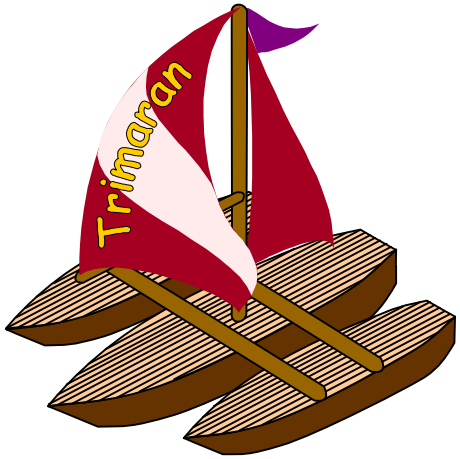


ELCOR Modules

Elcor Functional Overview

Elcor is a collection of compiler components and scripts that analyze and transform Rebel

- Elementary data structures
  - Container classes
  - Data structures for compiler algorithms
- Intermediate Representation data structures(*)
- I/O modules
  - Rebel reader/writer
  - Lcode reader/writer
- Mdes interface(*)
- Analysis modules
  - Control dependence
  - Data flow
- Transformation and optimization modules
- Scheduling modules
  - Acyclic schedulers
  - Loop schedulers
- Rotating register allocator
- Static register allocator(*)

  by ReaCT-ILP

(*)*Covered separately*

Basic Elcor Data Structures

- Container classes
  - Lists, sets, hash tables, vectors, maps etc.
- Data structures for compiler algorithms
  - Bitvector
  - Matrices,
    - **Solving shortest/longest path problems**
  - Directed graphs
    - **Topological sort**
    - **Depth-first/breadth-first graph traversals**

Control Flow Analysis

- Dominator analysis
- Control Dependence Analysis
- Loop detection
- Induction variable detection

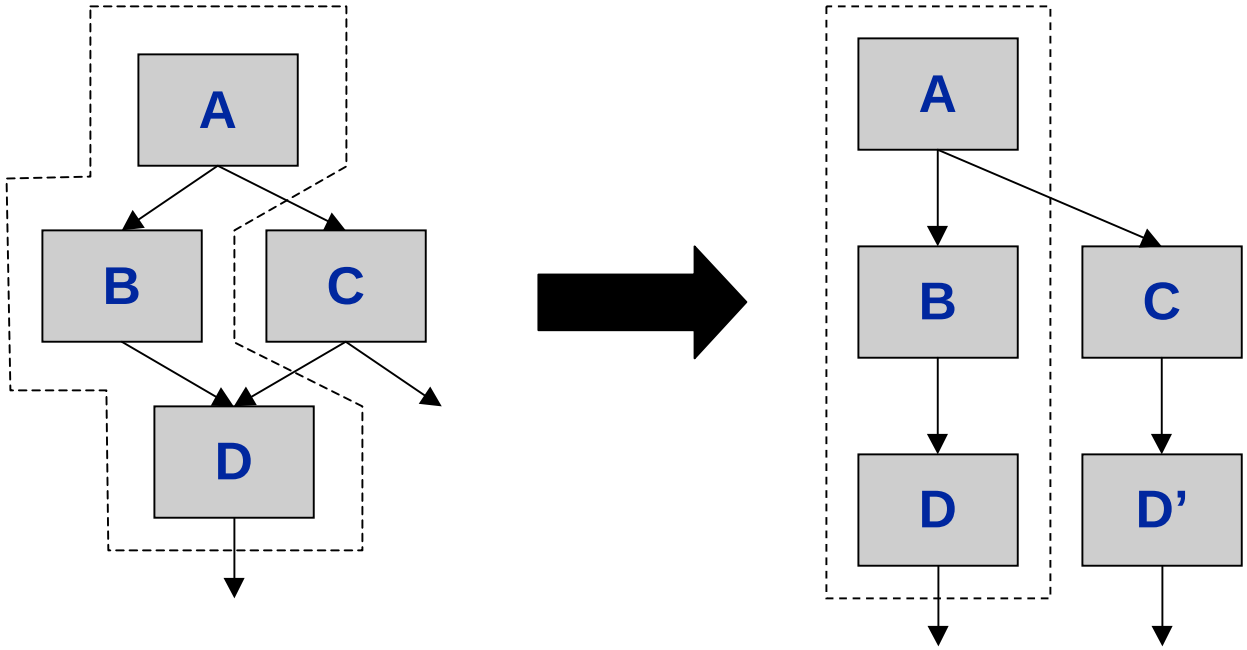
Control Flow Transformations

- Loop region construction
  - Constructs a cyclic region which can be modulo scheduled
    - **Single back edge**
    - **Counted do loop**
- Structural region formation
  - Identifies acyclic subgraphs of CFG that are single entry and multiple exit.
- Branch normalization/denormalization
  - Constructs a memory layout independent form of CFG that can be transformed easily.

Control Flow Transformations

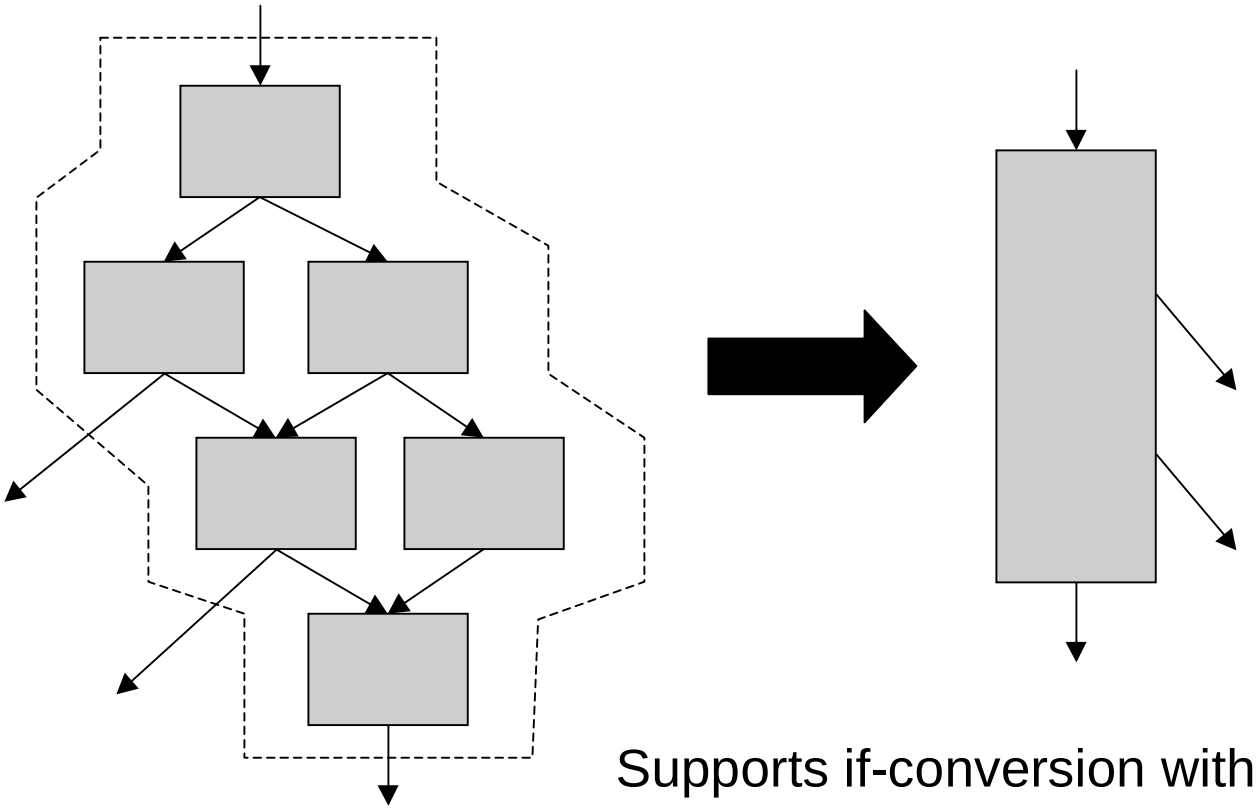
- Tail duplication:

  Useful for constructing single entry multiple exit regions



Control Flow Transformations

- If-conversion of single entry multiple exit basic block regions



Supports if-conversion with or without fully resolved predicates (FRP's)

Data-flow analysis

- Live variable analysis
  - Live variable information is annotated on the IR

  Can also compute
  - Up exposed defined variables
  - Down exposed used variables
  - Down exposed defined variables
- Reaching definitions analysis
  - A data structure for def-use chains is annotated on the IR
- Available expression analysis
  - Queries for expression availability is provided at any point on the control-flow graph

These analysis can be performed on any region

- Predicate Query System(s) for hyperblocks
  - One-bit representation of expressions (TRUE or FALSE)
  - Single symbol representation (symbols from program text)

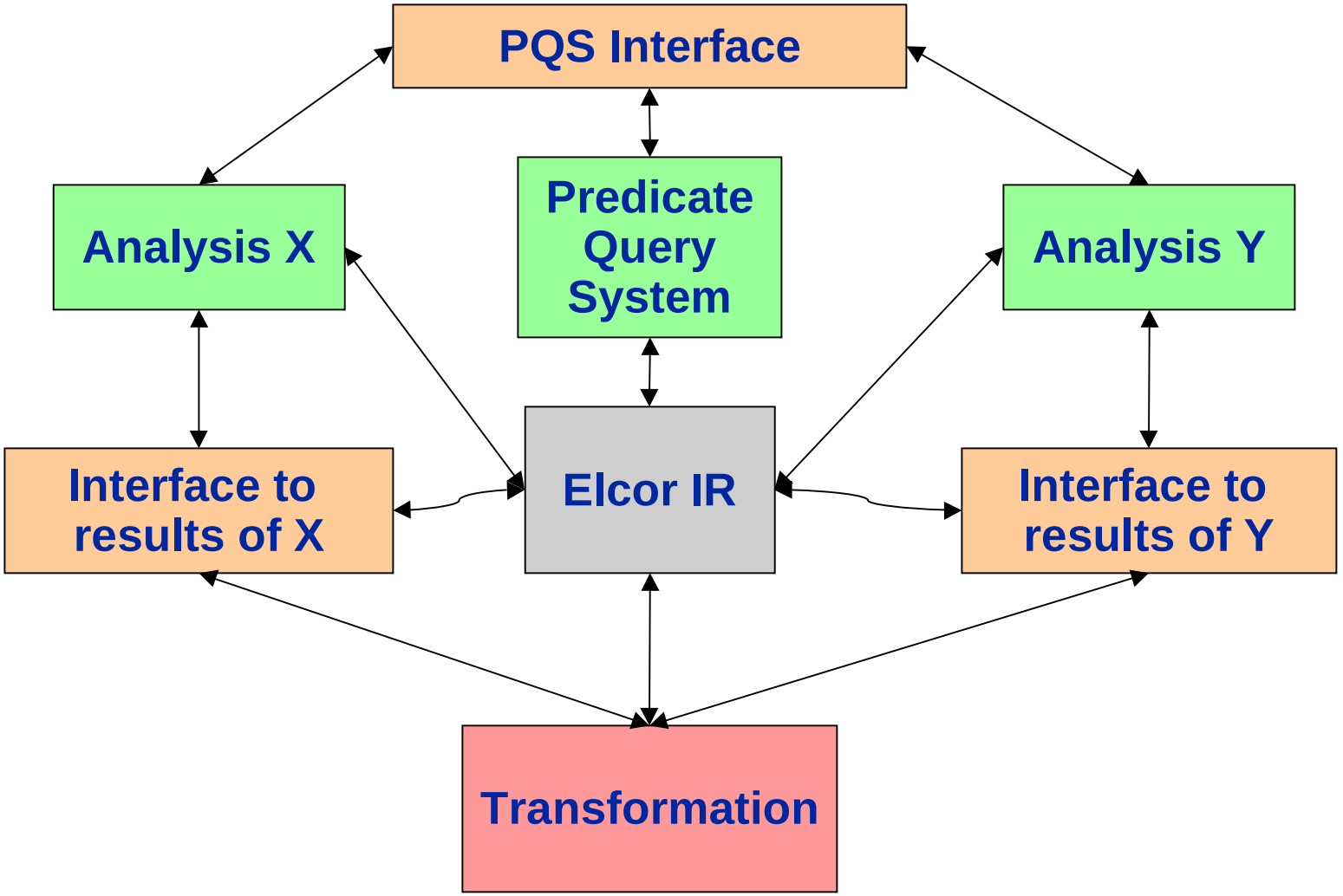
Data-flow analysis architecture

Uses a CFG consisting of basic-blocks and hyperblocks. Such a cut has to exist in the region hierarchy

Region based analysis has three steps

- Transfer functions are constructed for each entry-exit pair on a CFG node
  - Transfer functions are constructed using local predicate relationships. The transfer functions themselves are conventional bitvectors.
  - Transfer function construction handles REMAP operations
- Global iterative solver is conventional
  - Solves data-flow equations at basic/hyperblock entry exit points.
- Local analysis is used to determine data-flow equation solutions at points within a block using global solver results
  - Local analysis is predicate and REMAP aware

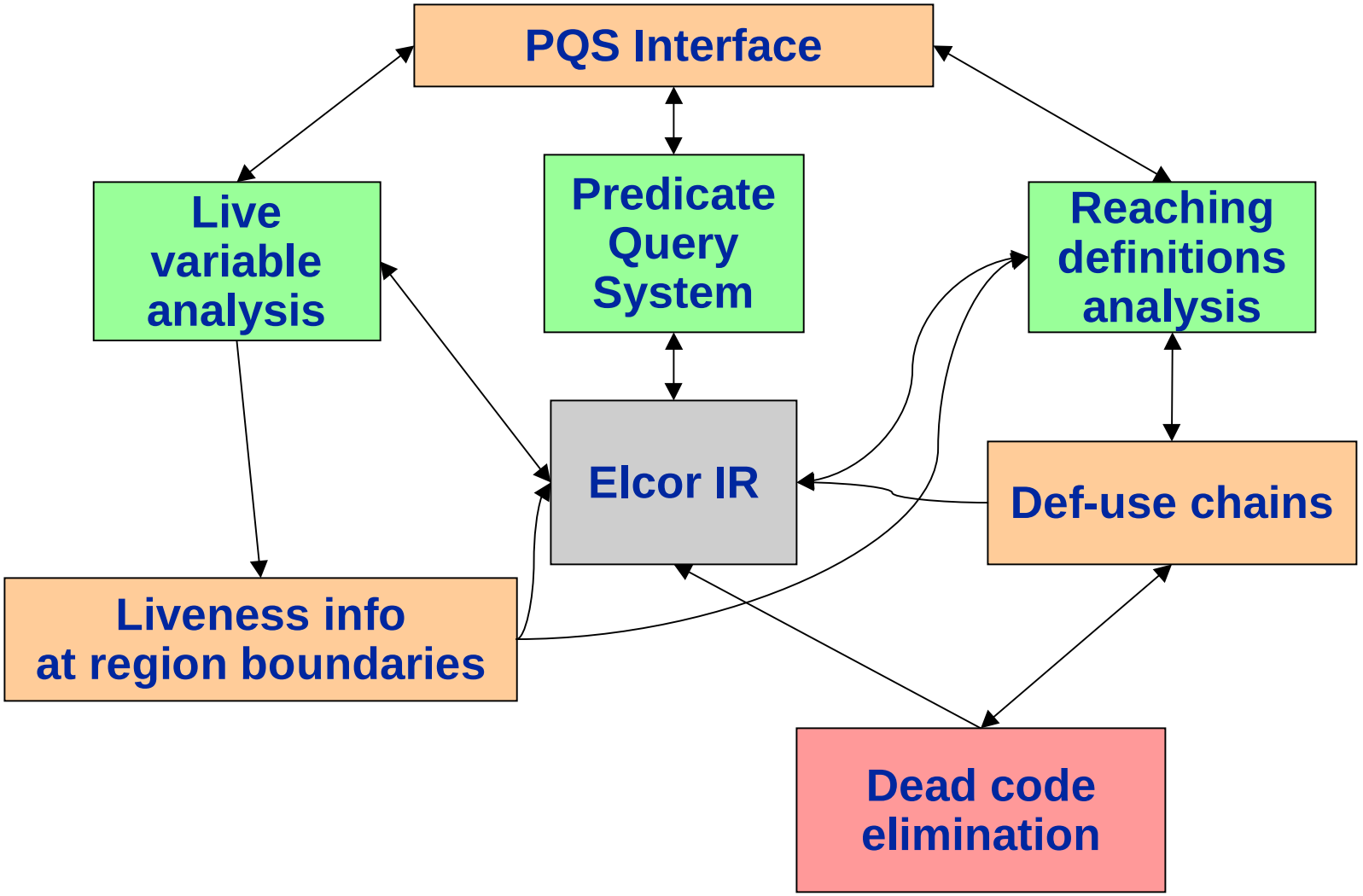
Elcor Optimization Architecture



Optimizations

- Predicate speculation
- Dead code elimination
- Global copy propagation (forward)
- Local copy propagation (forward and backward)
- Global common sub-expression elimination
- Loop-invariant code removal
- Global register renaming
  - Including web splitting

Region Based Dead Code Elimination



Region Based Predicated Dead Code Elimination

region **A**

region **B**

```
Op1 x := ... if p
Op2 x := ... if TRUE
Op3 x := ... if q

branch L2 if cond
```

region **C**

```
L2:
   y := x if TRUE
```

Assume:

- Region A is a procedure and B and C are regions in A
- x is live at region B exit on taken path only

region **B**

```
Op1 x := ... if p
Op2 x := ... if TRUE
Op3 x := ... if q

branch L2 if cond
(uses x if taken)
```

1) Op1 is always dead

2) Op2 is dead if branch is taken implies Op3 executes

Edge Drawing

Edge drawing modules draw dependence edges threaded through control-flow within a region

If the region is cyclic, edge-drawn graph may contain dependence cycles

- Register name changes are tracked through REMAP operations in drawing edges

Edge drawing inserts register flow/anti/output dependence, memory dependence, and control dependence edges.

- Dependence graph is used by scheduling modules
- Dependence graph is used to compute dependence height (to guide optimizations/transformations)

Loop Scheduling And Register Allocation

- Modulo scheduler
  - Iterative modulo scheduler which generates kernel-only code for counted loops
- Stage scheduler
  - A post-pass to the modulo scheduler to decrease register pressure
- Rotating register allocator
  - Allocates EVR's within the kernel only code to rotating registers.
  - Does not allocate static registers

Acyclic Scheduling

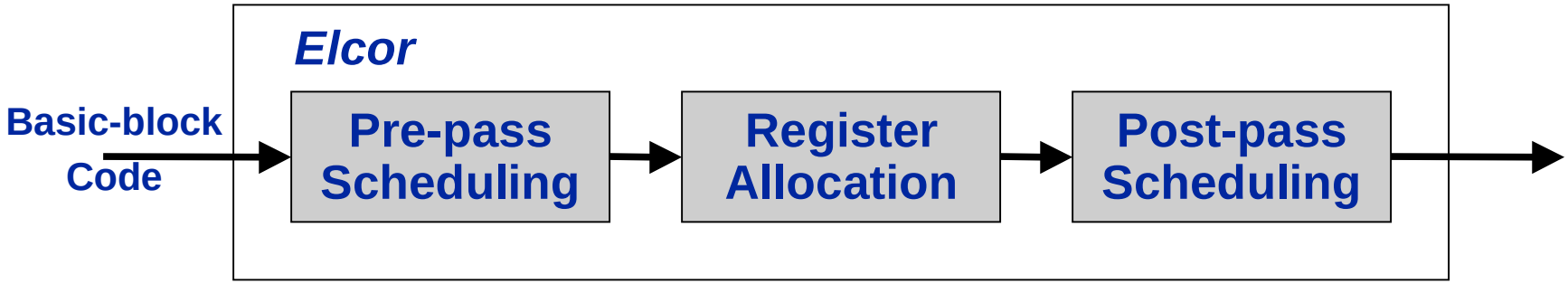
Schedulers for superblocks or hyperblocks

- Cycle scheduler
  - Generates a schedule by building instructions for each issue cycle in order
  - Supports cache miss sensitive scheduling if cache miss profile information is available
  - Supports use-of data speculative loads (LDS/LDV pair) if memory dependence profile information is available
- Backtracking scheduler
  - Limited backtracking, only for branches with delay slots and operations displaced by branches
  - Unlimited backtracking
- Meld scheduler
  - Propagates latency constraints across region boundaries

Execution Paths

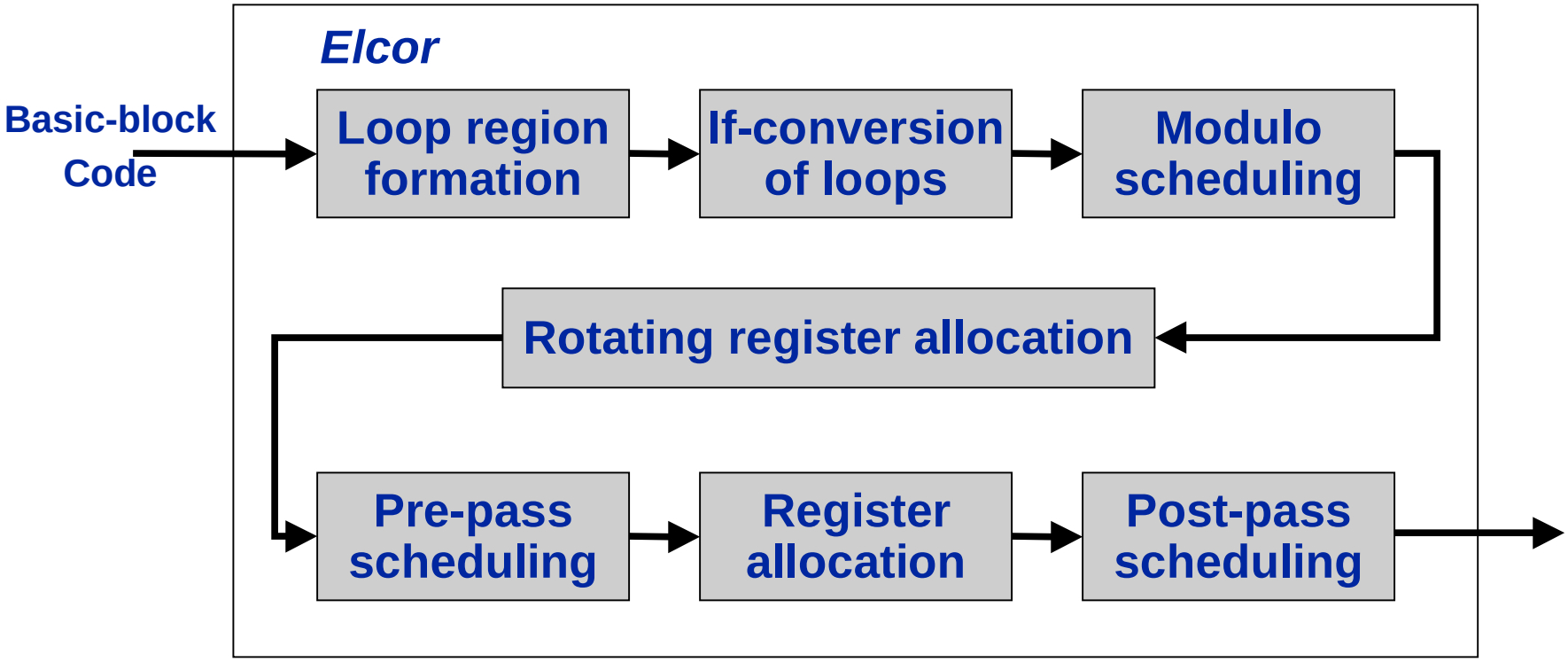
It is possible to craft many execution scripts depending on the state of the input and desired output.

A simple path for executing a program would be:



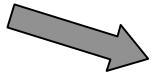
Execution Paths (cont.)

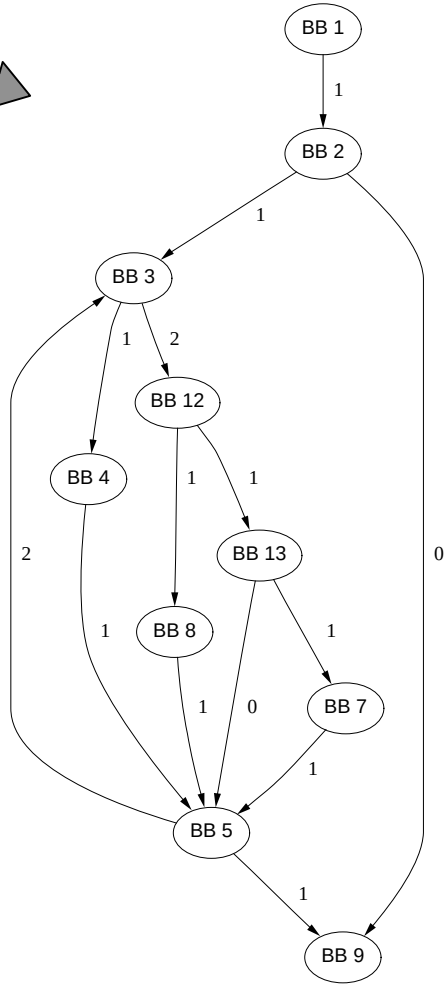
A more complex path may be:



Instrumentation/Visualization

- Control flow graph visualization using Dot
- Dependence graph visualization using Dot





- Built in timer for measuring time spent in large modules.
- Compile time statistics on
  - Schedule length of regions
  - Opcode usage
  - Program and per procedure estimates of
    - **Execution time**
    - **Number of operations**

Summary

Elcor is a collection of software components that are designed to be flexible, modular, and interface with each other.

These components:

- Are predicate and REMAP aware.
- Region based.
- Perform ILP optimizations/transformations.
- Provide common program analysis.
- Provide conventional optimizations.
- Can be used as building blocks for exploring new ILP compilation techniques.