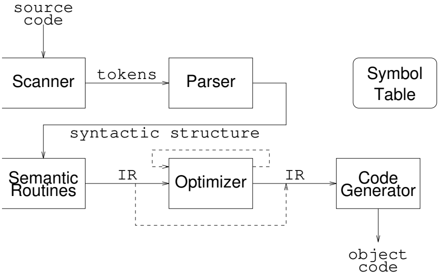
Liveness Analysis and Register Allocation

The Compiler



Challenges in the back end

- The input to the backend: IR.
- The target program – instruction set, constraints, number of registers, etc.
- Instruction selection (undecidable): maps groups of IR instructions to one or more machine instructions. Why not say each IR instruction maps to one more more machine level instructions?
  - Easy, if we don't care about the efficiency.
  - Choices may be involved (add / inc); may involve understanding of the context in which the instruction appears.
- Register Allocation (NP-complete): Intermediate code has unbounded number of temporaries. Need to translate them to registers (fastest storage).
  - Finite number of registers.
  - If we cannot allocate on registers, store in the memory – will be expensive.
  - Sub problems: Register allocation, register assignment, spill location, coalescing. All NP-complete.

The Memory Hierarchy

	Access Time	Size
Registers	1 cycle	256-8000 bytes
Cache	5-10 cycles	256 KB - 40 MB
Main Memory	20-100 cycles	4 GB - 32+ GB
Disk	0.5-5M cycles	1-10 TB

Managing the Memory Hierarchy

- Most programs are written as if there are only two kinds of memory: main memory and disk.
- Programmer is responsible for moving data from disk to memory (i.e. file I/O).
- Hardware is responsible for moving data between memory and caches.
- Compiler is responsible for moving data between memory and registers.

Managing the Memory Hierarchy

- Note that there is an order of magnitude difference between register/cache access and main memory access.
- Hence, it is very important to
  - Manage caches properly.
  - Manage registers properly.
- Cache behaviour is in general unpredictable (actually undecidable).
  - Hence, as a compiler designer, managing registers properly becomes even more important!

Register allocation

Register allocation:

- have value in a register when used
- limited resources
- can effect the instruction choices
- can move loads and stores
- optimal allocation is difficult
  $\Rightarrow$ NP-complete for $k \geq 1$ registers

Basic blocks

A graph representation of intermediate code.

Each basic block is a maximal sequence of 3-address code instructions with following properties:

- The flow of control can only enter the basic block through the first instruction in the block.
- No jumps into the middle of the block.
- Control leaves the block without halting / branching (except may be the last instruction of the block).

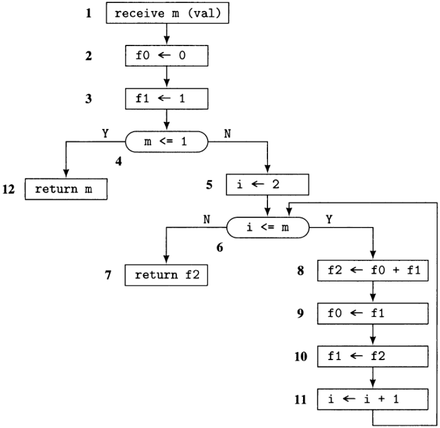
The basic blocks become the nodes of a flow graph, whose edges indicate which blocks can follow which other blocks.

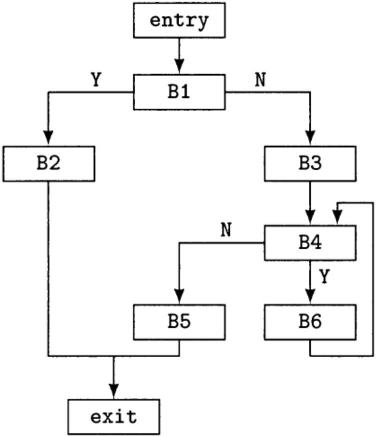
Example

```
unsigned int fib(m)
   unsigned int m;
{  unsigned int f0 = 0, f1 = 1, f2, i;
   if (m <= 1) {
      return m;
   }
   else {
      for (i = 2; i <= m; i++) {
         f2 = f0 + f1;
         f0 = f1;
         f1 = f2;
      }
      return f2;
   }
}
```

```
     receive m (val)
     f0 ← 0
     f1 ← 1
     if m <= 1 goto L3
     i ← 2
L1:  if i <= m goto L2
     return f2
L2:  f2 ← f0 + f1
     f0 ← f1
     f1 ← f2
     i ← i + 1
     goto L1
L3:  return m
```

Example - flow chart and control-flow





- The high-level abstractions might be lost in the IR.
- Control-flow analysis can expose control structures not obvious in the high level code.

CFG - Control flow graph

Definition:

- A rooted directed graph $G = (N, E)$, where N is given by the set of basic blocks + two special BBs: `entry` and `exit`.
- And edge connects two basic blocks b_1 and b_2 if control can pass from b_1 to b_2.
- An edge(s) from `entry` node to the initial basic block(s?)
- From each final basic blocks (with no successors) to `exit` BB.

Liveness analysis

Problem:

- IR contains an unbounded number of temporaries
- machine has bounded number of registers

Approach:

- temporaries with disjoint live ranges can map to same register
- if not enough registers then spill some temporaries
  (i.e., keep them in memory)

The compiler must perform liveness analysis for each temporary:

It is live if it holds a value that may be needed in future

Example

$$
\begin{aligned}
L_1 : \quad & a \leftarrow 0 \\
& b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& \text{if } c < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$

a and b can be allocated to the same register

Example

$$
\begin{aligned}
L_1:\quad & a \leftarrow 0 \\
& b \leftarrow a + 1 \\
& c \leftarrow c + b \\
& \text{if } c < N \text{ goto } L_1 \\
& \text{return } c
\end{aligned}
$$

$$
\begin{aligned}
L_1:\quad & r_1 \leftarrow 0 \\
& r_1 \leftarrow r_1 + 1 \\
& r_2 \leftarrow r_2 + r_1 \\
& \text{if } r_2 < N \text{ goto } L_1 \\
& \text{return } r_2
\end{aligned}
$$

Liveness analysis

Gathering liveness information is a form of data flow analysis operating over the CFG:

- We will treat each statement as a different basic block.
- liveness of variables "flows" through the edges of the graph
- assignments define a variable, v:
  - *def*$(v) = \text{set of graph nodes that define } v$
  - *def*$[n] = \text{set of variables defined by } n$ where n is a BB.
- occurrences of v in expressions use it:
  - *use*$(v) = \text{set of nodes that use } v$
  - *use*$[n] = \text{set of variables used in } n$ where n is a BB.

Definitions

- v is live on edge e if there is a directed path from e to a use of v that does not pass through any *def*(v)
- v is live-in at node n if live on any of n's in-edges
- v is live-out at n if live on any of n's out-edges
- $v \in$ *use*$[n] \Rightarrow v$ live-in at n (recall: each statement is its own basic block).
- v live-in at $n \Rightarrow v$ live-out at all $m \in$ *pred*$[n]$
- v live-out at $n \wedge v \notin$ *def*$[n] \Rightarrow v$ live-in at n

Liveness analysis

Define:

$$\begin{aligned} \mathit{in}[n] &= \text{variables live-in at } n \\ \mathit{out}[n] &= \text{variables live-out at } n \end{aligned}$$

Then:

$$\mathit{out}[n] = \bigcup_{s \in \mathit{succ}(n)} \mathit{in}[s]$$

$$\mathit{succ}[n] = \phi \quad \Rightarrow \quad \mathit{out}[n] = \phi$$

Note:

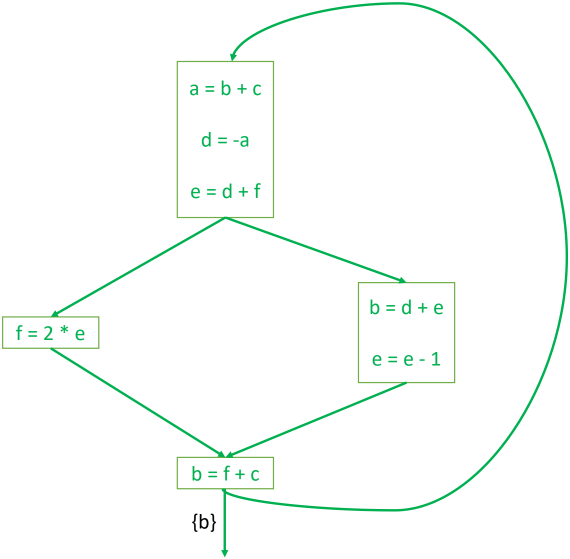
$$\begin{aligned} \mathit{in}[n] &\supseteq \mathit{use}[n] \\ \mathit{in}[n] &\supseteq \mathit{out}[n] - \mathit{def}[n] \end{aligned}$$

$\mathit{use}[n]$ and $\mathit{def}[n]$ are constant (independent of control flow)
Now, $v \in \mathit{in}[n]$ iff. $v \in \mathit{use}[n]$ or $v \in \mathit{out}[n] - \mathit{def}[n]$
Thus, $\mathit{in}[n] = \mathit{use}[n] \cup (\mathit{out}[n] - \mathit{def}[n])$

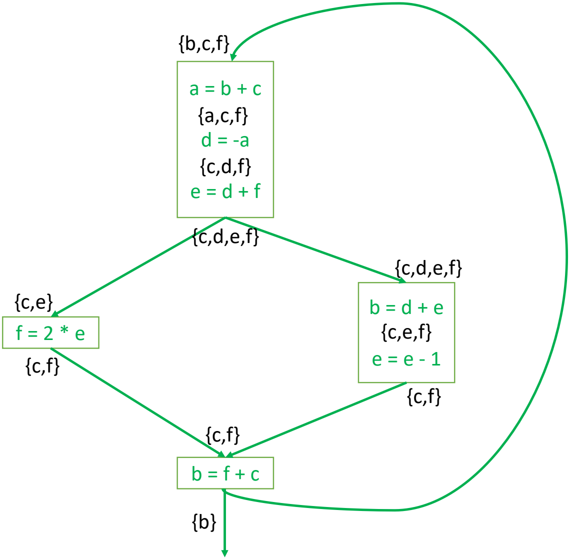
Iterative solution for liveness

N : Set of nodes of CFG;
foreach $\underline{n \in N}$ **do**
    $in[n] \leftarrow \phi$;
    $out[n] \leftarrow \phi$;
end
repeat
    foreach n $\in$ Nodes **do**
        $in'[n] \leftarrow in[n]$;
        $out'[n] \leftarrow out[n]$;
        $in[n] \leftarrow use[n] \cup (out[n] - def[n])$;
        $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$;
    end
until $\underline{\forall n, in'[n] = in[n] \wedge out'[n] = out[n]}$;

Example



Example



Notes

- should order computation of inner loop of the data-flow analysis algorithm to follow the “flow”
- liveness flows backward along control-flow arcs, from out to in
- nodes can just as easily be basic blocks to reduce CFG size
- could do one variable at a time, from uses back to defs, noting liveness along the way

Iterative solution for liveness

Complexity: for input program of size N

- $\leq N$ nodes in CFG
  $\Rightarrow \leq N$ variables
  $\Rightarrow N$ elements per *in/out*
  $\Rightarrow O(N)$ time per set-union
- **for** loop performs constant number of set operations per node
  $\Rightarrow O(N^2)$ time for **for** loop
- each iteration of **repeat** loop can only add to each set
  sets can contain at most every variable
  $\Rightarrow$ sizes of all in and out sets sum to $2N^2$,
  bounding the number of iterations of the **repeat** loop

$\Rightarrow$ worst-case complexity of $O(N^4)$

- ordering can cut **repeat** loop down to 2-3 iterations
  $\Rightarrow O(N)$ or $O(N^2)$ in practice

Least fixed points

There is often more than one solution for a given dataflow problem.

- For example, if a variable x is never used or defined, then adding x to the in and out sets of every basic block is also a valid solution.

Many possible solutions but we want the “smallest”: the least fixpoint.
The iterative algorithm computes this least fixpoint.
Any solution to dataflow equations is a conservative approximation:

- v has some later use downstream from n
  $\Rightarrow v \in \mathit{out}(n)$
- but not the converse

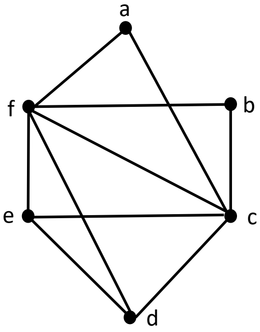
Conservatively assuming a variable is live does not break the program; just means more registers may be needed.
Assuming a variable is dead when really live will break things.

Register Interference Graph

- An undirected graph
  - A node for each temporary/variable.
  - An edge between t_1 and t_2 if they are live simultaneously at some point in the program.
- Two temporaries can be allocated to the same register if there is no edge connecting them.
- The register interference graph extracts exactly the information needed to perform legal register assignments.
- Also gives a global picture (i.e. over the entire flow graph) of the register requirements.

Example: Register Interference Graph

The Register Interference Graph for our example program:

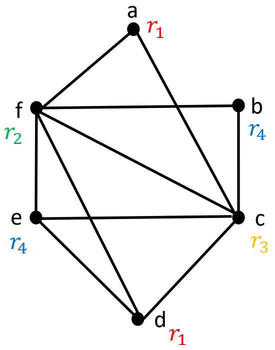


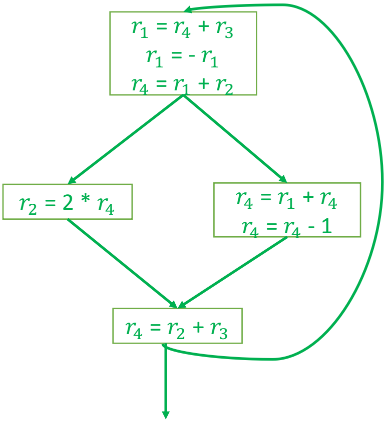
- b and c cannot be assigned the same register.
- b and d can be assigned the same register.
- How many registers?Same as the number of colors required for coloring the graph.

Register allocation - by Graph coloring

- Step 1:
  - Select target machine instructions assuming infinite registers (temps).
  - If a instruction requires a special register – replace that temp with that register.
- Step 2:
  - Construct the interference graph.
  - Solve the register allocation problem by coloring the graph.
  - A graph is said to be colored if each pair of neighboring nodes have different colors.

Example: Colored RIG





Computing Graph Colorings

- How do we compute graph colorings?
- The problem is NP-Hard. No efficient algorithms are known.
  - Solution: We will use heuristics.
- A coloring may not exist for a given number of registers/colors.
  - Solution: We will use systematic spilling.

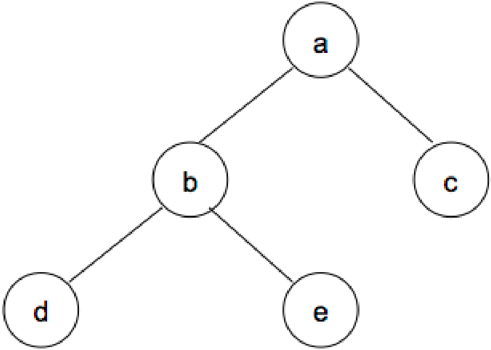
Graph Coloring Heuristic

- Observation:
  - Pick a node t with fewer than k neighbours in RIG.
  - Eliminate t and its edges from RIG.
  - If the resulting graph is k-colorable, then so is the original graph.
- Why?
  - Let $c_1, c_2, \ldots, c_n$ be the colors assigned to neighbours of t in the reduced graph.
  - Since $n < k$, we can pick a color for t among k colors that is different from those of its neighbours.

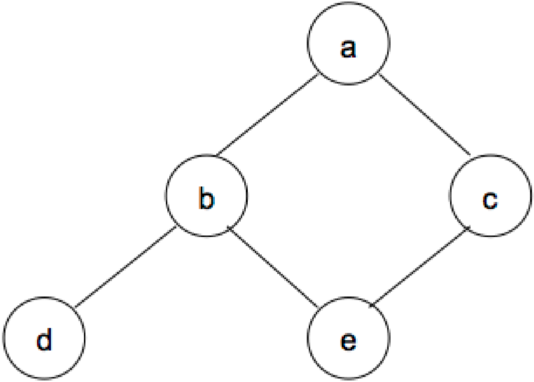
Graph coloring - a simplistic approach

```
Input: G - the interference graph, k - number of colors
repeat
    repeat
        Remove a node n and all its edges from G, such that degree of n is
          less than K;
        Push n onto a stack;
    until G has no node with degree less than k;
    // G is either empty or all of its nodes have
       degree ≥ k
    if G is not empty then
        Take one node m out of G, and mark it for spilling;
        Remove all the edges of m from G;
    end
until G is empty;
Take one node at a time from the stack and assign a non conflicting color.
```

Example 1, available colors = 2



Example 2



We have to spill.

Graph coloring - Kempe's heuristic

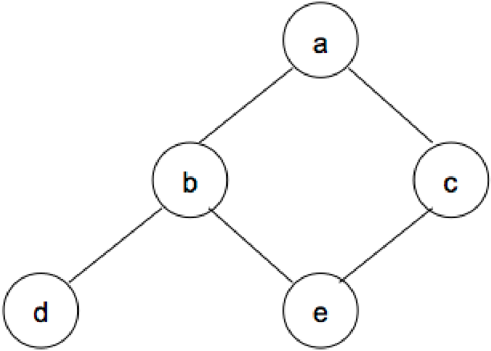
- Algorithm dating back to 1879.
- Also called 'optimistic coloring'.

Input: G - the interference graph, K - number of colors

```
repeat
    repeat
        Remove a node n and all its edges from G, such that degree of n is
          less than K;
        Push n onto a stack;
    until G has no node with degree less than K;
    // G is either empty or all of its nodes have
       degree ≥ K
    if G is not empty then
        Take one node m out of G.;
        push m onto the stack;
    end
until G is empty;
```

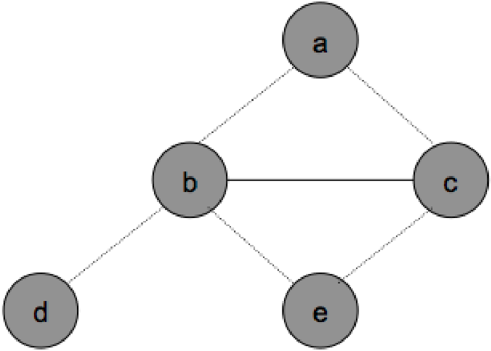
Take one node at a time from the stack and assign a non conflicting color (if possible, else spill).

Example 2 (revisited)



We don't have to spill.

Example 3



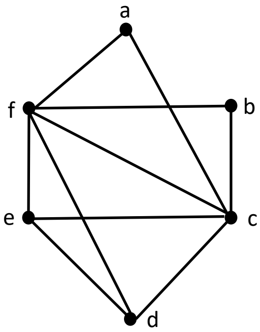
Don’t have a choice. Have to spill.

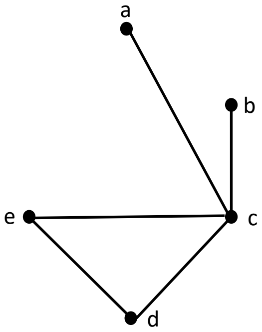
Spilling

- We need to generate extra instructions to load variables from the stack and store them back.
- The load and store may require registers again:
  - Naive approach: Keep a separate register (wasteful).
  - Rewrite the code - by introducing a temporary; rerun the liveness + register-allocation
    (Note: the new temp has much smaller live range).

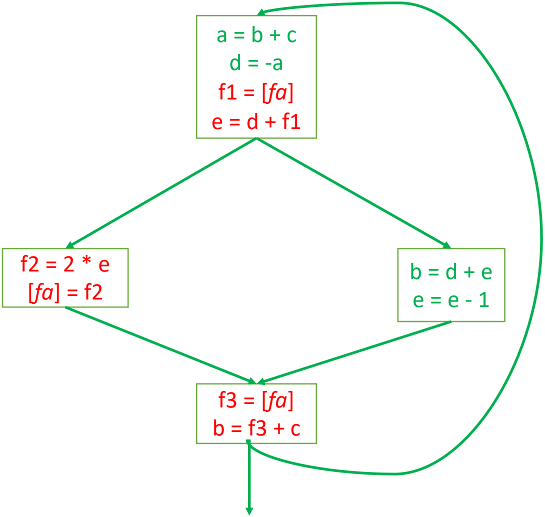
Example: Spilling

Going back to our running example, suppose we only have 3 registers, and decide to spill f.

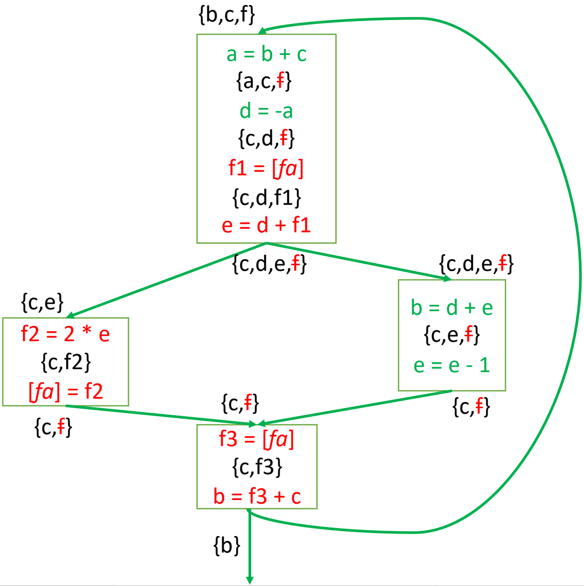




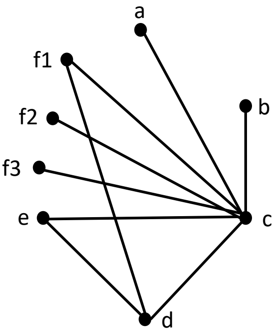
Example: Rewrite for spilled variable



Example: Recomputing Liveness after rewrite



Example: RIG after rewrite



Homework: Find a register allocation using 3 registers, and rewrite the program using the allocated registers.

Register allocation - Linear scan

Register allocation is **expensive**.

- Many algorithms use heuristics for graph coloring.
- Allocation may take time quadratic in the number of live intervals.

Not suitable

- Online compilers – need to generate code quickly. e.g. JIT compilers.
- Sacrifice efficient register allocation for compilation speed.

Linear scan register allocation - Massimiliano Poletto and Vivek Sarkar, ACM TOPLAS 1999

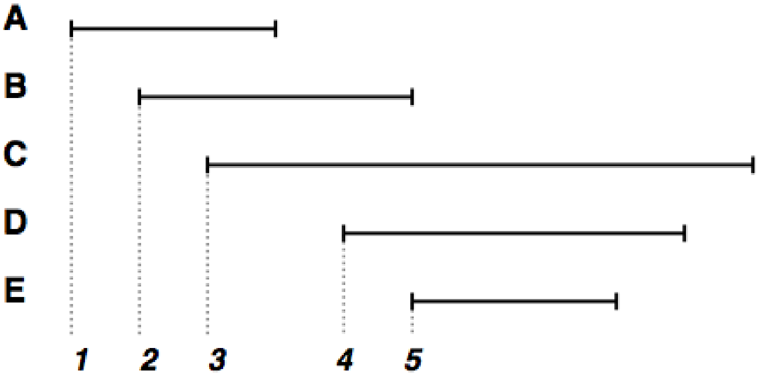
Linear Scan algorithm

```
LINEARSCANREGISTERALLOCATION
    active ← {}
    foreach live interval i, in order of increasing start point
        EXPIREOLDINTERVALS(i)
        if length(active) = R then
            SPILLATINTERVAL(i)
        else
            register[i] ← a register removed from pool of free registers
            add i to active, sorted by increasing end point

EXPIREOLDINTERVALS(i)
    foreach interval j in active, in order of increasing end point
        if endpoint[j] ≥ startpoint[i] then
            return
        remove j from active
        add register[j] to pool of free registers

SPILLATINTERVAL(i)
    spill ← last interval in active
    if endpoint[spill] > endpoint[i] then
        register[i] ← register[spill]
        location[spill] ← new stack location
        remove spill from active
        add i to active, sorted by increasing end point
    else
        location[i] ← new stack location
```

Example



- Say, available registers = 2

Linear Scan algorithm - analysis

- Each live range gets either a register or a spill location.
- Note: The number of overlapping intervals changes only at the start and end points of an interval.
- Live intervals are stored in a list that is sorted in order of increasing start point.
- The active list is kept sorted in order of increasing end point. Adv: need to scan only those intervals (+1 at most) that have to be removed.
- Complexity: $O(V)$ – if number of registers is assumed to be a constant. Else? $O(V \times logR)$
- Many more details can be found in the paper.
  - Study section 4 of the paper (Poletto et al. ACM TOPLAS 1999).