Exame 2022

(1)

Problem 1. Run-Time Environments [10 points]

Many modern imperative programming languages support the abstraction of procedures and functions. To support the execution of procedures, compilers make use of activation records or AR that capture data related to the execution of procedures and functions. In this context, answer the following questions:

a. [05 points] What information is typically stored at run-time and why?
b. [05 points] Where are ARs allocated and why?

a) The informations stored at run-time are parameters, local variables, return value, return address, ARP Link and Access Link.
The AR's need to track the execution environment of procedures/Functions.

a. The Activation Records tracks the execution environment of a procedure (or function) and thus must capture among other value, the return address and the frame pointer of the enclosed procedure (i.e., the procedure that immediately called the currently-executing procedure). In addition, and for procedures that allow for nested procedures (as is the case of PASCAL), the activation record also capture the Access Link, which allows an executing environment to access non-local variables, i.e., variables that are local to other active procedures (elsewhere on the stack) but are visible from a scoping standpoint. In addition, the activation records also capture the usual actual parameters of the procedure as local variables.

b) The AR's can be allocated in 3 ways: statically, Stack and Heap.
The more eficient way is statically, but it is needed that the procedure doesn't use recursion.
When the programming language allows recursion, the AR's are allocated in the Stack, as their activation on deactivation follows a LIFO order.

→ only allows a single activation of the given procedure.

↓ older programming languages

Stack, is still needed to save and restore the return address of each procedure.

(2)

Problem 2. Control-Flow Analysis and Register Allocation [40 points]

Consider the three-address code below for a procedure with input/output arguments `p0` and `p1` and using several temporary variables, named `t0` through `t4`.

```
01:       t1 = 0
02:       t0 = p1
03:       t2 = 0
04:       t1 = p2
05:       if (t1 > 0) goto L1
06:       t1 = t0
07:       t0 = 1
08:       t2 = 0
09: L1:   t1 = t0
10:       t4 = t0
11:       if (t2 > t4) goto L2
12:       t2 = t2 + 1
13:       goto L1
14: L2:   t0 = p1
15:       t3 = t0
16:       t1 = t2 + t3
17:       ret t1
```

Questions:
For this code determine the following:

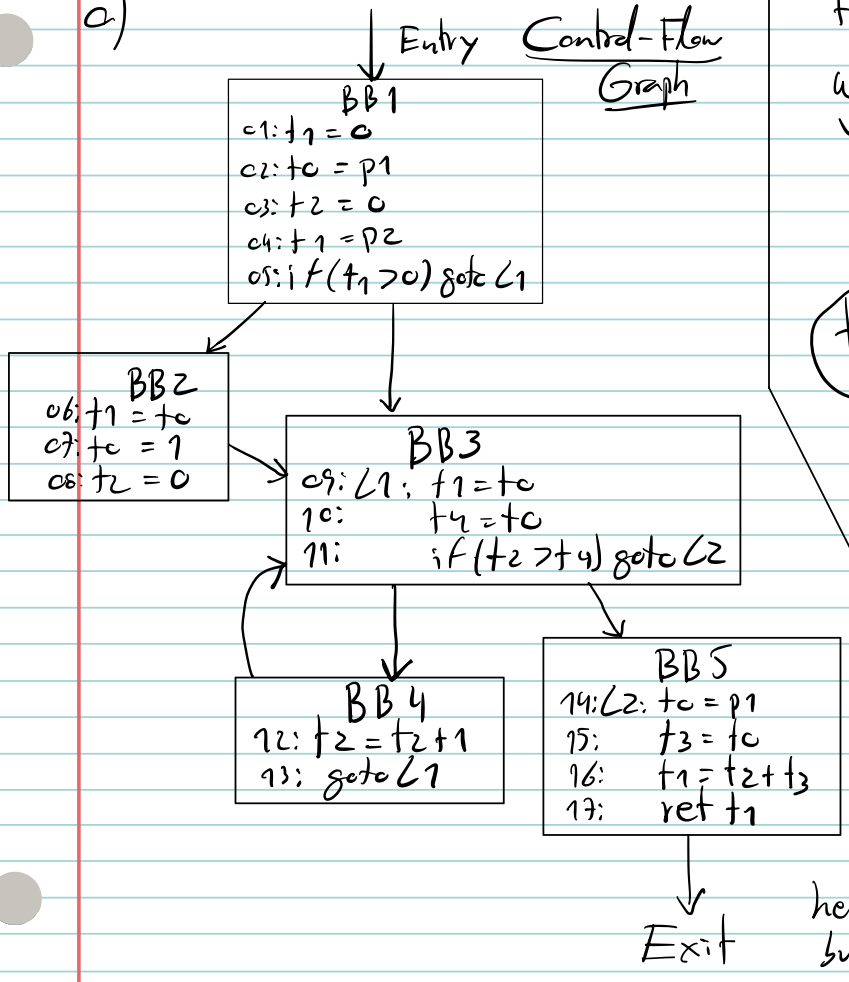
a. [05 points] Basic blocks and the corresponding control-flow graph (CFG) indicating for each basic block the corresponding line numbers of the code above.

b. [05 points] Dominator tree and the natural loops in this code (if any) along with the corresponding back edge(s).

c. [20 points] Determine the live ranges for the variables `t0`, `t1`, `t2`, `t3` and `t4`, the corresponding webs and interference graph, for the refined notion of interference discussed in class. Assume that you do not need registers for the parameters p0 and p1 and assume that on exit of the procedure. In this analysis you should ignore the parameter variables `p0` and `p1`. In this analysis you should ignore the last basic block (the one ending with the return instruction) `t2` is live but the remainder temporaries are dead on exit of the procedure.

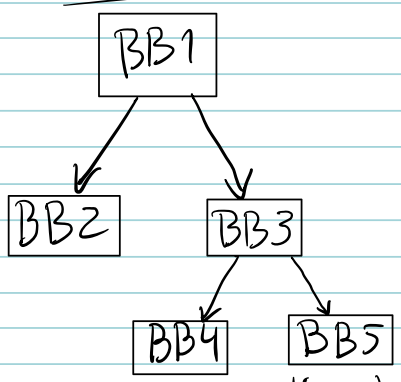
d. [10 points] Can you color the resulting interference graphs with 3 colors? If you cannot, suggest a source code transformation that will reduce the connectivity of the corresponding interference graph so that it becomes 3-colorable. If you can use 3 colors, just show a possible color assignment as you do not need to show the coloring that would result if you were to follow the graph coloring algorithm described in class.

a)



b)

Dominator Tree

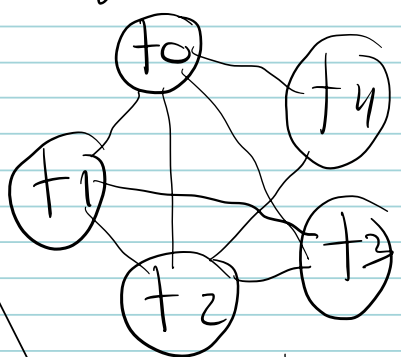


c) Live Ranges And webs

t0: {2,3,4,5,6,9,10} + {7,8,9,10}
t1: {1,7} + {4,5} + {6} + {9} + {16,17}
t2: {3} + {8,9,10,11,12,13,14,15,16}
t3: {15,16}
t4: {10,11} + {14,15}

t0 has {2,3,4,5,6,7,8,9,10} + {14,15}

Webs are the sets in each var group.



- t1 e t4 não interferem
- t3 e t4 não interferem

(excelto 1 par pintar com a mesma cor)

We need 5 - 1 colors to color the graph, so we need 4 registers.

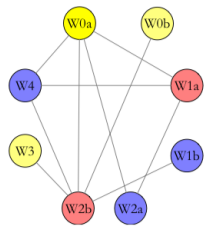
Simple notion of interference

d) Here we need to compare the webs and not the live ranges in full.
↳ refined notion discussed in class.

The professor doesn't count an assignment without any use a web.

The figure below depicts the interference graph for the temporaries `t0` through `t4` using the refined notion of interference as discussed in class. As it can be seen there are several cliques of size 3, so clearly it can be colored using 3 colors.

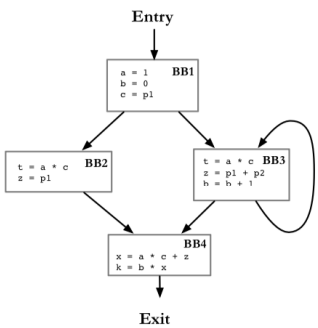
```
web0a: {02-14}
web0b: {15-16}
web1a: {04-05}
web1b: {16-17}
web2a: {03-05}
web2b: {08-16}
 web3: {15-16}
 web4: {10-11}
```



3

Problem 3. Code Optimization [20 points]

Consider the snippet 3-address code below using temporary variables `a`, `b`, `c`, `t`, `z`, `x` and `k` as well as the argument value `p1` and `p2` (used via the corresponding procedure's parameters).



Questions:

a. [10 points] Identify opportunities for the application of constant propagation, available expression and propagation, algebraic simplification and strength reduction.

b. [10 points] Identify opportunities for loop invariant code motion (LICM), as well as induction variables (i.e., variables that increase or decrease in value at every iteration of the loop). Consider also, redundant variable elimination (i.e., variables that can be removed by copy propagation).

In both cases, reasons about the legality of the code transformation with respect to the control-flow and the notion of dominance.

a) Constant Propagation

$a = 1 \Rightarrow$ como a é uma constante, podemos propagar o seu valor para as operações que usam a variável a.
É a única definição de a e está presente no bloco que domina todos os restantes.
O mesmo acontece com c que é = p1 em que tem essa definição.

Strength Reduction

A expressão a*c presente nos blocos bb2, bb3 e bb4 que será alterada para 1*c pela constant propagation, pode ser reduzida para "c".

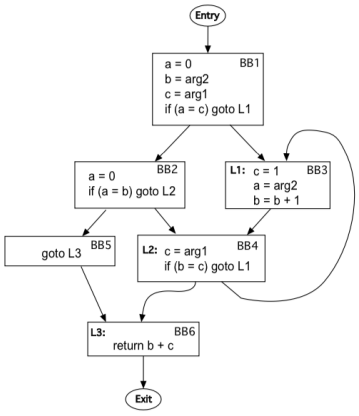
b) Loop Invariant Code Motion

- a*c é uma expressão que não varia com a iteração do ciclo, pois nem a nem c são alterados neste bloco. Acontece o mesmo com a definição da variável z que é igual a p1+p2 e nenhuma destes valores é alterado. Portanto, estes dois casos podem ser deslocados para um bloco pré-loop antes do bloco bb3.
- A única variável de indução presente no loop é a var b.

4

Problem 4: Iterative Data-Flow Analysis [30 points]

Your task is to formalize and apply the Copy-Propagation data flow analysis for a problem where we want to determine which pairs of variables "bound" by an assignment of the form a = b at program point p reach a given program point q if along any program path neither or a nor b are redefined. For instance, the assignment b = arg2 reaches BB2 and thus the predicate (a = b) can be rewritten as (a = arg2).



Questions:

Describe your approach to anticipation analysis by answering the following questions:

a. [05 points] What is the set of values in the lattice and the initial values?

b. [05 points] What is the direction of the problem, backwards or forward and why?

c. [05 points] What is the meet function for this data-flow problem, i.e., the GEN and KILL and the equations the iterative approach needs to solve?

d. [05 points] How do you construct the transfer function of a basic block based on the GEN and KILL at the instruction level or another algorithmic method?

e. [10 points] Describe transformations and optimizations that can leverage the information uncovered by this analysis in general and in particular to the example code provided. Provide a couple of examples to illustrate your points.

a) At each program point we will keep track of which pairs of variables are bound to the same value at run time.
We will have an unordered set of tuples of the form $\langle u, v \rangle$.
The initial values will be the empty set.

b) The direction of the problem is forward, because at each program point we want to know if the binding of two variables is steel value. As such as progress forward until there is one assignment to either of the variables that affect the tuples. As such the flow of information is naturally forward along the control flow.

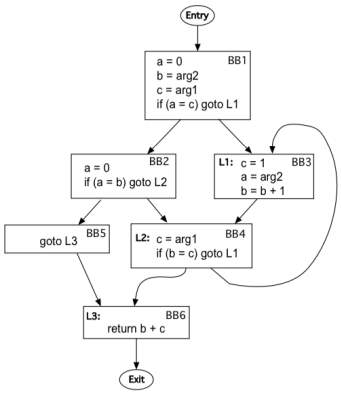
c) The meet operation is supported by the following equations:
$OUT = Gen \cup (IN - Kill)$
$Gen = \{\langle u, v \rangle \mid u = v \text{ is a statement}\}$
$Kill = \{\langle u, v \rangle \mid LHS \text{ of an assignment that is either } u \text{ or } v\}$

d) The Gen set can be accomplished by a single forward pass on the instructions of a basic block where we just keep track at each instruction what variables are assigned on the LHS of the assignment and are not subsequently killed by other assignments.
The kill set is simply the merge of all the variables that show up on the LHS of the assignments.

4

Problem 4: Iterative Data-Flow Analysis [30 points]

Your task is to formalize and apply the Copy-Propagation data flow analysis for a problem where we want to determine which pairs of variables "bound" by an assignment of the form a = b at program point p reach a given program point q if along any program path neither or a nor b are redefined. For instance, the assignment b = arg2 reaches BB2 and thus the predicate (a = b) can be rewritten as (a = arg2).



Questions:

Describe your approach to anticipation analysis by answering the following questions:

a. [05 points] What is the set of values in the lattice and the initial values?

b. [05 points] What is the direction of the problem, backwards or forward and why?

c. [05 points] What is the meet function for this data-flow problem, i.e., the GEN and KILL and the equations the iterative approach needs to solve?

d. [05 points] How do you construct the transfer function of a basic block based on the GEN and KILL at the instruction level or another algorithmic method?

e. [10 points] Describe transformations and optimizations that can leverage the information uncovered by this analysis in general and in particular to the example code provided. Provide a couple of examples to illustrate your points.

a) Queremos contabilizar que pares de variáveis e valores estão presentes em cada ponto do programa. Ou seja, teremos um unordered set de tuplos da forma {v, V} sendo v a variável e V o valor atribuído. Os valores iniciais serão o conjunto vazio.

b) A direção do problema é forward, porque temos que verificar se nos pontos seguintes à definição da variável, o value continua a ser igual ao inicial.

c) OUT = Gen U(IN - KILL)
Gen = {<v,V> | v = V is the statement}
Kill = {<v,V> | LHS variable of an assignment is either v or V}

d) Podemos usar uma single forward pass nas instruções de cada basic block e verificar que variáveis são (re)definidas. Todas as vars que estão presentes no LHS de um assignment serão introduzidas no KILL set.

The Gen can be accomplished by a single forward pass on the instructions of a basic block where we just keep track at each instructions what variables are assigned on the LHS of the assignment and are not subsequently killed by other assignments. The Kill set is simply the merge of all the variables that show up on the LHS of the assignments.

e) This fact that we know that two variables will hold the same value will allow use to replace one of them, in the hope that the corresponding variable (and consequently the instructions that make the assignments, become dead. In addition, and when combined with constant propagation this can lead to even more dead-code elimination. In this particular example the assignment to the c variable is basic block BB1 and BB3 are dead. The only association that reaches the use of c in BB4 for example is the assignment c = arg1. As such the predicate in this BB4 can be rewritten as if (b = arg1) goto L1. Notice also that the two associations that reach BB6 for c are the same {c, arg1} and as a result the return statement can be rewritten as return b+arg1. After this the references to c can simply be removed from the code