```
{-# LANGUAGE CPP, RankNTypes, ScopedTypeVariables, GADTs, TypeFamilies, MultiPar
amTypeClasses #-}
#if __GLASGOW_HASKELL__ >= 709
{-# LANGUAGE Safe #-}
#elif __GLASGOW_HASKELL__ >= 701
{-# LANGUAGE Trustworthy #-}
#endif
#if __GLASGOW_HASKELL__ >= 703
{-# OPTIONS_GHC -fprof-auto #-}
#endif
#if __GLASGOW_HASKELL__ < 701
{-# OPTIONS_GHC -fno-warn-incomplete-patterns #-}
#endif

module Compiler.Hoopl.Dataflow
  ( DataflowLattice(..), JoinFun, OldFact(..), NewFact(..), Fact, mkFactBase

  , ChangeFlag(..), changeIf

  , FwdPass(..)
  , FwdTransfer(..), mkFTransfer, mkFTransfer3
  , FwdRewrite(..),  mkFRewrite,  mkFRewrite3, noFwdRewrite
  , wrapFR, wrapFR2

  , BwdPass(..)
  , BwdTransfer(..), mkBTransfer, mkBTransfer3
  , wrapBR, wrapBR2
  , BwdRewrite(..),  mkBRewrite,  mkBRewrite3, noBwdRewrite

  , analyzeAndRewriteFwd,  analyzeAndRewriteBwd

  -- * Respecting Fuel

  -- $fuel
  )
where

import Compiler.Hoopl.Block
import Compiler.Hoopl.Collections
import Compiler.Hoopl.Checkpoint
import Compiler.Hoopl.Fuel
import Compiler.Hoopl.Graph hiding (Graph) -- hiding so we can redefine
                                           -- and include definition in paper
import Compiler.Hoopl.Label

import Control.Monad
import Data.Maybe

------------------------------------------------------------------------------
--               DataflowLattice
------------------------------------------------------------------------------


data DataflowLattice a = DataflowLattice
 { fact_name       :: String          -- Documentation
 , fact_bot        :: a               -- Lattice bottom element
 , fact_join       :: JoinFun a       -- Lattice join plus change flag
                                      -- (changes iff result > old fact)
 }
-- ^ A transfer function might want to use the logging flag
-- to control debugging, as in for example, it updates just one element
-- in a big finite map.  We don't want Hoopl to show the whole fact,
-- and only the transfer function knows exactly what changed.

type JoinFun a = Label -> OldFact a -> NewFact a -> (ChangeFlag, a)
  -- the label argument is for debugging purposes only
newtype OldFact a = OldFact a
newtype NewFact a = NewFact a

data ChangeFlag = NoChange | SomeChange deriving (Eq, Ord)
changeIf :: Bool -> ChangeFlag
changeIf changed = if changed then SomeChange else NoChange


-- | 'mkFactBase' creates a 'FactBase' from a list of ('Label', fact)
-- pairs.  If the same label appears more than once, the relevant facts
-- are joined.

mkFactBase :: forall f. DataflowLattice f -> [(Label, f)] -> FactBase f
mkFactBase lattice = foldl add mapEmpty
  where add :: FactBase f -> (Label, f) -> FactBase f
        add map (lbl, f) = mapInsert lbl newFact map
          where newFact = case mapLookup lbl map of
                            Nothing -> f
                            Just f' -> snd $ join lbl (OldFact f') (NewFact f)
                join = fact_join lattice


------------------------------------------------------------------------------
--               Analyze and rewrite forward: the interface
------------------------------------------------------------------------------

data FwdPass m n f
  = FwdPass { fp_lattice  :: DataflowLattice f
            , fp_transfer :: FwdTransfer n f
            , fp_rewrite  :: FwdRewrite m n f }

newtype FwdTransfer n f
  = FwdTransfer3 { getFTransfer3 ::
                     ( n C O -> f -> f
                     , n O O -> f -> f
                     , n O C -> f -> FactBase f
                     ) }

newtype FwdRewrite m n f   -- see Note [Respects Fuel]
  = FwdRewrite3 { getFRewrite3 ::
                    ( n C O -> f -> m (Maybe (Graph n C O, FwdRewrite m n f))
                    , n O O -> f -> m (Maybe (Graph n O O, FwdRewrite m n f))
                    , n O C -> f -> m (Maybe (Graph n O C, FwdRewrite m n f))
                    ) }

{-# INLINE wrapFR #-}
wrapFR :: (forall e x. (n  e x -> f  -> m  (Maybe (Graph n  e x, FwdRewrite m  n
  f )))
                   -> (n' e x -> f' -> m' (Maybe (Graph n' e x, FwdRewrite m' n
' f')))
          )
            -- ^ This argument may assume that any function passed to it
            -- respects fuel, and it must return a result that respects fuel.
       -> FwdRewrite m  n  f
       -> FwdRewrite m' n' f'      -- see Note [Respects Fuel]
wrapFR wrap (FwdRewrite3 (f, m, l)) = FwdRewrite3 (wrap f, wrap m, wrap l)

{-# INLINE wrapFR2 #-}
wrapFR2
  :: (forall e x . (n1 e x -> f1 -> m1 (Maybe (Graph n1 e x, FwdRewrite m1 n1 f1
))) ->
                    (n2 e x -> f2 -> m2 (Maybe (Graph n2 e x, FwdRewrite m2 n2 f2
))) ->
                    (n3 e x -> f3 -> m3 (Maybe (Graph n3 e x, FwdRewrite m3 n3 f3
)))
     )
            -- ^ This argument may assume that any function passed to it
            -- respects fuel, and it must return a result that respects fuel.
  -> FwdRewrite m1 n1 f1
  -> FwdRewrite m2 n2 f2
  -> FwdRewrite m3 n3 f3      -- see Note [Respects Fuel]
wrapFR2 wrap2 (FwdRewrite3 (f1, m1, l1)) (FwdRewrite3 (f2, m2, l2)) =
    FwdRewrite3 (wrap2 f1 f2, wrap2 m1 m2, wrap2 l1 l2)


mkFTransfer3 :: (n C O -> f -> f)
             -> (n O O -> f -> f)
             -> (n O C -> f -> FactBase f)
             -> FwdTransfer n f
mkFTransfer3 f m l = FwdTransfer3 (f, m, l)

mkFTransfer :: (forall e x . n e x -> f -> Fact x f) -> FwdTransfer n f
mkFTransfer f = FwdTransfer3 (f, f, f)

-- | Functions passed to 'mkFRewrite3' should not be aware of the fuel supply.
-- The result returned by 'mkFRewrite3' respects fuel.
mkFRewrite3 :: forall m n f. FuelMonad m
            => (n C O -> f -> m (Maybe (Graph n C O)))
            -> (n O O -> f -> m (Maybe (Graph n O O)))
            -> (n O C -> f -> m (Maybe (Graph n O C)))
            -> FwdRewrite m n f
mkFRewrite3 f m l = FwdRewrite3 (lift f, lift m, lift l)
  where lift :: forall t t1 a. (t -> t1 -> m (Maybe a)) -> t -> t1 -> m (Maybe (
a, FwdRewrite m n f))
        lift rw node fact = liftM (liftM asRew) (withFuel =<< rw node fact)
        asRew :: forall t. t -> (t, FwdRewrite m n f)
        asRew g = (g, noFwdRewrite)

noFwdRewrite :: Monad m => FwdRewrite m n f
noFwdRewrite = FwdRewrite3 (noRewrite, noRewrite, noRewrite)

noRewrite :: Monad m => a -> b -> m (Maybe c)
noRewrite _ _ = return Nothing



-- | Functions passed to 'mkFRewrite' should not be aware of the fuel supply.
-- The result returned by 'mkFRewrite' respects fuel.
mkFRewrite :: FuelMonad m => (forall e x . n e x -> f -> m (Maybe (Graph n e x))
)
           -> FwdRewrite m n f
mkFRewrite f = mkFRewrite3 f f f


type family   Fact x f :: *
type instance Fact C f = FactBase f
type instance Fact O f = f

-- | if the graph being analyzed is open at the entry, there must
--   be no other entry point, or all goes horribly wrong...
analyzeAndRewriteFwd
   :: forall m n f e x entries. (CheckpointMonad m, NonLocal n, LabelsPtr entrie
s)
   => FwdPass m n f
   -> MaybeC e entries
   -> Graph n e x -> Fact e f
   -> m (Graph n e x, FactBase f, MaybeO x f)
analyzeAndRewriteFwd pass entries g f =
  do (rg, fout) <- arfGraph pass (fmap targetLabels entries) g f
     let (g', fb) = normalizeGraph rg
     return (g', fb, distinguishedExitFact g' fout)

distinguishedExitFact :: forall n e x f . Graph n e x -> Fact x f -> MaybeO x f
distinguishedExitFact g f = maybe g
    where maybe :: Graph n e x -> MaybeO x f
          maybe GNil       = JustO f
          maybe (GUnit {}) = JustO f
          maybe (GMany _ _ x) = case x of NothingO -> NothingO
                                          JustO _  -> JustO f

------------------------------------------------------------------
--       Forward Implementation
------------------------------------------------------------------

type Entries e = MaybeC e [Label]

arfGraph :: forall m n f e x .
            (NonLocal n, CheckpointMonad m) => FwdPass m n f ->
            Entries e -> Graph n e x -> Fact e f -> m (DG f n e x, Fact x f)
arfGraph pass@FwdPass { fp_lattice = lattice,
                        fp_transfer = transfer,
                        fp_rewrite  = rewrite } entries = graph
  where
    {- nested type synonyms would be so lovely here
    type ARF  thing = forall e x . thing e x -> f        -> m (DG f n e x, Fact
x f)
    type ARFX thing = forall e x . thing e x -> Fact e f -> m (DG f n e x, Fact
x f)
    -}
    graph ::              Graph n e x -> Fact e f -> m (DG f n e x, Fact x f)
-- @ start block.tex -2
    block :: forall e x .
             Block n e x -> f -> m (DG f n e x, Fact x f)
-- @ end block.tex
-- @ start node.tex -4
    node :: forall e x . (ShapeLifter e x)
         => n e x -> f -> m (DG f n e x, Fact x f)
-- @ end node.tex
-- @ start bodyfun.tex
    body  :: [Label] -> LabelMap (Block n C C)
          -> Fact C f -> m (DG f n C C, Fact C f)
-- @ end bodyfun.tex
                     -- Outgoing factbase is restricted to Labels *not* in
                     -- in the Body; the facts for Labels *in*
                     -- the Body are in the 'DG f n C C'
-- @ start cat.tex -2
    cat :: forall e a x f1 f2 f3.
           (f1 -> m (DG f n e a, f2))
        -> (f2 -> m (DG f n a x, f3))
        -> (f1 -> m (DG f n e x, f3))
-- @ end cat.tex

    graph GNil            = \f -> return (dgnil, f)
    graph (GUnit blk)     = block blk
    graph (GMany e bdy x) = (e `ebcat` bdy) `cat` exit x
     where
      ebcat :: MaybeO e (Block n O C) -> Body n -> Fact e f -> m (DG f n e C, Fa
ct C f)
      exit  :: MaybeO x (Block n C O)           -> Fact C f -> m (DG f n C x, Fa
ct x f)
      exit (JustO blk) = arfx block blk
      exit NothingO    = \fb -> return (dgnilC, fb)
      ebcat entry bdy = c entries entry
       where c :: MaybeC e [Label] -> MaybeO e (Block n O C)
                 -> Fact e f -> m (DG f n e C, Fact C f)
             c NothingC (JustO entry)   = block entry `cat` body (successors ent
ry) bdy
             c (JustC entries) NothingO = body entries bdy
             c _ _ = error "bogus GADT pattern match failure"
```

```
    -- Lift from nodes to blocks
-- @ start block.tex -2
    block BNil          = \f -> return (dgnil, f)
    block (BlockCO l b)   = node l 'cat' block b
    block (BlockCC l b n) = node l 'cat' block b 'cat' node n
    block (BlockOC   b n) =              block b 'cat' node n

    block (BMiddle n)  = node n
    block (BCat b1 b2) = block b1 'cat' block b2
-- @ end block.tex
    block (BSnoc h n)  = block h  'cat' node n
    block (BCons n t)  = node  n  'cat' block t

-- @ start node.tex -4
    node n f
     = do { grw <- frewrite rewrite n f
          ; case grw of
              Nothing -> return ( singletonDG f n
                                , ftransfer transfer n f )
              Just (g, rw) ->
                  let pass' = pass { fp_rewrite = rw }
                      f'    = fwdEntryFact n f
                  in  arfGraph pass' (fwdEntryLabel n) g f' }

-- @ end node.tex

    -- | Compose fact transformers and concatenate the resulting
    -- rewritten graphs.
    {-# INLINE cat #-}
-- @ start cat.tex -2
    cat ft1 ft2 f = do { (g1,f1) <- ft1 f
                       ; (g2,f2) <- ft2 f1
                       ; return (g1 'dgSplice' g2, f2) }
-- @ end cat.tex
    arfx :: forall thing x .
            NonLocal thing
         => (thing C x ->        f -> m (DG f n C x, Fact x f))
         -> (thing C x -> Fact C f -> m (DG f n C x, Fact x f))
    arfx arf thing fb =
      arf thing $ fromJust $ lookupFact (entryLabel thing) $ joinInFacts lattice
 fb
     -- joinInFacts adds debugging information


                    -- Outgoing factbase is restricted to Labels *not* in
                    -- in the Body; the facts for Labels *in*
                    -- the Body are in the 'DG f n C C'
-- @ start bodyfun.tex
    body entries blockmap init_fbase
      = fixpoint Fwd lattice do_block entries blockmap init_fbase
      where
        do_block :: forall x. Block n C x -> FactBase f
                 -> m (DG f n C x, Fact x f)
        do_block b fb = block b entryFact
          where entryFact = getFact lattice (entryLabel b) fb
-- @ end bodyfun.tex


-- Join all the incoming facts with bottom.
-- We know the results _shouldn't change_, but the transfer
-- functions might, for example, generate some debugging traces.
joinInFacts :: DataflowLattice f -> FactBase f -> FactBase f
joinInFacts (lattice @ DataflowLattice {fact_bot = bot, fact_join = fj}) fb =
  mkFactBase lattice $ map botJoin $ mapToList fb
    where botJoin (l, f) = (l, snd $ fj l (OldFact bot) (NewFact f))
```

```
forwardBlockList :: (NonLocal n, LabelsPtr entry)
                 => entry -> Body n -> [Block n C C]
-- This produces a list of blocks in order suitable for forward analysis,
-- along with the list of Labels it may depend on for facts.
forwardBlockList entries blks = postorder_dfs_from blks entries

--------------------------------------------------------------------------------
--              Backward analysis and rewriting: the interface
--------------------------------------------------------------------------------

data BwdPass m n f
  = BwdPass { bp_lattice  :: DataflowLattice f
            , bp_transfer :: BwdTransfer n f
            , bp_rewrite  :: BwdRewrite m n f }

newtype BwdTransfer n f
  = BwdTransfer3 { getBTransfer3 ::
                     ( n C O -> f          -> f
                     , n O O -> f          -> f
                     , n O C -> FactBase f -> f
                     ) }
newtype BwdRewrite m n f
  = BwdRewrite3 { getBRewrite3 ::
                    ( n C O -> f          -> m (Maybe (Graph n C O, BwdRewrite m
 n f))
                    , n O O -> f          -> m (Maybe (Graph n O O, BwdRewrite m
 n f))
                    , n O C -> FactBase f -> m (Maybe (Graph n O C, BwdRewrite m
 n f))
                    ) }

{-# INLINE wrapBR #-}
wrapBR :: (forall e x .
               Shape x
            -> (n  e x -> Fact x f  -> m  (Maybe (Graph n  e x, BwdRewrite m  n
  f )))
            -> (n' e x -> Fact x f' -> m' (Maybe (Graph n' e x, BwdRewrite m' n
' f')))
           )
           -- ^ This argument may assume that any function passed to it
           -- respects fuel, and it must return a result that respects fuel.
       -> BwdRewrite m  n  f
       -> BwdRewrite m' n' f'      -- see Note [Respects Fuel]
wrapBR wrap (BwdRewrite3 (f, m, l)) =
  BwdRewrite3 (wrap Open f, wrap Open m, wrap Closed l)

{-# INLINE wrapBR2 #-}
wrapBR2 :: (forall e x . Shape x
            -> (n1 e x -> Fact x f1 -> m1 (Maybe (Graph n1 e x, BwdRewrite m1 n1
 f1)))
            -> (n2 e x -> Fact x f2 -> m2 (Maybe (Graph n2 e x, BwdRewrite m2 n2
 f2)))
            -> (n3 e x -> Fact x f3 -> m3 (Maybe (Graph n3 e x, BwdRewrite m3 n3
 f3))))
            -- ^ This argument may assume that any function passed to it
            -- respects fuel, and it must return a result that respects fuel.
        -> BwdRewrite m1 n1 f1
        -> BwdRewrite m2 n2 f2
        -> BwdRewrite m3 n3 f3      -- see Note [Respects Fuel]
wrapBR2 wrap2 (BwdRewrite3 (f1, m1, l1)) (BwdRewrite3 (f2, m2, l2)) =
    BwdRewrite3 (wrap2 Open f1 f2, wrap2 Open m1 m2, wrap2 Closed l1 l2)



mkBTransfer3 :: (n C O -> f -> f) -> (n O O -> f -> f) ->
                (n O C -> FactBase f -> f) -> BwdTransfer n f
mkBTransfer3 f m l = BwdTransfer3 (f, m, l)

mkBTransfer :: (forall e x . n e x -> Fact x f -> f) -> BwdTransfer n f
mkBTransfer f = BwdTransfer3 (f, f, f)

-- | Functions passed to 'mkBRewrite3' should not be aware of the fuel supply.
-- The result returned by 'mkBRewrite3' respects fuel.
mkBRewrite3 :: forall m n f. FuelMonad m
            => (n C O -> f          -> m (Maybe (Graph n C O)))
            -> (n O O -> f          -> m (Maybe (Graph n O O)))
            -> (n O C -> FactBase f -> m (Maybe (Graph n O C)))
            -> BwdRewrite m n f
mkBRewrite3 f m l = BwdRewrite3 (lift f, lift m, lift l)
  where lift :: forall t t1 a. (t -> t1 -> m (Maybe a)) -> t -> t1 -> m (Maybe (
a, BwdRewrite m n f))
        lift rw node fact = liftM (liftM asRew) (withFuel =<< rw node fact)
        asRew :: t -> (t, BwdRewrite m n f)
        asRew g = (g, noBwdRewrite)

noBwdRewrite :: Monad m => BwdRewrite m n f
noBwdRewrite = BwdRewrite3 (noRewrite, noRewrite, noRewrite)

-- | Functions passed to 'mkBRewrite' should not be aware of the fuel supply.
-- The result returned by 'mkBRewrite' respects fuel.
mkBRewrite :: FuelMonad m
           => (forall e x . n e x -> Fact x f -> m (Maybe (Graph n e x)))
           -> BwdRewrite m n f
mkBRewrite f = mkBRewrite3 f f f


--------------------------------------------------------------------------------
--              Backward implementation
--------------------------------------------------------------------------------

arbGraph :: forall m n f e x .
            (NonLocal n, CheckpointMonad m) => BwdPass m n f ->
            Entries e -> Graph n e x -> Fact x f -> m (DG f n e x, Fact e f)
arbGraph pass@BwdPass { bp_lattice  = lattice,
                        bp_transfer = transfer,
                        bp_rewrite  = rewrite } entries = graph
  where
    {- nested type synonyms would be so lovely here
    type ARB  thing = forall e x . thing e x -> Fact x f -> m (DG f n e x, f)
    type ARBX thing = forall e x . thing e x -> Fact x f -> m (DG f n e x, Fact
e f)
    -}
    graph ::             Graph n e x -> Fact x f -> m (DG f n e x, Fact e f)
    block :: forall e x . Block n e x -> Fact x f -> m (DG f n e x, f)
    node  :: forall e x . (ShapeLifter e x)
                       => n e x       -> Fact x f -> m (DG f n e x, f)
    body  :: [Label] -> Body n -> Fact C f -> m (DG f n C C, Fact C f)
    cat :: forall e a x info info' info''.
           (info' -> m (DG f n e a, info''))
        -> (info  -> m (DG f n a x, info'))
        -> (info  -> m (DG f n e x, info''))

    graph GNil            = \f -> return (dgnil, f)
    graph (GUnit blk)     = block blk
    graph (GMany e bdy x) = (e 'ebcat' bdy) 'cat' exit x
     where
      ebcat :: MaybeO e (Block n O C) -> Body n -> Fact C f -> m (DG f n e C, Fa
ct e f)
      exit  :: MaybeO x (Block n C O)           -> Fact x f -> m (DG f n C x, Fa
ct C f)
      exit (JustO blk) = arbx block blk
      exit NothingO    = \fb -> return (dgnilC, fb)
      ebcat entry bdy = c entries entry
       where c :: MaybeC e [Label] -> MaybeO e (Block n O C)
                  -> Fact C f -> m (DG f n e C, Fact e f)
             c NothingC (JustO entry)   = block entry 'cat' body (successors ent
ry) bdy
             c (JustC entries) NothingO = body entries bdy
             c _ _ = error "bogus GADT pattern match failure"

    -- Lift from nodes to blocks
    block BNil          = \f -> return (dgnil, f)
    block (BlockCO l b)   = node l 'cat' block b
    block (BlockCC l b n) = node l 'cat' block b 'cat' node n
    block (BlockOC   b n) =              block b 'cat' node n

    block (BMiddle n)  = node n
    block (BCat b1 b2) = block b1 'cat' block b2
    block (BSnoc h n)  = block h  'cat' node n
    block (BCons n t)  = node  n  'cat' block t

    node n f
      = do { bwdres <- brewrite rewrite n f
           ; case bwdres of
               Nothing -> return (singletonDG entry_f n, entry_f)
                          where entry_f = btransfer transfer n f
               Just (g, rw) ->
                          do { let pass' = pass { bp_rewrite = rw }
                             ; (g, f) <- arbGraph pass' (fwdEntryLabel n) g f
                             ; return (g, bwdEntryFact lattice n f)} }

    -- | Compose fact transformers and concatenate the resulting
    -- rewritten graphs.
    {-# INLINE cat #-}
    cat ft1 ft2 f = do { (g2,f2) <- ft2 f
                       ; (g1,f1) <- ft1 f2
                       ; return (g1 'dgSplice' g2, f1) }

    arbx :: forall thing x .
            NonLocal thing
         => (thing C x -> Fact x f -> m (DG f n C x, f))
         -> (thing C x -> Fact x f -> m (DG f n C x, Fact C f))

    arbx arb thing f = do { (rg, f) <- arb thing f
                          ; let fb = joinInFacts lattice $
                                       mapSingleton (entryLabel thing) f
                          ; return (rg, fb) }
     -- joinInFacts adds debugging information

                    -- Outgoing factbase is restricted to Labels *not* in
                    -- in the Body; the facts for Labels *in*
                    -- the Body are in the 'DG f n C C'
    body entries blockmap init_fbase
      = fixpoint Bwd lattice do_block (map entryLabel (backwardBlockList entries
 blockmap)) blockmap init_fbase
      where
        do_block :: forall x. Block n C x -> Fact x f -> m (DG f n C x, LabelMap
 f)
        do_block b f = do (g, f) <- block b f
                          return (g, mapSingleton (entryLabel b) f)



backwardBlockList :: (LabelsPtr entries, NonLocal n) => entries -> Body n -> [Bl
ock n C C]
-- This produces a list of blocks in order suitable for backward analysis,
-- along with the list of Labels it may depend on for facts.
```

```
backwardBlockList entries body = reverse $ forwardBlockList entries body

{-

The forward and backward cases are not dual.  In the forward case, the
entry points are known, and one simply traverses the body blocks from
those points.  In the backward case, something is known about the exit
points, but this information is essentially useless, because we don't
actually have a dual graph (that is, one with edges reversed) to
compute with.  (Even if we did have a dual graph, it would not avail
us---a backward analysis must include reachable blocks that don't
reach the exit, as in a procedure that loops forever and has side
effects.)

-}


-- | if the graph being analyzed is open at the exit, I don't
--   quite understand the implications of possible other exits
analyzeAndRewriteBwd
   :: (CheckpointMonad m, NonLocal n, LabelsPtr entries)
   => BwdPass m n f
   -> MaybeC e entries -> Graph n e x -> Fact x f
   -> m (Graph n e x, FactBase f, MaybeO e f)
analyzeAndRewriteBwd pass entries g f =
  do (rg, fout) <- arbGraph pass (fmap targetLabels entries) g f
     let (g', fb) = normalizeGraph rg
     return (g', fb, distinguishedEntryFact g' fout)

distinguishedEntryFact :: forall n e x f . Graph n e x -> Fact e f -> MaybeO e f
distinguishedEntryFact g f = maybe g
    where maybe :: Graph n e x -> MaybeO e f
          maybe GNil      = JustO f
          maybe (GUnit {}) = JustO f
          maybe (GMany e _ _) = case e of NothingO -> NothingO
                                          JustO _  -> JustO f

--------------------------------------------------------------------------------
--      fixpoint: finding fixed points
--------------------------------------------------------------------------------

     -- See Note [TxFactBase invariants]

updateFact :: DataflowLattice f
           -> LabelMap (DBlock f n C C)
           -> Label -> f       -- out fact
           -> ([Label], FactBase f)
           -> ([Label], FactBase f)
-- See Note [TxFactBase change flag]
updateFact lat newblocks lbl new_fact (cha, fbase)
  | NoChange <- cha2, lbl `mapMember` newblocks  = (cha,     fbase)
  | otherwise         = (lbl:cha, mapInsert lbl res_fact fbase)
  where
    (cha2, res_fact) -- Note [Unreachable blocks]
       = case lookupFact lbl fbase of
           Nothing -> (SomeChange, new_fact_debug)  -- Note [Unreachable blocks]
           Just old_fact -> join old_fact
         where join old_fact =
                 fact_join lat lbl
                   (OldFact old_fact) (NewFact new_fact)
               (_, new_fact_debug) = join (fact_bot lat)


{-
-- this doesn't work because it can't be implemented
class Monad m => FixpointMonad m where
  observeChangedFactBase :: m (Maybe (FactBase f)) -> Maybe (FactBase f)
-}

-- @ start fptype.tex
data Direction = Fwd | Bwd
fixpoint :: forall m n f. (CheckpointMonad m, NonLocal n)
 => Direction
 -> DataflowLattice f
 -> (Block n C C -> Fact C f -> m (DG f n C C, Fact C f))
 -> [Label]
 -> LabelMap (Block n C C)
 -> (Fact C f -> m (DG f n C C, Fact C f))
-- @ end fptype.tex
-- @ start fpimp.tex
fixpoint direction lat do_block entries blockmap init_fbase
  = do
       -- trace ("fixpoint: " ++ show (case direction of Fwd -> True; Bwd -> Fa
lse) ++ " " ++ show (mapKeys blockmap) ++ show entries ++ " " ++ show (mapKeys i
nit_fbase)) $ return()
       (fbase, newblocks) <- loop init_fbase entries mapEmpty
       -- trace ("fixpoint DONE: " ++ show (mapKeys fbase) ++ show (mapKeys new
blocks)) $ return()
       return (GMany NothingO newblocks NothingO,
               mapDeleteList (mapKeys blockmap) fbase)
    -- The successors of the Graph are the the Labels
    -- for which we have facts and which are *not* in
    -- the blocks of the graph
  where
    -- mapping from L -> Ls.  If the fact for L changes, re-analyse Ls.
    dep_blocks :: LabelMap [Label]
    dep_blocks = mapFromListWith (++)
                       [ (l, [entryLabel b])
                       | b <- mapElems blockmap
                       , l <- case direction of
                                Fwd -> [entryLabel b]
                                Bwd -> successors b
                       ]

    loop
       :: FactBase f  -- current factbase (increases monotonically)
       -> [Label]     -- blocks still to analyse (Todo: use a better rep)
       -> LabelMap (DBlock f n C C)  -- transformed graph
       -> m (FactBase f, LabelMap (DBlock f n C C))

    loop fbase []         newblocks = return (fbase, newblocks)
    loop fbase (lbl:todo) newblocks = do
      case mapLookup lbl blockmap of
        Nothing  -> loop fbase todo newblocks
        Just blk -> do
          -- trace ("analysing: " ++ show lbl) $ return ()
          (rg, out_facts) <- do_block blk fbase
          let (changed, fbase') = mapFoldWithKey
                                    (updateFact lat newblocks)
                                    ([],fbase) out_facts
          -- trace ("fbase': " ++ show (mapKeys fbase')) $ return ()
          -- trace ("changed: " ++ show changed) $ return ()

          let to_analyse
                = filter (`notElem` todo) $
                  concatMap (\l -> mapFindWithDefault [] l dep_blocks) changed

          -- trace ("to analyse: " ++ show to_analyse) $ return ()

          let newblocks' = case rg of
                             GMany _ blks _ -> mapUnion blks newblocks

          loop fbase' (todo ++ to_analyse) newblocks'


{-  Note [TxFactBase invariants]
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
The TxFactBase is used only during a fixpoint iteration (or "sweep"),
and accumulates facts (and the transformed code) during the fixpoint
iteration.

* tfb_fbase increases monotonically, across all sweeps

* At the beginning of each sweep
      tfb_cha  = NoChange
      tfb_lbls = {}

* During each sweep we process each block in turn.  Processing a block
  is done thus:
    1.  Read from tfb_fbase the facts for its entry label (forward)
        or successors labels (backward)
    2.  Transform those facts into new facts for its successors (forward)
        or entry label (backward)
    3.  Augment tfb_fbase with that info
  We call the labels read in step (1) the "in-labels" of the sweep

* The field tfb_lbls is the set of in-labels of all blocks that have
  been processed so far this sweep, including the block that is
  currently being processed.  tfb_lbls is initialised to {}.  It is a
  subset of the Labels of the *original* (not transformed) blocks.

* The tfb_cha field is set to SomeChange iff we decide we need to
  perform another iteration of the fixpoint loop. It is initialsed to NoChange.

  Specifically, we set tfb_cha to SomeChange in step (3) iff
    (a) The fact in tfb_fbase for a block L changes
    (b) L is in tfb_lbls
  Reason: until a label enters the in-labels its accumuated fact in tfb_fbase
  has not been read, hence cannot affect the outcome

Note [Unreachable blocks]
~~~~~~~~~~~~~~~~~~~~~~~~~
A block that is not in the domain of tfb_fbase is "currently unreachable".
A currently-unreachable block is not even analyzed.  Reason: consider
constant prop and this graph, with entry point L1:
  L1: x:=3; goto L4
  L2: x:=4; goto L4
  L4: if x>3 goto L2 else goto L5
Here L2 is actually unreachable, but if we process it with bottom input fact,
we'll propagate (x=4) to L4, and nuke the otherwise-good rewriting of L4.

* If a currently-unreachable block is not analyzed, then its rewritten
  graph will not be accumulated in tfb_rg.  And that is good:
  unreachable blocks simply do not appear in the output.

* Note that clients must be careful to provide a fact (even if bottom)
  for each entry point. Otherwise useful blocks may be garbage collected.

* Note that updateFact must set the change-flag if a label goes from
  not-in-fbase to in-fbase, even if its fact is bottom.  In effect the
  real fact lattice is
       UNR
       bottom
       the points above bottom

* Even if the fact is going from UNR to bottom, we still call the
  client's fact_join function because it might give the client
  some useful debugging information.

* All of this only applies for *forward* fixpoints.  For the backward
  case we must treat every block as reachable; it might finish with a
  'return', and therefore have no successors, for example.
-}

--------------------------------------------------------------------------------
--      DG: an internal data type for 'decorated graphs'
--          TOTALLY internal to Hoopl; each block is decorated with a fact
--------------------------------------------------------------------------------

-- @ start dg.tex
type Graph = Graph' Block
type DG f  = Graph' (DBlock f)
data DBlock f n e x = DBlock f (Block n e x) -- ^ block decorated with fact
-- @ end dg.tex
instance NonLocal n => NonLocal (DBlock f n) where
  entryLabel (DBlock _ b) = entryLabel b
  successors (DBlock _ b) = successors b

--- constructors

dgnil  :: DG f n O O
dgnilC :: DG f n C C
dgSplice  :: NonLocal n => DG f n e a -> DG f n a x -> DG f n e x

---- observers

normalizeGraph :: forall n f e x .
                  NonLocal n => DG f n e x
               -> (Graph n e x, FactBase f)
                 -- A Graph together with the facts for that graph
                 -- The domains of the two maps should be identical

normalizeGraph g = (mapGraphBlocks dropFact g, facts g)
    where dropFact :: DBlock t t1 t2 t3 -> Block t1 t2 t3
          dropFact (DBlock _ b) = b
          facts :: DG f n e x -> FactBase f
          facts GNil = noFacts
          facts (GUnit _) = noFacts
          facts (GMany _ body exit) = bodyFacts body `mapUnion` exitFacts exit
          exitFacts :: MaybeO x (DBlock f n C O) -> FactBase f
          exitFacts NothingO = noFacts
          exitFacts (JustO (DBlock f b)) = mapSingleton (entryLabel b) f
          bodyFacts :: LabelMap (DBlock f n C C) -> FactBase f
          bodyFacts body = mapFoldWithKey f noFacts body
            where f :: forall t a x. (NonLocal t) => Label -> DBlock a t C x ->
LabelMap a -> LabelMap a
                  f lbl (DBlock f _) fb = mapInsert lbl f fb

--- implementation of the constructors (boring)

dgnil  = GNil
dgnilC = GMany NothingO emptyBody NothingO

dgSplice = splice fzCat
  where fzCat :: DBlock f n e O -> DBlock t n O x -> DBlock f n e x
        fzCat (DBlock f b1) (DBlock _ b2) = DBlock f (b1 `blockAppend` b2)

--------------------------------------------------------------------
--       Utilities
--------------------------------------------------------------------

-- Lifting based on shape:
--  - from nodes to blocks
--  - from facts to fact-like things
```

```
-- Lowering back:
--  - from fact-like things to facts
-- Note that the latter two functions depend only on the entry shape.
-- @ start node.tex
class ShapeLifter e x where
 singletonDG   :: f -> n e x -> DG f n e x
 fwdEntryFact  :: NonLocal n => n e x -> f -> Fact e f
 fwdEntryLabel :: NonLocal n => n e x -> MaybeC e [Label]
 ftransfer :: FwdTransfer n f -> n e x -> f -> Fact x f
 frewrite  :: FwdRewrite m n f -> n e x
           -> f -> m (Maybe (Graph n e x, FwdRewrite m n f))
-- @ end node.tex
 bwdEntryFact :: NonLocal n => DataflowLattice f -> n e x -> Fact e f -> f
 btransfer    :: BwdTransfer n f -> n e x -> Fact x f -> f
 brewrite     :: BwdRewrite m n f -> n e x
              -> Fact x f -> m (Maybe (Graph n e x, BwdRewrite m n f))

instance ShapeLifter C O where
  singletonDG f n = gUnitCO (DBlock f (BlockCO n BNil))
  fwdEntryFact     n f  = mapSingleton (entryLabel n) f
  bwdEntryFact lat n fb = getFact lat (entryLabel n) fb
  ftransfer (FwdTransfer3 (ft, _, _)) n f = ft n f
  btransfer (BwdTransfer3 (bt, _, _)) n f = bt n f
  frewrite  (FwdRewrite3  (fr, _, _)) n f = fr n f
  brewrite  (BwdRewrite3  (br, _, _)) n f = br n f
  fwdEntryLabel n = JustC [entryLabel n]

instance ShapeLifter O O where
  singletonDG f = gUnitOO . DBlock f . BMiddle
  fwdEntryFact   _ f = f
  bwdEntryFact _ _ f = f
  ftransfer (FwdTransfer3 (_, ft, _)) n f = ft n f
  btransfer (BwdTransfer3 (_, bt, _)) n f = bt n f
  frewrite  (FwdRewrite3  (_, fr, _)) n f = fr n f
  brewrite  (BwdRewrite3  (_, br, _)) n f = br n f
  fwdEntryLabel _ = NothingC

instance ShapeLifter O C where
  singletonDG f n = gUnitOC (DBlock f (BlockOC BNil n))
  fwdEntryFact   _ f = f
  bwdEntryFact _ _ f = f
  ftransfer (FwdTransfer3 (_, _, ft)) n f = ft n f
  btransfer (BwdTransfer3 (_, _, bt)) n f = bt n f
  frewrite  (FwdRewrite3  (_, _, fr)) n f = fr n f
  brewrite  (BwdRewrite3  (_, _, br)) n f = br n f
  fwdEntryLabel _ = NothingC

-- Fact lookup: the fact `orelse` bottom
getFact  :: DataflowLattice f -> Label -> FactBase f -> f
getFact lat l fb = case lookupFact l fb of Just  f -> f
                                           Nothing -> fact_bot lat



{-  Note [Respects fuel]
~~~~~~~~~~~~~~~~~~~~~~~~~
-}
-- $fuel
-- A value of type 'FwdRewrite' or 'BwdRewrite' /respects fuel/ if
-- any function contained within the value satisfies the following properties:
--
--   * When fuel is exhausted, it always returns 'Nothing'.
--
--   * When it returns @Just g rw@, it consumes /exactly/ one unit
--     of fuel, and new rewrite 'rw' also respects fuel.
--
```

```
-- Provided that functions passed to 'mkFRewrite', 'mkFRewrite3',
-- 'mkBRewrite', and 'mkBRewrite3' are not aware of the fuel supply,
-- the results respect fuel.
--
-- It is an /unchecked/ run-time error for the argument passed to 'wrapFR',
-- 'wrapFR2', 'wrapBR', or 'warpBR2' to return a function that does not respect
fuel.
```