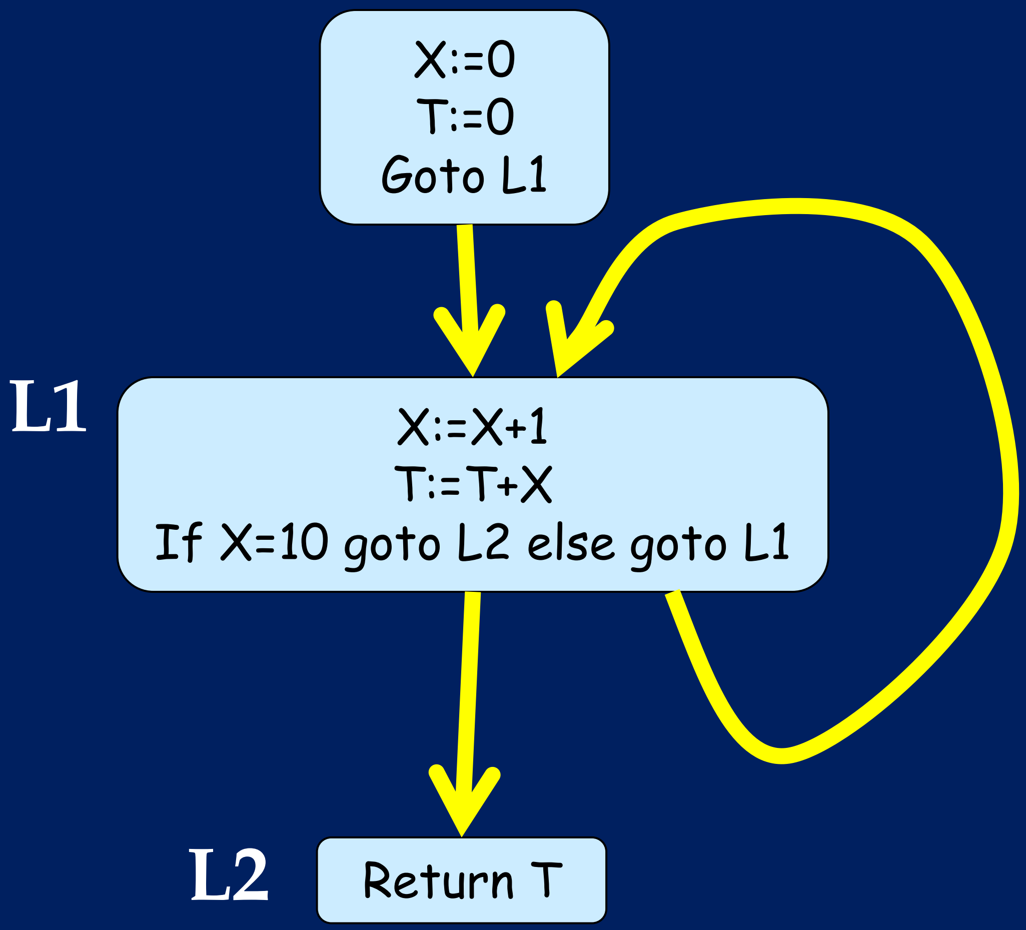
Using Type Functions in Dataflow Optimiation

Simon Peyton Jones (Microsoft Research)
Norman Ramsey, John Dias (Tufts University)

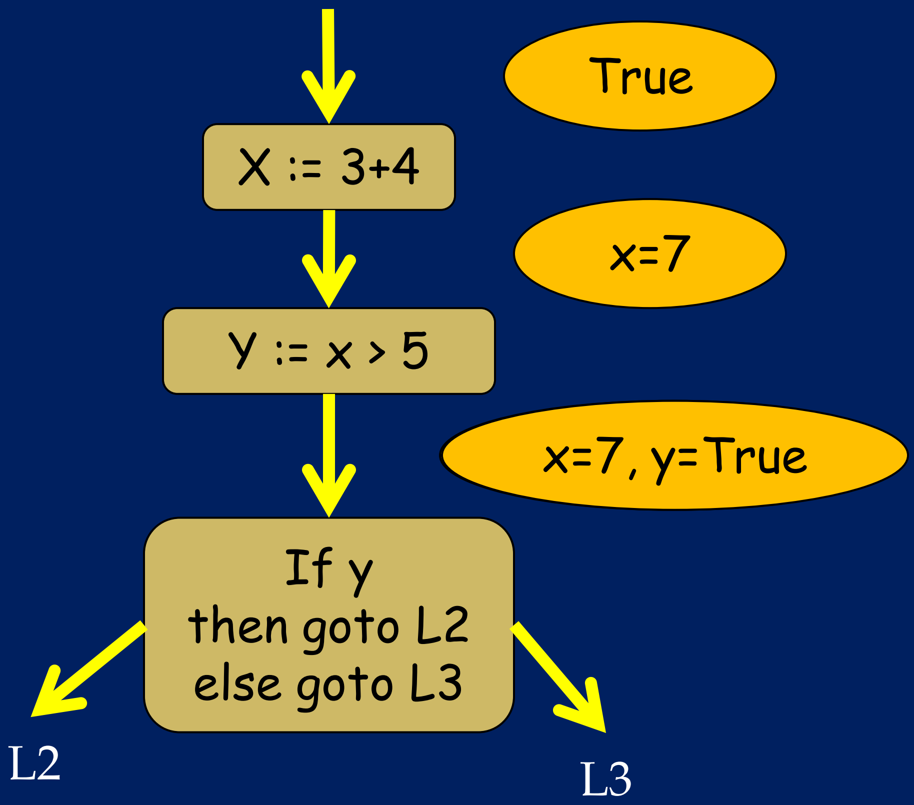
March 2010

Control flow graphs



- One entry, perhaps many exits
- Each block has a label
- Each block is a sequence of nodes
- Control transfers at end of block
- Arbitrary control flow

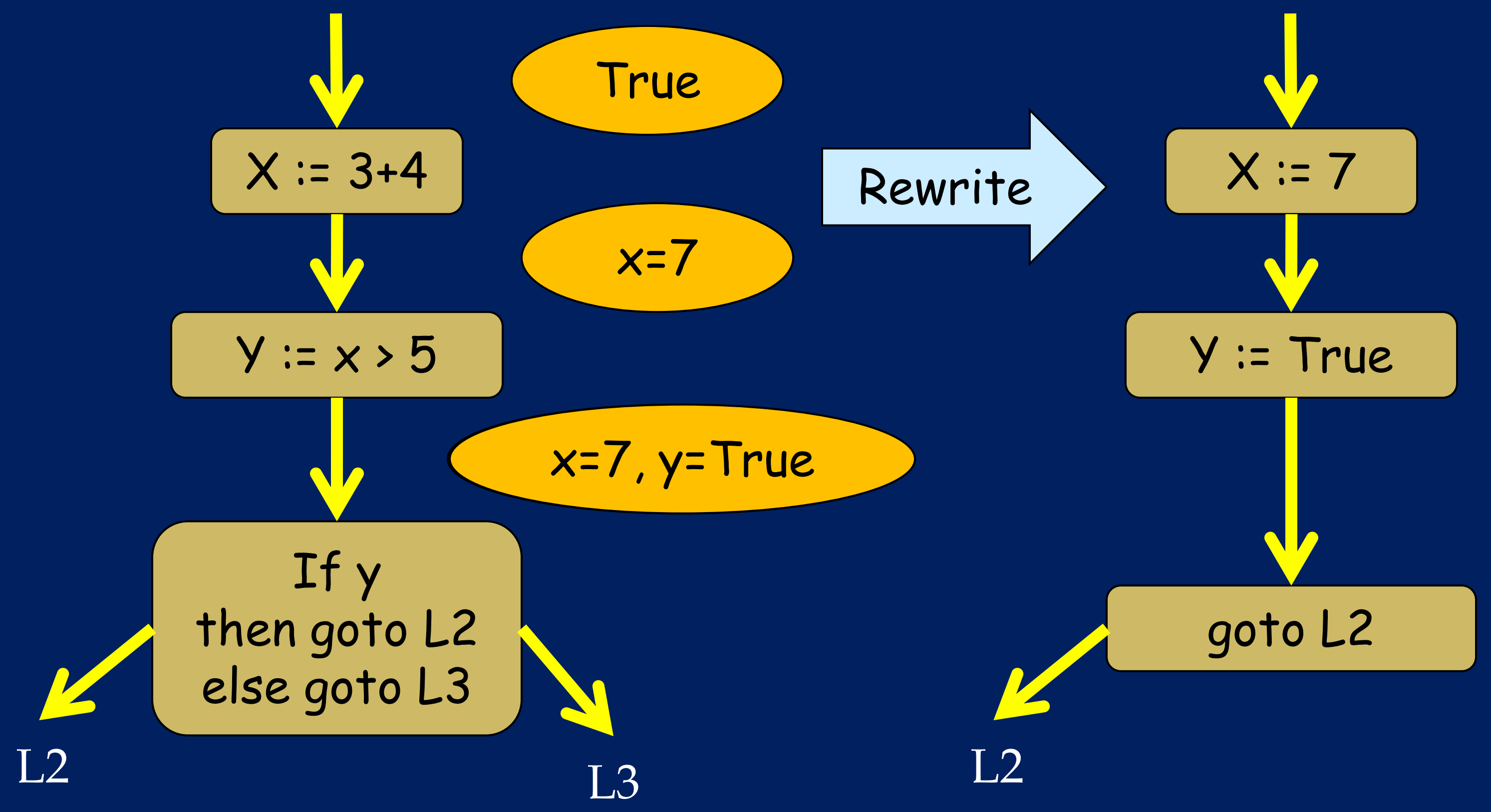
Data flow analysis



Each analysis has

- Data flow “facts”
- Transfer function for each node

Data flow transformation



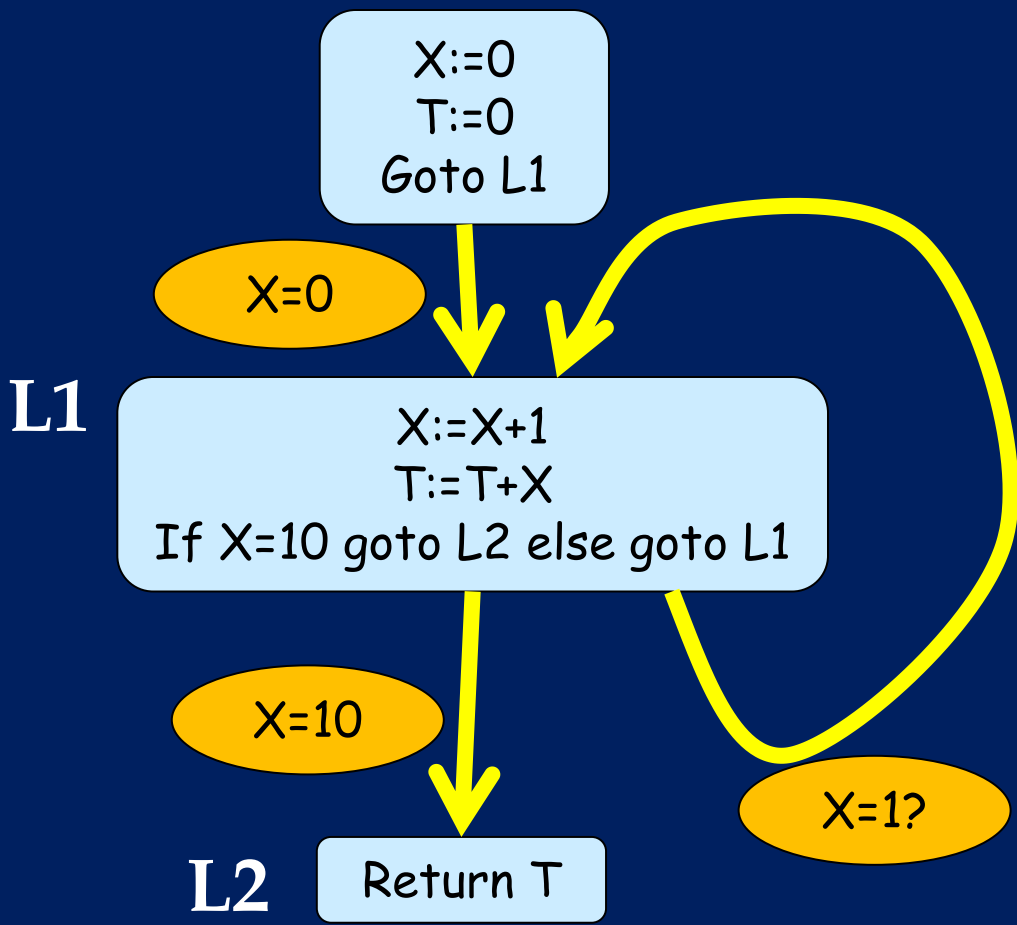
- Rewrite each node based on incoming dataflow fact
- Feed rewritten node to the transfer function

Rewriting in general

- Each rewrite takes
  - A node
  - The dataflow fact flowing to that node

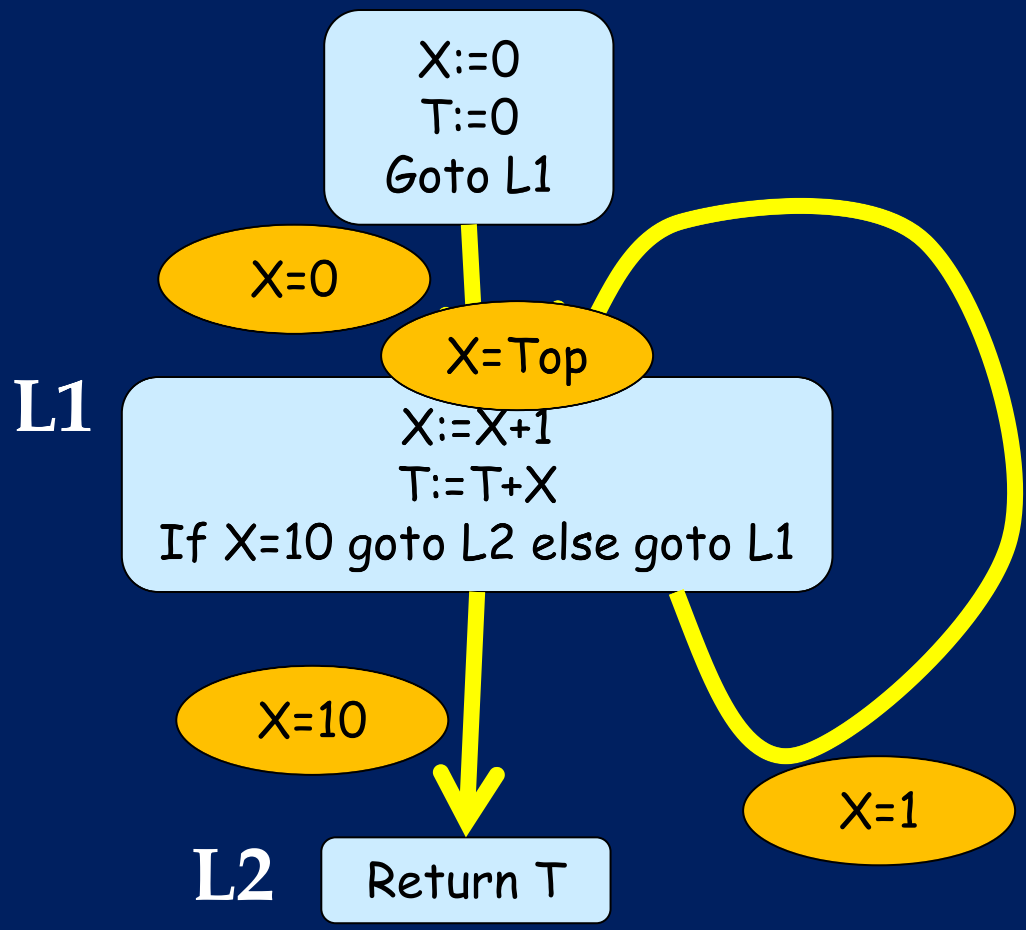
  and returns…what???
- Correct answer: an arbitrary graph!
- Examples: rewrite
  - an instruction to a no-op
  - a block-copy "instruction" to a loop
  - a switch "instruction" to a tree of conditionals
  - a call to the instantiated procedure body (inlining)

Fixpoints



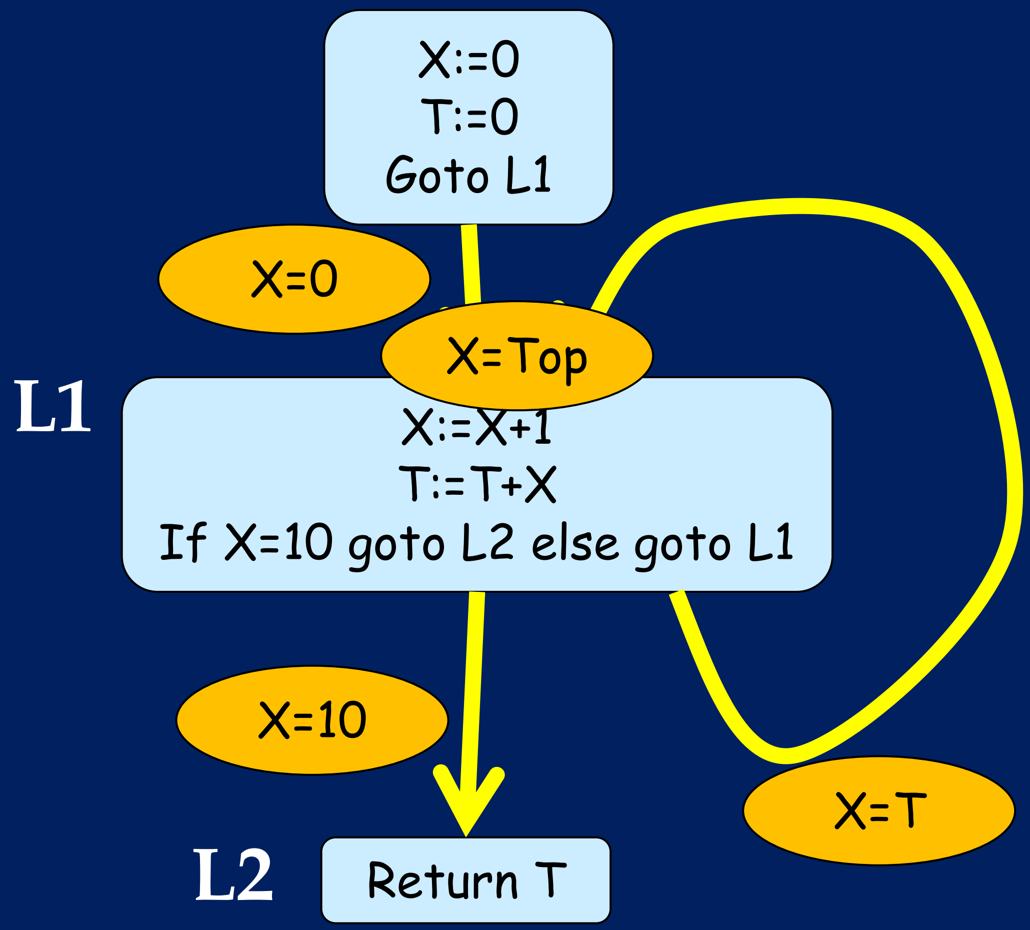
- First time round, we may have bogus information

Fixpoints



- First time round, we may have bogus information
- Combine facts flowing into a block

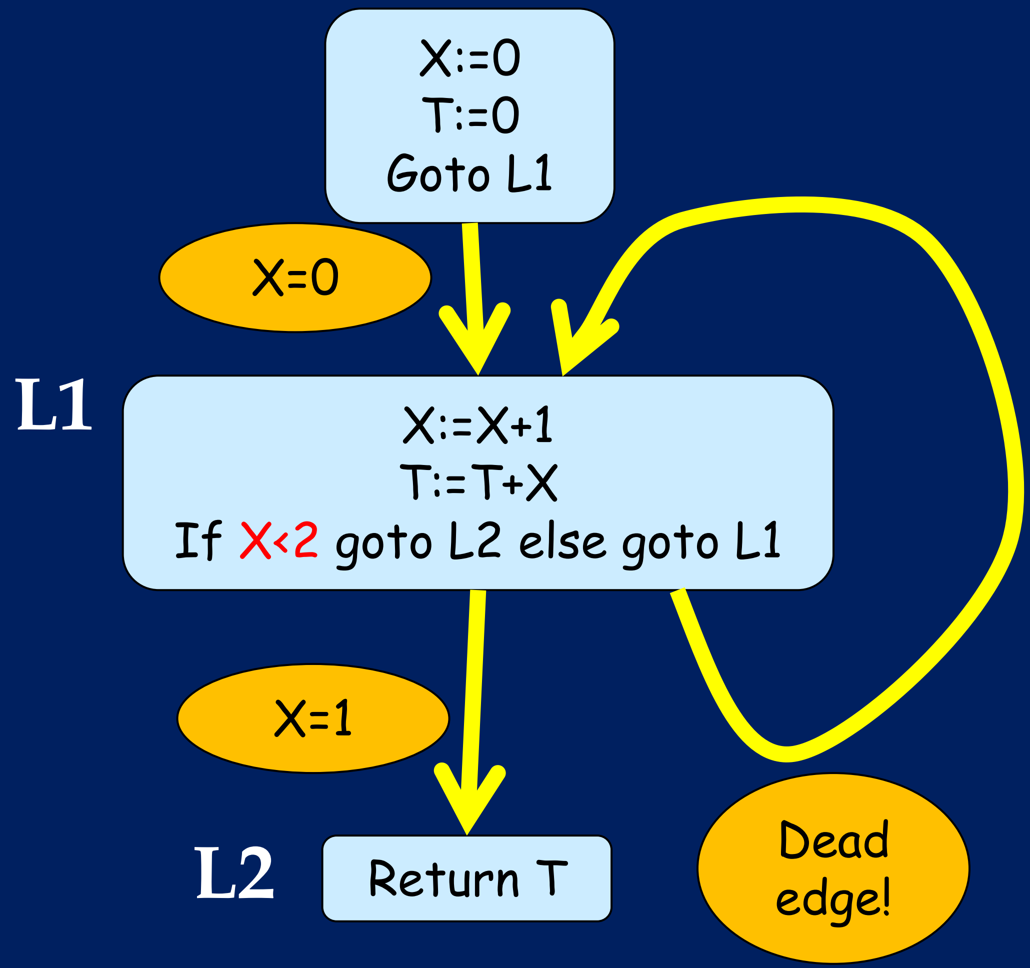
Fixpoints



- First time round, we may have bogus information
- Combine facts flowing into a block
- And iterate to fixed point

Rewrites with fixpoints

- Rewrites based on bogus (non-final) "facts" must be discarded
- But they must still be done (speculatively) in order to exploit current "fact"



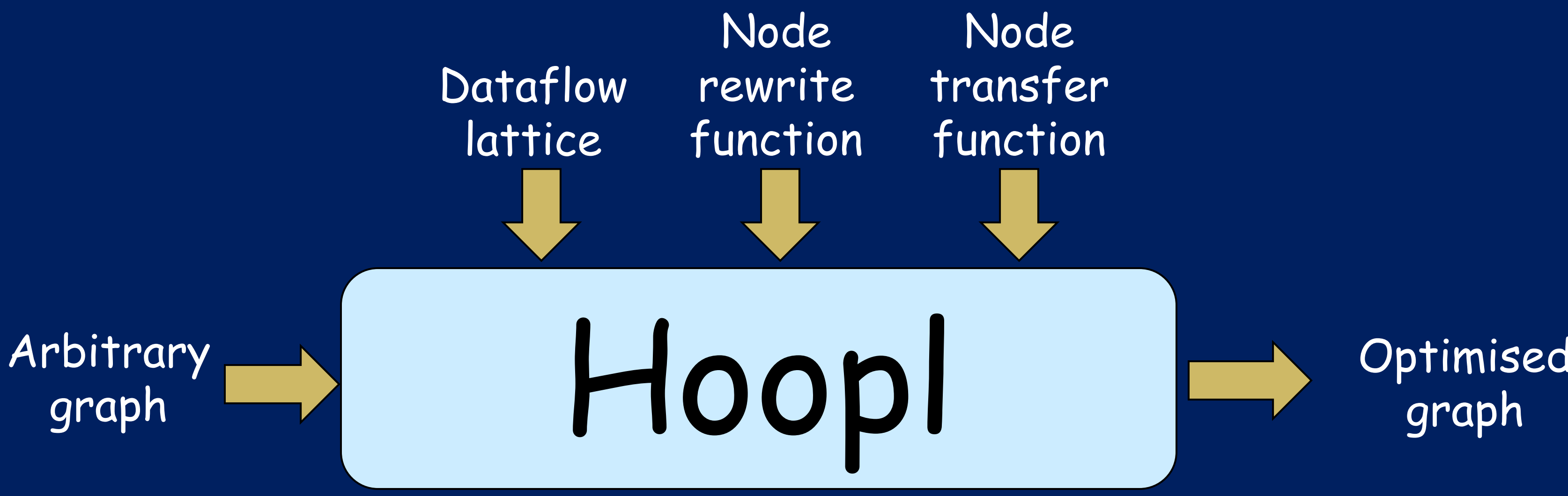
Lerner/Grove/Chambers

- Many dataflow analyses and optimisations can be done in this "analyse-and-rewrite" framework
- Interleaved rewriting and analysis is essential
- Can combine analyses into "super-analyses". Instead of A then B then A then B, do A&B.
- Lerner, Grove, Chambers POPL 2002

Conventional implementations

- Graph implemented using pointers
- Facts decorate the graph; keeping them up to date is painful
- Rewrites implements as mutation; undoing bogus rewrites is a major pain
- Difficult and scary

Hoopl: making it easy



- Interleaved rewriting and analysis
- Shallow and deep rewriting
- Fixpoint finding for arbitrary control flow
- One function for forward dataflow, one for backward
- Polymorphic in **node** and **fact** types

Open and closed

In Hoopl we have:

- Nodes
- Blocks
- Graphs

All are parameterised by whether "shape"

- Open/Closed on entry
- Open/Closed on exit

What is a node?

- Defined by **client** of Hoopl
- Hoopl is **polymorphic** in node type

```
data O          -- Defined
data C          --   by Hoopl

data Node e x where -- Defined by client
  Head        :: Node C O
  Assign      :: Reg -> Expr -> Node O O
  Store       :: Expr -> Expr -> Node O O
  Branch      :: BlockId -> Node O C
  CondBranch :: BlockId -> BlockId -> Node O C
  ...more constructors...
```

What is a block?

```
data Block n e x where      -- Defined by Hoopl
  BUnit :: n e x -> Block n e x
  BCat  :: Block n e O -> Block n O x -> Block n e x
```

- Blocks are **non-empty** **sequences** of nodes
- Only open/open joins are allowed
- Type of block describes its "shape"

```
BUnit (Assign x e) :: Block O O

BUnit (Assign x e) `BCat` BUnit (Branch l1) :: Block O C

BUnit (Branch l1) `BCat` BUnit (Assign x e)   -- ILL-TYPED
```

What is a graph?

```
type LBlocks n = Data.IntMap (Block n C C)
```

- LBlocks is a collection of closed/closed Blocks
  - Used for the main body of a graph

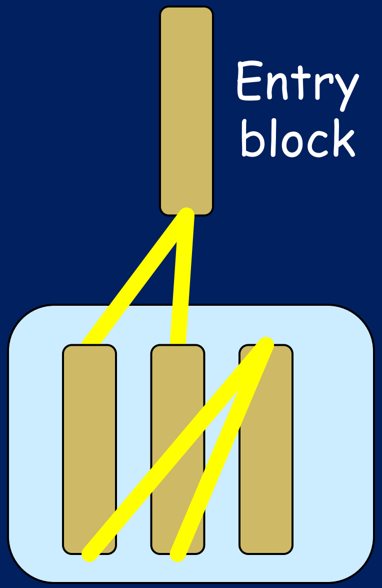
What is a graph?

```
type LBlocks n = Data.IntMap (Block n C C)

data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
```

- GUnit lifts a Block to be a Graph
- GNil is the empty graph (open both ends). Remember, blocks are non-empty, so GUnit won't do for this.

What is a graph?

```
type LBlocks n = Data.IntMap (Block n C C)

data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x
```

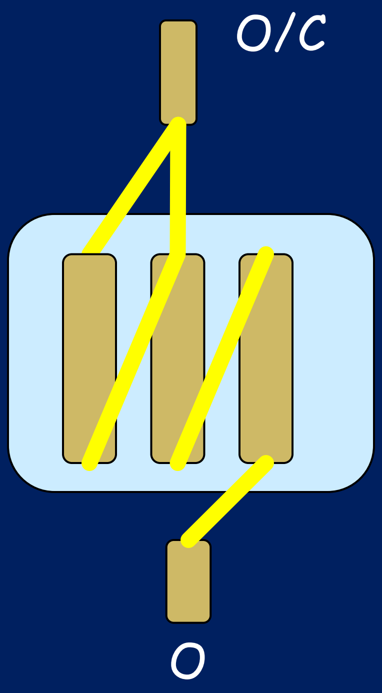


GMany has

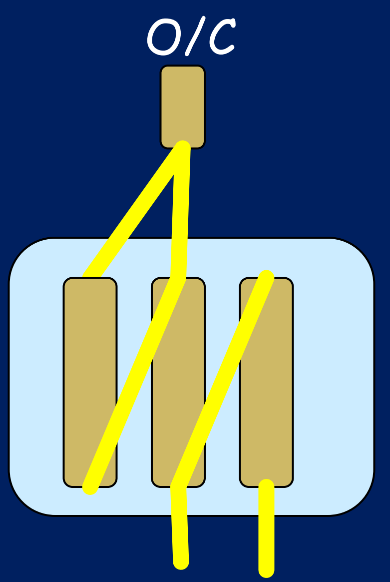
- a distinguished entry block (closed at end)
- an arbitrary graph of internal LBlocks (all C/C)
- a "tail" of some kind

What is a graph?

```
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x

data Tail n x where
  NoTail :: Tail n C
  Tail   :: BlockId -> Block n C O -> Tail n O
```



- Tail id b => control flows out through b
- NoTail => control leaves graph by gotos only



Unique representation

```
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x

data Tail n x where
  NoTail :: Tail n C
  Tail   :: BlockId -> Block n C O
         -> Tail n O
```

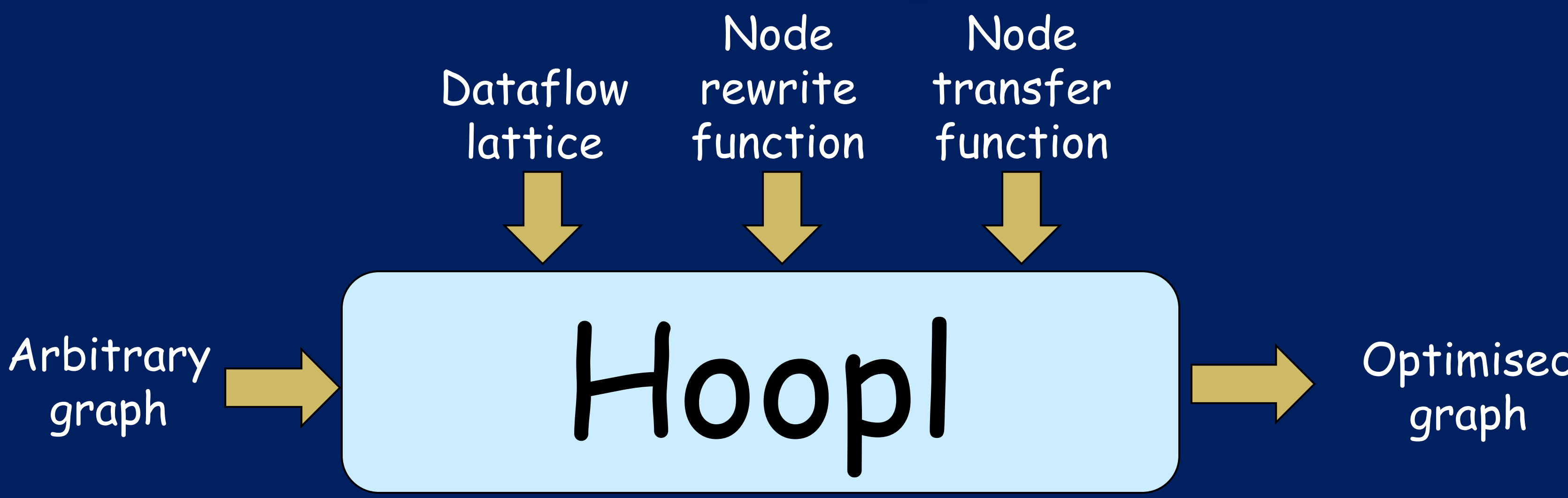
- No blocks: GNil
- 1 block:
  - Open at end: (GUnit b)
  - Closed at end : GMany b [] NoTail
- 2 or more blocks:
  - Open at end: GMany be bs (Tail bx)
  - Closed at end: GMany b bs NoTail

Constant-time graph concatenation

```
gCat :: Graph n e O -> Graph n O x -> Graph n e x
gCat GNil g2 = g2
gCat g1 GNil = g1
gCat (GUnit b1) (GUnit b2) = GUnit (b1 `BCat` b2)
gCat (GUnit b) (GMany e bs x) = GMany (b `BCat` e) bs x
gCat (GMany e bs (Tail bid x)) (GUnit b2)
   = GMany e bs (Tail bid (x `BCat` b2))
gCat (GMany e1 bs1 (Tail bid x1)) (GMany e2 bs2 x2)
   = GMany e1 (LB bid (x1 `BCat` e2) : bs1 ++ bs2) x2
```

```
data LBlock n x = LB BlockId (Block n C x)
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C -> [LBlock n C] -> Tail n x -> Graph n e x
data Tail n x where
  NoTail :: Exit n C
  Tail   :: BlockId -> Block n C O -> Exit n O
```

Hoopl: making it easy




```
analyseAndRewriteFwd ::
  forall n f. Edges n
   => DataflowLattice f
   -> ForwardTransfers n f
   -> ForwardRewrites n f
   -> RewritingDepth
   -> Graph n e C
   -> f
   -> HooplM(Graph n e C, ...)
```

What is HooplM?

- It supports
  - Allocating fresh blockIds
  - Supply of “optimisation fuel”
- When optimisation fuel is exhausted, no more rewrites are done
- Allows binary search to pin-point a buggy rewrite

What is a dataflow lattice?

```
data DataflowLattice a = DataflowLattice  {
  fact_bot    :: a,
  fact_extend :: a -> a -> (a, ChangeFlag)
}

data ChangeFlag = NoChange | SomeChange
```

- `fact_extend` takes
  - The "current fact"
  - A "new fact"

and returns

  - Their least upper bound
  - A flag indicating whether the result differs from the "current fact"

What is a rewrite function?

```
type ForwardRewrites n f
  = forall e x. n e x -> f -> Maybe (AGraph n e x)
```

- Takes a node, and a fact and returns
  - Nothing => No rewrite, thank you
  - Just g => Please rewrite to this graph
- AGraph is a Graph, except that it needs a supply of fresh BlockIds:

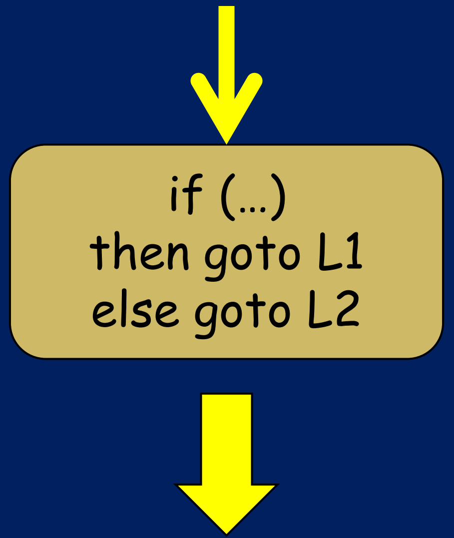
```
type AGraph n e x = BlockIdSupply
                  -> (Graph n e x, BlockIdSupply)
```

- Returned graph is **same shape as input**!

What is a transfer function?

```
type ForwardTransfers n f
  = forall e x. n e x -> f -> f    -- WRONG
```

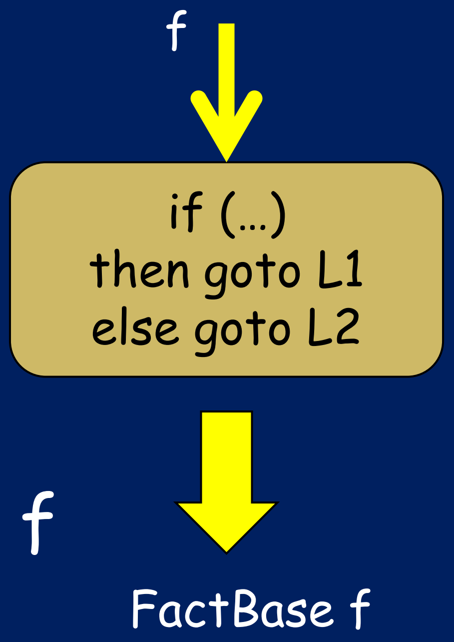
- What if x=C?



What comes out???
Clearly not one fact!

What is a transfer function?

```
type ForwardTransfers n f
  = forall e x. n e x -> f -> f     -- WRONG
```

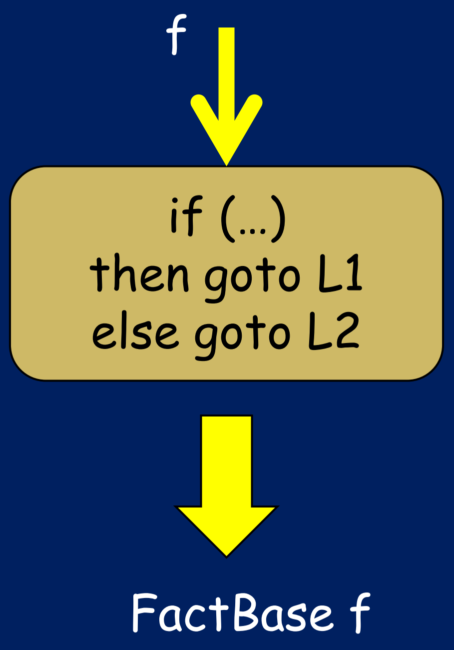


- What if x=C?
- Then what comes out is
  type FactBase f = Map BlockId f
- So the result type depends on f
- Type functions to the rescue!

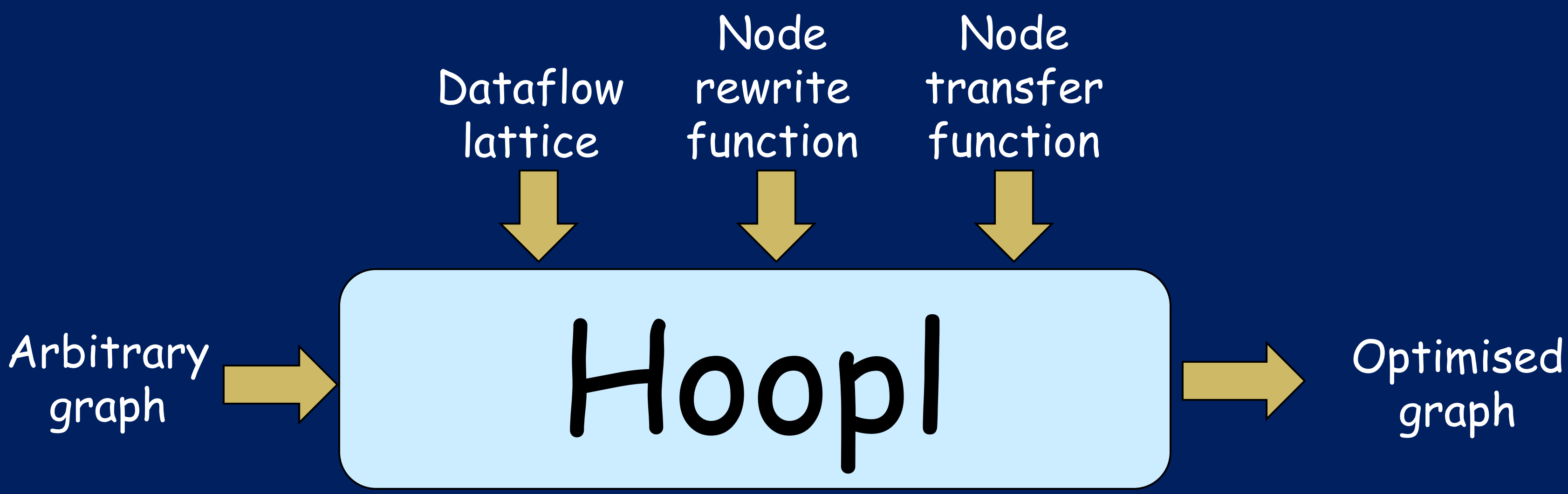
What is a transfer function?

```
type ForwardTransfers n f
  = forall e x. n e x -> f -> OutFact x f

type family OutFact x f
type instance OutFact O f = f
type insatanc OutFact C f = FactBase f
```

- "Fact" coming out depends on the "x" flag (only)



Hoopl: making it easy




```
analyseAndRewriteFwd ::
  forall n f. Edges n
   => DataflowLattice f
   -> ForwardTransfers n f
   -> ForwardRewrites n f
   -> RewritingDepth
   -> Graph n e C
   -> f
   -> HooplM(Graph n e C, ...)
```

Implementing Hoopl

- The grand plan

```
arfNode :: ForwardTransfers n f
        -> ForwardRewrites n f
        -> ARF n f

arfBlock :: ARF n f -> ARF (Block n) f

arfGraph :: ARF (Block n) f -> ARF (Graph n) f
```

Deals with sequence of nodes in a block

Deals with fixpoints

Implementing Hoopl

- The grand plan

```
arfNode :: ForwardTransfers n f
        -> ForwardRewrites n f
        -> ARF (Graph n) f
        -> ARF n f

arfBlock :: ARF n f -> ARF (Block n) f

arfGraph :: DataflowLattice f
         -> ARF (Block n) f -> ARF (Graph n) f
```

How to analyse and rewrite a rewritten graph

Deals with fixpoints

What is ARF?

```
arfNode :: ForwardTransfers n f
        -> ForwardRewrites n f
        -> ARF (Graph n) f
        -> ARF n f

arfBlock :: ARF n f -> ARF (Block n) f

arfGraph :: ARF (Block n) f -> ARF (Graph n) f
```

```
type ARF thing f
   = forall e x. thing e x
              -> f
              -> HooplM (Graph e x, OutFact x f)
```

Input thing

Input fact

Rewritten thing

Output fact

Writing arfBlock

```
type ARF thing f
   = forall e x. thing e x
                 -> f
                 -> HooplM (Graph e x, OutFact x f)

data Block n e x where
  BUnit :: n e x -> Block n e x
  BCat  :: Block n e O -> Block n O x -> Block n e x
```

```
arfBlock :: ARF n f -> ARF (Block n) f
arfBlock arf_node (BUnit n)
  =
arfBlock arf_node (b1 `BCat` b2)
  =
```

Writing arfNode

```
type ARF thing f
   = forall e x. thing e x
                 -> f
                 -> HooplM (Graph e x, OutFact x f)
type ForwardTransfers n f
  = forall e x. n e x -> f -> OutFact f
type ForwardRewrites n f
  = forall e x. n e x -> f -> Maybe (AGraph n e x)

graphOfAGraph :: AGraph n e x -> HooplM (Graph n e x)
nodeToGraph :: n e x -> Graph n e x   -- URK!
```

```
arfNode :: ForwardTransfers n f
        -> ForwardRewrites n f
        -> ARF (Graph n) f
        -> ARF n f

arfNode tf rw arf_graph n f
  = case (rw f n) of
      Nothing -> return (nodeToGraph n, tf f n)
      Just ag -> do { g <- graphOfAGraph ag
                    ; arf_graph g f }
```

nodeToGraph

```
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x
```

```
nodeToGraph :: n e x -> Graph n e x
nodeToGraph n = GUnit (BUnit n)
```

Cannot unify 'e' with 'O'

- Could generalise type of GUnit
- Or add class constraint to nodeToGraph

nodeToGraph

```
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x
```

```
class LiftNode x where
  nodeToGraph :: n e x -> Graph n e x

instance LiftNode O where
  nodeToGraph n = GUnit (BUnit n)

instance LiftNode C where
  nodeToGraph n = GMany (BUnit n) [] NoTail
```

- But since nodeToGraph is overloaded, so must arfNode be overloaded...

Writing arfNode

```
type ARF thing f
   = forall e x. LiftNode x
                 => thing e x
                 -> f
                 -> HooplM (Graph e x, OutFact x f)
```

```
arfNode :: ForwardTransfers n f
        -> ForwardRewrites n f
        -> ARF (Graph n) f
        -> ARF n f

arfNode tf rw arf_graph n f
  = case (rw f n) of
      Nothing -> return (nodeToGraph n, tf f n)
      Just ag -> do { g <- graphOfAGraph ag
                    ; arf_graph g f }
```

arfGraph

```
data Graph n e x where
  GNil  :: Graph n O O
  GUnit :: Block n e O -> Graph n e O
  GMany :: Block n e C
        -> LBlocks n
        -> Tail n x
        -> Graph n e x
```

```
arfGraph :: DataflowLattice f
        -> ARF (Block n) f -> ARF (Graph n) f
```

- More complicated: 30 lines of code (!)
  - Three constructors (GNil, GUnit, GMany)
  - The optional Tail
  - Fixpoint
  - Put blocks in topological order to improve convergence

The pièce de resistance

```
analyseAndRewriteFwd
   :: forall n f. Edges n
   => DataflowLattice f   -> ForwardTransfers n f
   -> ForwardRewrites n f -> RewritingDepth
   -> ARF_Graph n f

analyseAndRewriteFwd depth lat tf rw
 = anal_rewrite
 where
  anal_only, anal_rewrite, rec :: ARF_Graph n f
  anal_only    = arfGraph lat $ arfBlock $ analNode tf
  anal_rewrite = arfGraph lat $ arfBlock $ arfNode tf rw rec

  rec = case depth of
              RewriteShallow -> anal_only
              RewriteDeep    -> anal_rewrite

analNode :: ForwardTransfers n f -> ARF_Node n f
analNode tf n f = return (nodeToGraph n f, tf f n)
```

Conclusion

- Old code was 250+ lines, impossible to understand, and probably buggy
- New code is < 100 lines, has many more static checks, and is much easier to understand
- GADTs and type functions play a crucial role