLn $ show sourceVars
            -- lift $ putStrLn $ show ndV
          (src, _trg) ← lift $ createNet lang ndV
          modifyPortVars (foldr ($) `flip` zipWith Map.insert sourceVars (V.toList src))
          interpret qs
        clashes → lift $ do
          putStrLn $ unlines $
            "\ninterpret: Inconsistent net description:"
             : show sourceVars
             : show ndV : "Clashes:" : map (("   "⧺) ∘ show) clashes
          exitFailure
interpret (Right ts : qs) = do    -- this is "printNet ts"
    lang ← liftM iLang get
    splitAttribsMap ← liftM iSplitAttribsMap get
    portVars ← liftM iPortVars get
    let pts = map (toPTerm "printNet" splitAttribsMap) ts
        printPTerm (ConnVar v) = do
          putStr $ "   " ⧺ v ⧺ " --> "
          case Map.lookup v portVars of
            Nothing → putStrLn "undefined!"
            Just p → do
              r ← getPTerm p
              putStrLn $ showsPTerm shows (error "showsVar") r ""
        printPTerm pt = putStrLn $ "printNet: variable expected: " ⧺ show pt
    lift $ mapM_ printPTerm pts
    interpret qs



runCompilationUnit :: CompilationUnit → IO ()
runCompilationUnit cu = let
  ((symbLangMap, polMap), (splitAttribsMap, lang)) = cuINetLang cu
  in evalStateT (interpret $ filterCU cu)
                (InterpState lang splitAttribsMap Map.empty Map.empty)
```

## 4.5 INet.InetsFile.MkExamples — Create Examples

```
module INet.InetsFile.MkExamples where
  -- import INet.PTerm
import System.Random
import System.IO
```

Generating alternative start lists for sort ∘ inet:

```
randomIntList :: Int → Int → IO [Int]
randomIntList max len = mapM (const $ randomRIO (0, max)) [1 .. len]
randomSortStart :: Int → Int → IO ()
randomSortStart max len =
  randomIntList max len ≫= putStrLn ∘ listExpr "Nil"
listExpr :: String → [Int] → String
listExpr arg ks = foldr (λi t → "Num(" ++ show i ++ ", " ++ t ++ ")") arg ks
chunk :: Int → [a] → [[a]]
chunk k [] = []
chunk k xs = case splitAt k xs of
  (ys, zs) → ys : chunk k zs
randomSortChunked :: Int → Int → IO ()
randomSortChunked max len = do
    h ← openFile outFile WriteMode
    readFile (dir ++ "sortC.inet-pre") ≫= hPutStrLn h
    list ← randomIntList max len
    hPutStr h "    "
    hPutStr h
      $ foldr1 (λxs ys → xs ++ ",\n    " ++ ys)
      $ reverse
      $ zipWith3 mkEq ("BSort(r)" : chunkVars) (chunkVars ++ ["Nil"]) (chunk chunkSize list)
    hPutStrLn h ";"
    readFile (dir ++ "sortC.inet-post") ≫= hPutStrLn h
    hClose h
  where
    dir = "../data/Inets/"
    outFile = dir ++ "sortC" ++ shows len ".inet"
    mkEq result arg is = result ++ "~" ++ listExpr arg is
    chunkSize = 10
    lastChunk = pred (len `div` chunkSize)
    chunkVars = map (("chunk"++) ∘ show) [1 .. lastChunk]
```

# Chapter 5

# Auxiliary Material

## 5.1 INet.Utils.Usage — Generating Usage Message

```
module INet.Utils.Usage where
```

```
usage :: String → String → String
usage progName argDescr = "Usage:    " ⧺ progName
   ⧺ " +RTS -N<#cores> -H<size> -M<size> -RTS " ⧺ argDescr
```

## 5.2 MVar Utilities

```
module INet.Utils.MVar where
import Control.Concurrent.MVar
```

```
moveMVar :: MVar a → MVar a → IO ()
moveMVar v1 v2 = takeMVar v1 ≫= putMVar v2
```

## 5.3 Vector Utilities

This module is the import interface for Data.Vector, so we have a single place for switching between safe and interface functions.

```
module INet.Utils.Vector
   (module INet.Utils.Vector
   , V.length
   , V.head
   , V.tail
   , V.cons
   , (V.⧺)
   , V.imap
   , V.foldr
   , V.ifoldr
   , V.zip
   , V.empty
   , V.singleton
   , V.generate
```

```
  , V.replicateM
  , V.mapM
  , V.mapM_
  , V.fromList
  , V.toList
  , V.unzip
  , V.zipWithM
  , V.zipWithM_
  , Vector ()
  ) where
import Data.Vector (Vector)
import qualified Data.Vector as V
import Control.Monad (liftM2)
```

```
(!) :: Vector a → Int → a
(!) = (V.!)
  -- (!) = V.unsafeIndex
(!?) :: Vector a → Int → Maybe a
(!?) = (V.!?)
  -- v !? i = Just (v ! i)
```

```
bounds :: Vector a → (Int, Int)
bounds v = (0, pred (V.length v))
atErr :: String → Vector a → Int → a
  -- atErr s = (V.!)
atErr s v i = case v V.!? i of
  Nothing → error $ s ⧺ show i
  Just x → x
```

```
assocs :: Vector a → [(Int, a)]
assocs = V.toList ∘ V.indexed
```

```
imapM′ :: Monad m ⇒ (Int → a → m b) → Vector a → m [b]
imapM′ f v = V.ifoldr′ (λi a m → liftM2 (:) (f i a) m) (return [ ]) v
```

(The $\eta$-expansion via v reduces allocation and time quite drastically. However, $\eta$-expanding the argument function to V.ifoldr′ does not seem to make a significant difference.)

imapM is more expensive, due to the intermediate vector.

```
imapM :: Monad m ⇒ (Int → a → m b) → Vector a → m (Vector b)
imapM f v = V.sequence (V.imap f v)
```

```
pick :: Int → Vector a → (a, Vector a)
pick i v = let
  (pre, post) = V.splitAt i v
  in (V.head post, preV. ⧺ V.tail post)
```

## 5.4 Map Utilities

```
 {-# LANGUAGE ScopedTypeVariables #-}
module INet.Utils.Map where
```

```
import Data.Map (Map)
import qualified Data.Map as Map
import Data.List (foldl′)
```

The result of `mkLookup2 triples` will raise run-time errors for arguments outside the domain defined by `triples`.

```
mkLookup2 :: forall a b c ∘ (Show a, Show b, Ord a, Ord b) ⇒ [((a, b), c)] → a → b → c
mkLookup2 triples = let
  f m0 ((a, b), c) = Map.insertWith Map.union a (Map.singleton b c) m0
  m :: Map a (Map b c)
  m = foldl′ f Map.empty triples
  inλx y → (mMap. ! x)Map. ! y
```

## 5.5 List Utilities

```
{-# LANGUAGE FlexibleContexts #-}
module INet.Utils.List where

import Control.Monad.State.Class
import Data.Function (on)
import Data.List (groupBy)
```

Since I couldn't find an appropriate queue interface, here is an auxiliary function that traverses a list only once, combining `Eq`-based index lookup with append for new elements. New elements need to be appended at the end in order to keep the index of the other elements stable. (An experiment with explicit length enabling reverse indices might be interesting.) [ **WK:** *In monadic context, this could be implemented using destructive list update. How to get a compiler to do this?* ]

```
  -- Left (gb i) for “found y satisfying f x at i”;
  -- Right (gc x i) for “inserted (appended) a at i”.
checkElem :: MonadState [a] m ⇒ (x → Int → c) → (x → a → Maybe (Int → b)) → x → a → m (Either b c)
checkElem gc f x a = state (h 0 id)
  where
    h i acc [ ] = (Right (gc x i), acc [a])
    h i acc ys@(y : ys′) = case f x y of
      Just gb → (Left (gb i), acc ys)
      Nothing → h (succ i) (acc ∘ (y:)) ys′
```

```
groupByOnFst :: (a → a → Bool) → [(a, b)] → [(a, [b])]
groupByOnFst rel = map h ∘ groupBy (rel ‘on‘ fst)
  where
    h ((a, b) : ps) = (a, b : map snd ps)
    h [ ] = error $ "IMPOSSIBLE: groupByOnFst.h []"
```

`removeDuplicates` removes *both* occurrences of all elements occurring twice. (Actually it removes all occurring of all elements occurring an even number of times.)

```
removeDuplicates :: (Eq a) ⇒ [a] → [a]
removeDuplicates [ ] = [ ]
removeDuplicates (x : xs) = if x ∈ xs then removeDuplicates $ removeOnce x xs
                                      else x : removeDuplicates xs
removeOnce :: (Eq a) ⇒ a → [a] → [a]
removeOnce x [ ] = [ ]
removeOnce x (y : ys) = if x ≡ y then ys else y : removeOnce x ys
```

## 5.6 Temp File Utilities

```
{-# LANGUAGE CPP #-}
module INet.Utils.TmpFile (writeTmp) where
#ifdef mingw32_HOST_OS
import Data.IORef
import System.IO.Unsafe
#else
import System.Posix.Temp
#endif
import System.IO


writeTmp :: String → String → IO FilePath


#ifdef mingw32_HOST_OS
{-# NOINLINE tmpx #-}
tmpx :: IORef Int
tmpx = unsafePerformIO $ newIORef 3241
writeTmp suffix s = do
  i ← atomicModifyIORef tmpx (λk → (succ k, k))
  let tmpFile = "TempFile" ++ shows i suffix
  writeFile tmpFile s
  return tmpFile
#else
writeTmp _suffix s = do
  (tmpFile, h) ← mkstemp "/tmp/Data.Rel.Utils.TmpFile.XXXXXX"
  hPutStr h s
  hClose h
  return tmpFile
#endif
```

## 5.7 Calling GhostView

```
{-# LANGUAGE CPP #-}
module INet.Utils.GhostView where
import INet.Utils.TmpFile
import System.Process (system)


ghostViewString :: String → IO ()
ghostViewString s = do
  tmpFile ← writeTmp ".eps" s
  ghostViewFile tmpFile


ghostViewFile :: FilePath → IO ()
ghostViewFile file = do
#ifdef mingw32_HOST_OS
  -- This is for MS Windows:
  --
  -- If you do not have gsview32 in you PATH, then
  -- comment the first “system” line,
  -- uncomment the second "system" line,
  -- and edit the hard-coded path to fit your installation.
  system $ unwords ["gsview32", file]
    -- system $ unwords ["\"J:\\Program Files\\Ghostgum\\gsview\\gsview32.exe\"", file]
# elif darwin_HOST_OS
  -- This is for MacOS X:
  let eps = file ⧺ ".eps"
  -- system $ unwords ["mv", file, eps, ";", "open -a /Applications/Preview.app", eps, "&"]
  system $ unwords ["mv", file, eps, ";", "open", eps, "&"]
# else
  -- This is the default, and should work on most UNIX-like systems
  -- (including Linux) with gv installed.
  -- http://www.gnu.org/software/gv/
  system $ unwords ["gv", file, "&"]
# endif
  return ()
```

## 5.8 Dot Graphs

```
module INet.Utils.Dot where
import INet.Utils.GhostView (ghostViewFile)
import INet.Utils.TmpFile
import System.Process (system)
import System.FilePath (addExtension, splitExtension)
```

```
type Attr = (String, String)
escape :: Char → ShowS
escape '\n' = ('\\':) ∘ ('n':)
escape '\t' = ('\\':) ∘ ('t':)
escape '\r' = ('\\':) ∘ ('r':)
escape '\b' = ('\\':) ∘ ('b':)
escape '\f' = ('\\':) ∘ ('f':)
escape '\\' = ('\\':) ∘ ('\\':)
escape '"' = ('\\':) ∘ ('"':)
escape c    = (c:)
showsString s = ('"':) ∘ flip (foldr escape) s ∘ ('"':)
showsAttr (name, value) = (name⧺) ∘ ('=':) ∘ showsString value
showsAttrs [] = id
showsAttrs attrs = (' ':) ∘ ('[':) ∘
  sepShowsList (',':) showsAttr attrs ∘
  (']':) ∘ (';':)
type Attrs = [Attr]
```

```
data Stmt = Setting Attr
   | NodeSettings Attrs
   | EdgeSettings Attrs
   | Node String Attrs
   | Edge String String Attrs
   | UndirEdge String String Attrs
   | Edges [String] Attrs
```

```
  | Subgraph String [Stmt]
data DotGraphKind = Graph | Digraph deriving (Eq, Ord)
instance Show DotGraphKind where
  show Graph = "graph"
  show Digraph = "digraph"
data DotGraph = DotGraph DotGraphKind String [Stmt]


instance Show Stmt where
  showsPrec _ = showsStmt (' ':)
  showList = showsStmts (' ':)
showsStmt = showsStmt0 False
showsNeatoStmt = showsStmt0 True
showsStmt0 undir indent = f where
  f (Setting attr) = showsAttr attr ∘ (';':)
  f (NodeSettings attrs) = ("node"++) ∘ showsAttrs attrs
  f (EdgeSettings attrs) = ("edge"++) ∘ showsAttrs attrs
  f (Node name attrs) = showsString name ∘ showsAttrs attrs
  f (UndirEdge src trg attrs) =
    if undir then showsEdge undir src trg attrs else id
  f (Edge src trg attrs) = showsEdge undir src trg attrs
  f (Edges ns attrs) = sepShowsList (arrow undir) showsString ns ∘ showsAttrs attrs
  f (Subgraph name stmts) =
    ("subgraph "++) ∘ (name++) ∘ (" {\n"++) ∘
    showsStmts0 undir ((' ':) ∘ indent) stmts ∘
    indent ∘ ('}':)
noconstraint = ("constraint", "false")
showsEdge undir src trg attrs =
  ((++) src) ∘ arrow undir ∘ ((++) trg) ∘ showsAttrs attrs
  -- showsString src . arrow undir . showsString trg . showsAttrs attrs
arrow False = (" -> "++)
arrow True = (" -- "++)
showsStmts = showsStmts0 False
showsNeatoStmts = showsStmts0 True
showsStmts0 undir indent [] = id
showsStmts0 undir indent (stmt : stmts) =
  indent ∘ showsStmt0 undir indent stmt ∘ ('\n':) ∘
  showsStmts0 undir indent stmts


showsDotGraph (DotGraph kind name stmts) =
  shows kind ∘ (' ':) ∘ (name++) ∘ (" {\n"++) ∘
  (case kind of
    Graph → showsNeatoStmts
    Digraph → showsStmts
    ) (' ':) stmts ∘ ('}':) ∘ ('\n':)
instance Show DotGraph where
  showsPrec _ = showsDotGraph
  showList [] = id
  showList (g : gs) = shows g ∘ showList gs


sepShowsList sep shows [] = id
sepShowsList sep shows [x] = shows x
sepShowsList sep shows (x : xs) = shows x ∘ sep ∘ sepShowsList sep shows xs
```

### 5.8.1 Class HasDot

```
class HasDot a where
  dotGraph :: String → a → DotGraph

dotGraphAddSettings :: [Attr] → DotGraph → DotGraph
dotGraphAddSettings ss (DotGraph k n ss′) = DotGraph k n (map Setting ss ++ ss′)

dotGraphWithSetting :: HasDot a ⇒ [Attr] → String → a → DotGraph
dotGraphWithSetting ss name = dotGraphAddSettings ss ∘ dotGraph name

dotString :: HasDot a ⇒ String → a → String
dotString name = show ∘ dotGraph name

dotToFile :: HasDot a ⇒ FilePath → String → a → IO ()
dotToFile = dotToFileWithSetting [ ]

dotToFileWithSetting :: HasDot a ⇒ [Attr] → FilePath → String → a → IO ()
dotToFileWithSetting ss f name = writeFile f ∘ show ∘ dotGraphWithSetting ss name

dot :: HasDot a ⇒ a → IO ()
dot = showDot ∘ dotString "RATHRel"


dotFileFormat :: String → String → FilePath → IO ()
dotFileFormat format suffixOut fileIn = let
    (basename, suffixIn) = splitExtension fileIn
    fileOut = (if suffixIn ≡ ".dot" then basename else fileIn)
      ‘addExtension‘ suffixOut
  in do
    system $ unwords ["dot -T" ++ format, "-o", fileOut, fileIn]
    return ()

dotFilePS :: FilePath → IO ()
dotFilePS = dotFileFormat "ps" ".eps"

dotGraphPS :: [Attr] → DotGraph → IO ()
dotGraphPS ss dg@(DotGraph kind name ss′) = let fileDot = name ++ ".dot"
  in do
    writeFile fileDot ∘ show $ dotGraphAddSettings ss dg
    dotFilePS fileDot


showDot :: String → IO ()
showDot s = do
  tmpFile ← writeTmp ".dot" s
  let (basename, suffix) = splitExtension tmpFile
    psfile = (if suffix ≡ ".dot" then basename else tmpFile)
              ‘addExtension‘ ".eps"
  system $ unwords ["dot -Tps ", tmpFile, ">", psfile]
  ghostViewFile psfile
```

### 5.8.2 Generating Dot Graphs from Relations

For simple generation of dot graphs from relations, we assume the node labels to be given as a list of strings, and the edges as index pairs. We also insert some useful default settings.

```
dotOfSepPairs :: DotGraphKind → String → [String] → [Int] → [(Int, Int)] → [(Int, Int)] → DotGraph
dotOfSepPairs kind name labels loops syms arrows =
    DotGraph kind name ∘ (settings++) ∘ (nodes++) $ edges
  where
    mkSymEdge (x, y) = Edge (show x) (show y) [("dir", "both")]
    symEdges = map mkSymEdge syms
```

```
        mkLoop x = Edge (show x) (show x) [("dir", "none"), ("tailclip", "false"), ("headclip", "false ")]
        loopEdges = map mkLoop loops
        mkEdge (x, y) = Edge (show x) (show y) []
        mkNode i n = Node (show i) [("label", n)]
        nodes = zipWith mkNode [0 ..] labels
        edges = loopEdges ++ symEdges ++ map mkEdge arrows
  settings =
    NodeSettings
      [("shape", "plaintext"), ("height", "0"), ("width", "0")
      , ("fontsize", "20")
      ]
    : map Setting [("nodesep", "0.1")
      , ("nslimit", "100"), ("mclimit", "100")]   -- make dot work harder
```

The first three attributes produce outline-free nodes with as little free space around them as possible. The choice of font size, with otherwise standard settings, makes arrows reasonably thin and short (relative to the nodes).

Since the generated dot file can be edited and dot run again, and dot settings can also be supplied on the dot command-line, the lack of possibility to influence the settings chosen here should not be a big problem.

# Bibliography

Richard Banach and George A. Papadopoulos. A study of two graph rewriting formalisms: Interaction nets and MONSTR. *Journal of Programming Languages*, 5:210–231, 1997.

Levent Erkök and John Launchbury. A recursive do for Haskell. In Manuel Chakravarty, editor, *Proc. Haskell Workshop 2002*, pages 29–37. ACM Press, 2002. ISBN 1-58113-605-6. doi: 10.1145/581690.581693.

Abubakar Hassan and Eugen Jiresch. Interaction nets programming language. `https://gna.org/projects/inets/`, `https://gna.org/svn/?group=inets`, last accessed 2015-02-06, March 2012. (Source code for the "Inets" system.).

Abubakar Hassan, Ian Mackie, and Shinya Sato. Compilation of interaction nets. *ENTCS*, 253(4):73–90, 2009. doi: 10.1016/j.entcs.2009.10.018. Proc. TERMGRAPH 2009.

Abubakar Hassan, Ian Mackie, and Shinya Sato. A lightweight abstract machine for interaction nets. In Jochen Küster and Emilio Tuosto, editors, *Proc. GT-VMT 2010*, volume 29 of *ECEASST*, pages 9.1–9.12, August 2010.

Yves Lafont. Interaction nets. In *17th POPL*, pages 95–108, New York, NY, USA, January 1990. ACM. doi: 10.1145/96709.96718.

Ian Mackie. Efficient λ-evaluation with interaction nets. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, RTA 2004*, volume 3091 of *LNCS*, pages 155–169. Springer, 2004. ISBN 978-3-540-22153-1. doi: 10.1007/978-3-540-25979-4_11.

Ian Mackie. Towards a programming language for interaction nets. *ENTCS*, 127(5):133–151, 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.02.015. Proc. TERMGRAPH 2004.