Code Graphs as Data-Flow Graphs, Control-Flow Graphs, and Interaction Nets

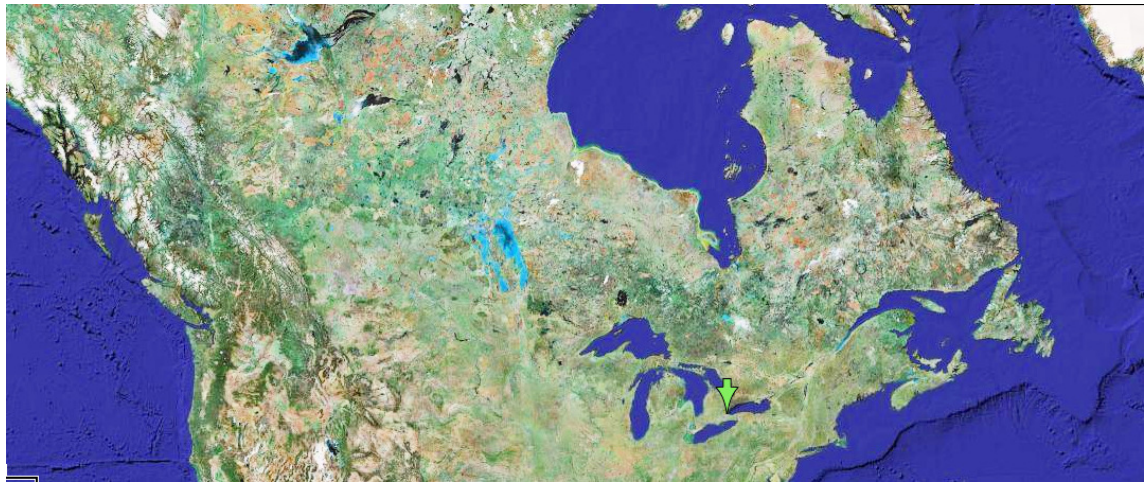
Wolfram Kahl

kahl@cas.mcmaster.ca

Software Quality Research Laboratory

McMaster University

Hamilton, Ontario, Canada



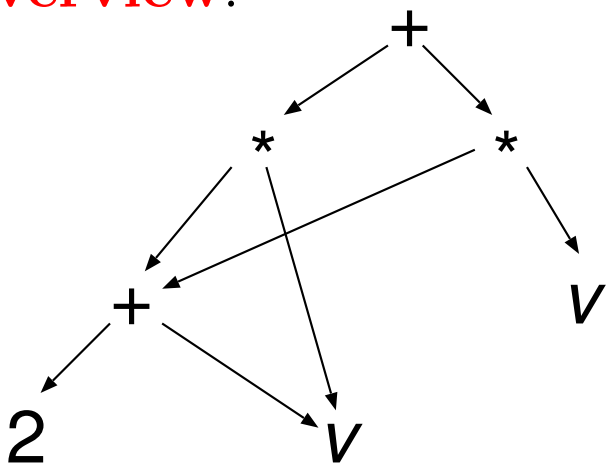
7 October 2015 — Universidade do Minho, Braga

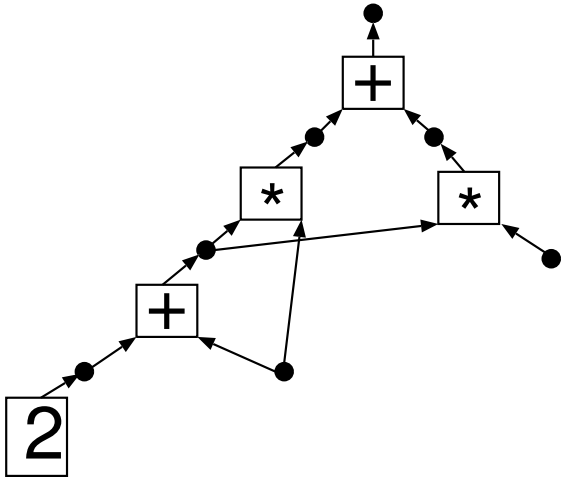
Overview

- From Term Graphs via Jungles To Data-Flow Code Graphs
- Software-Pipelining as Code Graph Transformation
- Calculating Software-Pipelining on Nested CF/DF Code Graphs
- From Code Graphs to Interaction Nets
- Parallel Interaction Net Implementations in Haskell and Go

Terms, Term Graphs, Jungles

Term $(2 + x) * x + (2 + x) * y$ represented with sharing —
quick overview:



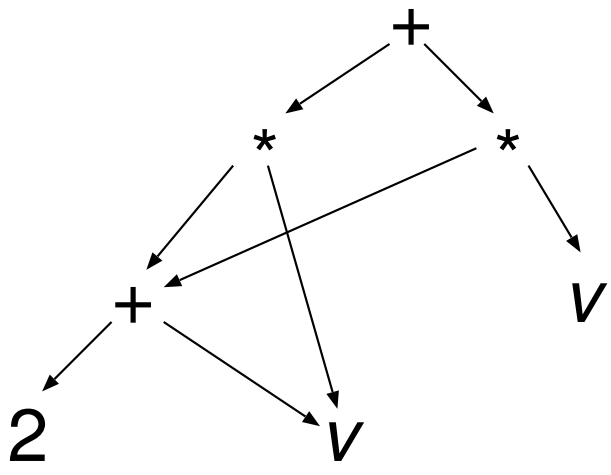


Term Graph **Jungle**

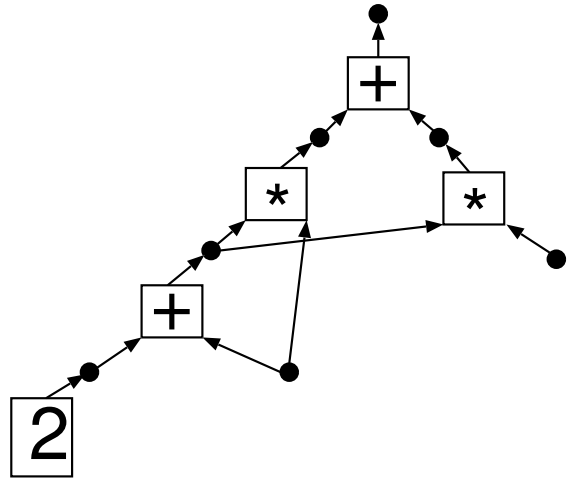
- **both:** directed, (implicit) incidence ordering
- **Term graph:** Symbols as node labels
- **Jungles:** Symbols as hyperedge labels

Terms, Term Graphs, Jungles

Term $(2 + x) * x + (2 + x) * y$ represented with sharing:



Term Graph

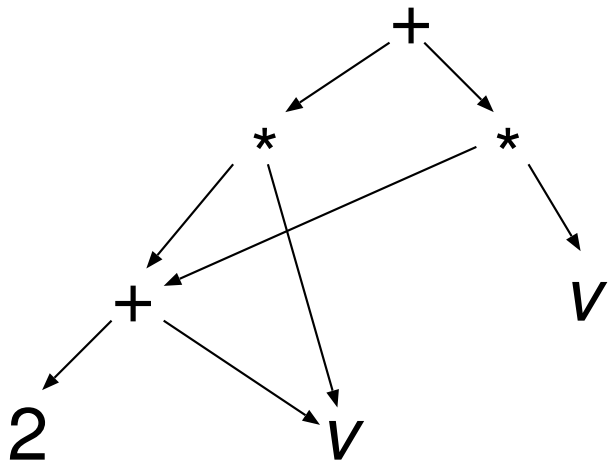


Jungle

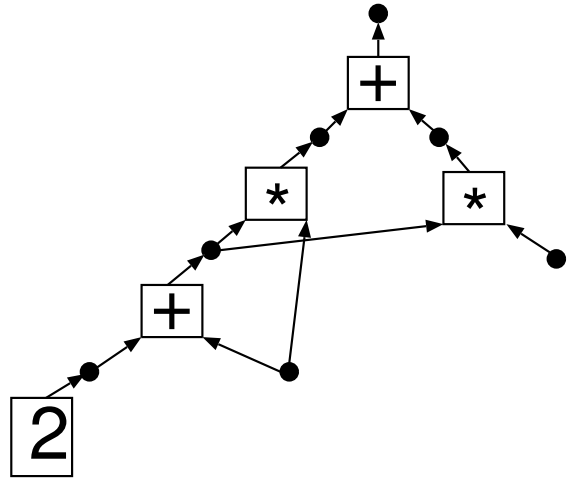
- Term graphs: arguments as successors
- Jungles:
  - arguments supplied via input tentacles
  - result tentacles: bijection between edges and non-var. nodes
- both:
  - direction motivated by denotational semantics
  - variable nodes, no variable names
  - variable nodes are input interface; root is output interface
  - "garbage" is not reachable from root

Terms, Term Graphs, Jungles (ctd.)

Term $(2 + x) * x + (2 + x) * y$ represented with sharing:



Term Graph

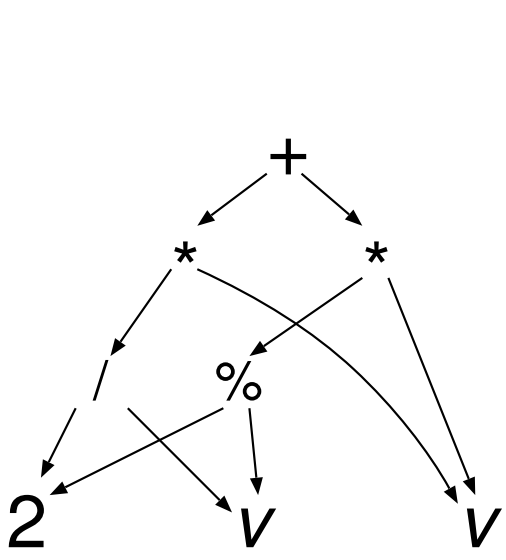


Jungle

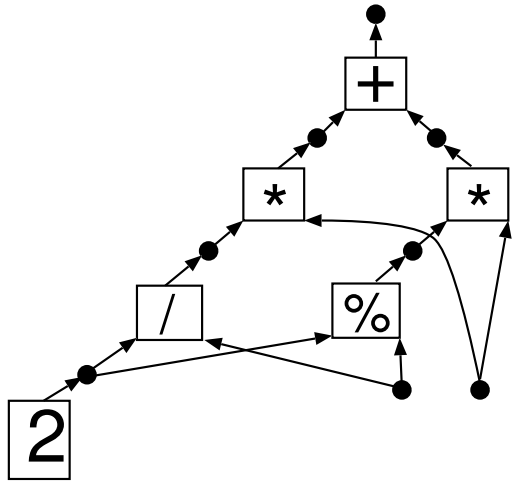
- Without meta-variables, term DAGs and acyclic jungles are equivalent
- TG representation of CRS rewriting requires "unnatural" matching concept for completeness
- Acyclic jungles are the morphisms of the free gs-monoidal category — functorial semantics [Corradini and Gadducci, 1999]
- Cyclic jungles: traced monoidal categories

A Second Example: From Jungles to Code Graphs

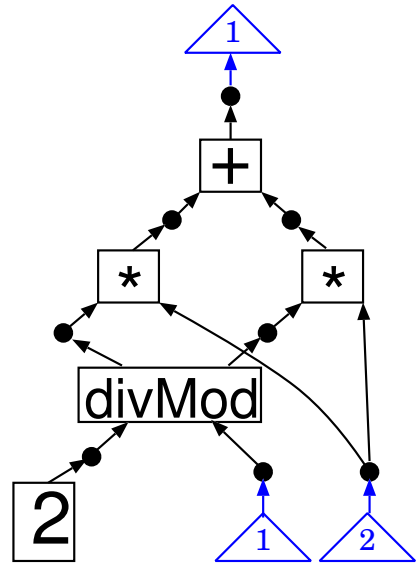
Term $(2/x) * y + (2\%x) * y$ represented with sharing:



Term Graph



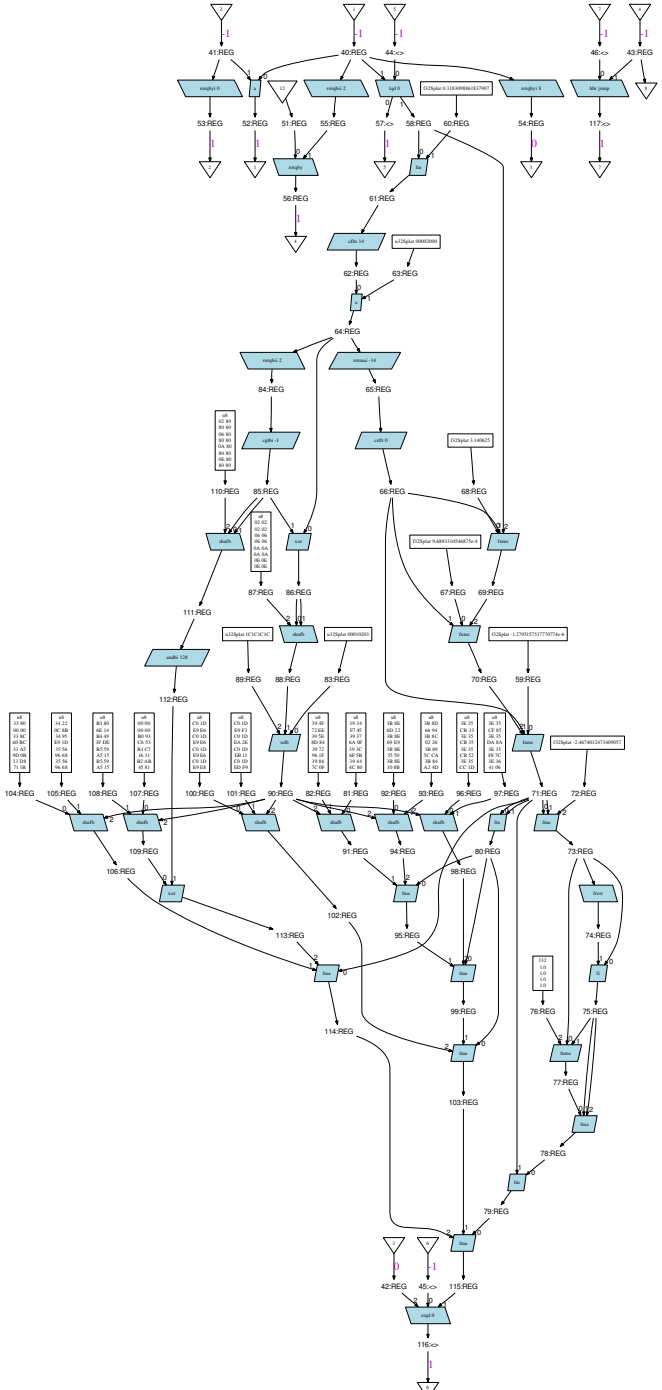
Jungle



Code Graph

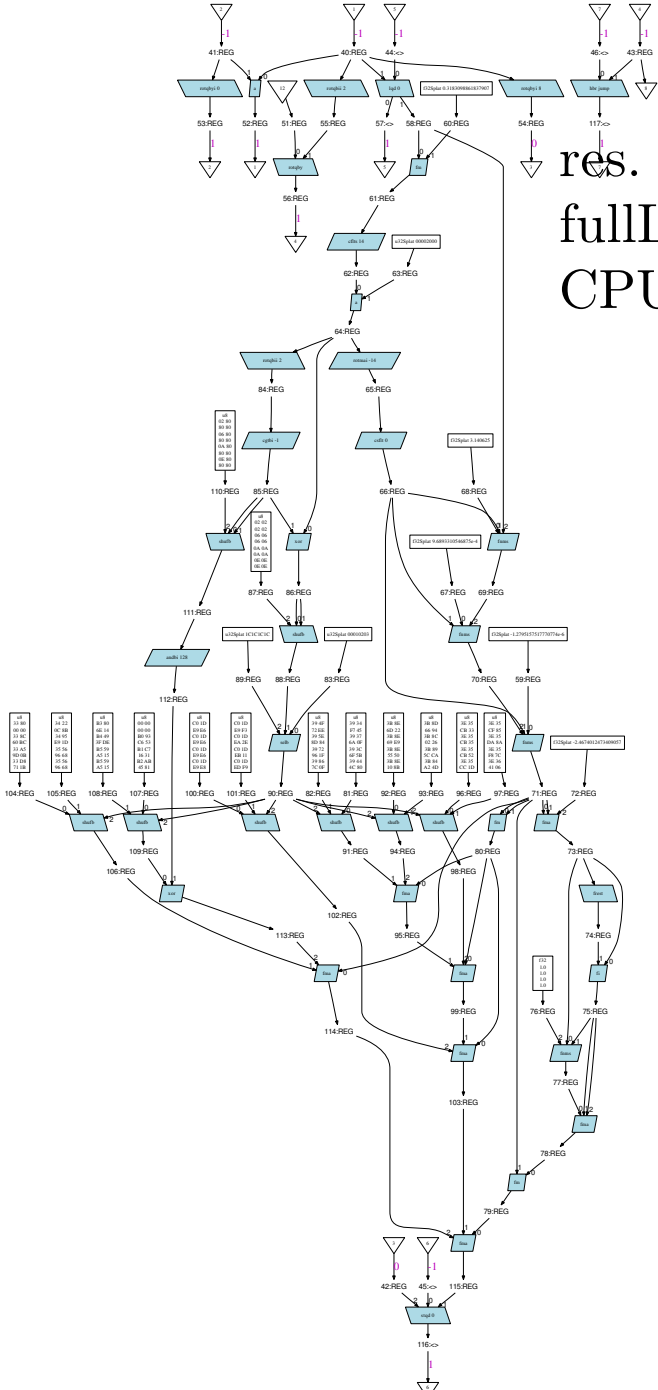
- Operations like divMod can have multiple results
- Code graph hyperedges can have multiple result tentacles
- The sequences of input and output nodes are graphically indicated

Dependencies in Real Code



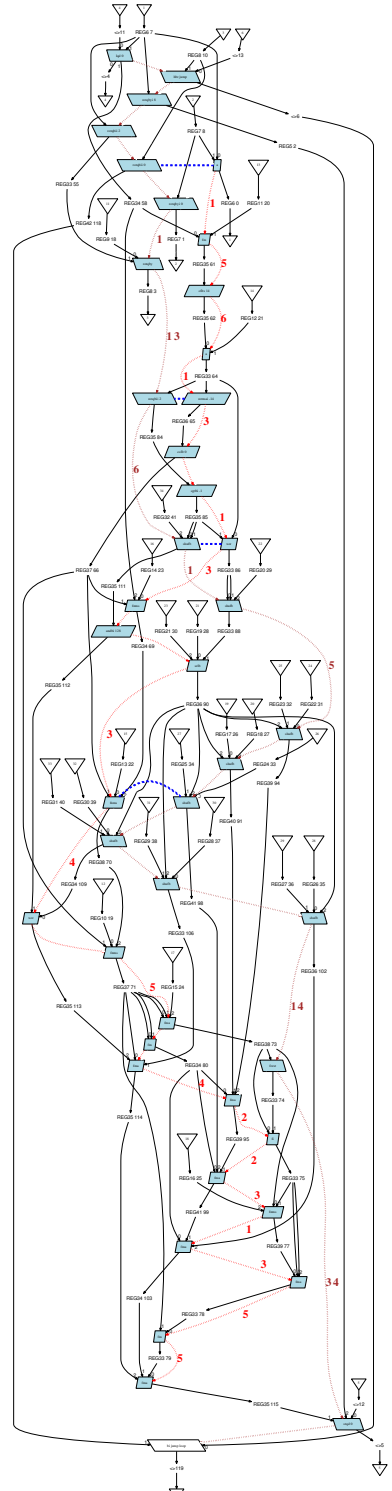
$$\texttt{result[i]} := \tan(\text{arg[i]})$$

Dependencies in Real Code



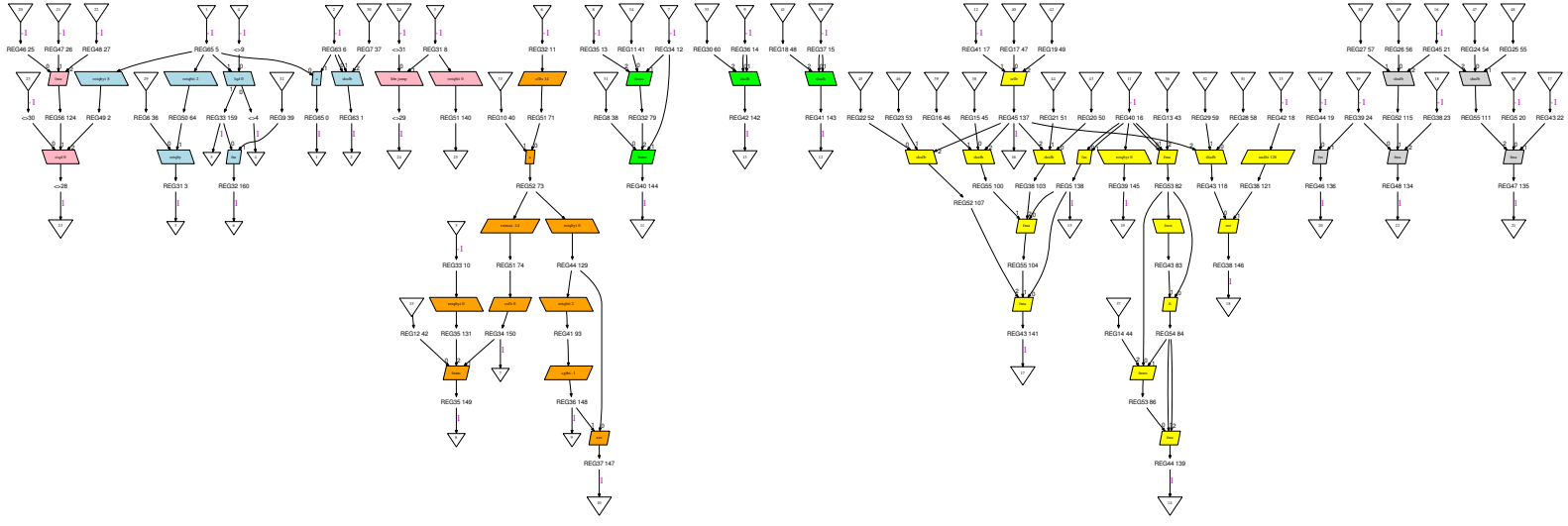
res. lower bounds: [25,19]
fullLength: 93
CPU utilisation ratio: **26.9%**

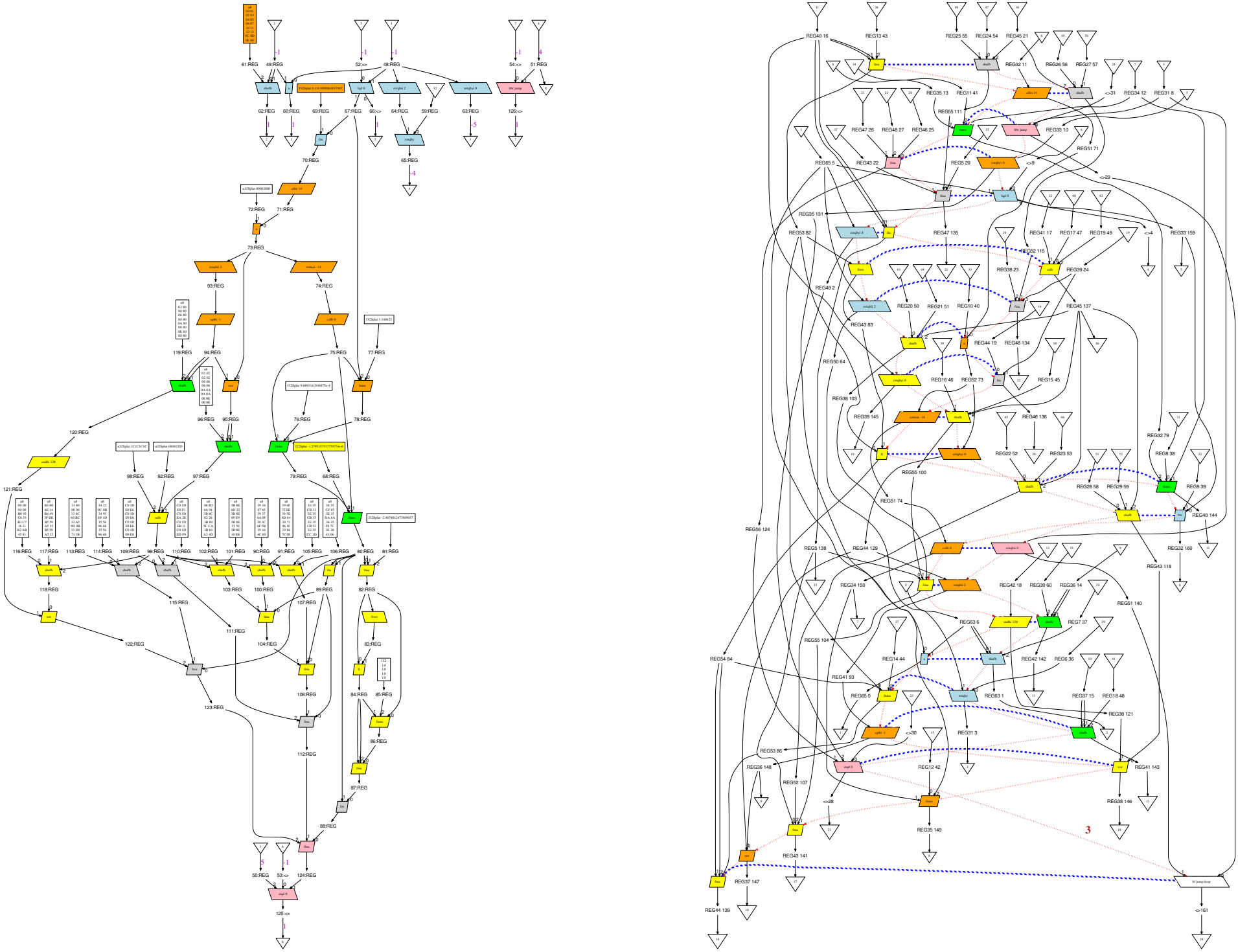
```
loop:   lqd       $34, 0($6)
        hbr       jump, $8
        rotqbyi   $5, $6, 8
        rotqbii   $33, $6, 2
        a         $6, $6, $7
        rotqbii   $42, $8, 0
        rotqbyi   $7, $7, 0
        fm        $35, $34, $11
        rotqby    $8, $9, $33
        cflts     $35, $35, 14
        a         $33, $35, $12
        lnop
        rotmai    $36, $33, -14
        rotqbii   $35, $33, 2
        csflt     $37, $36, 0
        cgtbi     $35, $35, -1
        xor       $33, $33, $35
        shufb     $35, $35, $35, $32
        shufb     $33, $33, $33, $20
        fnms      $34, $14, $37, $34
        andbi     $35, $35, 128
        selb      $36, $19, $33, $21
        shufb     $39, $22, $23, $36
        shufb     $40, $17, $18, $36
        fnms      $38, $13, $37, $34
        shufb     $41, $24, $25, $36
        shufb     $34, $30, $31, $36
        shufb     $33, $28, $29, $36
        shufb     $36, $26, $27, $36
        xor       $35, $34, $35
        fnms      $37, $10, $37, $38
        fma       $38, $37, $37, $15
        fm        $34, $37, $37
        fma       $35, $37, $33, $35
        frest     $33, $38
        fma       $39, $34, $40, $39
        fi        $33, $38, $33
        fma       $41, $34, $39, $41
        fnms      $39, $38, $33, $16
        fma       $34, $34, $41, $36
        fma       $33, $39, $33, $33
        fm        $33, $33, $37
        fma       $35, $33, $34, $35
        stqd      $35, 0($5)
        nop       $0
jump:
        bi        $42
```



Breaking Dependencies by Software Pipelining

	Hardware Pipelining	**Software** Pipelining
Unit:	instruction	loop body
Segments:	pipeline stages	body stages
Independence:	different instructions	different iterations

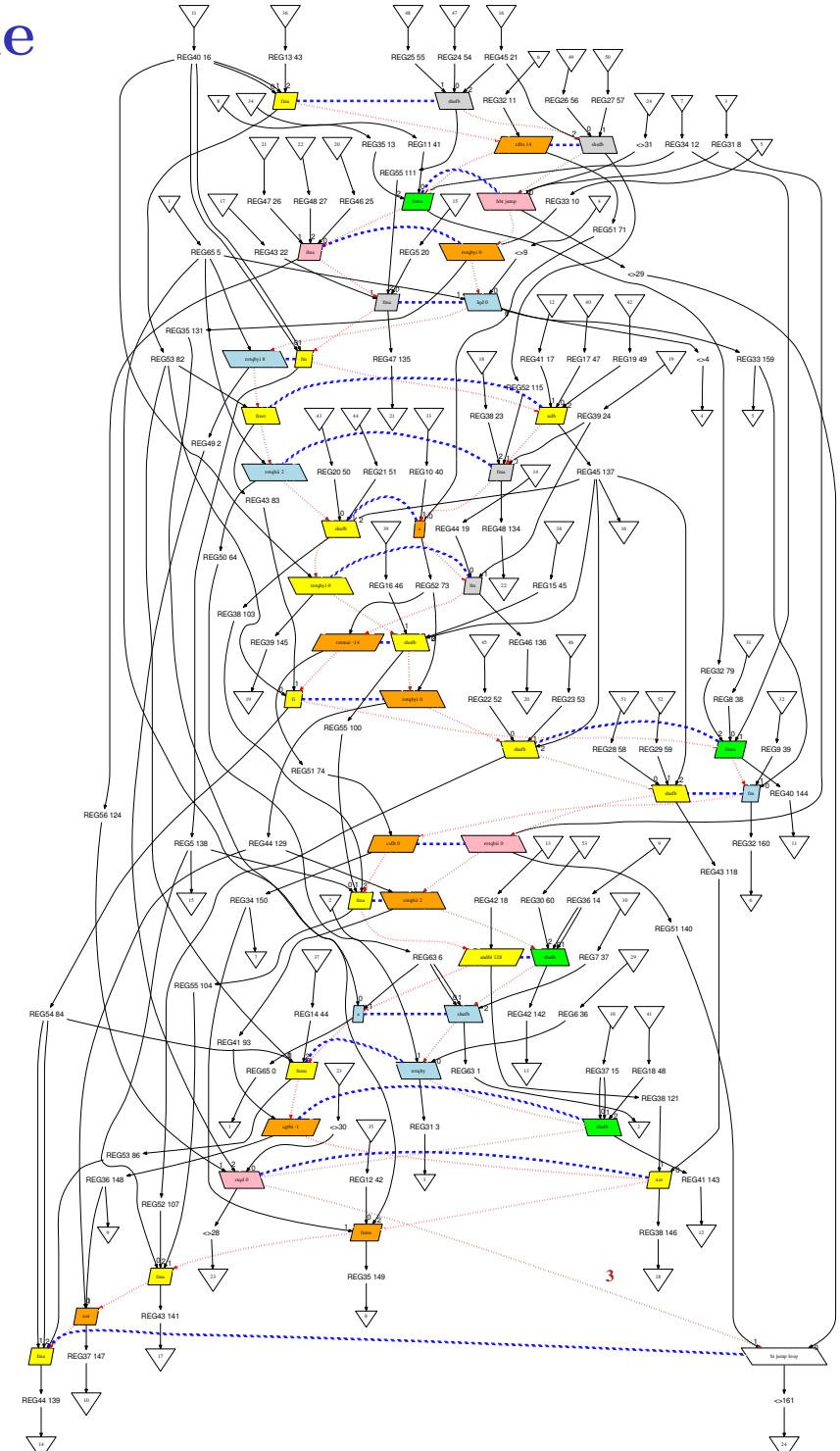



61:REG
49:REG
52:<>
48:REG
54:<>
51:REG
62:REG
60:REG
69:REG
67:REG
66:<>
64:REG
59:REG
63:REG
126:<>
70:REG
65:REG
72:REG
71:REG
73:REG
93:REG
74:REG
119:REG
94:REG
75:REG
77:REG
96:REG
95:REG
76:REG
78:REG
120:REG
98:REG
92:REG
97:REG
79:REG
68:REG
121:REG
116:REG
117:REG
113:REG
114:REG
109:REG
99:REG
110:REG
102:REG
101:REG
90:REG
91:REG
105:REG
106:REG
80:REG
81:REG
118:REG
115:REG
103:REG
100:REG
107:REG
89:REG
82:REG
122:REG
111:REG
104:REG
83:REG
123:REG
108:REG
84:REG
85:REG
112:REG
86:REG
87:REG
88:REG
50:REG
53:<>
124:REG
125:<>
REG40 16
REG13 43
REG25 55
REG24 54
REG45 21
REG32 11
REG26 56
REG27 57
REG35 13
REG11 41
REG34 12
REG31 8
REG47 26
REG48 27
REG46 25
REG55 111
REG33 10
REG51 71
REG65 5
REG43 22
REG20 20
<>9
<>29
REG35 131
REG53 82
REG47 135
REG41 17
REG17 47
REG19 49
<>4
REG33 159
REG52 115
REG49 2
REG38 23
REG39 24
REG20 50
REG21 51
REG10 40
REG45 137
REG43 83
REG44 19
REG48 134
REG50 64
REG16 46
REG52 73
REG15 45
REG38 103
REG39 145
REG46 136
REG32 79
REG22 52
REG23 53
REG38 38
REG55 100
REG28 58
REG29 59
REG39 39
REG51 74
REG40 144
REG56 124
REG5 138
REG44 129
REG42 18
REG30 60
REG36 14
REG43 118
REG34 150
REG51 140
REG63 6
REG37 38
REG55 104
REG14 44
REG42 142
REG66 36
REG54 84
REG41 93
REG65 0
REG63 1
REG37 15
REG18 48
REG38 121
REG53 86
<>30
REG31 3
REG36 148
REG12 42
REG41 143
REG52 107
<>28
REG38 146
REG35 149
REG43 141
REG37 147
REG44 139
<>161
3

Software-Pipelined Schedule

res. lower bounds: [25,19]
fullLength: 25
CPU utilisation ratio: **100%**

```
loop:   fma       $53, $40, $40, $13
        shufb     $55, $24, $25, $45
        cflts     $51, $32, 14
        shufb     $52, $26, $27, $45
        fnms      $32, $11, $34, $35
        hbr       jump, $31
        fma       $56, $46, $47, $48
        rotqbyi   $35, $33, 0
        fma       $47, $5, $43, $55
        lqd       $33, 0($65)
        fm        $5, $40, $40
        rotqbyi   $49, $65, 8
        selb      $45, $17, $41, $19
        frest     $43, $53
        fma       $48, $39, $52, $38
        rotqbii   $50, $65, 2
        a         $52, $51, $10
        shufb     $38, $20, $21, $45
        fm        $46, $44, $39
        rotqbyi   $39, $40, 0
        rotmai    $51, $52, -14
        shufb     $55, $15, $16, $45
        fi        $54, $53, $43
        rotqbyi   $44, $52, 0
        fnms      $40, $8, $34, $32
        shufb     $52, $22, $23, $45
        fm        $32, $33, $9
        shufb     $43, $28, $29, $45
        csflt     $34, $51, 0
        rotqbii   $51, $31, 0
        fma       $55, $5, $55, $38
        rotqbii   $41, $44, 2
        andbi     $38, $42, 128
        shufb     $42, $36, $36, $30
        a         $65, $65, $63
        shufb     $63, $63, $63, $7
        fnms      $53, $53, $54, $14
        rotqby    $31, $6, $50
        cgtbi     $36, $41, -1
        shufb     $41, $37, $37, $18
        xor       $38, $43, $38
        stqd      $56, 0($49)
        fnms      $35, $12, $34, $35
        fma       $43, $5, $55, $52
        xor       $37, $44, $36
        lnop
        fma       $44, $53, $54, $54
jump:
        bi        $51
```



Hardware Parallelism on the Cell BE

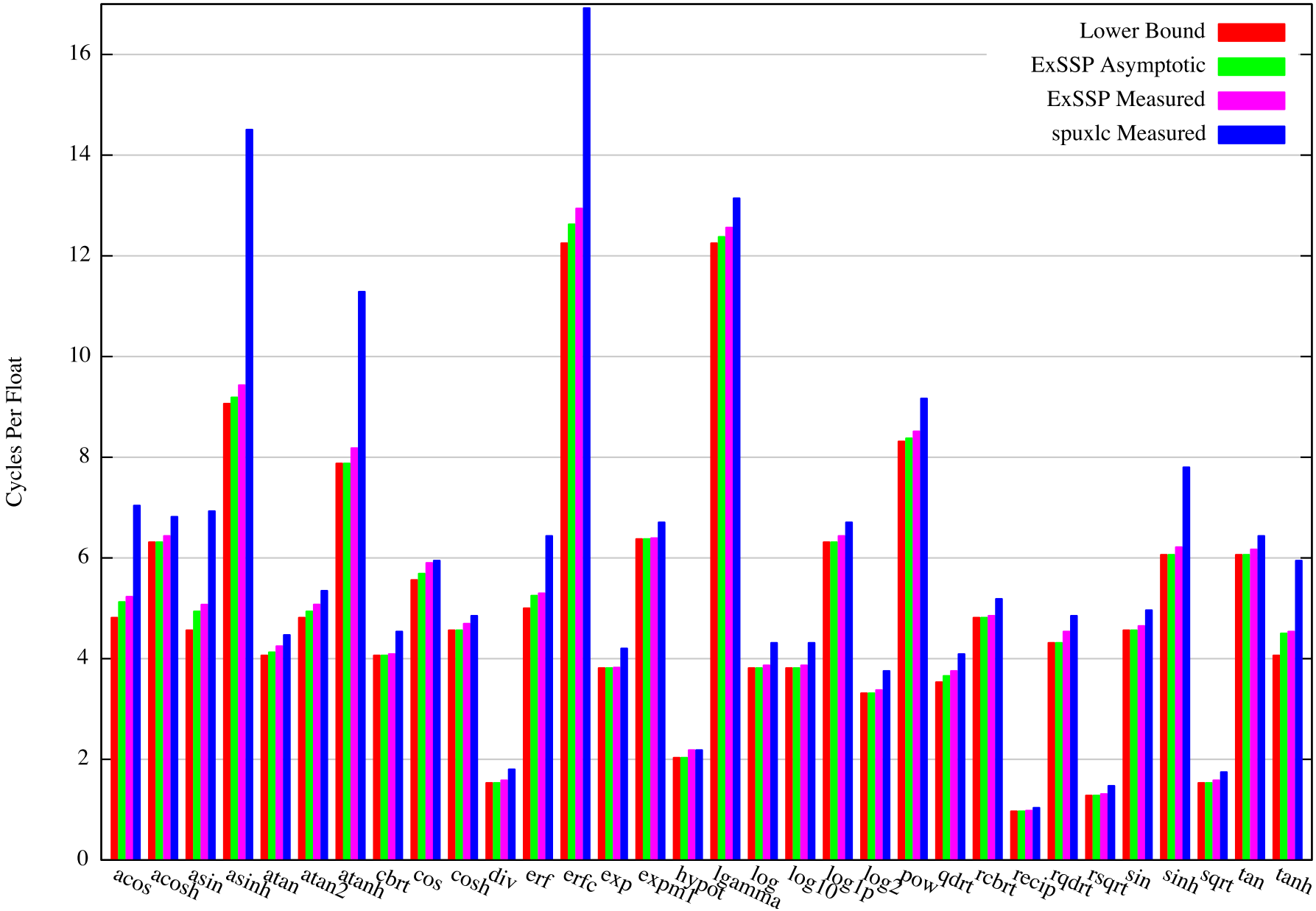
- **1** PPE + **8** SPEs
- DMA independent of execution (**double-buffering**)

Hardware Parallelism on the Cell SPU

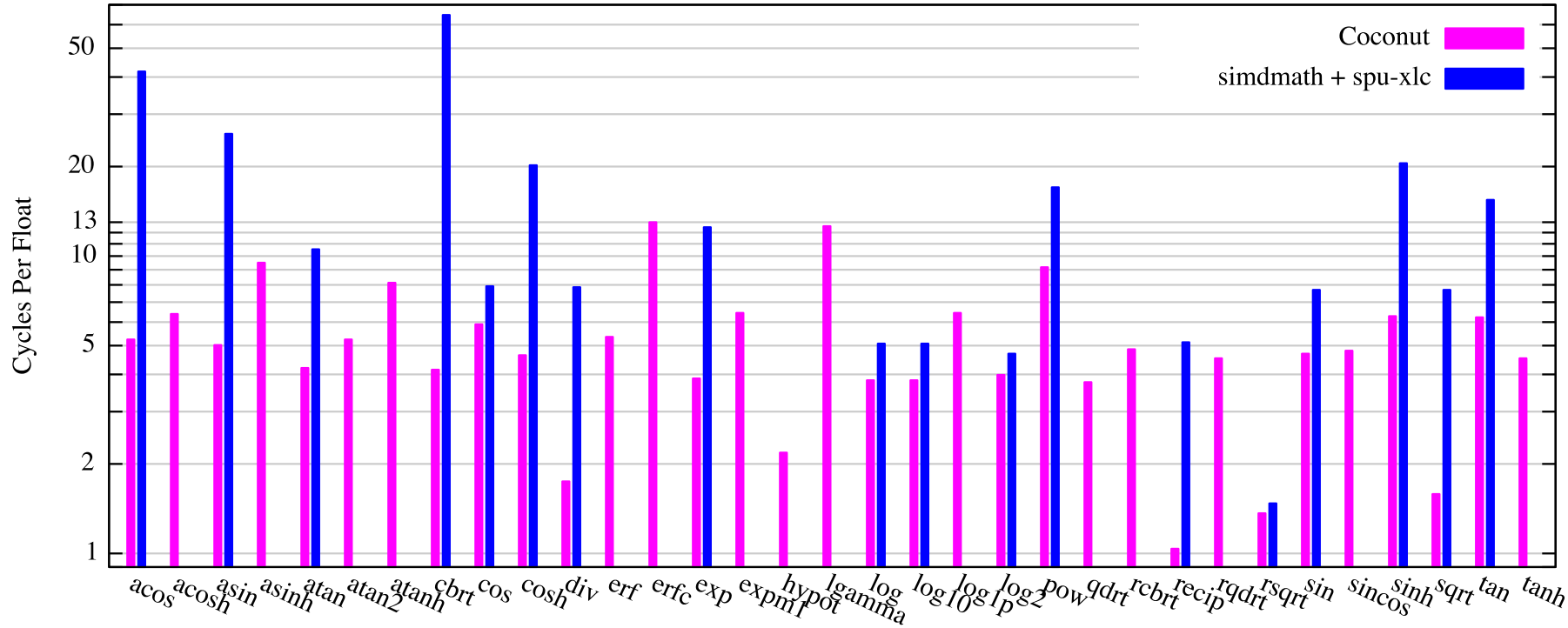
- SIMD instructions act in parallel on "polymorphic" registers:
  128bit = 2 `double` = **4** `float` = 4 `int` = 8 `short` = 16 `byte`
- **2** instruction units (arithmetic and data movement) with double issue
- Both units are *pipelined*:
  Up to **6** instructions "in-flight" simultaneously

Parallelism requires independence!

Coconut-Scheduled Special Function Library



Coconut-Scheduled Special Function Library



- Shipped with IBM Cell SDK

Intermediate Representation: Code Graphs

Hypergraphs are a kind of *typed term graphs* with:

- node-labels (register or state **types**)
- edge-labels (functions, **operation names**, state transformations)
- each edge is assigned
  - a sequence of argument nodes
  - a sequence of result nodes

Code graphs are hypergraphs with:

- designated *input* and *output* node sequences

Simple Transformation:

rewrite step is DPO in **hypergraph category**
rule is span in **code graph category**

Control-Flow Rearrangment

- Original view of pipelining transformation: *Recomposition*
- More flexible: *Nested graphs*
- Data-flow graphs inside control-flow graph hyperedges
- Type interaction
- **Semantically justified transformations**
- Control-flow **rearrangement** for **software pipelining**

Software Pipelining by Calculation

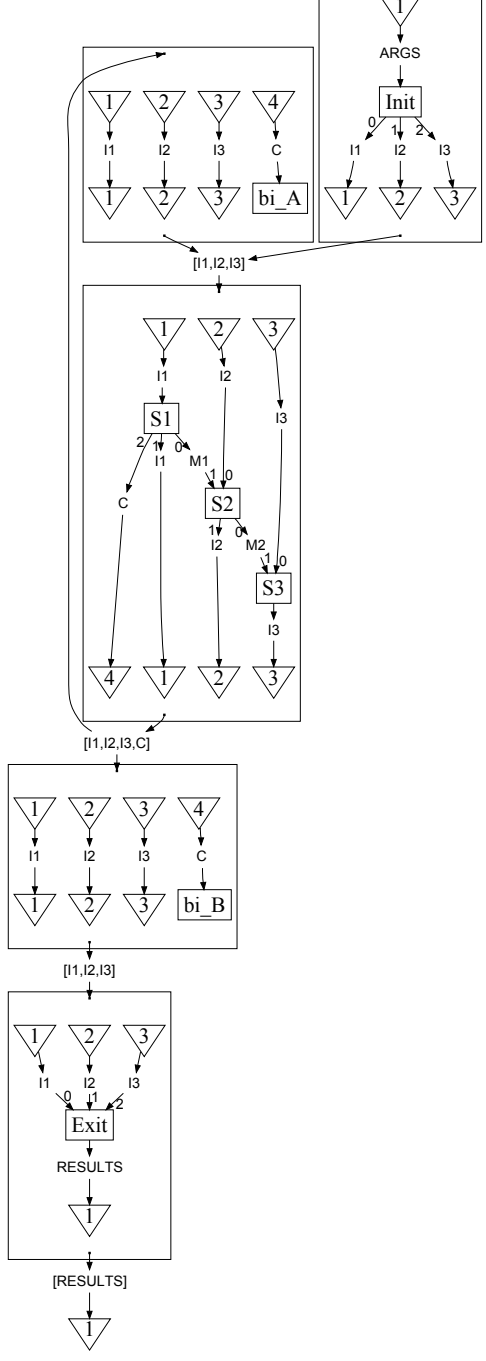
Simple Control-Flow Code Graphs

State transformations as edge labels:

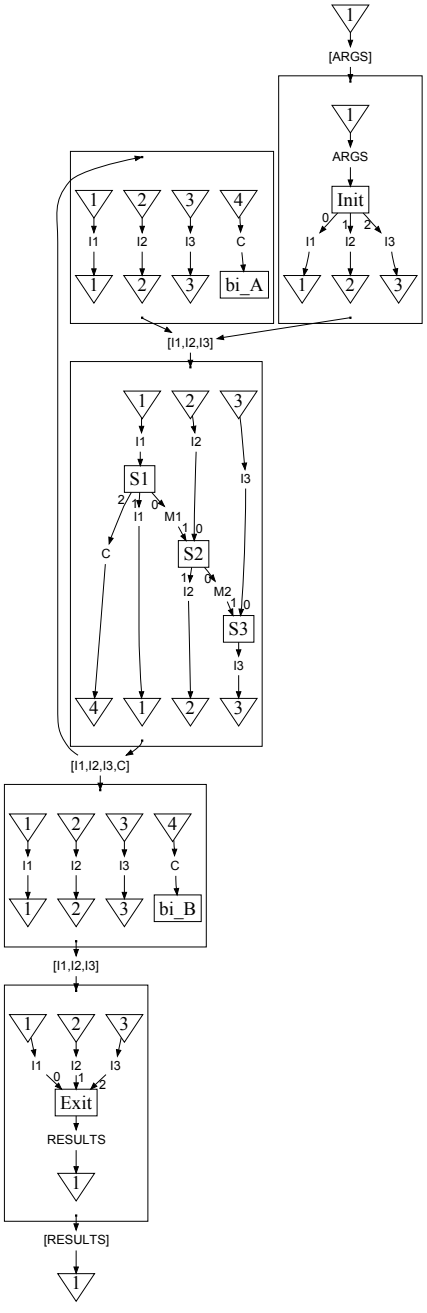
- Deterministic control flow:
  single argument, single result
- Cycles and joins allowed
- Different state types:
  live data sets
- **Nested** data-flow graphs

Standard semantics:

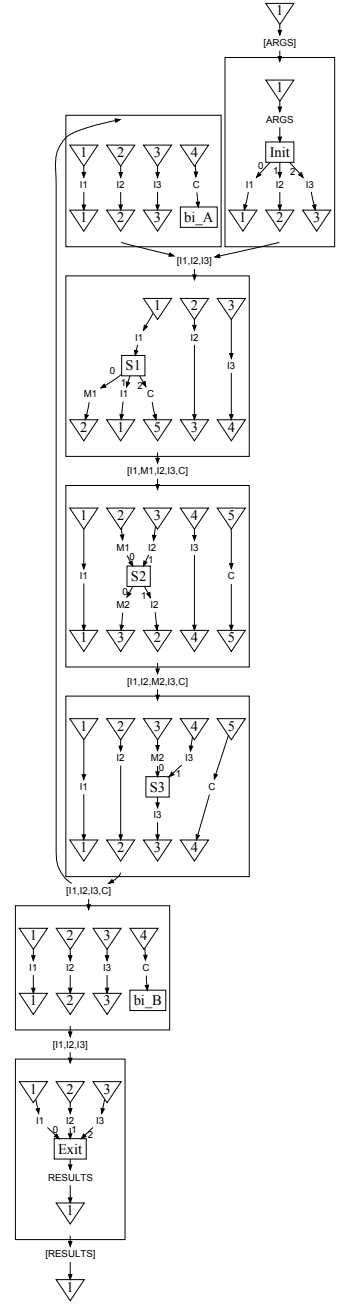
- Kleene categories



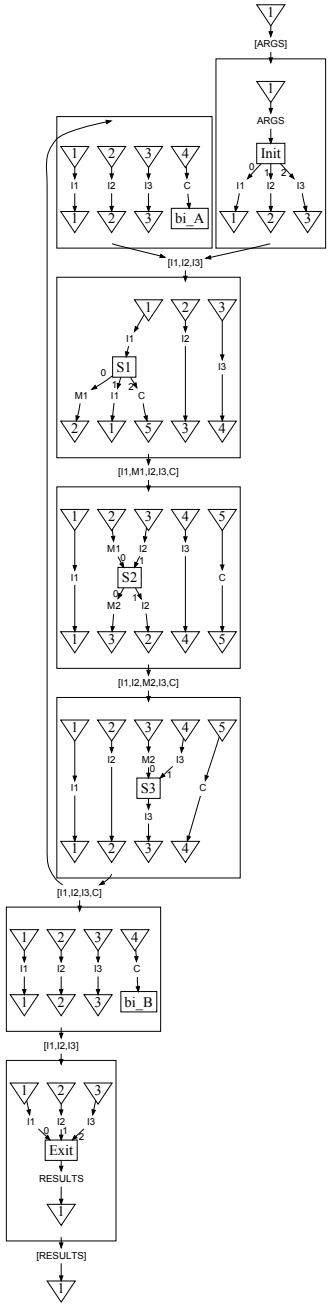
Sequentially Decompose Staged Body



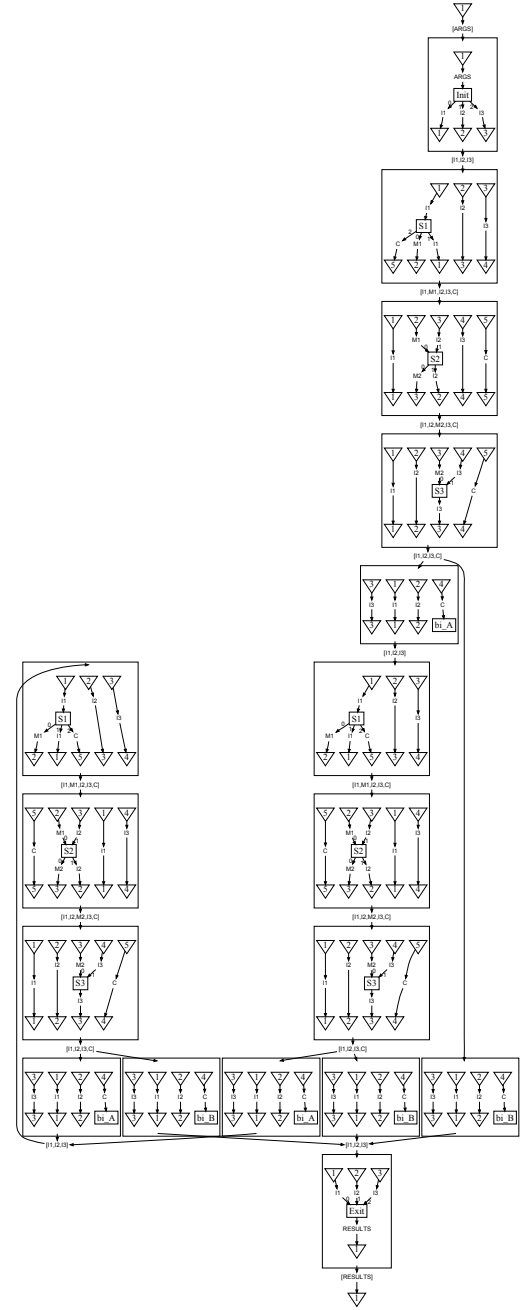
Common composition of both code graph layers



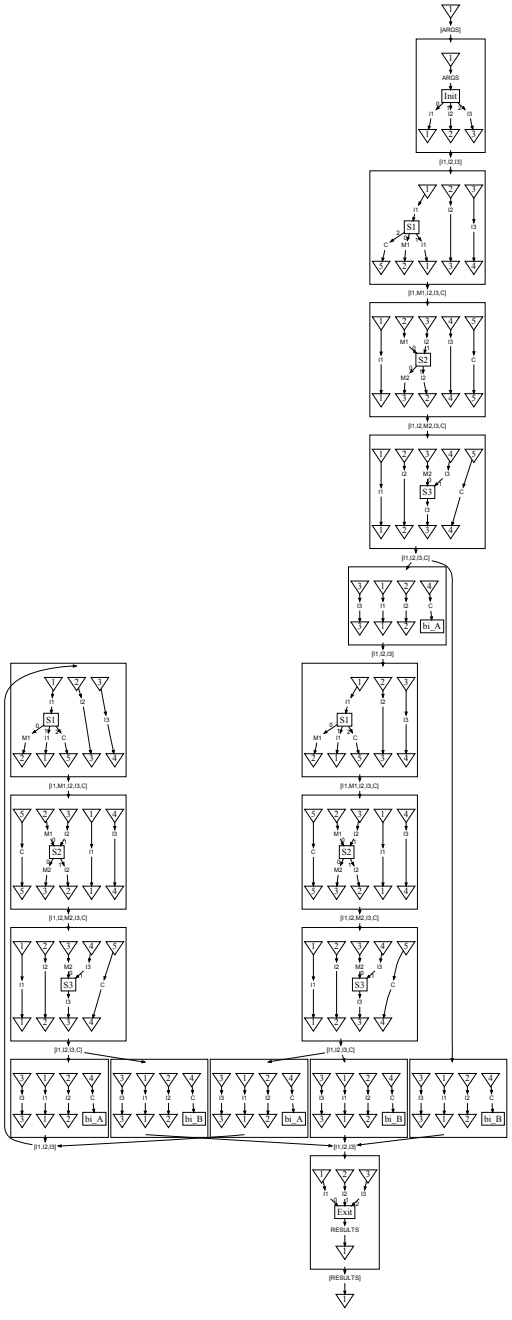
Unroll Loop Twice



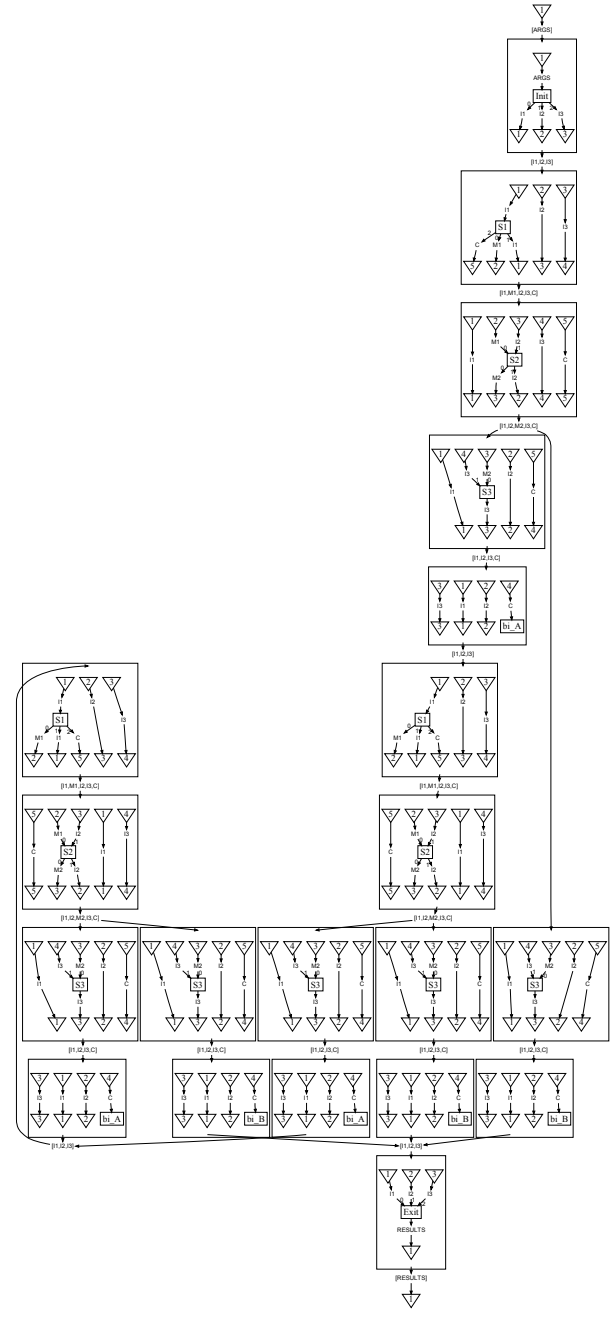
$$\textbf{while}\quad b\quad \textbf{do}S\quad =$$
$$\textbf{if}\ b\ \textbf{then}\ (S; \textbf{while}\ b\ \textbf{do}S)$$



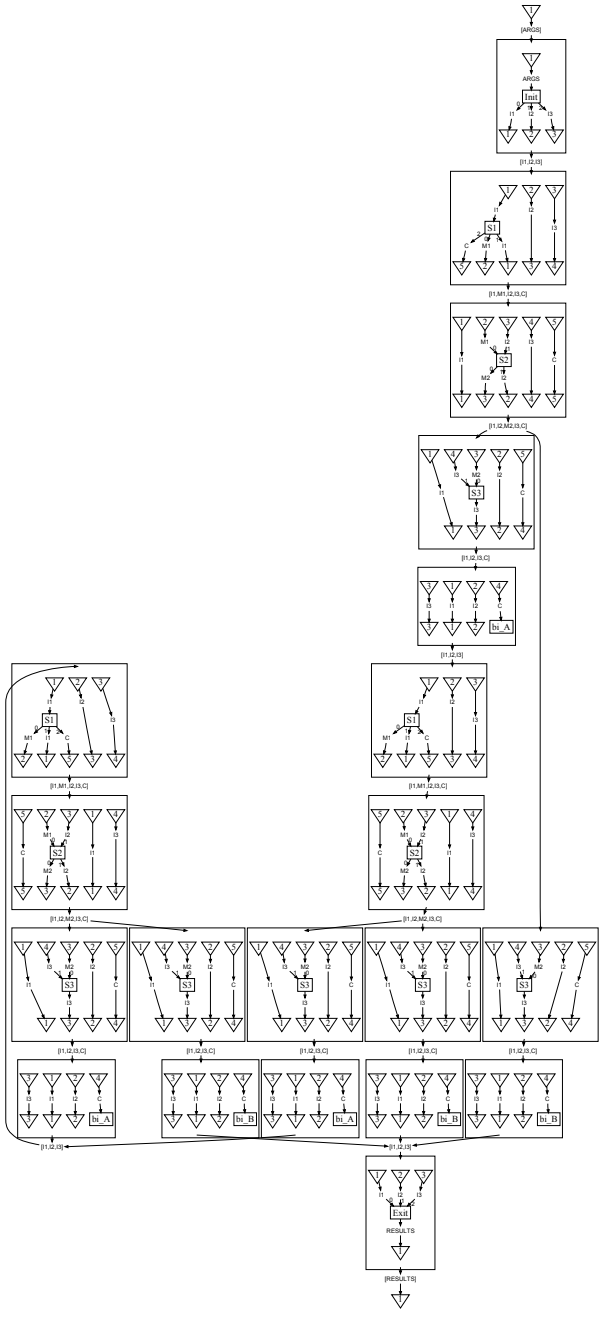
Distribute S3 into Branches



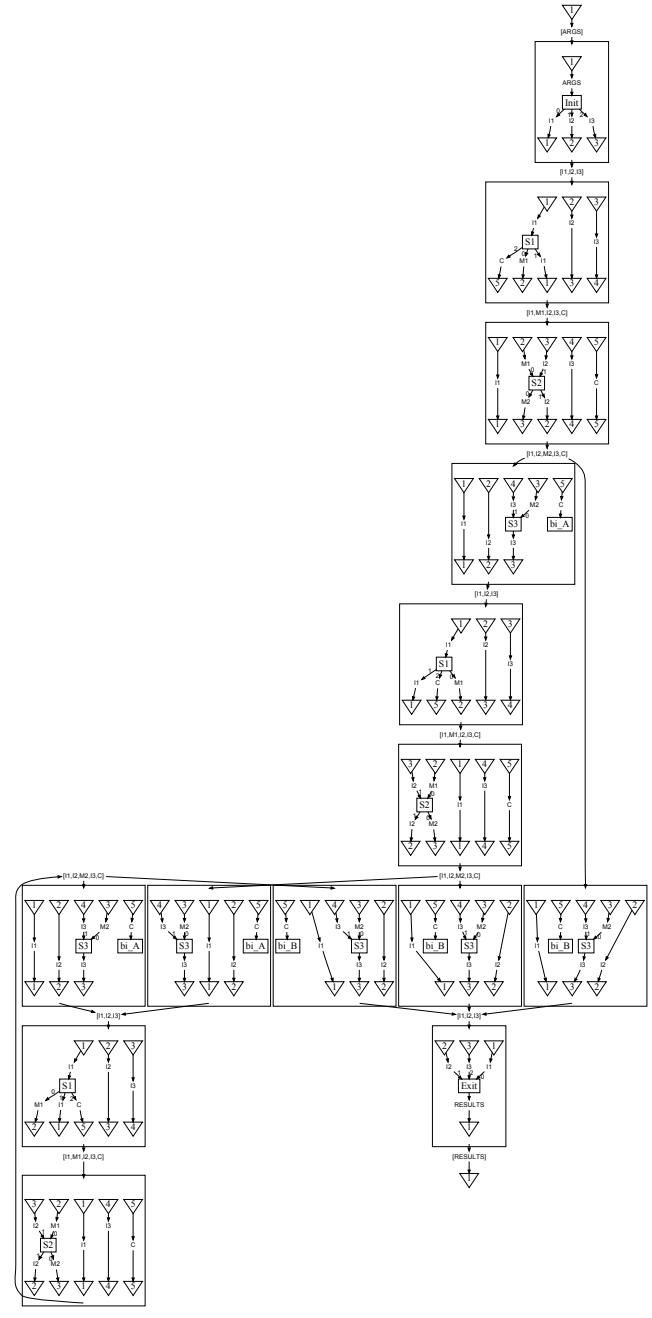
$$Q\,;(R \cup S)$$
$$= Q\,;R \cup Q\,;S$$



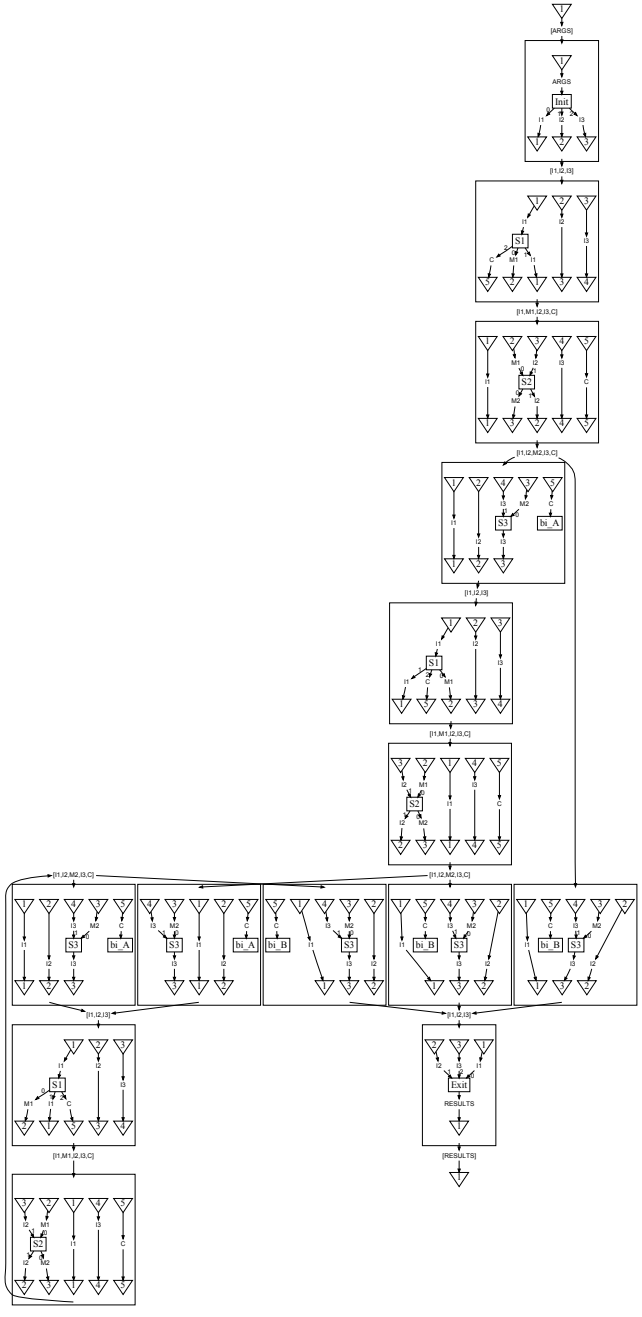
Compose S3 with Branches



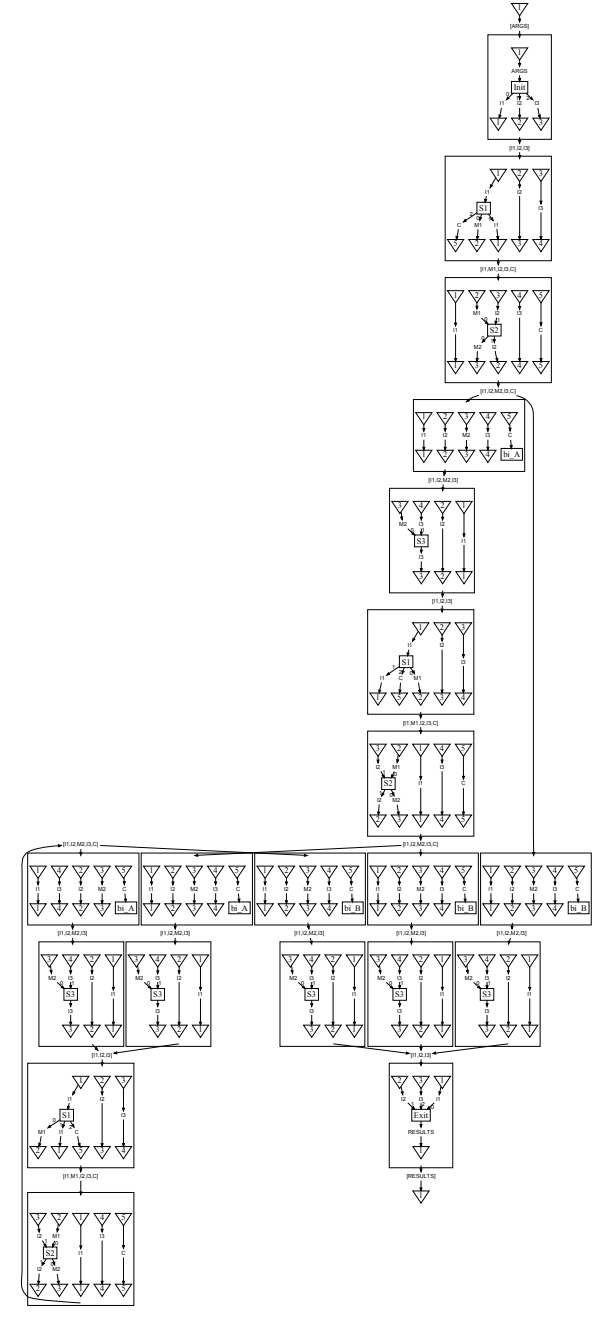
Common composition of both code graph layers



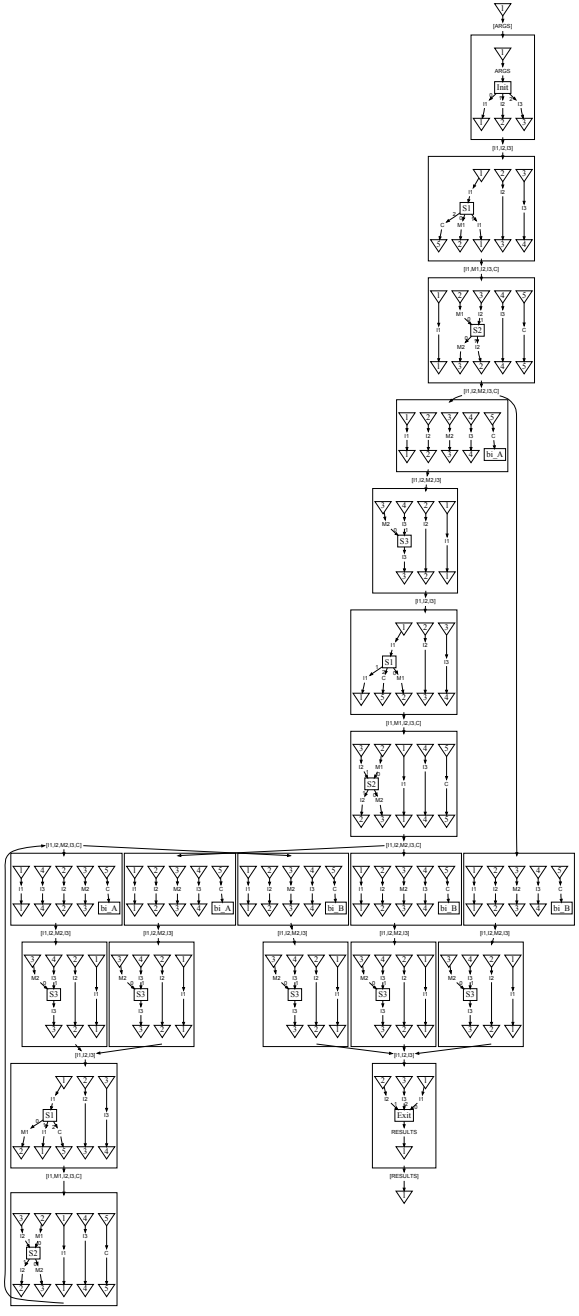
Sequentially Decompse S3 after Branches



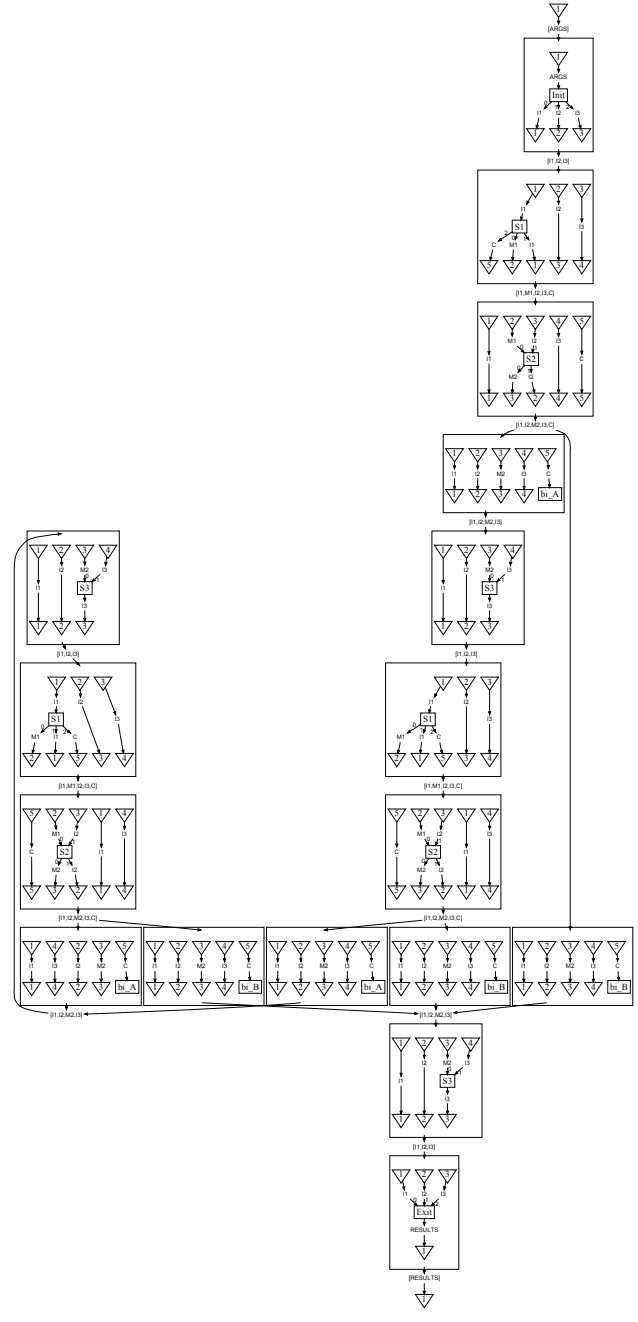
Functoriality of $\otimes$; common composition



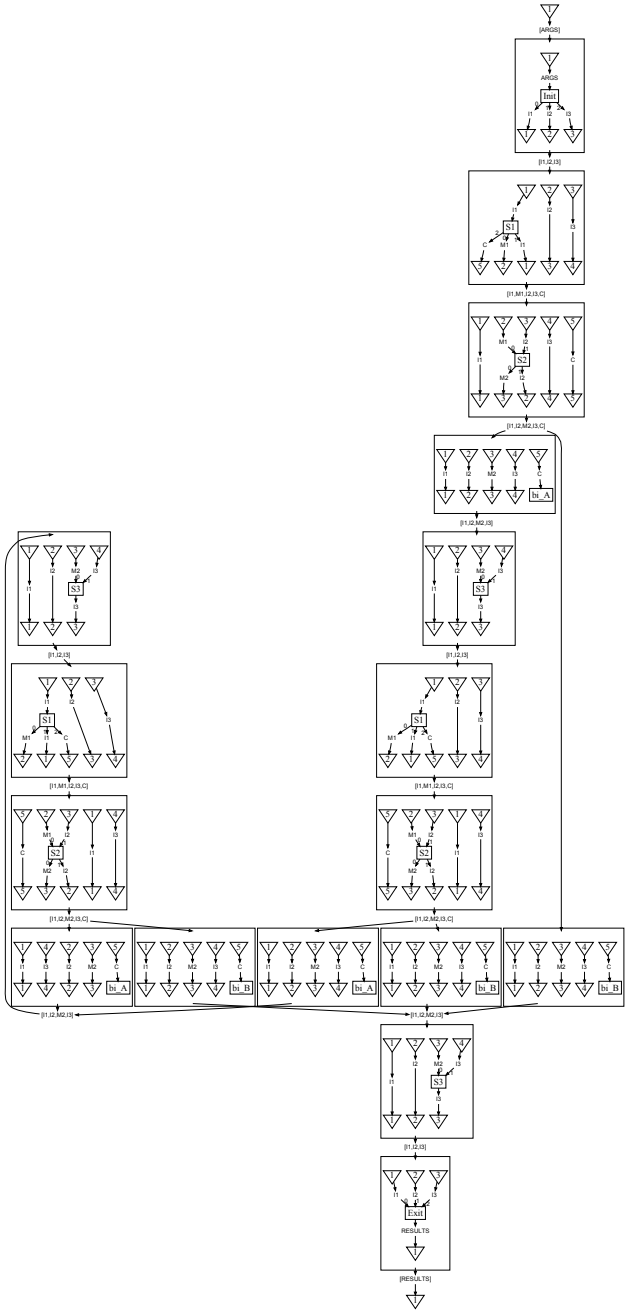
Un-Distribute S3

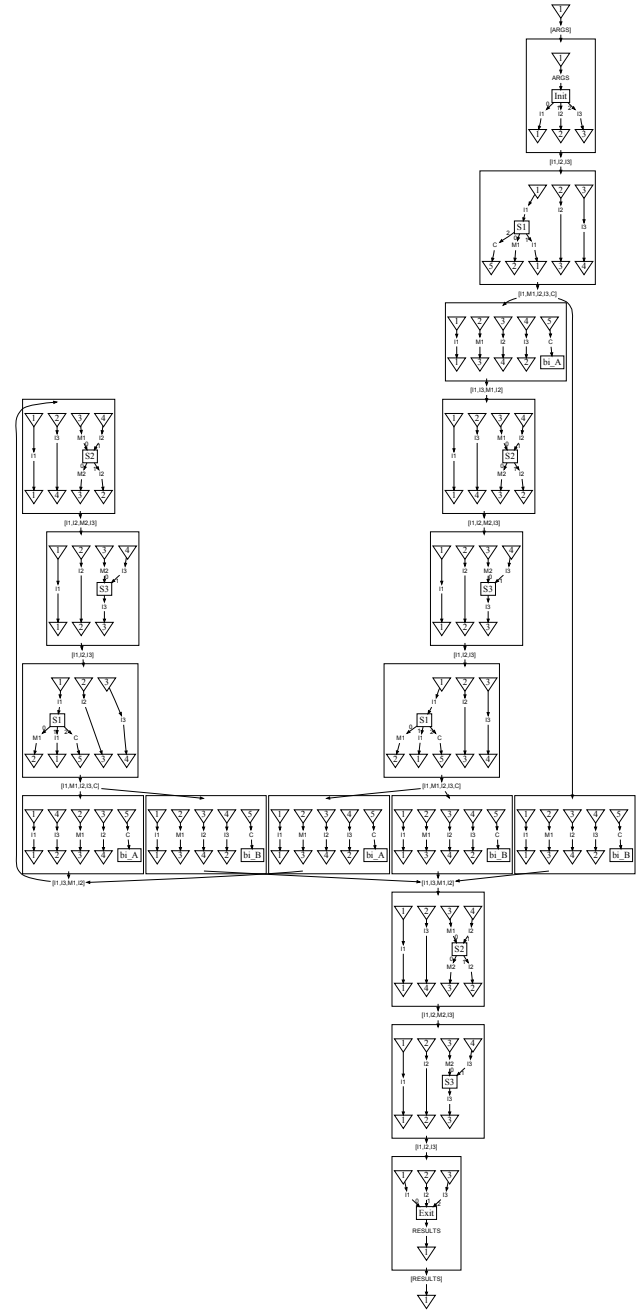


$$Q\,;R \cup Q\,;S$$
$$= Q\,;(R \cup S)$$

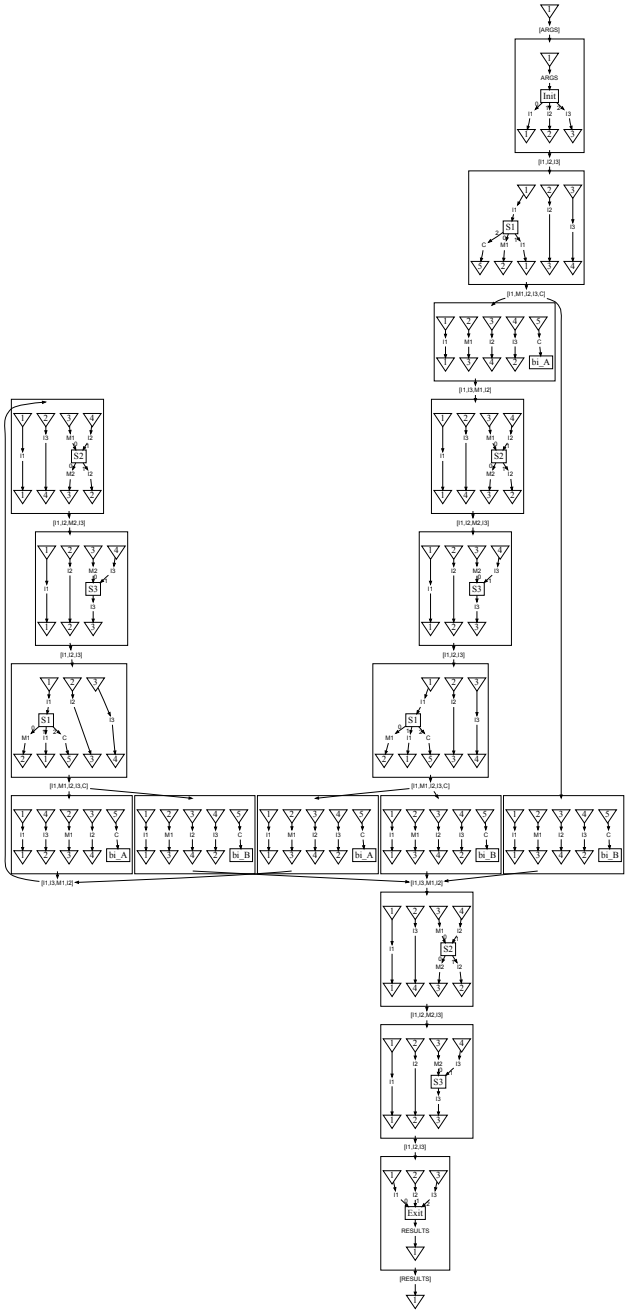


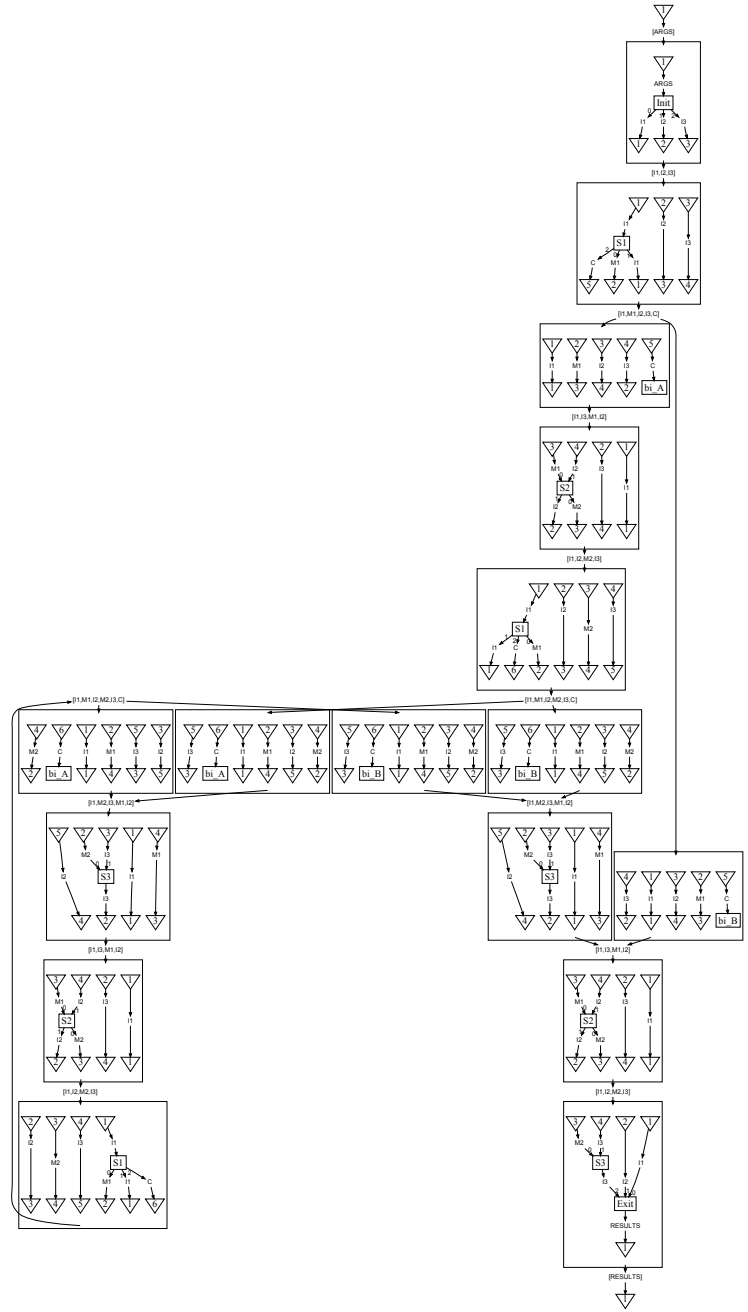
Move S2 over Branches Analogously



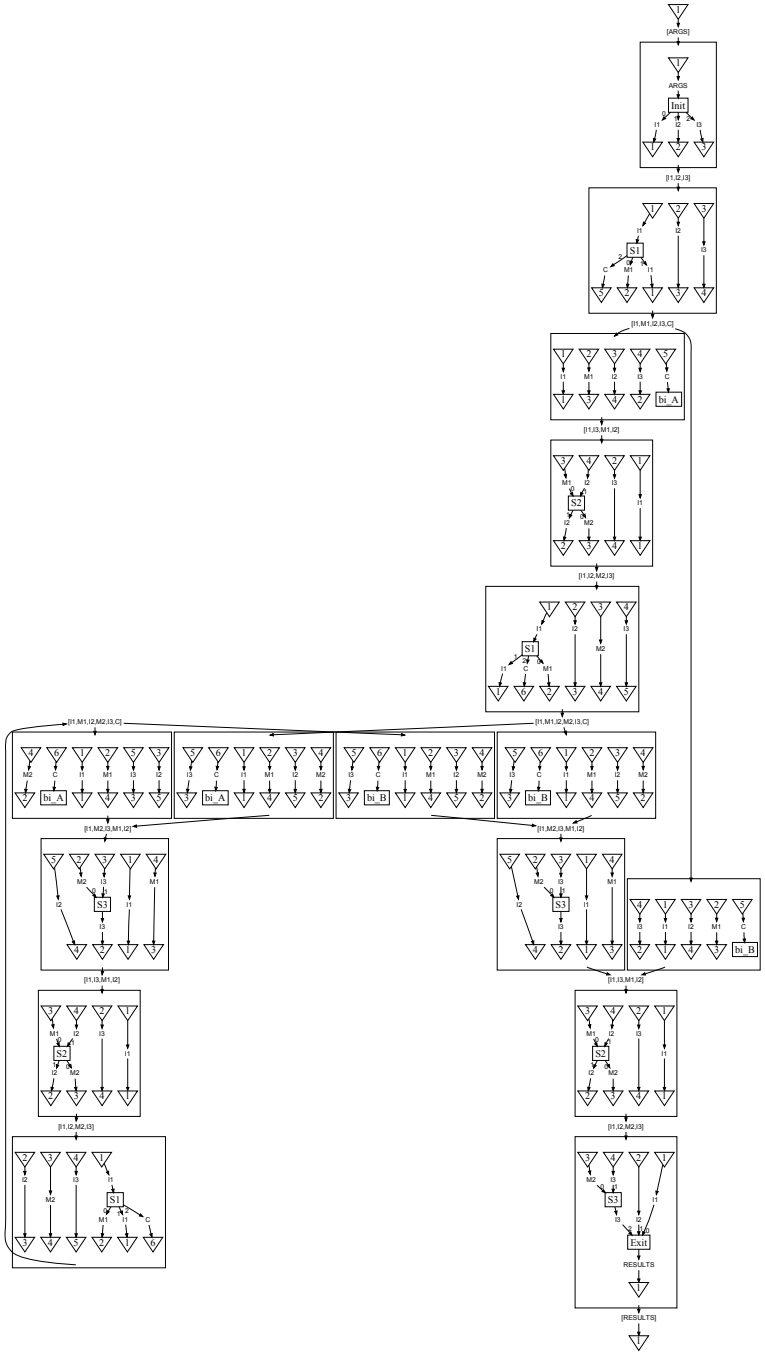


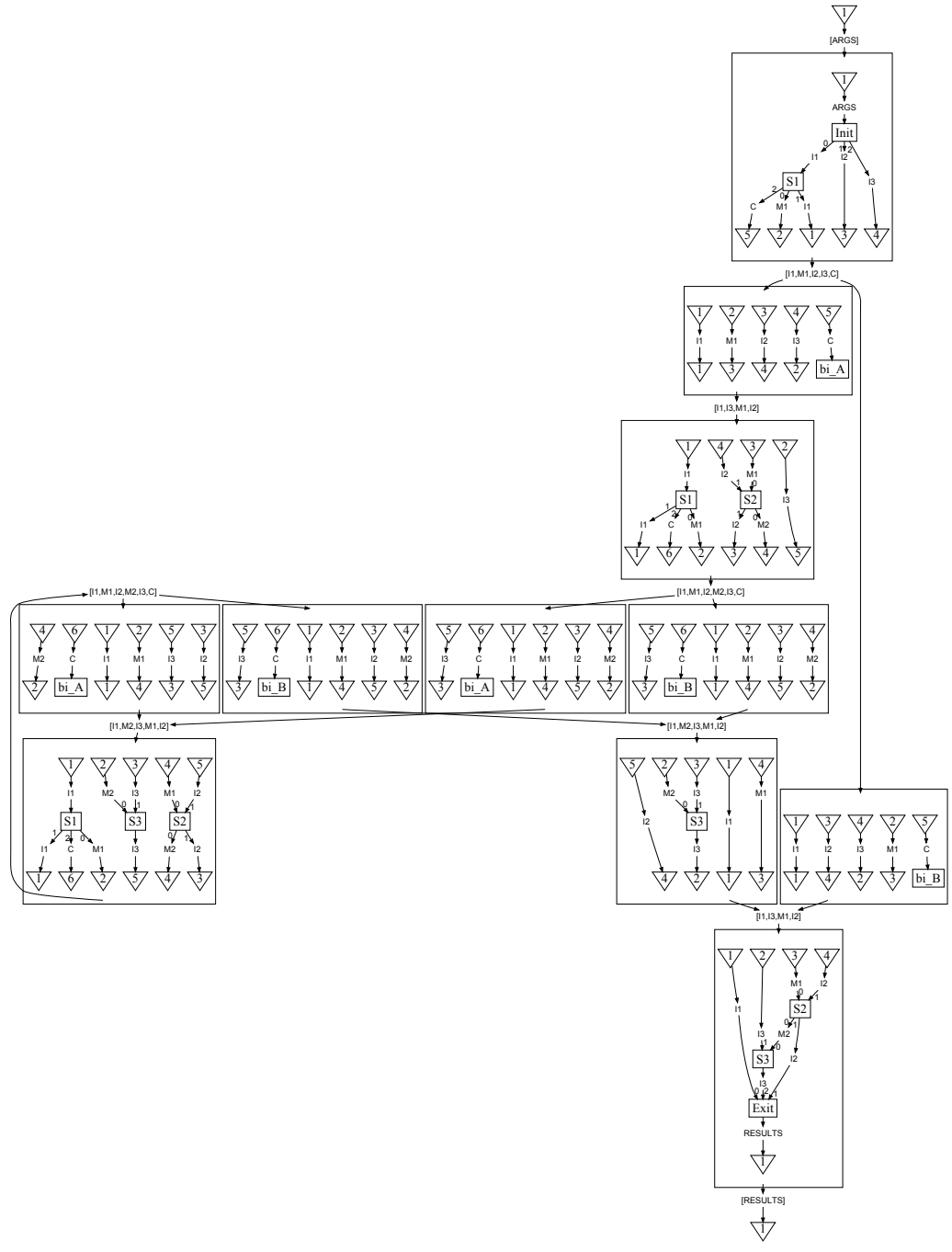
Move S3 Forward Again



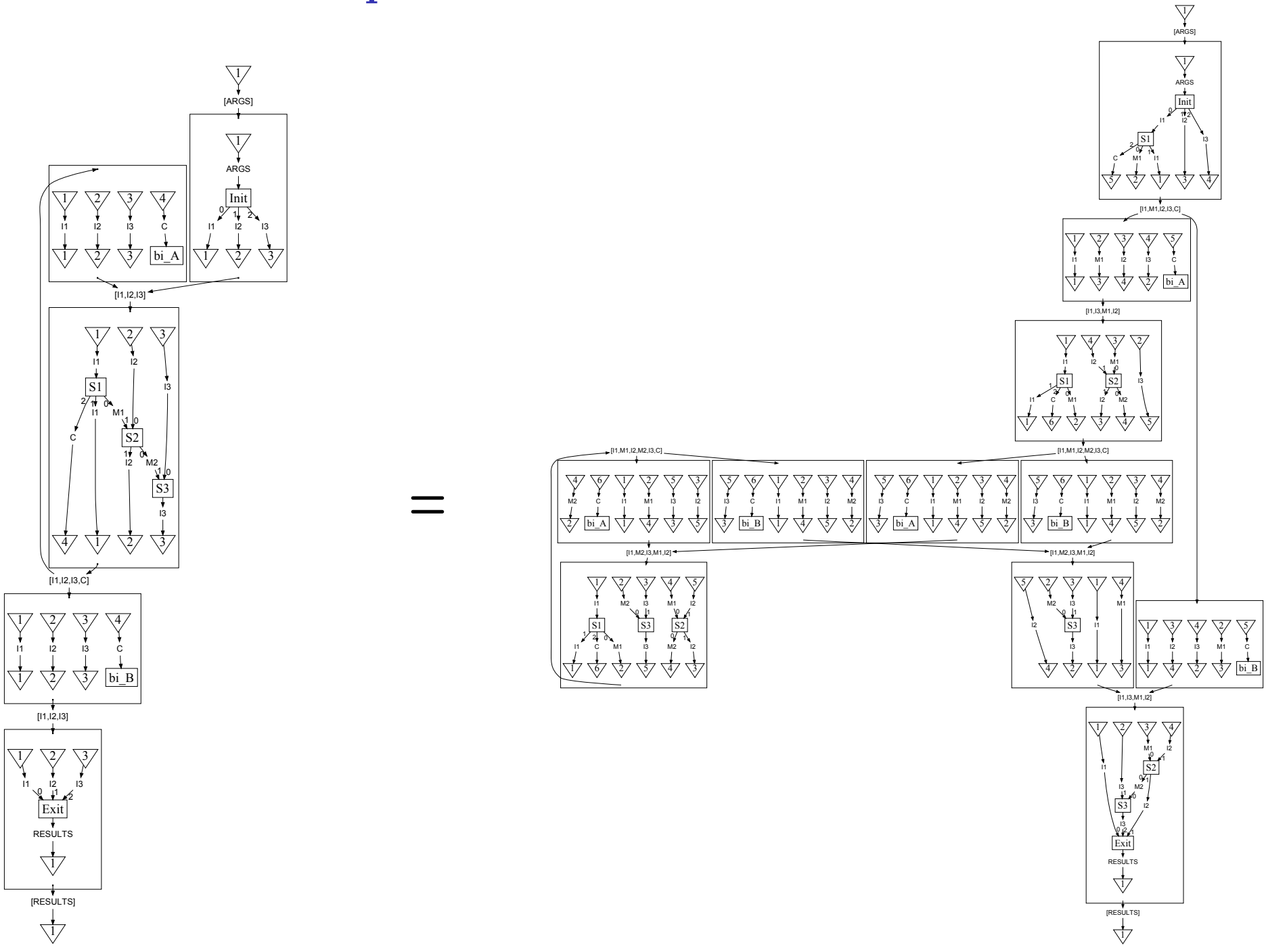


Merge Consecutive Straight-Line Code

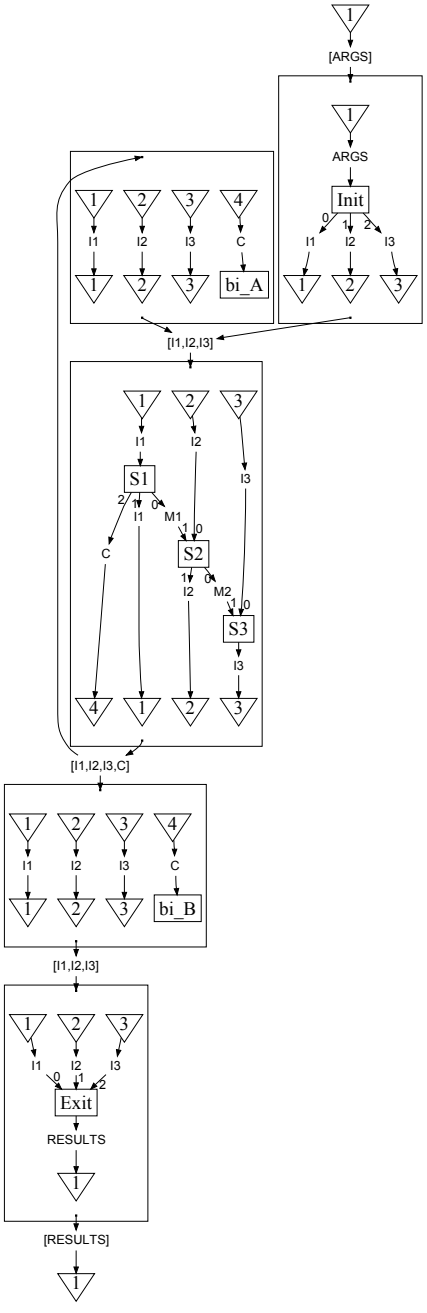




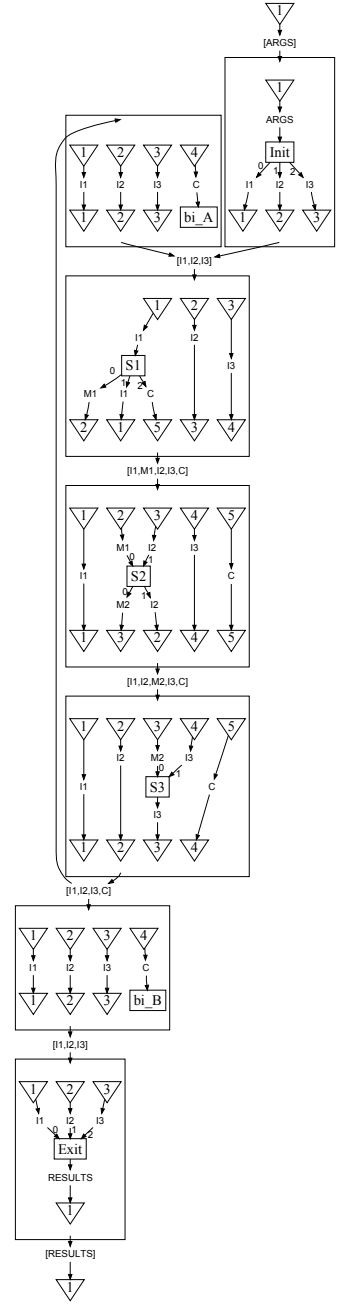
Complete Staging Transformation



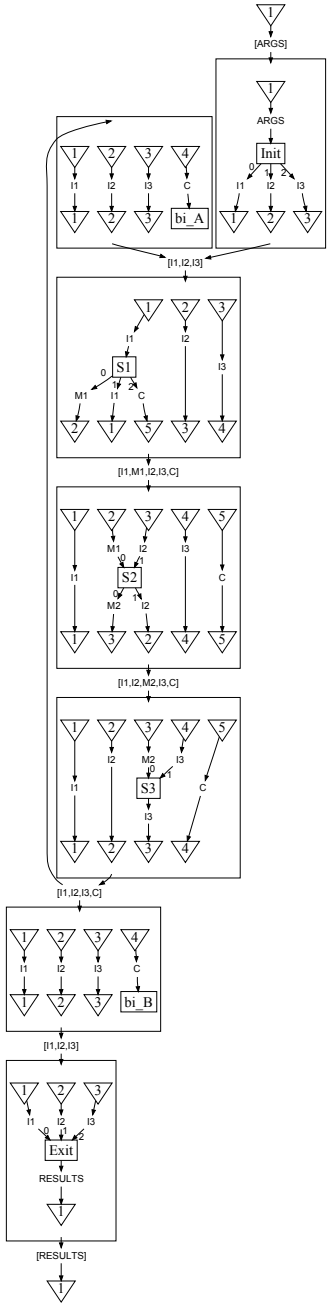
Sequentially Decompose Staged Body



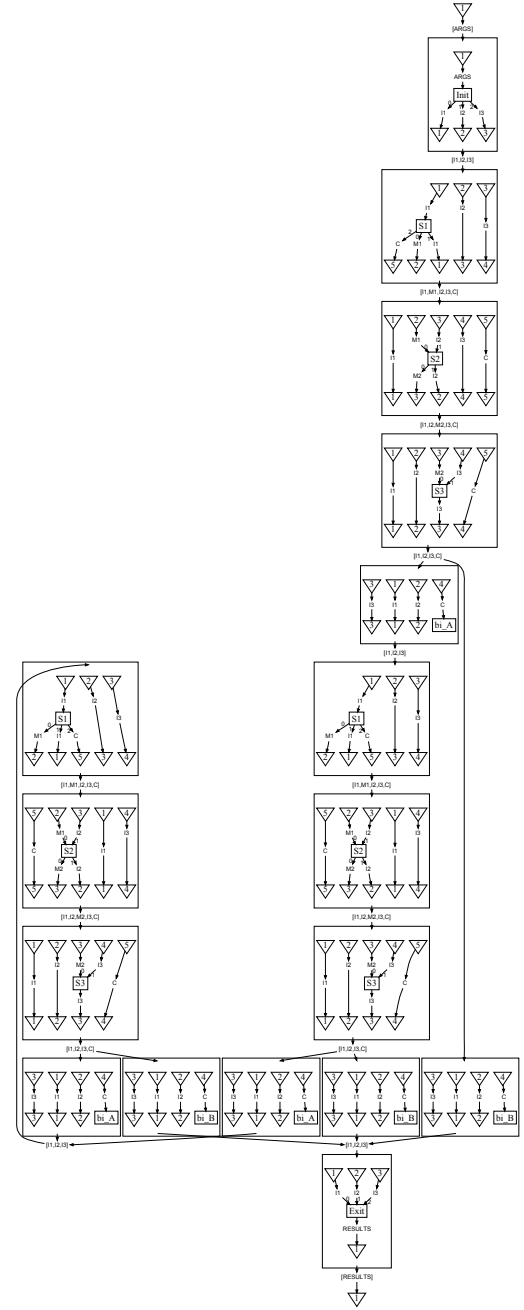
Common composition of both code graph layers



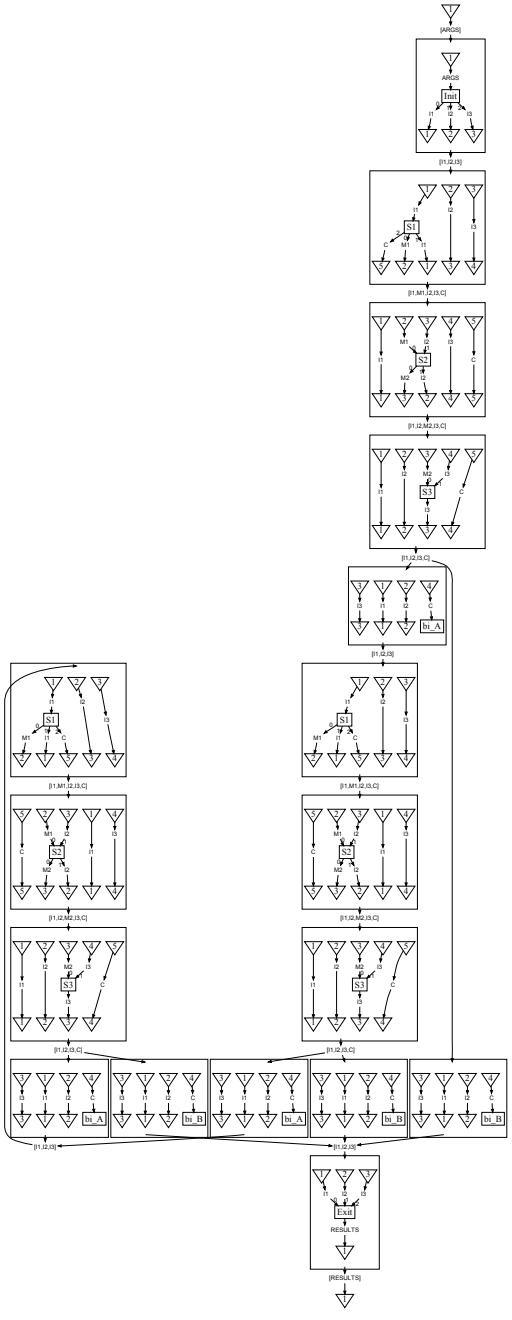
Unroll Loop Twice



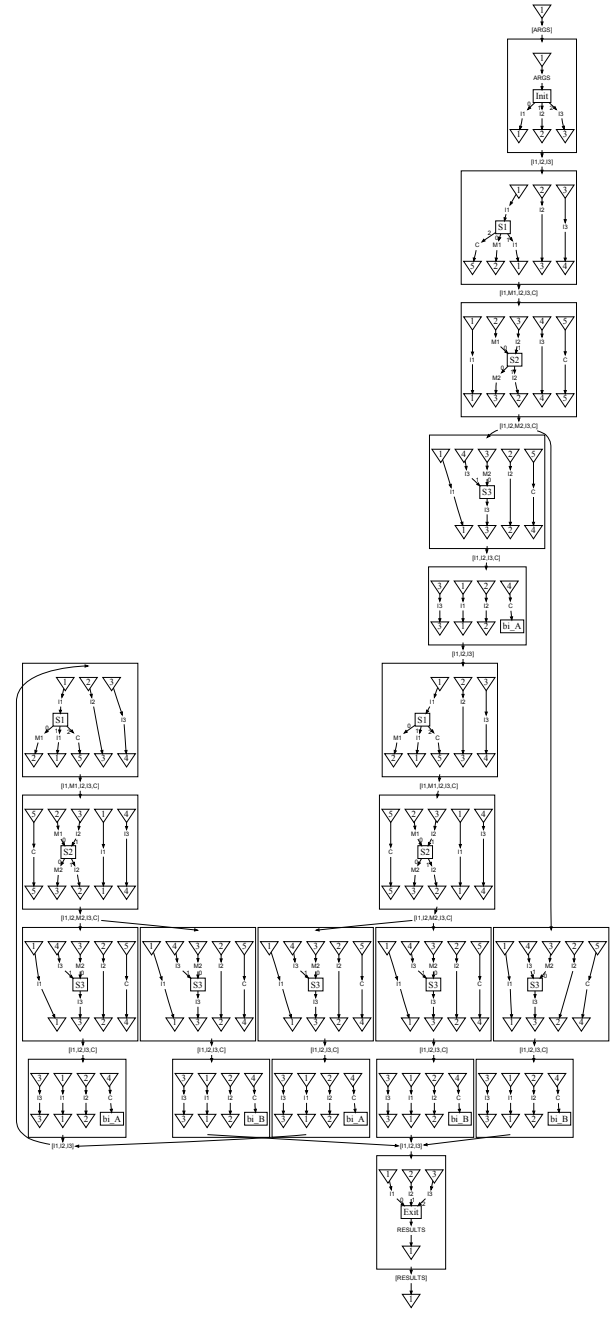
$$\textbf{while}\quad b\quad \textbf{do}S\quad =$$
$$\textbf{if}\ b\ \textbf{then}\ (S; \textbf{while}\ b\ \textbf{do}S)$$



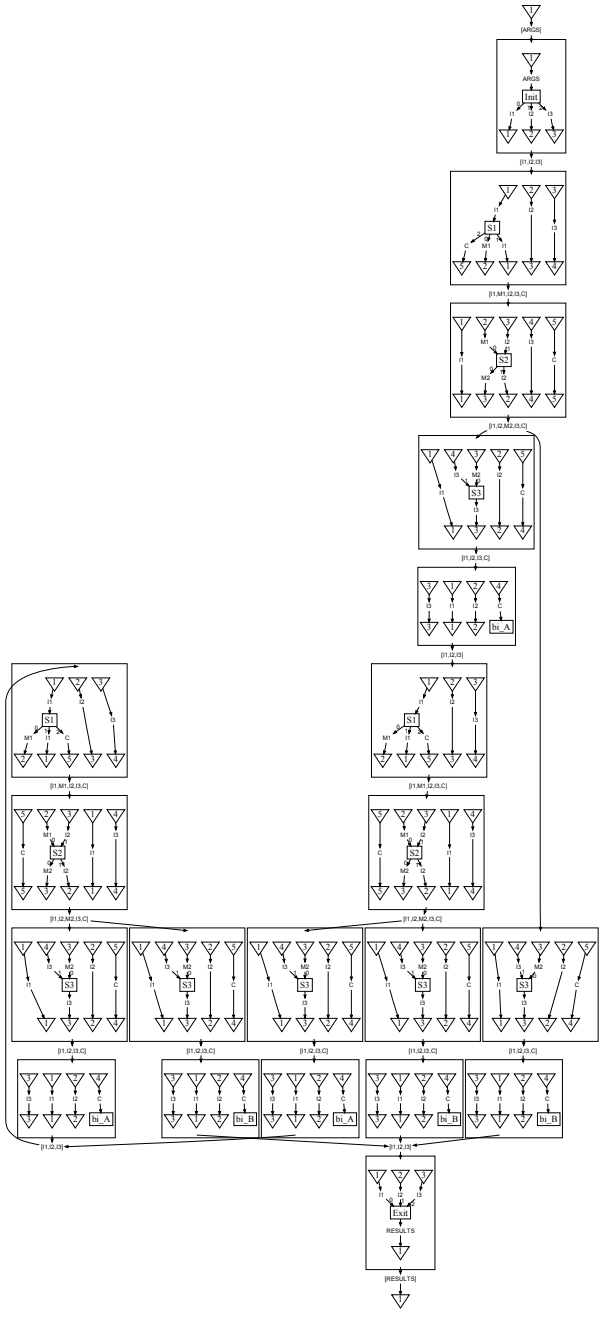
Distribute S3 into Branches



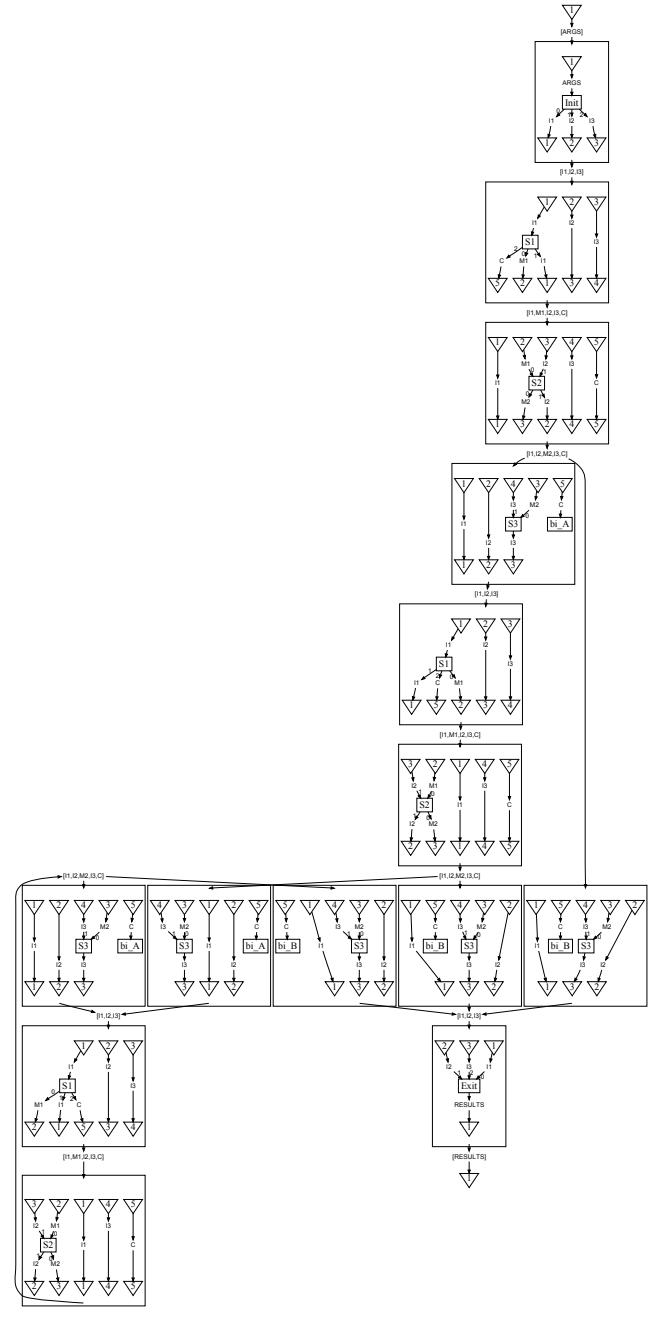
$$Q\,;(R \cup S)$$
$$= Q\,;R \cup Q\,;S$$



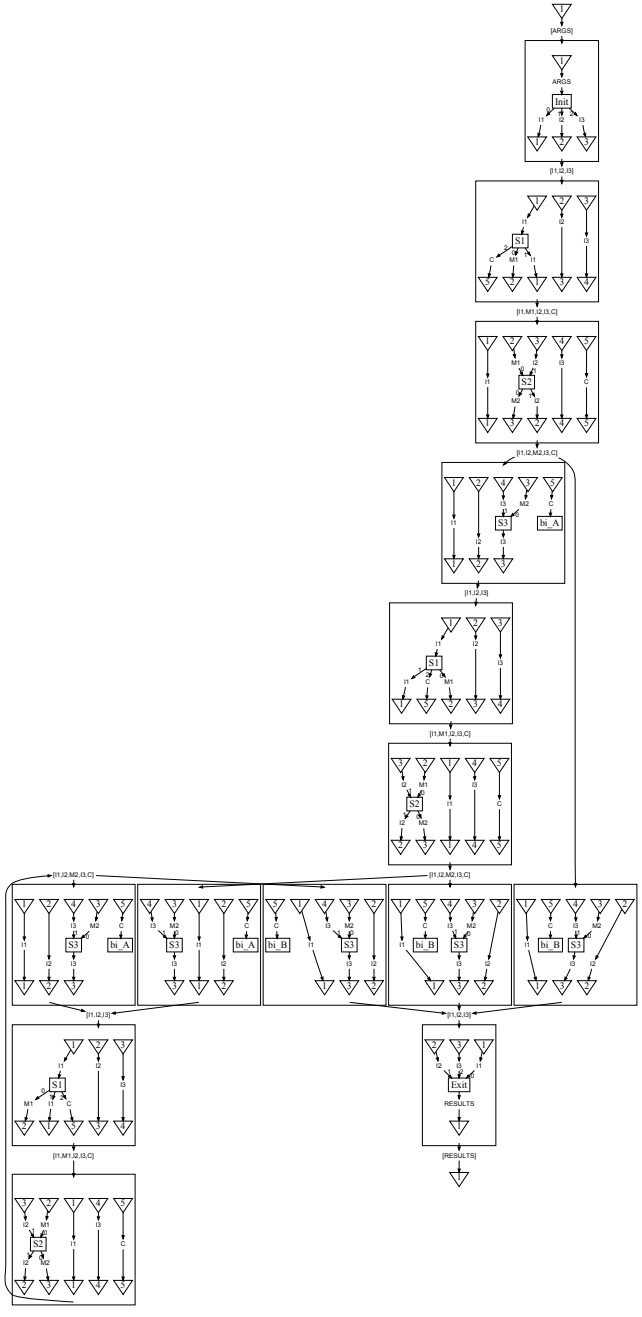
Compose S3 with Branches



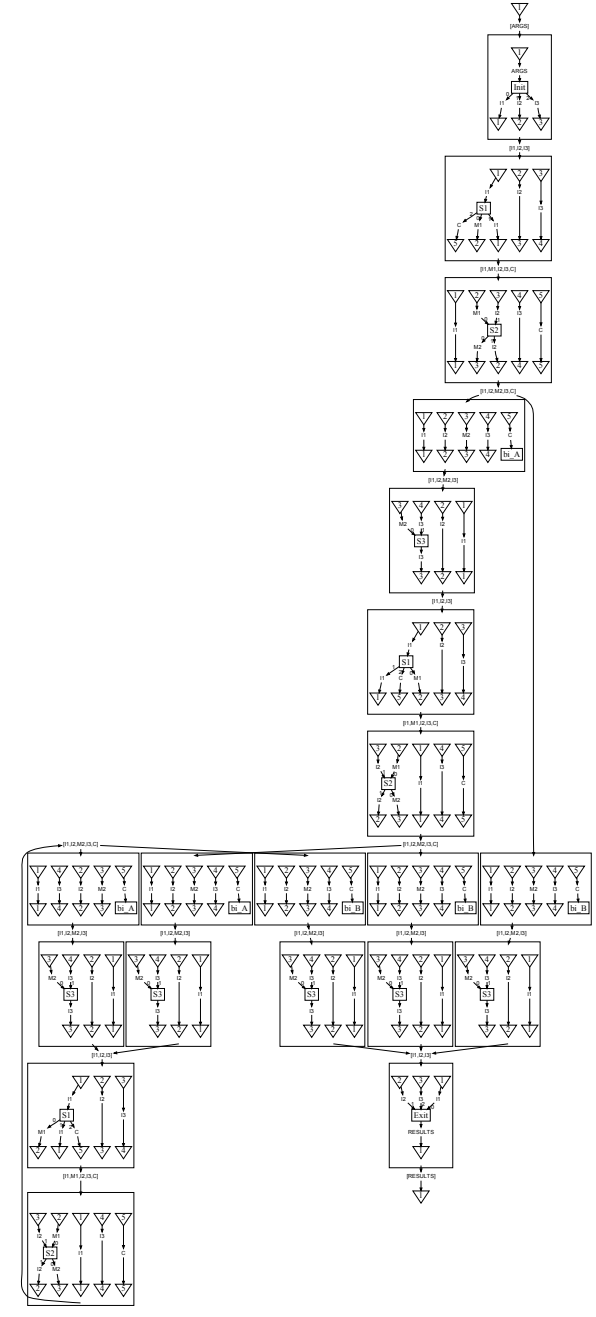
Common composition of both code graph layers



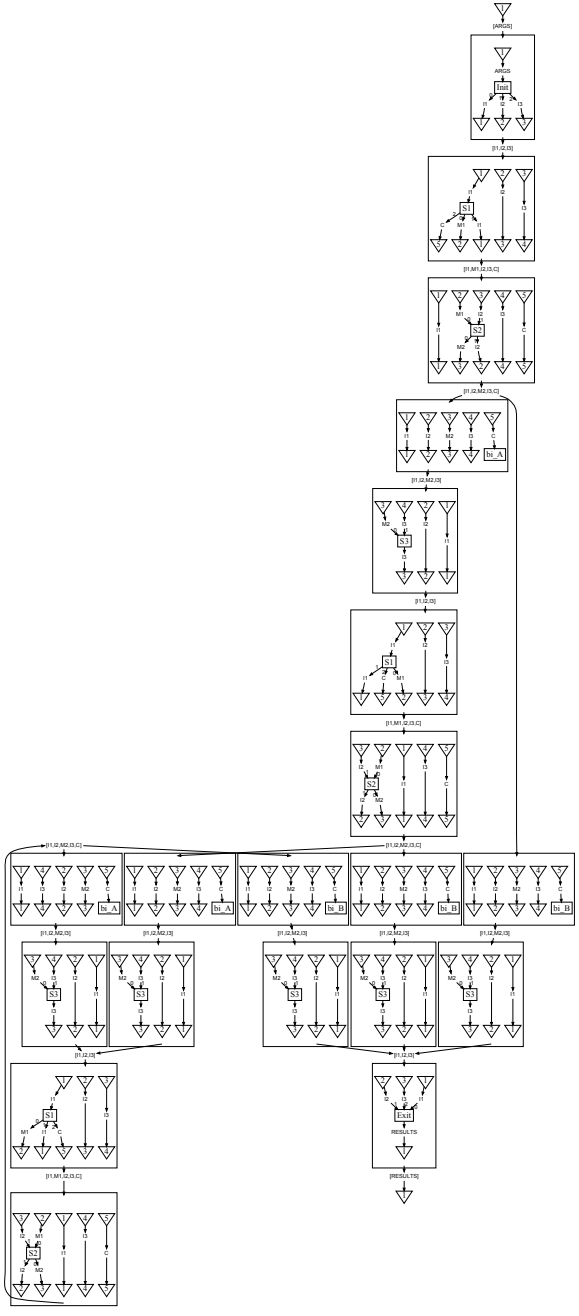
Sequentially Decompse S3 after Branches



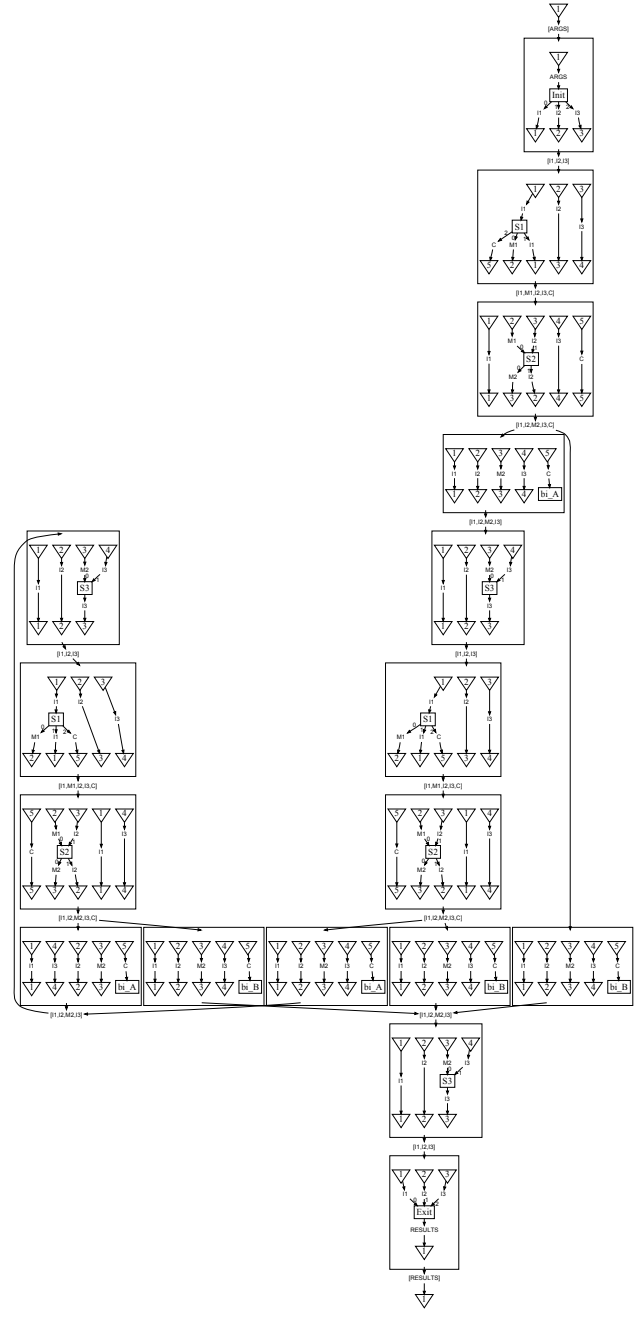
Functoriality of $\otimes$; common composition



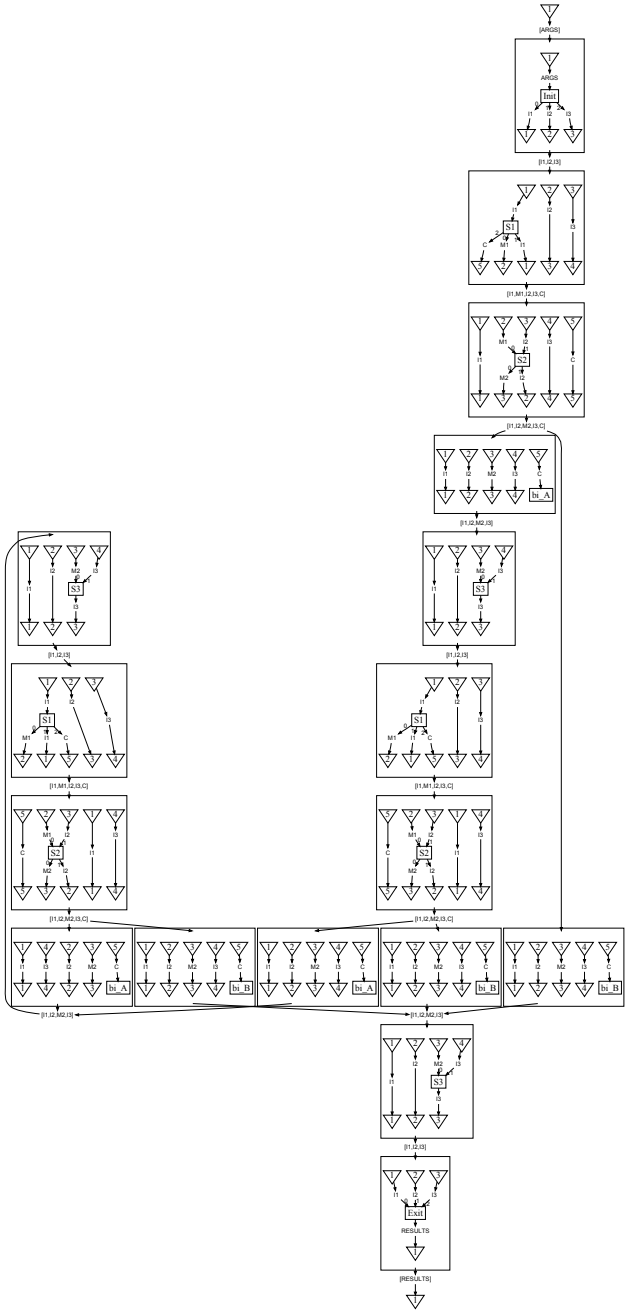
Un-Distribute S3

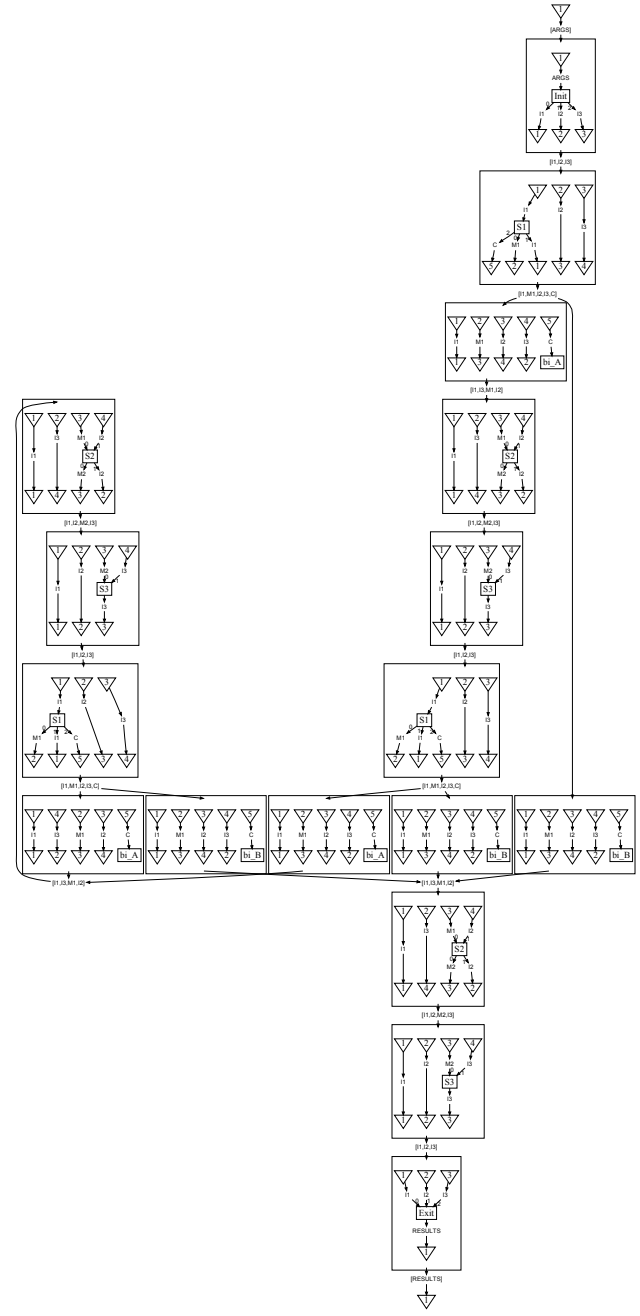


$$Q\,;R \cup Q\,;S$$
$$= Q\,;(R \cup S)$$

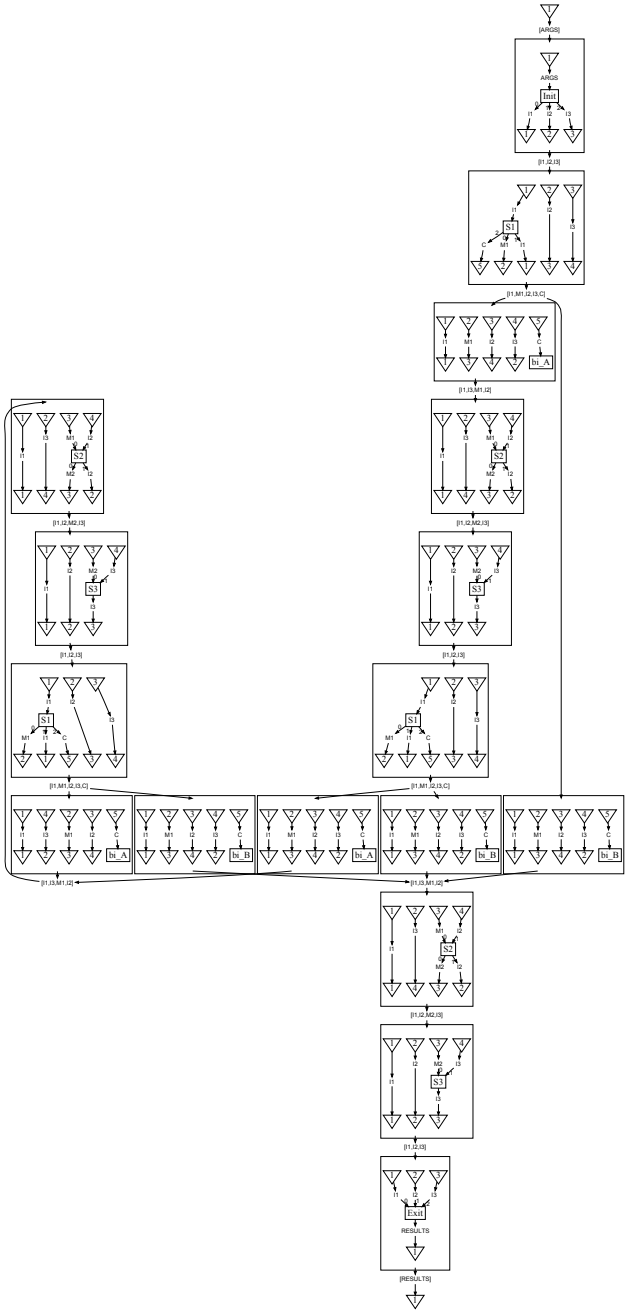


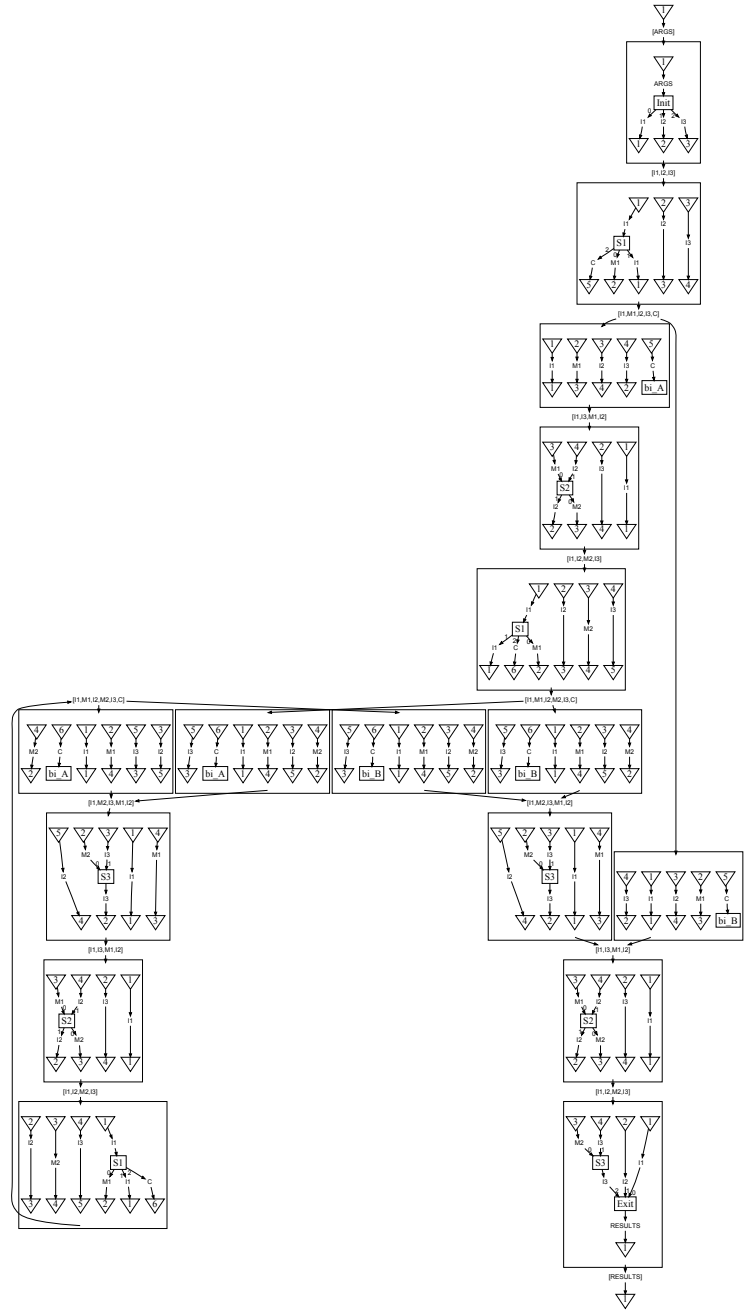
Move S2 over Branches Analogously



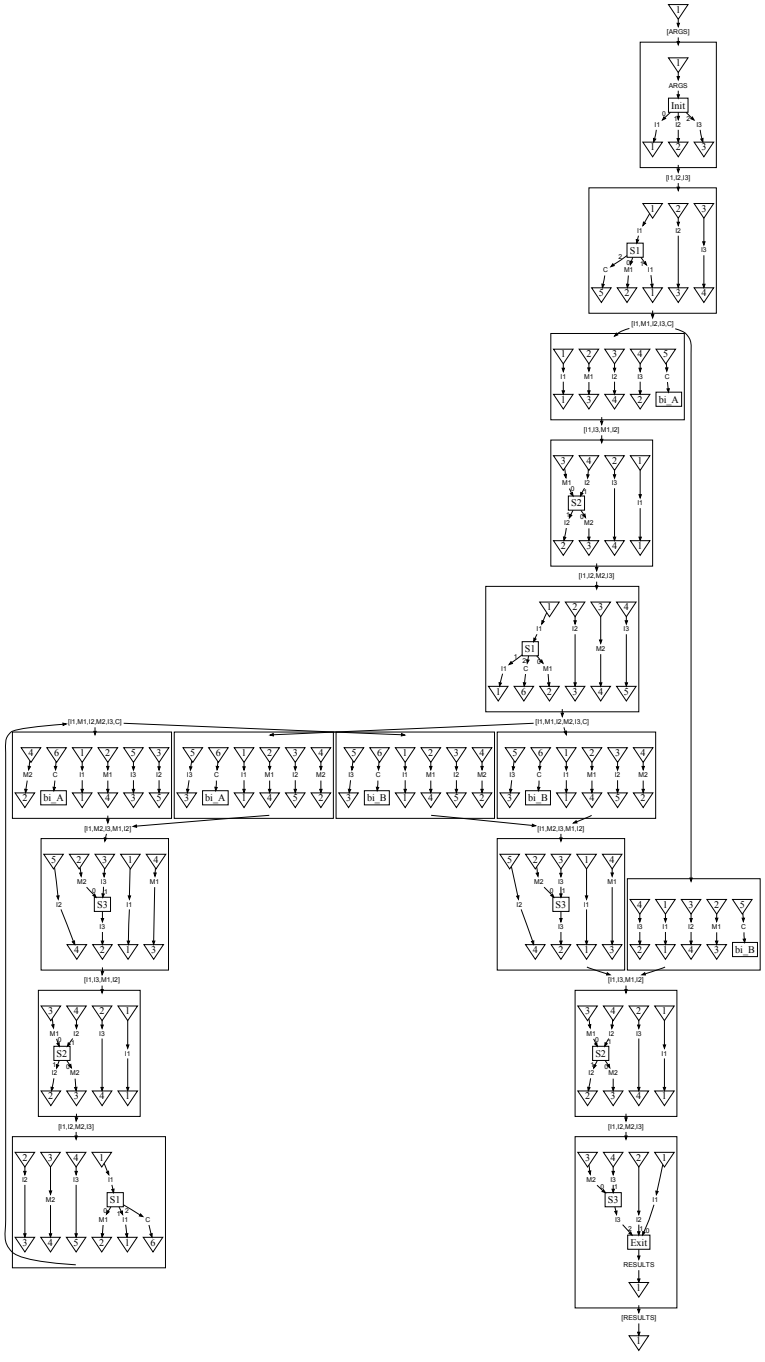


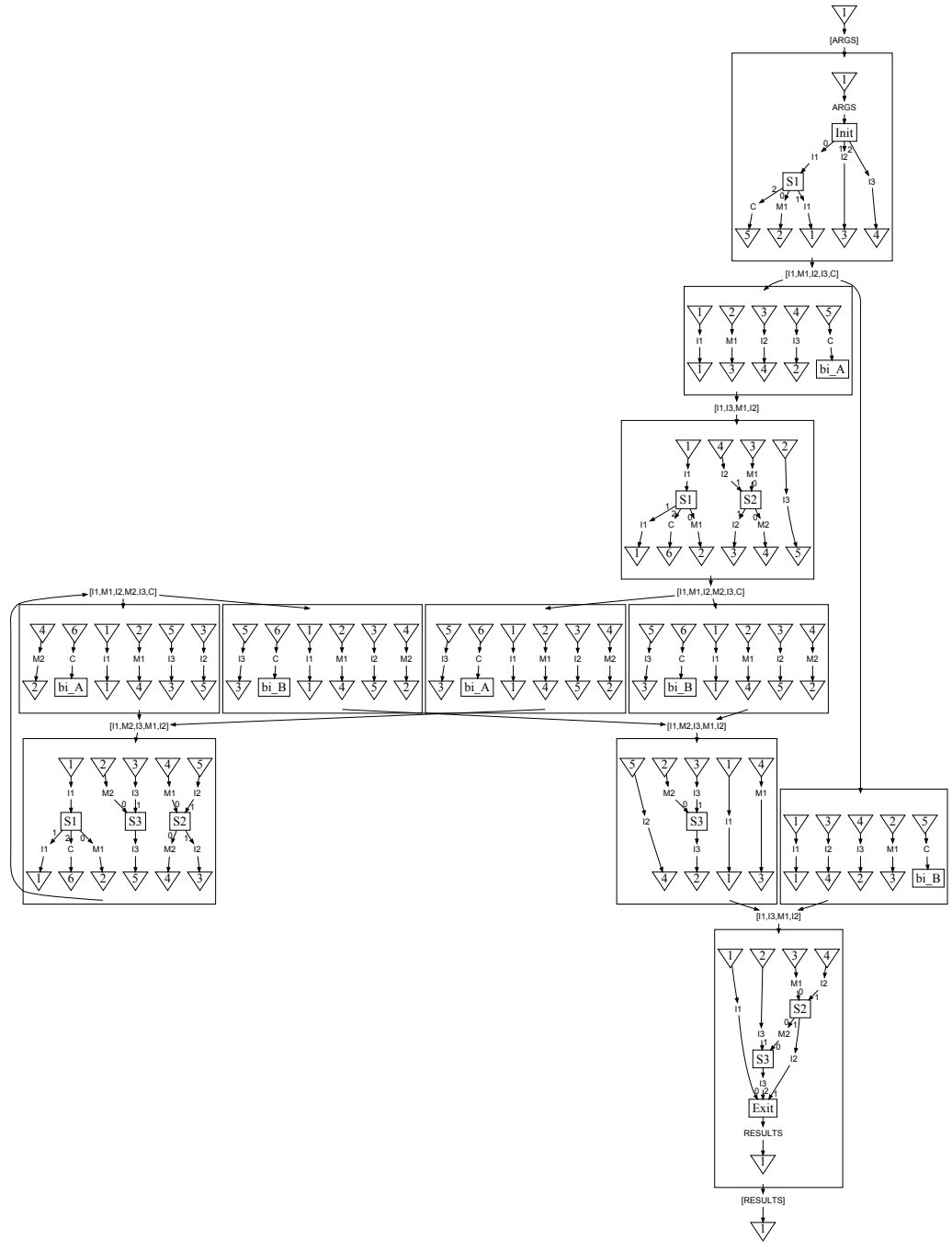
Move S3 Forward Again



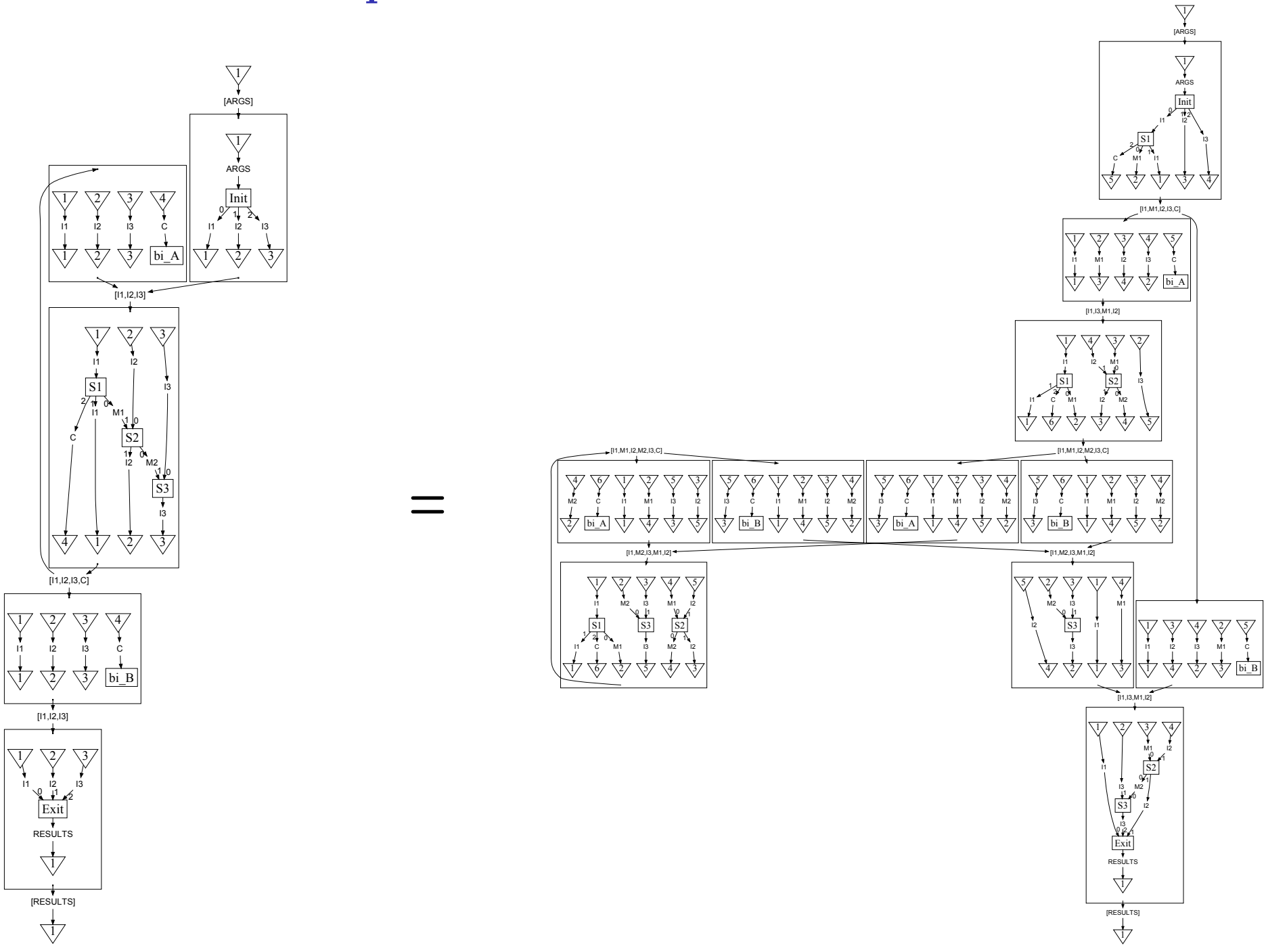


Merge Consecutive Straight-Line Code





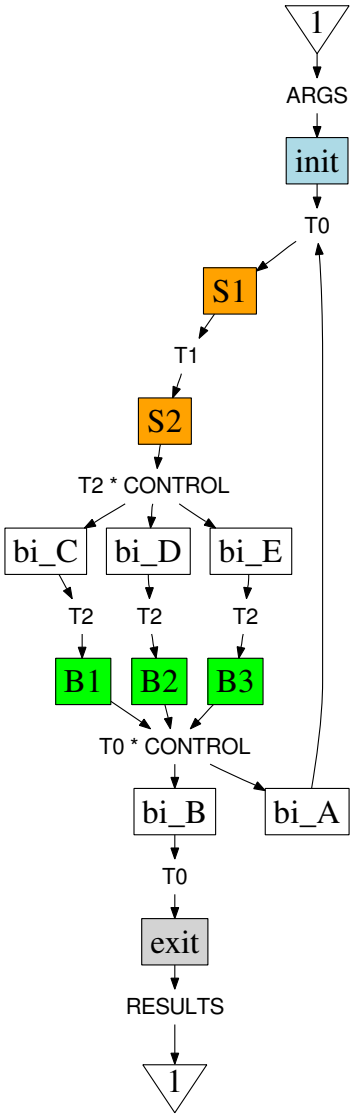
Complete Staging Transformation

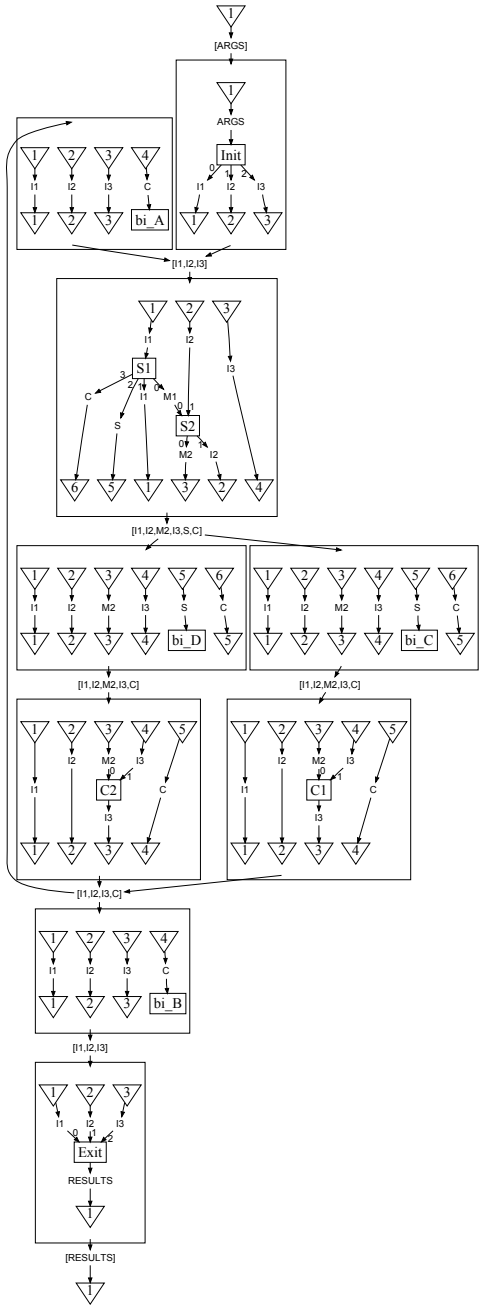


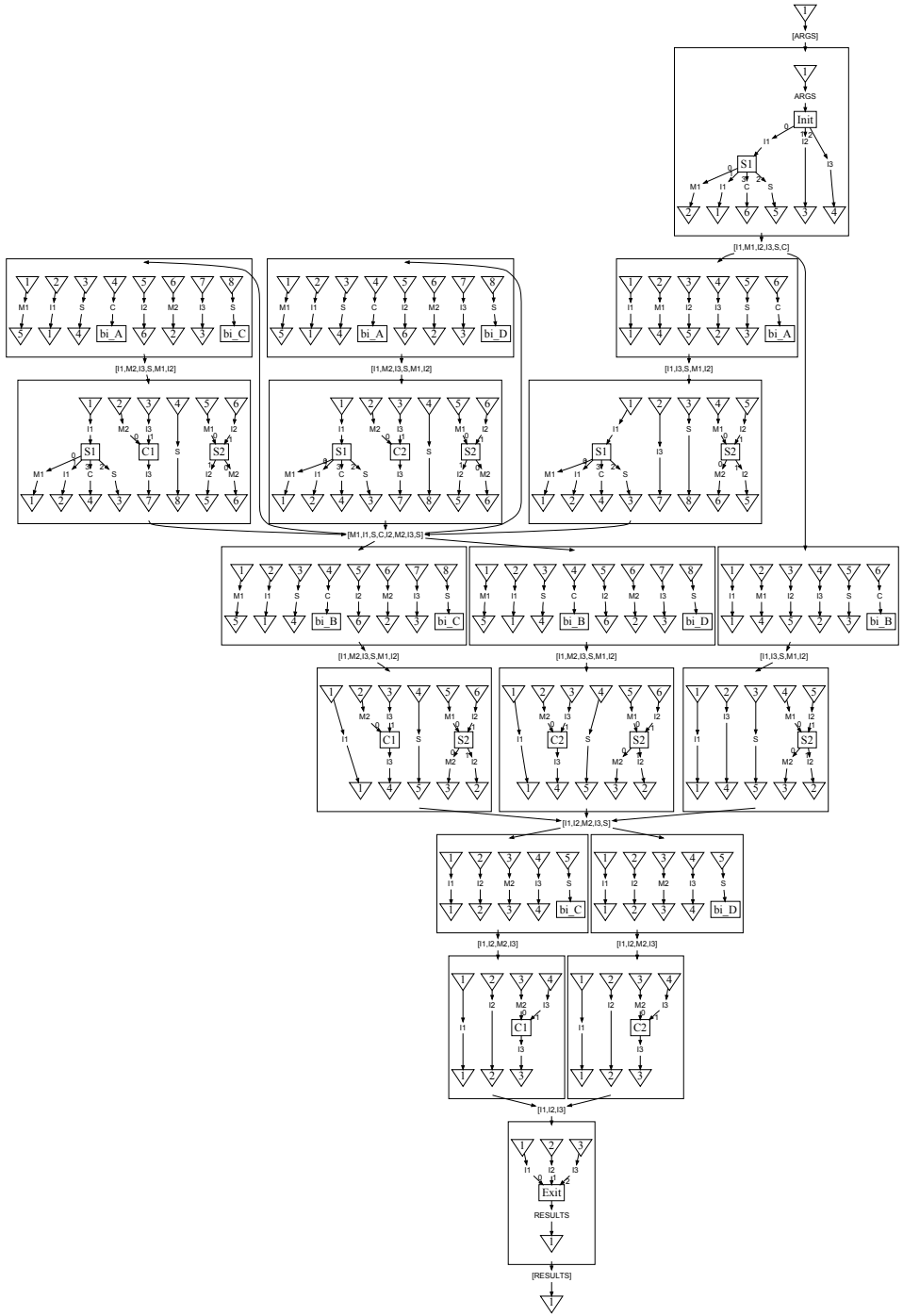
Software Pipelining by Calculation

- Simple loops — similar to modulo scheduling

Variant: Staging with Branch Inside Loop Body



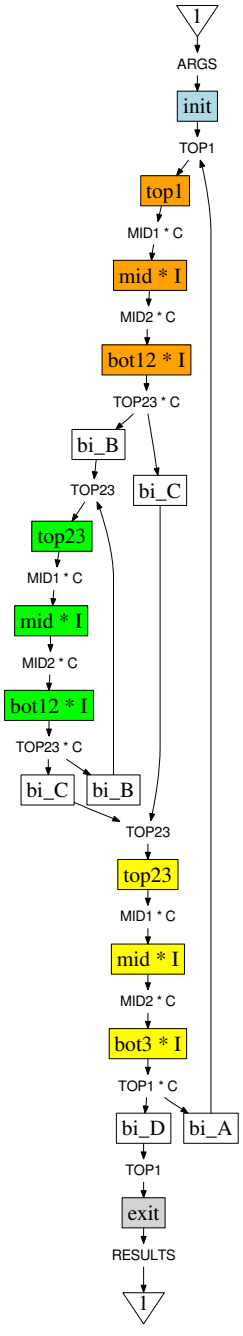


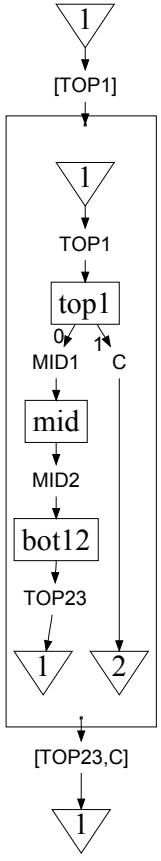


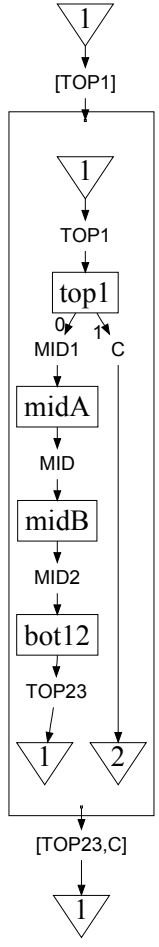
Software Pipelining by Calculation

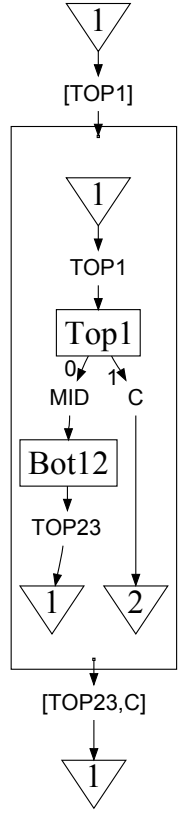
- Simple loops — similar to modulo scheduling
- Multi-way switch inside loop body

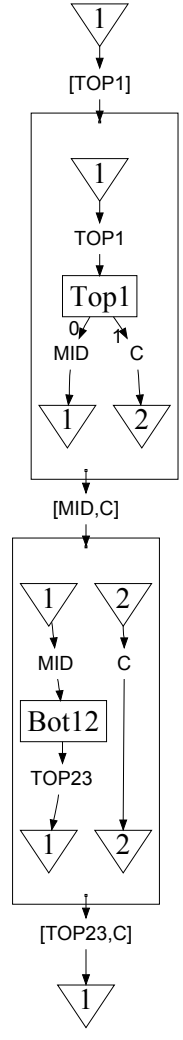
Control-Flow Rearrangement: Matrix Mult.

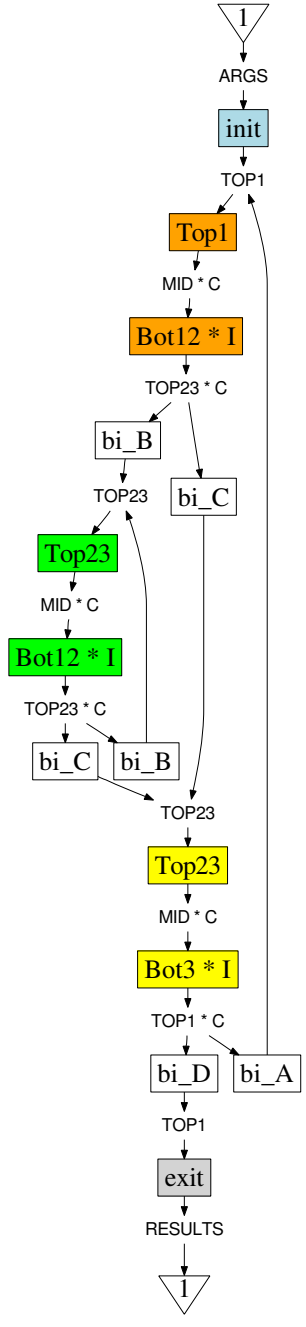




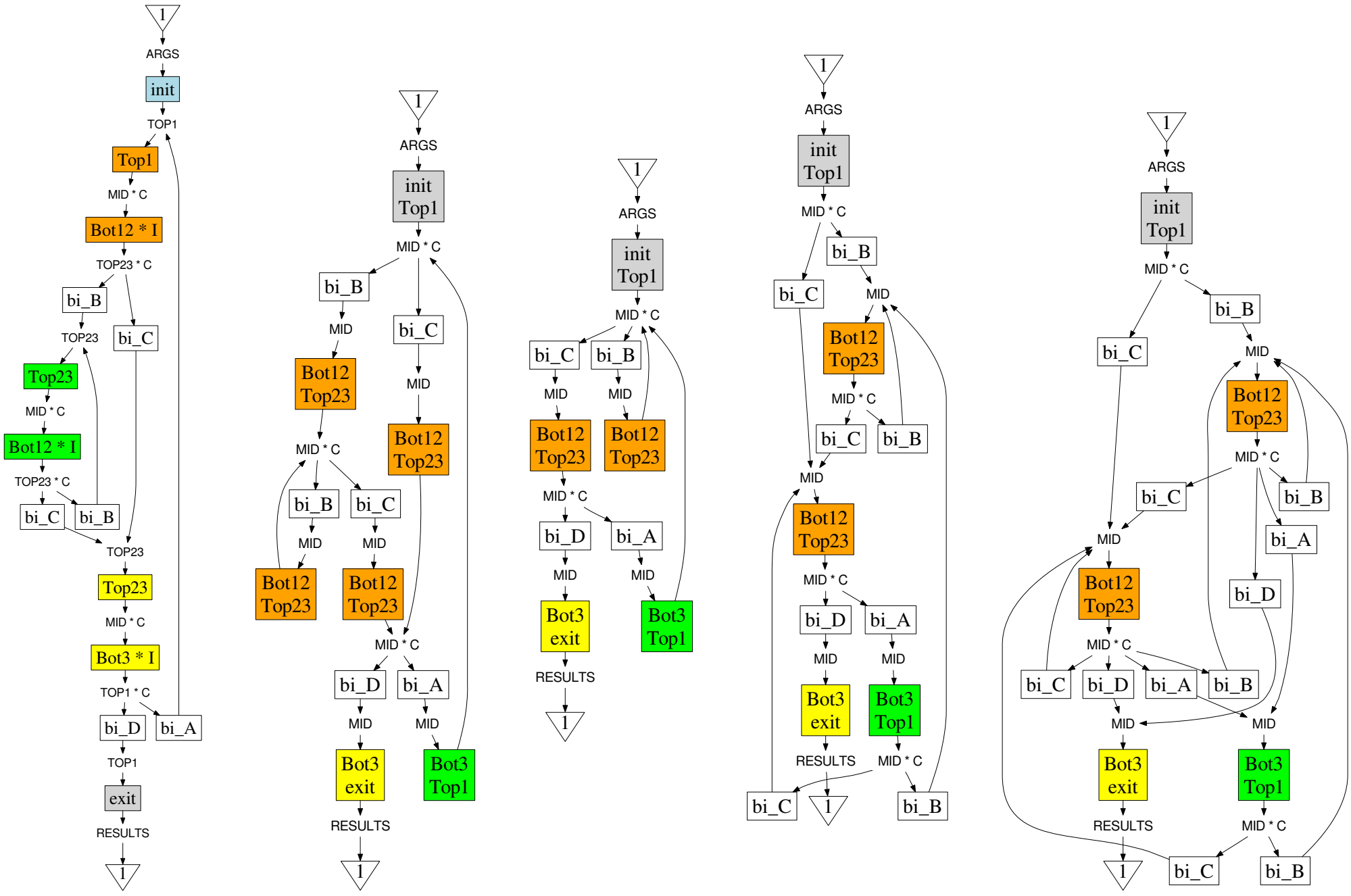




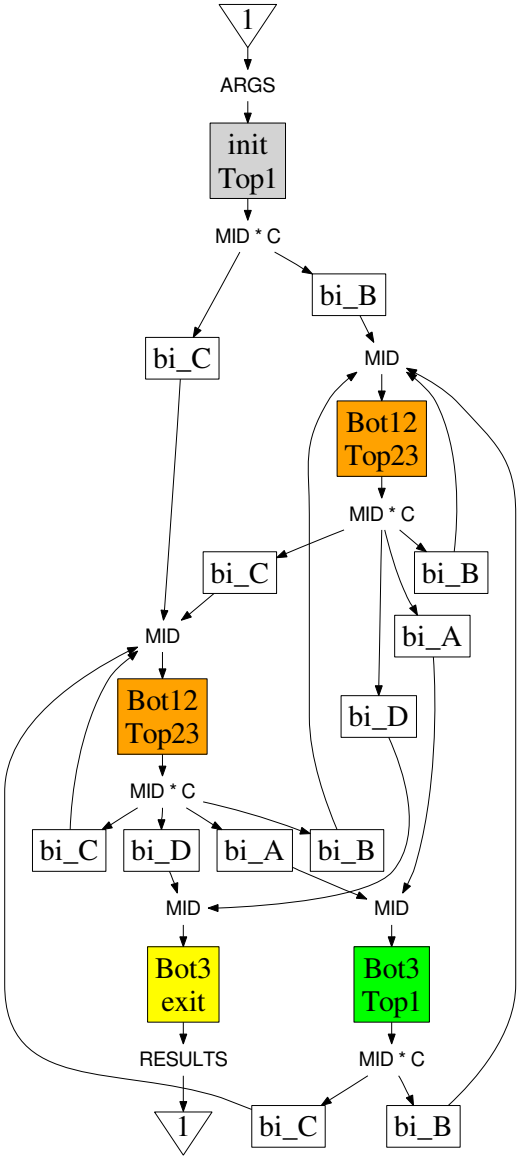


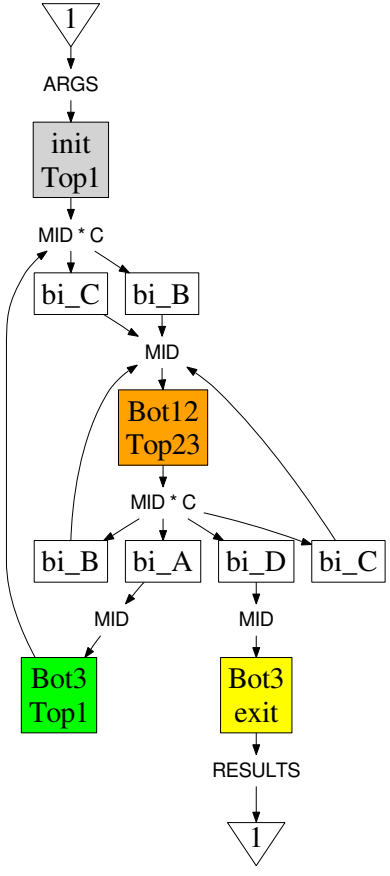


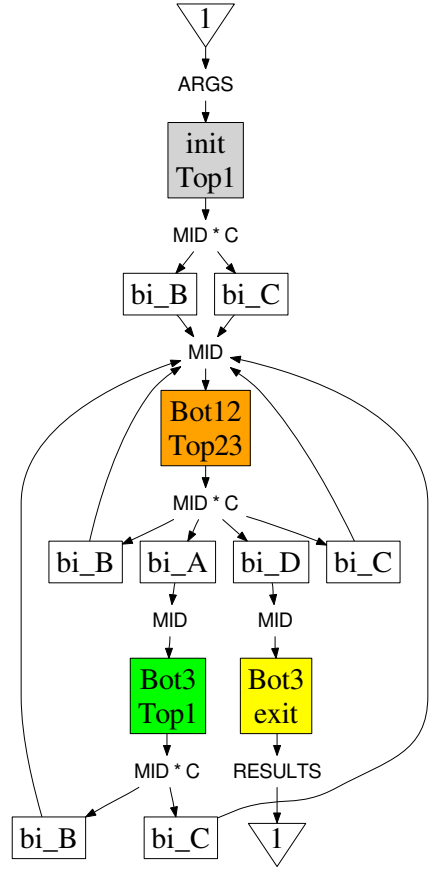
Key to Minimalisation: Impossible Branch Edges

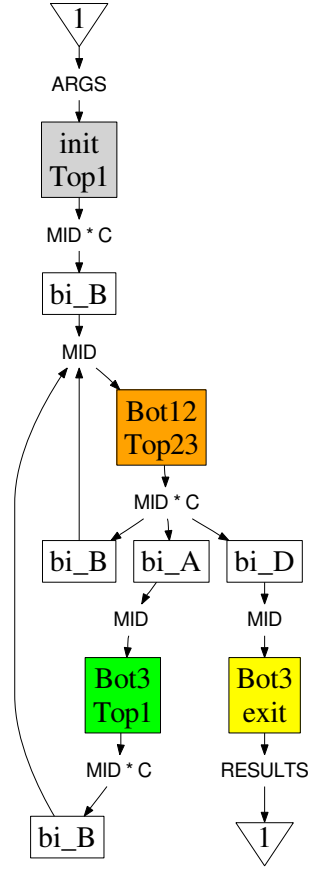


Matrix Mult.: Minimalisation and Simplification









Software Pipelining by Calculation

- Simple loops — similar to modulo scheduling
- Multi-way switch inside loop body
- MatMult nested loop — involves "synthetic loop overhead"
- FFT nested loop — different structure
- **General tool-box** for **loop pattern transformations with correctness proofs**
- Transformation of **nested graphs** with **modular semantics**

Limitations of Code Graph Tranformation in Haskell

- Coconut "code graphs" are intermediate presentation for the generation of highly optimised assembly code
- In Haskell: Code-graph-creation EDSL captures sharing introduced by **let**-definitions
- Haskell functions representing assembly-level operations construct hyperedges
- Haskell lacks full dependent typing: graph navigation, traversal, and manipulation operations cannot preserve the connection with the Haskell-level typing of the assembly operations
- Preservation of well-typedness by transformations requires proof or run-time checks
- "Simulator" implementation of code-graph-creation EDSL disconnected from code-graph typing

Agda

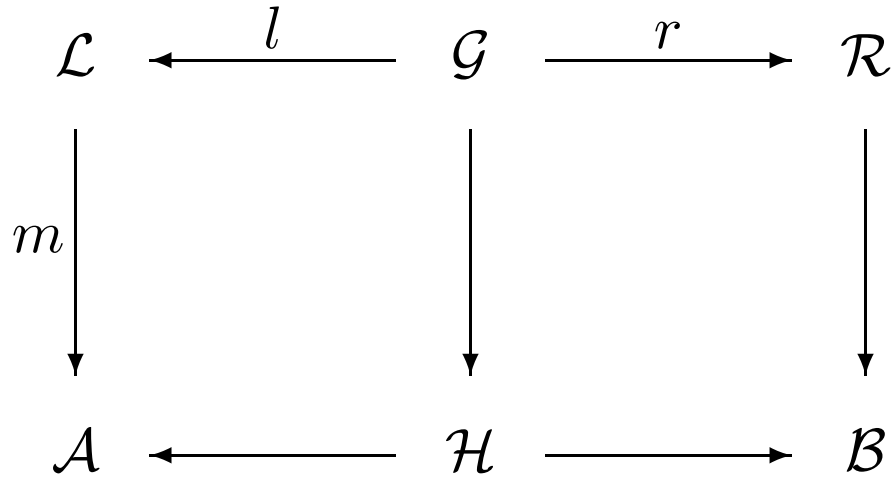
- Agda is a dependently typed functional programming language
- Agda is a proof assistant based on Per Martin-Löf's intuitionistic type theory
- Syntactically and "culturally" close to Haskell
- Different semantics: strongly normalising, no $\perp$ values
- Dependently typed: No distinction between terms, types, and kinds

RATH-Agda — The Plan

- **R**elation-**A**lgebraic **Th**eories in **Agda**
- Motivated by graph transformation and code generation
- Collagories etc. defined axiomatically
- Collagories etc. of abstract algebras proven correct
- Co-algebras for hypergraphs and symbolically attributed graphs
- Pushouts etc. defined axiomatically
- Constructions, e.g. pushouts from sums and coequalisers (quotients), proven correct
- High-level programming of transformation approaches
- Base categories implement interfaces
- *. . . currently can calculate pushouts of (at most) unary algebras*

Algebraic Graph Transformation

- uses **abstract category-theoretic** concepts
- defines general transformation approach
- predominant: **double-pushout approach**:
  - a rule is a span $\mathcal{L} \xleftarrow{l} \mathcal{G} \xrightarrow{r} \mathcal{R}$
  - application via a **matching** m rewrites $\mathcal{A}$ to $\mathcal{B}$



Code Graph Transformation: Conclusion

- Generalised software-pipelining can be derived calculationally
- This requires a lot of mathematics
- Formalising mathematical developments in Agda is **remarkably straight-forward**
- **Lots of work still to do**:
  - Justification of DPO code graph trafos
  - (De-)serialisation of more complex structures
  - Matching search
  - Integrate **decision procedures** where possible
- URL: http://relmics.mcmaster.ca/RATH-Agda/

Interaction Nets are Code Graphs, Too...

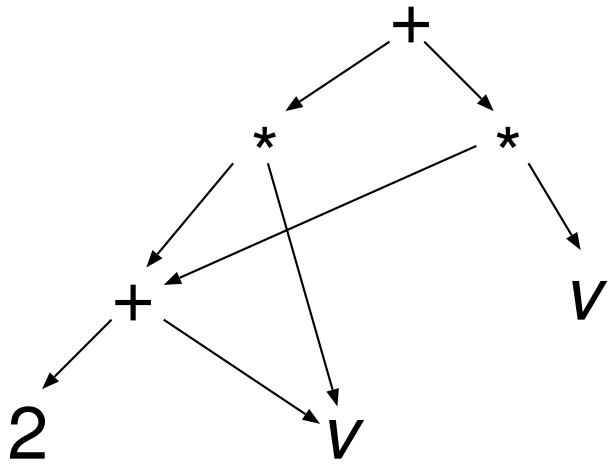
- A (biased, long, somewhat rushed) introduction to interaction nets
- Implementation principles
- Implementation in Haskell
- Compilation to Go
- Benchmarks of parallel reduction

To Remember:

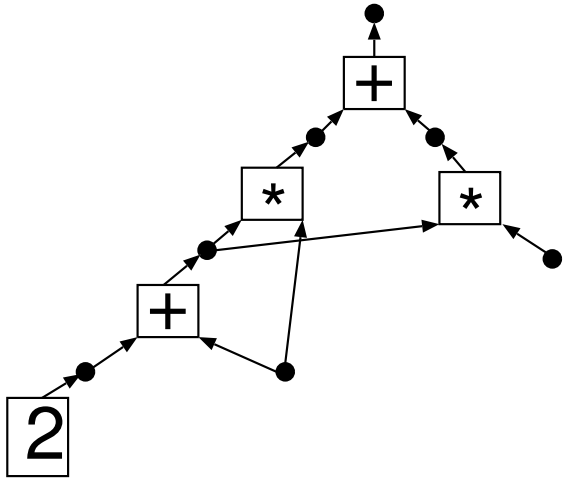
- Interaction Nets are an execution model,
  not a programming language
- Interaction Nets as an execution model
  - promise large speed-ups via parallelisation
  - still have to deliver...
- For some interaction net languages, even a **straight-forward**

Terms, Term Graphs, Jungles, Interaction Nets

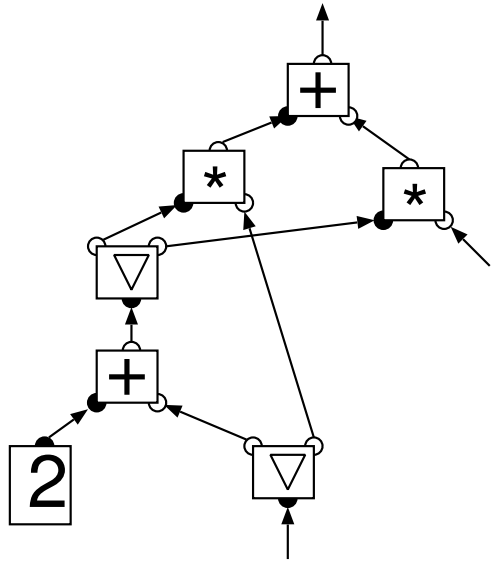
Term $(2 + x) * x + (2 + x) * y$ represented with sharing —
quick overview:



Term Graph



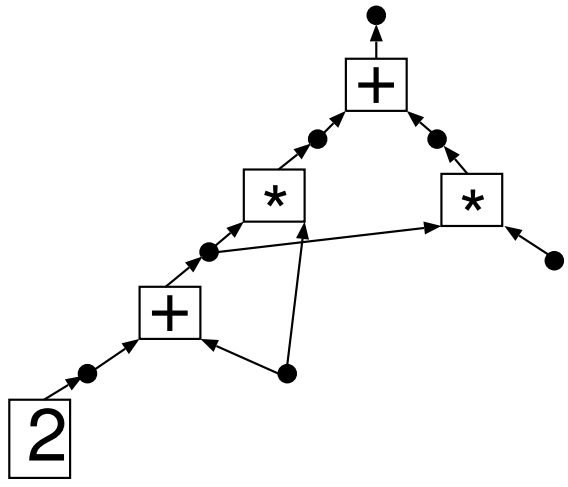
Jungle



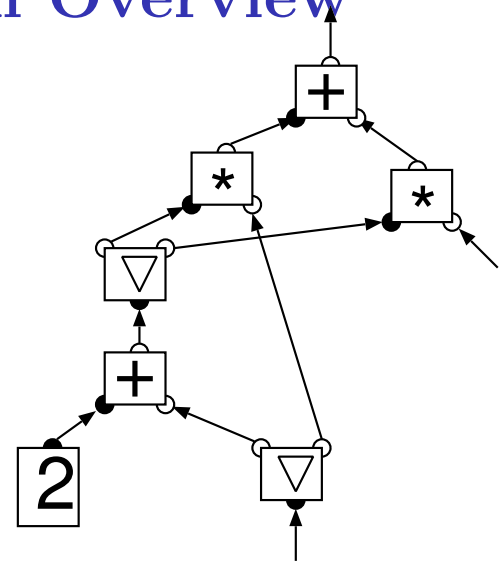
Interaction Net

- **all:** directed, (implicit) incidence ordering
- **Term graph:** Symbols as node labels
- **Jungles:** Symbols as hyperedge labels
- **Interaction Nets:**
  - Symbols as node labels
  - no sharing by connections
  - fixed operational semantics

Jungles, Interaction Nets — Initial Overview



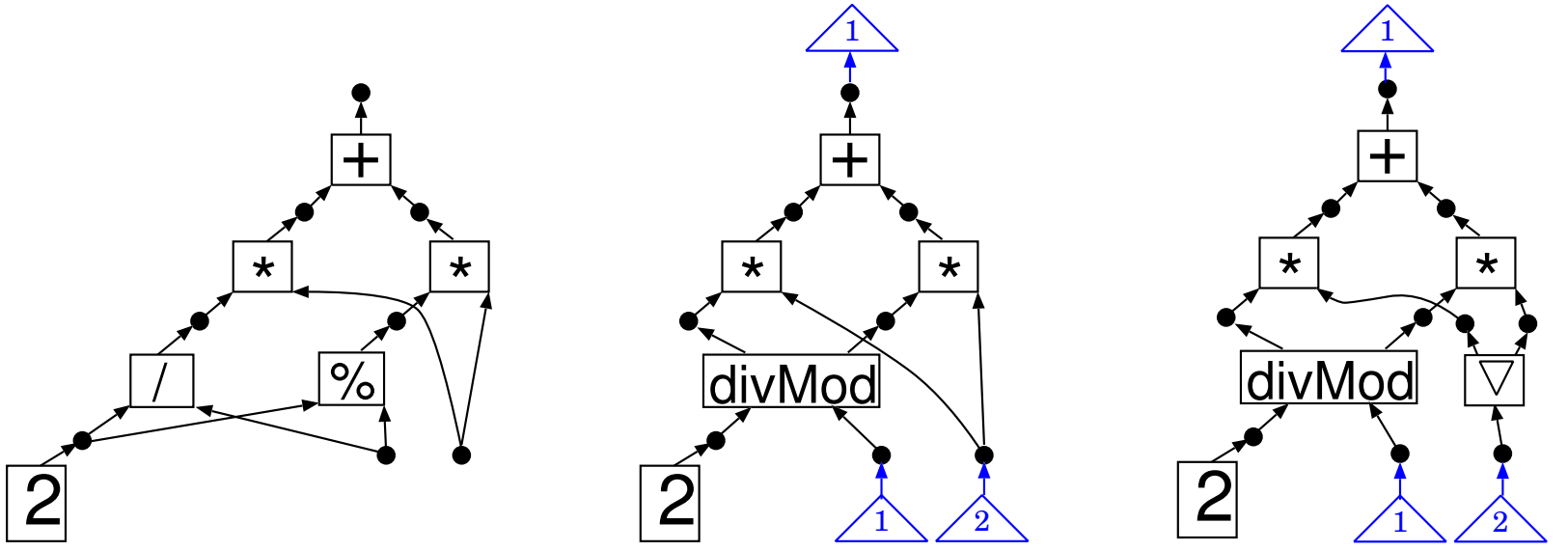
Jungle



Interaction Net

- TG, Jungle: Denotationally motivated directions
- INet:
  - Denotationally motivated directions sometimes called “polarities”, sometimes omitted
  - Operationally motivated direction of nodes (“actors”) from auxiliary ports to principal port
- TG, Jungle: One-sided sharing
- INet: Explicit sharing nodes

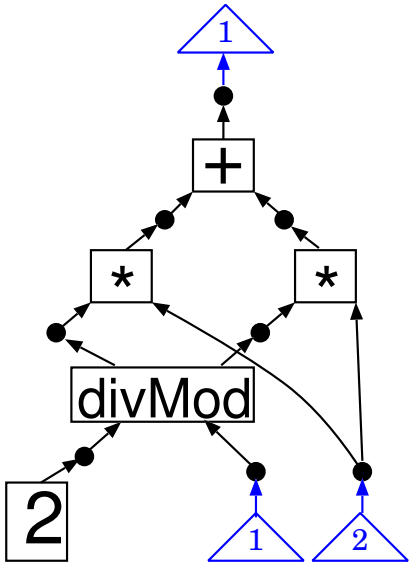
From Jungles to Code Graphs with Explicit Sharing



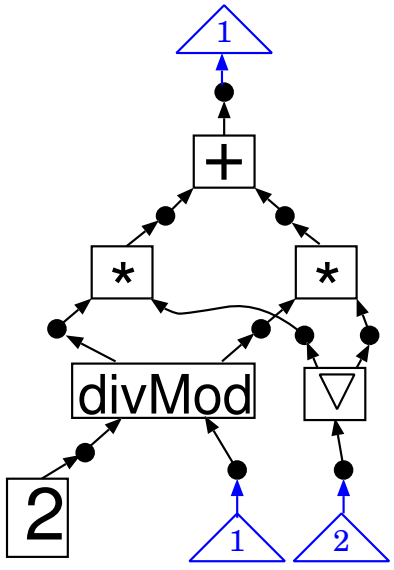
Jungle **Code Graph** **Code Graph w. Dup.**

- Operations like divMod can have multiple results
- Code graph hyperedges can have multiple result tentacles
- The sequences of input and output nodes are graphically indicated
- Implicit sharing can be replaced with multi-output sharing operations ∇
- (Garbage can be flagged by zero-output "terminator" !)
- ∇ and ! are primitive in gs-monoidal categories
  — explicit ∇ and ! operations move into monoidal categories

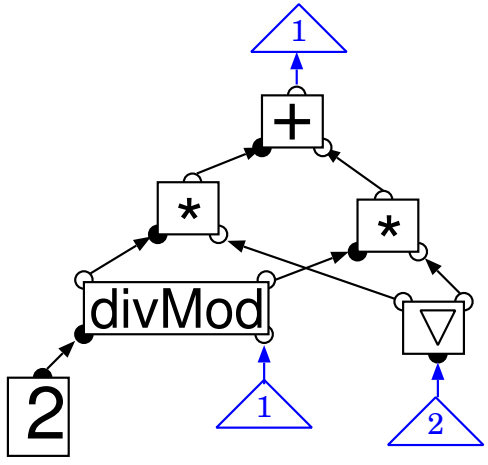
From Code Graphs to Interaction Nets



Code Graph



Code Graph w. Dup.



Interaction Net

Interaction nets are code graphs with neither sharing nor garbage, **with a reduction rule set** where

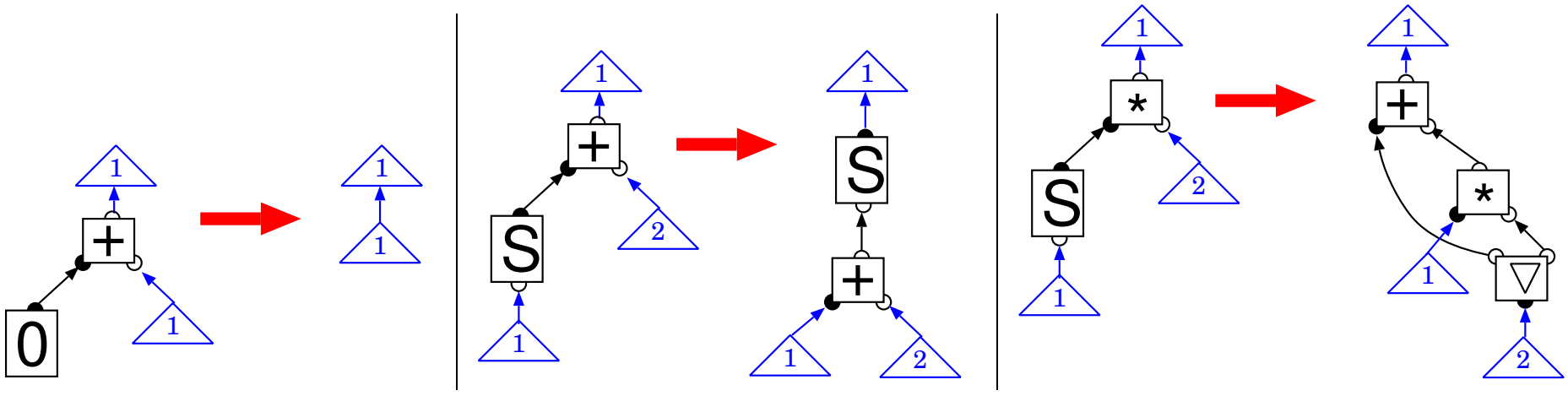
- all rule LHSs have exactly two nodes with exactly one internal connection
- for each node label, its internal connections in rule LHSs are all incident with the same port
- that port is called the **principal port** of that node label.

Operational Semantics of Interaction Nets

add 0 n = n

add (S m) n = add m n

mult (S m) n = add n (mult m n)



Replacement of subnets induced by two nodes connected via their principal ports:

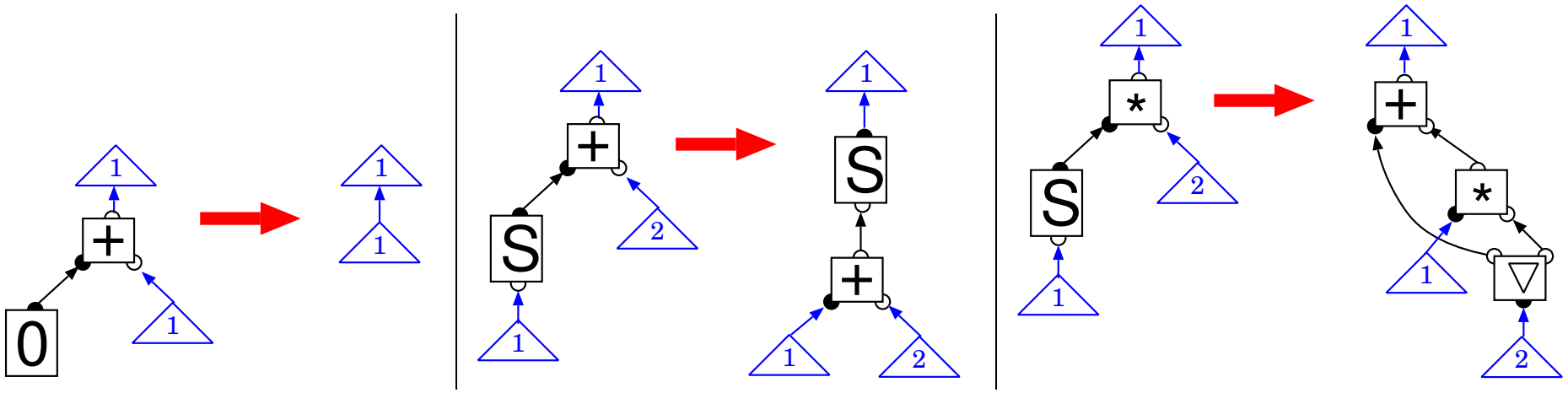
- Extreme locality (in the graph view)
- No critical pairs: Confluent
- Deterministic except for sequencing of independent reductions
- "Inherently parallel"
- Simple DPO step **if considered as code graphs**

Conventional Representations of Interaction Net Rules

add 0 n = n

add (S m) n = add m n

mult (S m) n = add n (mult m n)



Interaction net language "Inets" [Mackie *et. al*]:

```
add(r,n) >< 0()
   ⇒   r~n
```

```
add(r,n) >< S(m)
   ⇒   m~add(a,n),
       r~S(a)
```

```
mult(r,n) >< S(m)
   ⇒   m~mult(a,b),
       n~∇(add(r,a),b))
```

Implementations

- GUI-oriented:
  - de Falco [2006]: "Interaction Nets Laboratory"
  - Almeida et al. [2008]: "INblobs": In Haskell
- Sequential:
  - Hassan et al. [2009]: "INETS": In C, two-sided non-pointer links
    Hassan et al. [2010]: Term-based
  - Lippi [2002]: in^2: similar in spirit
- Parallel:
  - Banach and Papadopoulos [1997]: Never ran?
  - Pinto [2000, 2001]: Term-based
  - Jiresch [2014]: Term-based GPU implementation (C/CUDA)
  - Pedicini and Quaglia [2007] PELCR: distributed, for optimal λ-calculus reduction

Modelling Interaction Nets: Naïve Port-Based Version

Connection targets in rule sides:

```
data PortTargetDescription = SourcePort   PI
  |                        InternalPort NI PI   -- node, port
  |                        TargetPort   PI
```

A node has a label and an array of ports:

```
data NodeDescription nLab = NodeDescription
  {nLab                 :: nLab
  , portDescriptions :: Vector PortTargetDescription}
```

A rule RHS contains replacement net, and interface specification:

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

- Every connection specified twice
- $\Longrightarrow$ Consistency invariant required
- Shortcircuits specified only in source and/or target

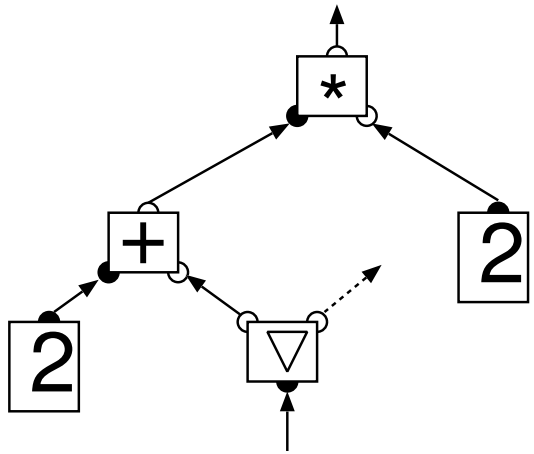
Naïve Connection-Based Version (INblobs)

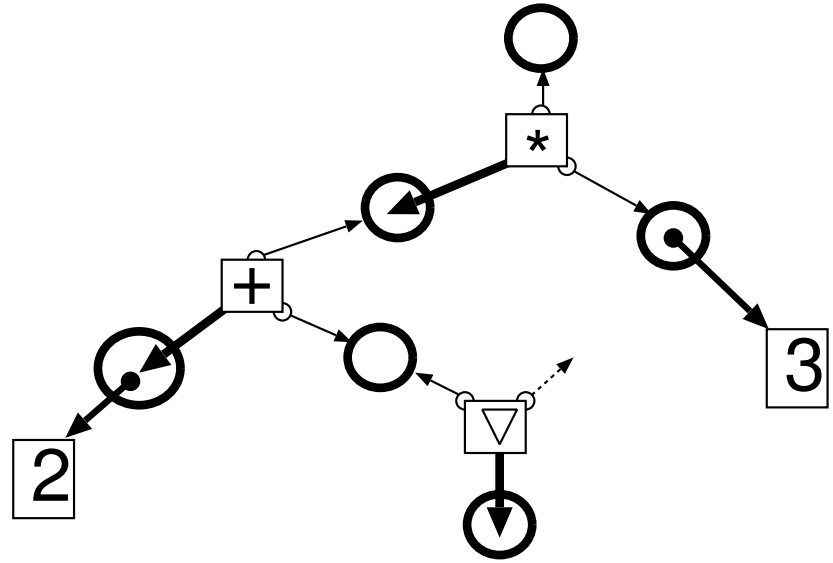
```
data Network g n e  =  Network
   {networkNodes :: IntMap (Node n)   -- node numbers to nodes
   , networkEdges  :: IntMap (Edge e)   -- edge numbers to edges
   , ...                  {-Layout information omitted -}      }
data Node n  =  Node
   {nodeInfo        :: n
   , ...                  {-Layout information omitted -}      }
data Edge e  =  Edge
   {edgeFrom :: NodeNr      -- where the edge starts
   , portFrom :: PortName   --
   , edgeTo :: NodeNr       -- where the edge points to
   , portTo  :: PortName    --
   , ...                  {-Layout information omitted -}      }
data INRule g n e  =  INRule
   {ruleLHS :: Network g n e   -- the rule LHS network
   , ruleRHS :: Network g n e   -- the rule RHS network
   , ruleMaps :: [ (NodeNr, NodeNr) ]  {-LHS to RHS -}  }
```

Preparing for Concurrency

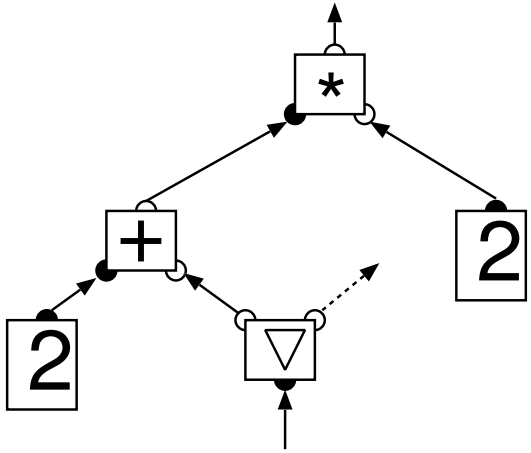
Main ideas proposed by Banach and Papadopoulos [1997]:

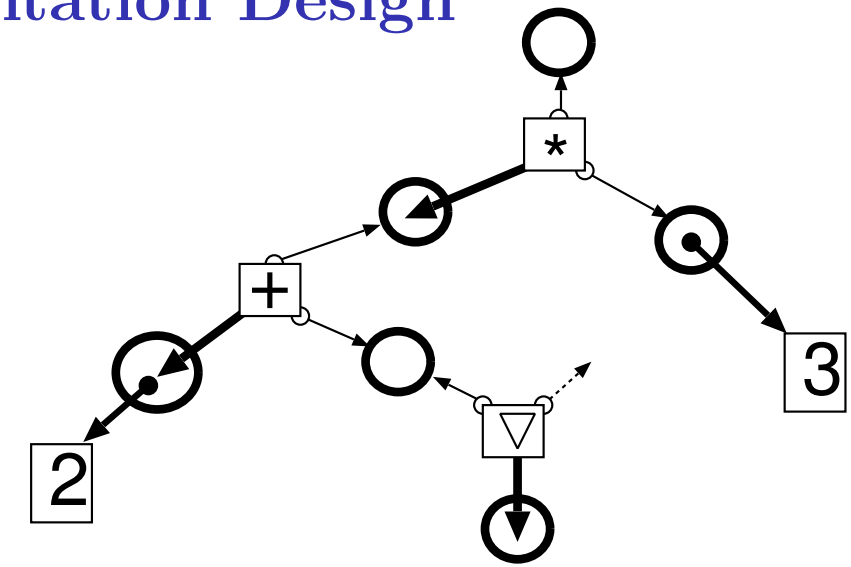
- Mutable two-way connections: opportunities for deadlock and race conditions
- Two-way connections can be avoided by using polarities:
  - The connections hold mutable state
  - Constructor nodes have no reference to their principal port connection
  - . . . but *vice versa*





Implementation Design





Directly implementing the approach of Banach and Papadopoulos [1997]:

- Connections are initially "empty"
- Each node has references to the connections attached to its auxiliary ports
- Attaching the PP of a constructor "fills" the connection with a reference to the constructor node
- Attaching the PP of a function starts a concurrent thread waiting for a constructor, and then starting the corresponding rule application
- Short-circuiting starts a thread waiting for a constructor on the positive side, to transfer it to the negative side

Mutable Net Representation — Haskell

Concurrent Haskell provides type MVar a for synchronisation:

- MVars can be empty, or contain an a
- newEmptyMVar creates an empty MVar
- putMVar m v (blocks until m is empty) and fills m with v
- takeMVar m blocks until m is full

```
type Conn  nLab  =  MVar (Node nLab)
data Polarity  =  Neg | Pos
data Port  nLab  =  Port  {pol :: Polarity
                          , conn :: Conn nLab
                          }
type Ports  nLab  =  Vector (Port nLab)
data Node nLab  =  Node {label :: nLab
                        , ports :: Ports nLab
                        }
```

Mutable Net Representation

Concurrent Haskell provides type MVar a for synchronisation:

- MVars can be empty, or contain an a
- newEmptyMVar creates an empty MVar
- putMVar m v (blocks until m is empty) and fills m with v
- takeMVar m blocks until m is full

```
type Conn  nLab  =  MVar (Node nLab)
data Polarity  =  Neg | Pos
data Port  nLab  =  Port  {pol :: Polarity
                          , conn :: Conn nLab
                          }
type Ports  nLab  =  Vector (Port nLab)
data Node nLab  =  Node {label :: nLab
                        , ports :: Ports nLab
                        }
```

Interaction Net Creation

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

createNet :: (Show nLab, Eq nLab   -- for debugging
  ) ⇒ INetLang nLab
      → NetDescription nLab → IO (Ports nLab, Ports nLab)

createNet lang descr = let
    mkConns = V.mapM (λ pl → fmap (Port pl) newEmptyMVar)
  in case interfacePolarity lang descr of
    Nothing → fail "createNet: insufficient polarity informatic
    Just (srcPol, trgPol) → do
      src ← mkConns srcPol
      trg ← mkConns trgPol
      replaceNet lang descr src trg
      return (src, trg)
```

Interaction Net Replacement (1)

```
replaceNet :: forall nLab ∘ INetLang nLab → NetDescription nLab
              → Ports nLab → Ports nLab → IO ()
replaceNet lang descr src trg = mdo
  nps ← let
    mkPort Pos (InternalPort _ _) = fmap (Port Pos) newEmptyMVar
    mkPort Neg ptd = return (portTarget ptd)
    mkNode (NodeDescription lab pds) = do
              ps ← V.zipWithM mkPort (polarity lang lab) pds
              return (Node {label = lab , ports = V.tail ps}
                      , V.head ps)
    in V.mapM mkNode (nodes descr)
  let portTarget :: PortTargetDescription → Port nLab
      portTarget (SourcePort i) = src ! pred i
      portTarget (TargetPort i) = trg ! pred i
      portTarget (InternalPort n i) = let (n', pp) = nps ! n
        in opPort (if i ≡ 0    then pp
                               else ports n' ! pred i)
```

Interaction Net Replacement (2)

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

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

Extraction of Natural Numbers

```
getNat :: Port NLab → IO Int
getNat = h 0 where
  h i (Port pl c) = do
    n ← takeMVar c
    case label n of
      NLab "Z" → return i
      NLab "S" → h (succ i) (ports n ! 0)
      lab → fail $ "getNat.h "
                 + shows i (": encountered " + show lab)


mainAck = do
  [m,n] ← fmap (map read) getArgs
  (src, trg) ← createNet simpleLang $ ackND m n
  i ← getNat $ src ! 0
  putStrLn $ "getNat result: " + show i
```

Mutable Net Representation — Go

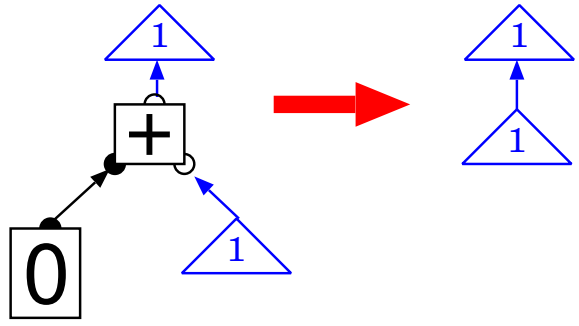
Go provides "channels" for synchronisation:

- Channels with capacity 1 are like MVars
- c ← v sends value v along channel c
- select with a <-c expression is used to wait on channel c

```
type (   conn chan node
         node struct { nodeLabel int
                       nodePorts [] conn
                     }
     )
```

Interaction Net Replacement — add 0 n = n

```
func replace_3_6(srcPorts, trgPorts []conn)
{ // Redex: + ⇔ 0
   var conn1, conn2 conn
   conn1 = srcPorts[0]
   conn2 = srcPorts[1]
   if conn1 != conn2
      { go chanForward(conn1, conn2)
      }
}
```

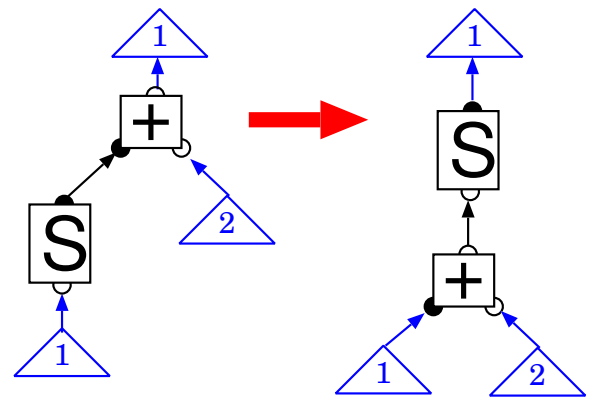


Short-circuiting: Forward constructor node references:

```
func chanForward(c1, c2 conn) {
   var n node
   select { case n = ←c1: // n received from c1
                          c2 ←n // n sent to c2
          }
}
```

Interaction Net Replacement: add (S m) n = S (add m n)

```
func replace_3_5(srcPorts, trgPorts  [] conn) // Redex: + ⇔ S
{ var  ( nodes [2]∗ node                      // new nodes
         pports  [2] conn )                   // their principal port connections
  { var  newNode ∗node = new(node)
    newNode.nodePorts = make([]conn, 1)       // create auxiliary port array
    nodes[0]  = newNode
    newNode.nodeLabel = 5                 }   // initialise node label "S"
  { var  newNode ∗node = new(node)
    newNode.nodePorts = make([]conn, 2)
    nodes[1]  = newNode
    newNode.nodeLabel = 3                     // "+"
    // create internal output connections:
    newNode.nodePorts[1] = make(conn, 1) } // capacity 1
  // connect remaining new ports:
  pports[0]  = srcPorts[1]
  nodes[0]. nodePorts[0]  = nodes[1].nodePorts[1]
  pports[1]  = trgPorts[0]
  nodes[1]. nodePorts[0]  = srcPorts[0]
  // send new constructors to their principal port connection:
  pports[0]  ← ∗nodes[0]
  // start new threads for new function agents:
  go reduce_3(pports[1],  nodes[1]. nodePorts)
}
```



Extraction of Natural Numbers — Go

```
func extractNat(c conn) {
  var ( count int = 0
        n node
        notDone bool = true )
  for notDone {
    select {
      case n = ←c:                    // receive n from channel c
        switch n.nodeLabel {
          case 6:                     // "Z"
               notDone = false
          case 5:                     // "S"
               count++
               c = n.nodePorts[0]
        }
    }
  }
  fmt.Println ("extractNat: ", count)
}
```

Benchmarks: Ackerman — Time

```
ack 0 n  =  S n                          ackAux m 0  =  ack m 1
ack (S m) n  =  ackAux m n               ackAux m (S n)  =  ack m (ack (S m) n)
```

4-core hyperthreading Intel Core i7 at 2.1GHz with 16GB RAM:

expr.	result	time (s)							
# cores		1	2	3	4	5	6	7	8
GHC-7.10.2, with 8GB fixed heap:									
ack 3 9	4093	46.74	28.41	22.21	18.29	17.23	16.53	15.90	15.73
ack 3 10	8189	205.73	125.75	95.00	80.37	74.96	70.60	67.69	65.88
gccgo-5.2.0 -O3									
ack 3 9	4093	42.63	23.47	17.19	15.20	13.29	12.19	11.35	11.04
ack 3 10	8189	176.55	97.32	70.54	62.37	55.84	51.47	47.90	46.01
Go-1.4.2									
ack 3 9	4093	22.73	12.12	8.45	7.62	6.87	6.51	6.38	6.55
ack 3 10	8189	97.36	52.18	39.45	33.52	30.61	27.39	26.49	26.15
Go-1.5									
ack 3 9	4093	14.20	7.34	5.30	5.11	4.62	5.03	5.23	5.66
ack 3 10	8189	65.68	38.39	27.91	25.73	23.39	23.44	23.01	23.34
ack 3 11	16381	292.44	155.82	114.48	105.83	100.10	100.05	95.86	95.86
ack 3 12	32765	1226.80	670.74	484.39	453.55	417.75	401.17	396.72	396.88
Go-1.5 without garbage collection:									
ack 3 9	4093	13.44	7.09	4.84	3.80	3.50	3.57	3.49	3.39
speedup over GC		*1.06*	*1.03*	*1.10*	*1.35*	*1.32*	*1.41*	*1.50*	*1.67*
ack 3 10	8189	54.74	28.93	20.09	15.96	15.77	15.29	14.74	14.31
speedup over GC		*1.20*	*1.33*	*1.39*	*1.61*	*1.48*	*1.53*	*1.56*	*1.63*

Benchmarks: Ackermann — Speed-ups

```
ack 0 n  =  S n
ack (S m) n  =  ackAux m n

ackAux m 0  =  ack m 1
ackAux m (S n)  =  ack m (ack (S m) n)
```

4-core hyperthreading Intel Core i7 at 2.1GHz with 16GB RAM:

expr.	result	time (s)	speedup factor over 1 core						
# cores		1	2	3	4	5	6	7	8
GHC-7.10.2, with 8GB fixed heap:									
ack 3 9	4093	46.74	1.65	2.10	2.56	2.71	2.83	2.94	**2.97**
ack 3 10	8189	205.73	1.64	2.17	2.56	2.74	2.91	3.04	**3.12**
gccgo-5.2.0 -O3									
ack 3 9	4093	42.63	1.82	2.48	2.81	3.21	3.50	3.76	**3.86**
ack 3 10	8189	176.55	1.81	2.50	2.83	3.16	3.43	3.69	**3.84**
Go-1.4.2									
ack 3 9	4093	22.73	1.87	2.69	2.98	3.31	3.49	**3.56**	3.47
ack 3 10	8189	97.36	1.87	2.47	2.90	3.18	3.55	3.68	**3.72**
Go-1.5									
ack 3 9	4093	14.20	1.94	2.68	2.78	**3.07**	2.83	2.72	2.51
ack 3 10	8189	65.68	1.71	2.35	2.55	2.81	2.80	**2.85**	2.81
ack 3 11	16381	292.44	1.88	2.55	2.76	2.92	2.92	**3.05**	3.05
ack 3 12	32765	1226.80	1.83	2.53	2.70	2.94	3.06	**3.09**	3.09
Go-1.5 without garbage collection:									
ack 3 9	4093	13.44	1.90	2.78	3.54	3.84	3.76	3.85	**3.96**
speedup over GC		*1.06*	*1.03*	*1.10*	*1.35*	*1.32*	*1.41*	*1.50*	*1.67*
ack 3 10	8189	54.74	1.89	2.72	3.43	3.47	3.58	3.71	**3.83**
speedup over GC		*1.20*	*1.33*	*1.39*	*1.61*	*1.48*	*1.53*	*1.56*	*1.63*

Benchmarks — Fibonacci

```
fib 0 = 0
fib (S n) = fibAux n

fibAux 0 = 1
fibAux (S n) = fib n + fibAux n
```

Times:

expr.	result	time (s)							
# cores		1	2	3	4	5	6	7	8
fib 25	75025	2.14	1.12	0.82	0.68	0.63	0.60	0.58	0.56
fib 28	317811	9.87	5.19	3.62	2.93	2.75	2.68	2.55	2.55
fib 30	832040	24.66	13.56	9.85	7.60	7.60	6.88	6.86	6.66
fib 31	1346269	45.79	25.60	9.90	14.09	9.39	8.70	8.76	7.20
fib 32	2178309	74.33	41.65	31.72	23.91	16.20	14.71	14.36	12.13
fib 33	3524578	122.97	66.25	31.20	38.40	25.70	25.11	24.74	20.30

Speed-ups:

expr.	result	time (s)	speedup factor over 1 core						
# cores		1	2	3	4	5	6	7	8
fib 25	75025	2.14	1.92	2.62	3.14	3.40	3.56	3.72	**3.80**
fib 28	317811	9.87	1.90	2.73	3.37	3.59	3.69	3.87	**3.87**
fib 30	832040	24.66	1.82	2.50	3.24	3.24	3.59	3.59	**3.70**
fib 31	1346269	45.79	1.79	**4.62**	3.25	4.88	5.26	5.23	**6.36**
fib 32	2178309	74.33	1.78	2.34	3.11	4.59	5.05	5.18	**6.13**
fib 33	3524578	122.97	1.86	**3.94**	3.20	4.78	4.90	4.97	**6.06**

Interaction Nets: Conclusion

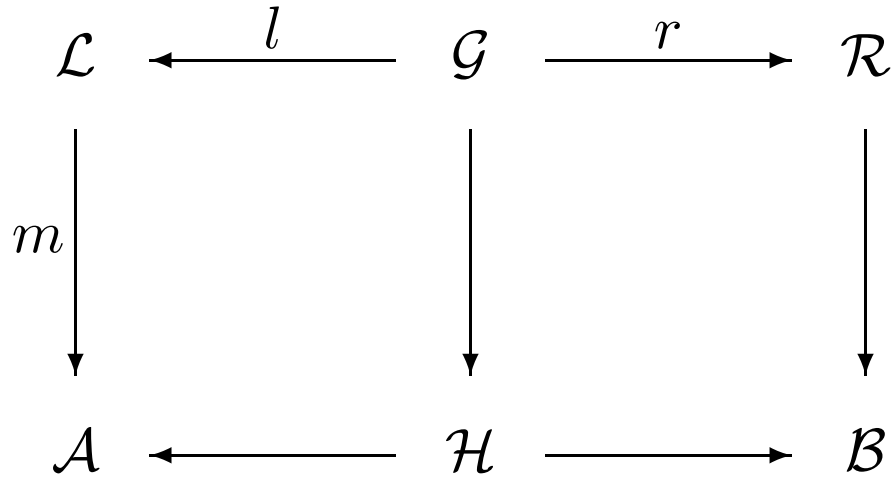
- A **direct** implementation of interaction net reduction
  - using the ideas of Banach and Papadopoulos [1997] for concurrency
  - using the **multi-core light-weight threads** of **Go**
  - implemented in Haskell, sharing code with the HINet interpreter
- Results:
  - 3× faster than Haskell on one core
  - Multi-core speedups mostly follow similar pattern as in Haskell
  - Multi-core speedups sometimes significantly better

Future Work

- Improve Go generation
- Complete code generation for "Inets" source — typechecking done
- Add interesting front-ends — real functional languages!
- Benchmark larger applications
- Try multi-core light-weight threads without garbage collection — candidates?

Algebraic Graph Transformation

- uses **abstract category-theoretic** concepts
- defines general transformation approach
- predominant: **double-pushout approach**:
  - a rule is a span $\mathcal{L} \xleftarrow{l} \mathcal{G} \xrightarrow{r} \mathcal{R}$
  - application via a **matching** m rewrites $\mathcal{A}$ to $\mathcal{B}$



Abstract Algebraic Graph Transformation

- DPO considers graphs as unary **algebras**
- Category setting $\Longrightarrow$ "High-Level Replacement Systems"
- Presentation of gluing condition usually not abstract
- Gluing condition has relation-algebraic formulation: [Kawahara 1990] uses relations in topos
- Pushout characterisation [Kawahara 1990] essentially dual to tabulations [Freyd, Scedrov 1990]
- [Kahl 2001] proposed amalgamated approach in Dedekind categories (relation algebras without complement)
- [Kahl 2004] abstracts pushouts to co-tabulations
- In **bi-tabular collagories**, the subcategory of mappings is adhesive [Kahl 2009 RelMiCS]
- Co-tabulations and bipushouts of mappings coincide [Kahl 2010 GT-VMT]
- DPO, DPB, and amalgamated "pullout" approach only need **collagories** [Kahl 2010 ICGT]

José Bacelar Almeida, Jorge Sousa Pinto, and Miguel Vilaça. A tool for programming with interaction nets. *ENTCS*, 219:83–96, 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.10.036. Proc. Eighth International Workshop on Rule Based Programming (RULE 2007).

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.

Andrea Corradini and Fabio Gadducci. An algebraic presentation of term graphs, via gs-monoidal categories. *Applied Categorical Structures*, 7(4):299–331, 1999.

Marc de Falco. Interaction nets laboratory, 2006.

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.

Eugen Jiresch. Towards a gpu-based implementation of interaction nets. In Benedikt Löwe and Glynn Winskel, editors, *8th International Workshop on Developments in Computational Models, DCM 2012*, volume 143 of *EPTCS*, pages 41–53, 2014. doi: 10.4204/EPTCS.143.4.

Sylvain Lippi. in^2: A graphical interpreter for interaction nets. In Sophie Tison, editor, *RTA 2002*, volume 2378 of *LNCS*, pages 380–385. Springer, Berlin Heidelberg, 2002. ISBN 978-3-540-43916-5. doi: 10.1007/3-540-45610-4_29.

Marco Pedicini and Francesco Quaglia. Pelcr: Parallel environment for optimal