Study Guide

Optimising Compilers @ Lunds Universitet

Basics

Lattice

It's a partially ordered set where every pair has least upper and greatest lower bounds (supremum and infimum) which are unique. They are called the *join* ($\vee$) and *meet* ($\wedge$), respectively.

Tarjan's algorithm for Strongly Connected Components

```
proc tarjan() ≡
  components = new Set
  dfsNumIndex = 0
  stack = new Stack
  foreach vertex v in V :
    if (dfsNum[v] = null) →
        strongConnect(v)

proc strongConnect(vertex v) ≡
  dfsNum[v] = dfsNumIndex
  lowLink[v] = dfsNumIndex
  dfsNumIndex += 1
  stack.push(v)
  foreach w in succ(v):
    if (dfsNum[w] = null) →
        strongConnect(w)
        lowLink[v] = min(lowLink[v], lowLink[w])
    elif (dfsNum[w] < dfsNum[v] ∧ stack.contains(w)) →
        lowLink[v] = min(lowLink[v], dfsNum[w])
  if (lowLink[v] = dfsNum[v]) →
      scc = new Set
      do
          w = stack.pop()
          scc.add(w)
      while (w ≠ v)
      components.add(scc)
```

Dominance Analysis

Compute dominance

Theory

1. $v \ggeq w$ **iff** every path from s to w includes v.
   v **dominates** w.

2. $dom(w) = \{v \mid v \ggeq w\}$

3. $\ggeq$ is a partial order relation.

4. $u \ggeq v$ **iff** $u \ggeq p_i$ for every $p_i \in pred(v)$

5. $dom(w) = \{w\} \cup \bigcap_{p \in pred(w)} dom(p)$

Algorithm

proc *computeDominance*() $\equiv$
  $dom[s] = \{s\}$
  foreach *vertex* v *in* V *except* s:
    $dom[v] = V$
  while *some* $dom[v]$ *has changed*:
      foreach *vertex* v *in* V *except* s:
        $dom[v] = \{v\} \cup \bigcap_{p \in pred(v)} dom[p]$

Lenguaer-Tarjan Algorithm (Dominator Tree)

Theory

1. $v \gg w$ **iff** $v \ggeq w \wedge v \neq w$
   v **strictly dominates** w.

2. $v \in dom(w) \wedge u \in dom(w) \Rightarrow u \ggeq v \vee v \ggeq u$

3. $v = iDom(w)$ **iff** $v \gg w \wedge (\forall u \mid u \gg w : v \not\gg u)$
   v **is the immediate dominator of** w.

4. The graph of the reflexive and transitive closure of the dominance relation is a tree, called the dominator tree (DT). $(u, v) \in DT$ **iff** $u = iDom(v)$.

5. $u \ggeq v$ **iff** $dfsNum(u) \leq dfsNum(v) \wedge$
   $dfsNum(v) \leq dfsNum(u) + numberOfDescendants(u)$

6. $v = sDom(w)$ **iff** v is the smallest vertex by $dfsNum$ such that there is a path $(v, u_1, u_2, ..., u_{k-1}, w)$ with $dfsNum(u_i) > dfsNum(w)$.
   The semidominator is frequently equal to the immediate dominator.

Algorithm *parent* is the parent in the DFS tree after calling $dfs()$.

```
proc lenguaerTarjan() ≡
  emptyBuckets()
  setSdominatorsToSelf()
  dfs()
  foreach vertex v in reverse dfsNumber order:
    foreach predecessor p of v:
      u = ancestorWithLeastSdom(p)
      if (dfsNum(sDom[u]) < dfsNum(sDom[v])) →
          sDom[v] = sDom[u]
    bucket[sDom[v]].add(v)
    ancestor[v] = parent(v)
    foreach vertex b in bucket[parent(v)]:
      remove b
      u = ancestorWithLeastSdom(b)
      if (dfsNum(sDom[u]) < dfsNum(sDom[b])) →
          iDom[b] = u
      else
          iDom[b] = parent(v)
  foreach vertex v except s:
    if iDom[v] ≠ sDom[v] →
        iDom[v] = iDom[iDom[v]]

proc ancestorWithLeastSdom(vertex w) ≡
  go through the ancestor chain of w
  return ancestor with the least sDom
```

Loop Analysis

Basics

1. A loop is a subgraph of the CFG that is a strongly connected component
2. An entry loop is a vertex with a predecessor that is not in the loop
3. Irreducible loops have multiple entry points

4. Reducible or natural loops only have one entry point: the loop header. It dominates all other vertices in the loop.

forward arc	$u < v$
back arc	$u > v \wedge$ v is left in the stack

proc *identifyNaturalLoops*() $\equiv$
  foreach *back edge* (v, w) *of* G:
    if $(w.dominates(v)) \rightarrow$
      search backwards all paths from v *to* w.
      the visited vertices belong to the natural loop

Dataflow Analysis

Basics

- It is concerned with how a procedure uses and defines its variables and expressions
- Every assignment statement is given an index
- Local analysis discovers what happens within a vertex: its output is $gen(v)$ and $kill(v)$
- Global analysis: each vertex has $in(v)$ and $out(v)$ in addition to gen and $kill$.

$in(v)$	definitions reaching the beginning of v
$out(v)$	definitions reaching the end of v
$gen(v)$	definitions generated in v
$kill(v)$	definitions killed in v

proc *reachingDefs*() $\equiv$
  workList = vertices in reverse postorder
  while *workList is not empty*:
    $v = workList.first()$
    $old = out[v]$
    $$in[v] = \bigcup_{p \in pred(v)} out[p]$$
    $out[v] = gen[v] \cup (in[v] - kill[v])$
    if $(old \neq out(v)) \rightarrow$
      foreach *succesor* s *of* v:
        if (s *is not in workList*) $\rightarrow$
          $workList.add(s)$

```
proc localLiveAnalysis() ≡
  foreach vertex v:
    foreach statement s:
      foreach used variable x of s:
        if (x ∉ kill[v]) →
            use[v].add(x)
      foreach defined variable x of s:
        if (x ∉ gen[v]) →
            kill[v].add(x)

proc globalLiveAnalysis() ≡
  while some in[v] has changed:
        foreach vertex v :
          out[v] = ⋃_{s∈succ(v)} in[s]
          in[v] = use[v] ∪ (out[v] − kill[v])
```

Static Single Assignment Form

- Every variable is defined as most once (statically)
- ϕ-functions: one argument for each predecessor at some join vertices

Dominance Frontiers

Theory

1. $DF(v) = \{w \mid (\exists p \mid p \in pred(w) : v \geqq p \wedge v \not\gg w)\}$
   the **dominance frontier of vertex v**
   the set of vertices w such that v dominates a predecessor of w, but does not strictly dominate w.
   if v dominates a predecessor of w but does not strictly dominate w, then w is in the dominance frontier of v.

2. $DF_{local}(v) = \{w \in succ(v) \mid v \not\gg w\}$

3. $DF_{up}(v) = \{w \in DF(v) \mid iDom(v) \not\gg w\}$

4. $DF(v) = DF_{local}(v) \cup \bigcup_{w \in children(v)} DF_{up}(w)$

5. $DF_{local}(v) = \{w \in succ(v) \mid iDom(w) \neq v\}$

6. $DF_{up}(v) = \{w \in DF(v) \mid iDom(w) \neq v\}$

Algorithm

proc *dominanceFrontiers*() $\equiv$
  foreach *vertex v in a postorder traversal of DT* :
    dominanceFrontier[*v*] = *new Set*
    foreach *vertex w in succ*(*v*):
      if (*iDom*(*w*) $\neq$ *v*) $\rightarrow$
          dominanceFrontier[*v*].*add*(*w*)
    foreach *vertex w in children*(*v*):
      foreach *vertex u in dominanceFrontier*[*w*]:
        if (*iDom*(*u*) $\neq$ *v*) $\rightarrow$
            dominanceFrontier[*v*].*add*(*u*)

Insertion of ϕ-functions

Theory

1. Two paths $p = (v_0, ..., v_X)$ and $q = (w_0, ..., w_Y)$ **converge** at u **iff**

   a) $v_0 \neq w_0$

   b) $v_X = w_Y = u$

   c) $(u_i = v_j) \Rightarrow i = X \vee j = Y$

2. Given a set A of vertices, the join of A ($J(A)$) is the set of all vertices w such that there are two distinct vertices u and v in A with paths which converge at w.

3. $J_{i+1}(A) = J(A \cup J_i(A))$, $J^+(A)$ is the fixed-point.

4. DF can be applied to sets of vertices, mapping DF over all the vertices of its argument.

5. $DF_{i+1}(A) = DF(A \cup DF_i(A))$, and $DF^+(A) = J^+(A)$.

6. The set of vertices which need ϕ-functions for any variable v is the iterated dominance frontier $DF_+(A)$, where A is the set of vertices with assignment statements to v.

Algorithm

```
proc insert_φ() ≡
  iteration = 0
  foreach vertex v:
    hasAlready[v] = 0
    work[v] = 0
  W = new Set
  foreach variable x:
    iteration += 1
    foreach vertex v ∈ A(x):
      work[v] = iteration
      W.add(v)
    while (W is not empty):
        v = take vertex from W
        foreach w in DF(v):
          if (hasAlready[w] < iteration) →
              place x = φ(x, ..., x) at w
              hasAlready[w] = iteration
              if (work[w] < iteration) →
                  work[w] = iteration
                  W.add(w)

proc renameVariables() ≡
  foreach variable x:
    C[x] = 0
    S[x] = new Stack
  search(s)      // s = the start vertex
```

proc *search*(*vertex* v) $\equiv$
  oldLHS = *new List*
  foreach *assignment statement* s *in* v:
    foreach *variable* x *in* $RHS(s)$:
      $i = S[x].top()$
      replace use of x *by use of* x_i
    foreach *variable* x *in* $LHS(s)$:
      oldLHS.add(x)
      $i = C[x]$
      replace x *by* x_i
      $S[x].push(i)$
      $C[x]$ `+=` 1
  foreach w *in* $succ(v)$:
    j = *which predecessor is v to w?* `// first, second, ... ?`
    foreach ϕ*-function in* w:
      $i = S[x].top()$
      replace use of the j*-th operand in* $RHS(\phi)$ *by use of* x_i
  foreach w *in* $children(v)$:
    search(w)
    `// pop every variable version pushed in v`
    foreach *variable* x *in* $oldLHS(s)$:
      $S[x].pop()$

SSA Optimisation

Copy propagation

```
int x;
int t;
t = a + b;
x = t;
```
is translated into...
```
int t;
t = a + b;
```

- It considers all copy statements such as `x = t` in a procedure and propagates the source by replacing uses of the destination by uses of the source instead.
- If the destination is no longer used anywhere, it can be removed.

- When SSA form is not used, copy propagation cannot be performed without limitations: careful about redefinitions. One can use iterative dataflow analysis.

- Consider `x = t` and an expression `x + y` to which we want to propagate `t`. On SSA form, it is always legal to propagate `t` by replacing uses of `x` with uses of `t`. Recall that **at any use of a variable, there is exactly one reaching definition of such a variable and the definition of the variable dominates its use**.

- Copy propagation can be performed during variable renaming: when the statement is a copy statement `x = t`, the original renaming algorithm replaces `t` with the top of the stack of `t`, say `t'`, and then replaces `x` with a new version of `x` which is pushed on stack `x`. The new copy statement becomes `x' = t'`. To do copy propagation during renaming, we simply push `t'` on the stack of `x` and delete the copy statement.

```
proc copy() ≡
  foreach variable x:
    if (def[x] is move ∧ source[def[x]] is var) →
        y = source(def(x))
        foreach use u of x:
          replace x by y
          add u to uses[y]
        remove def[x] from the program
        remove x from the program
```

Copy propagation

- Simplifies expressions with compile-time constant operands. It can change conditional branches to unconditional and delete code.

- Initially only the start vertex is known to be executable.

- CFG arcs on a worklist: it contains arcs which have been discovered to possibly be eecutable.

- At ϕ-functions, only values from operands corresponding to CFG arcs marked as executable are inspected. ϕ-functions can become constant.

- A lattice is used with three element types, $\top$, $\perp$ and c_i.

- Initially, each lattice cell is $\top$, the unknown value. If any of the source operands is $\bot$, the result becomes $\bot$, if both are constants the result becomes constant. Otherwise, the result remains $\top$.

- The first time a vertex is visited, all statements are interpreted. Subsequent visits to the vertex only need to interpret the ϕ-functions.

```
proc constantPropagation() ≡
  foreach definition d:
    value[d] = ⊤
  foreach vertex v:
    visited[v] = false
  visitVertex(s)
  while (arcWorkList ≠ ∅ ∨ ssaWorkList ≠ ∅):
        if (arcWorkList ≠ ∅) →
            arc = arkWorkList.first()
            if (¬executable[arc]) →
                executable[arc] = true
                visitVertex(head(arc))
        if (ssaWorkList ≠ ∅) →
            t = ssaWorkList.first()
            visitStatement(t)

proc visitVertex(v) ≡
  onlyPhi = visited[v]
  visited[v] = true
  foreach statement s in w:
    if (onlyPhi ∧ s is not φ) →
        continue
    visitStatement(s, v)
```

```
proc visitStatement(vertex v, statement s) ≡
  statementType = statementType(s)
  if (statementType is unconditional branch) →
       arcWorkList.add(v, succ(v))
  elif (statementType is conditional branch) →
         add appropriate arcs depending on what is known
         about the operands
  elif (statementType is add, mul, etc) →
         left = value of the first source operand
         right = value of the second source operand
         result = what can be determined from left and right
         if (result < value[s]) →
              add uses of destination of s to ssaWorkList
              value[s] = result
  elif (statementType is a φ-function) →
         result = ⊤
         foreach p ∈ pred(v):
           if (executable[(p, v)]) →
                value = value of φ-function operand for p
                result = infimum(result, value)
         if (result < value[s]) →
              add uses of destination of s to ssaWorkList
              value[s] = result
  ...
```

Dead Code Elimination

Control dependence

- Removes useless statements: a statement is useless if it cannot affect program output in any way
- First remove unreachable code: DFS from the start vertex and delete all unvisited vertices
- Early DCE algorithms: live variable analysis to remove assignments to local variables which have no use
- Other optimisations on SSA form frequently produce useless code

- New idea: statements which can directly affect output are `prelive` and are marked as `live`, then a search from their operands mark additional statements as `live`. These statements are put in `prelive`:

  1. Writes to global variables and through pointers
  2. Function calls to I/O routines
  3. Function calls with unknown side-effects
  4. Return statements

- When a statement is marked as `live`, all multiway branches which directly control whether that statement will be executed should also be marked as `live`.

Theory

1. The reverse control flow graph: s and e have switched roles.

2. Postdominance, $\lll$, is equivalent to dominance in the RCFG.

3. Control dependence is equivalent to dominance frontiers in the RCFG.

4. A vertex v is control dependent on $w \in CD^{-1}(v)$ if v postdominates a succesor of w but does not strictly postdominate w:
   $CD^{-1}(v) = \{w \mid (\exists s \in succ(w) \mid: v \lll s \ \wedge \ v \not\lll w)\}$

5. $w \in CD^{-1}(v)$ in the CFG **iff** $w \in DF(v)$ in the RCFG

Algorithm

proc *computeControlDependence*() $\equiv$
  build RCFG
  compute RDT
  compute RDF
  foreach *vertex* v:
    $CD[u]$ = *new Set*
  foreach *vertex* v:
    foreach *vertex* w *in* $RDF(v)$:
      $CD[w].add(v)$

Dead code elimination on SSA Form

`eliminateDeadCode` will return a CFG that consists of some vertices that are live and some that are not. Then, `simplify` will connect the live blocks and delete the others.

proc *eliminateDeadCode*() $\equiv$
  foreach *statement s*:
    if (*s is prelive*) $\rightarrow$
        live[s] $= true$
        workList.add(s)
    else
        live[s] $= false$
  while (*workList* $\neq \emptyset$):
        $s =$ *workList.first*()
        $v =$ *vertex*(s)
        live[v] $= true$
        foreach *source operand* ω *of* s:
          $t =$ *def*(ω)
          if ($\neg$*live*[t]) $\rightarrow$
              live[t] $= true$
              workList.add(t)
        foreach *vertex* $v \in CD^{-1}$(*vertex*(s)):
          $t =$ *multiway branch of v*
          if ($\neg$*live*[t]) $\rightarrow$
              live[t] $= true$
              workList.add(t)
  foreach *statement s*:
    if ($\neg$*live*[s] $\wedge$ *s is not a label* $\wedge$ *s is not a branch*) $\rightarrow$
        delete s
  simplify()

```
proc simplify() ≡
  live[e] = true
  change = false
  foreach vertex v in CFG:
    if (¬live[v]) →
        continue
    foreach w ∈ succ(v):
      if (live(w)) →
          continue
      u = iPdom(w)// inmediate postdominator
      replace (v, w) with (v, u)
      update the branch in v to its new target u
      change = true
  if (change) →
      delete vertices from CFG which have become unreachable
      update dominator tree
```

Instruction scheduling

Its purpose is to improve performance by reducing the number of pipeline stalls suffered during execution. The module of the optimiser which does this is the instruction scheduler, and it tries to remove pipeline stalls by changing the order of instructions in a basic block, for example. It is usually done before register allocation. Additional registers are needed to hold values when there are unrelated instructions between a producer and a consumer of a value (**register pressure**). It needs an instruction level data dependency graph.

The most fundamental instruction scheduling technique is **list scheduling**, and schedules one basic block at a time. First, an instruction level data dependency graph is built, based on the definition and uses of storage resources, e.g. variables, registers.

- `def(r)` is the instruction which most recently modified r while scanning backwards, or $\perp$.
- `uses(r)` is the set of instructions which use the current value of r.

Register allocation

If it is done one basic block at a time, it is called **local register allocation**. One of the best approaches is to model it as graph coloring. An undirected graph, **the interference graph** is constructed with live ranges as nodes and with an edge between the two nodes if the corresponding live ranges are live at the same time anywhere in the procedure. The graph should be colored with K colors (number of registers). One can decide to decide that certain variables must reside in memory (**spilling**).

- The interference graph is built
- The number of neighbors of a node is its **degree**
- Suppose K colors are available, and a node n has a degree less than K. If we were to remove n and the resulting graph is colorable with K colors, then the original graph is also colorable with K colors.
- The allocator keeps looking for a node n with fewer than K colors, removes it from the graph and pushes it into a stack. This is repeated until either the graph is empty or no node with degree less than K is found, in which case a node is removed from the graph and its variable is spilled.
- When the graph has become empty, one node n at a time is popped from the stack and reinserted into the graph. Then, a color which is not used by any neighbor of n is chosen and n is allocated this color.
- Register coalescing resembles copy propagation.

ADVANCED - SSA Optimisation

Partial Redundancy Elimination (PRE)

Basics

- Attempt to calculate expressions only once
- Partial redundancy: one of the expressions is inside an `if-else`
- Case examples

  1. ```
     if (a * b > max)
         max = a * b
     ```

     is translated into...

     ```
     t = a * b
     if (t > max)
         max = t
     ```
     ```

2. ```
   if (condition)
       x = a * b
   y = a * b
   ```

   is translated into...

   ```
   if (condition) {
       t = a * b
       x = t
   }
   else
       t = a * b
   y = t
   ```

3. ```
   do {
       x += a * b
   }
   while (x < y)
   ```

   is translated into...

   ```
   t = a * b
   do {
       x += t
   }
   while (x < y)
   ```

4. `do {`

   ```
   do {
       x += a * b
   }
   while (x < y)
   ```

   is translated into...

   ```
   t = a * b
   do {
       x += t
   }
   while (x < y)
   ```

5. ```
   if (condition) {
       // some code
       y = x + 4
   }
   else {
       // other code
   }
   z = x + 4
   ```

   is translated into...

   ```
   if (condition) {
       // some code
       t = x + 4
       y = t
   }
   else {
       // other code
       t = x + 4
   }
   z = t
   ```

- Loop improvement
- Global value numbering can optimise code that PRE cannot
- Common subexpression elimination and code motion of loop invariants are subsumed by PRE
   ```