Fourth Weekly IPS Assignment

This is the text of the fourth weekly assignment for the DIKU course Implementation of Programming Languages (IPS), 2023.

Hand in your solution in the form of a short report in text or PDF format.

Task 1

This task refers to compiler optimizations (see Lecture slides 6 - Optimizations in Fasto), and specifically dead-binding elimination (DBE).

Consider the following Fasto expression, with labeled subexpressions:

$_{(11)}$let x = $_{(1)}$u in $_{(10)}$let y = $_{(2)}$x+x in $_{(9)}$let y = $_{(5)}$(let x = $_{(3)}$foo(x) in $_{(4)}$7) in $_{(8)}$let t = $_{(6)}$x+u in $_{(7)}$y*x

Each subexpression extends from its label and as far to the right as syntactically possible. For example, subexpression (2) is `x+x`, and (5) is `let x = foo(x) in 7`, while (11) is the whole expression. `foo` is some user-defined function, about which you should assume nothing (in particular, it *might* contain I/O).

Show, in tabular form, the computation of DBE for this expression, by giving for each subexpression (n), the following information in corresponding columns:

a. IO[n]: whether the subexpression might perform I/O operations;

b. Elim[n]: for let-expressions only, blank otherwise: whether the let-binding may be safely eliminated;

c. UV[n]: the set of variables used in the subexpression *after* DBE optimization; and

d. OE[n] the DBE-optimized subexpression itself.

You should fill out the row for each subexpression based only on the information already computed (in earlier rows) for its immediate constituents, without re-traversing those entire subexpressions themselves.

In particular, for each let-subexpression "$_{(i)}$let ID = $_{(j)}$⋯ in $_{(k)}$⋯" (where both j and k will be less than i, such as for i = 9, where j = 5, k = 8, and ID = y), you should have:

- If IO[j] = `no` and ID ∉ UV[k], then:
  IO[i] = IO[k], Elim[i] = `yes`, UV[i] = UV[k], and OE[i] = OE[k].
- Otherwise (i.e., if IO[j] = `yes`, ID ∈ UV[k], or both), then:
  IO[i] = (IO[j] or IO[k]), Elim[i] = `no`, UV[i] = UV[j] ∪ (UV[k] \ {ID}) and OE[i] = (let ID = OE[j] in OE[k]).
  You should show explicitly how the set UV[i] is calculated, in addition to giving the result. (The notation A \ B stands for set difference, i.e., all elements of set A except those which are also in set B.)

For the leaf subexpression (i.e those that are *not* let-expressions), you can just fill out the row directly by looking at the text of the subexpression, and in particular OE[i] will be just the subexpression itself. The final result of the DBE optimization will then be OE[n], where n is the number of the topmost subexpression (such as n = 11 in the example above).

As an illustration of the format, for the following, simpler expression:

$_{(5)}$let x = $_{(1)}$read(int) in $_{(4)}$let y = $_{(2)}$x+y in $_{(3)}$x+1

the answer should be the following table:

```
n     IO      Elim    UV                  OE
-------------------------------------------------------------------
1     yes     -       {}                  read(int)
2     no      -       {x,y}               x+y
3     no      -       {x}                 x+1
4     no      yes     {x}                 x+1
5     yes     no      {}∪({x}\{x}) = {}   let x=read(int) in x+1
```

Task 2

This task refers to liveness analysis and register allocation (see corresponding lecture slides 7 - Register Allocation).

Given the following program:

```
   gcd(a,b) {
1:  LABEL start
2:  IF a < b THEN next ELSE swap
3:  LABEL swap
4:  t := a
5:  a := b
6:  b := t
7:  LABEL next
8:  z := 0
9:  b := b mod a
10: IF b = z THEN end ELSE start
11: LABEL end
12: RETURN a
   }
```

- a.) Show `succ`, `gen` and `kill` sets for every instruction in the program.
- b.) Compute `in` and `out` sets for every instruction, show the fix-point iteration.
- c.) Show the interference table (which documents the interference relation for each assignment instruction).
- d.) Draw the interference graph for `a`, `b`, `t`, and `z`
- e.) Color the interference graph with `3` colors. Show the stack.
- f.) Color the interference graph with `2` colors. Select variables to spill, perform the spilling transformation and show the resulted program after spilling. You are not required to perform the whole analysis again, i.e., don't do the liveness analysis and graph coloring on the "spilled" code.

Task 3

This task refers to regular expressions and NFAs (see corresponding lecture slides 8 - Lexical Analysis, and section 1.1–1.3 in the book).

Consider the following regular expression (over the alphabet {a,b}):

```
(a*b|a)*(b|ε)
```

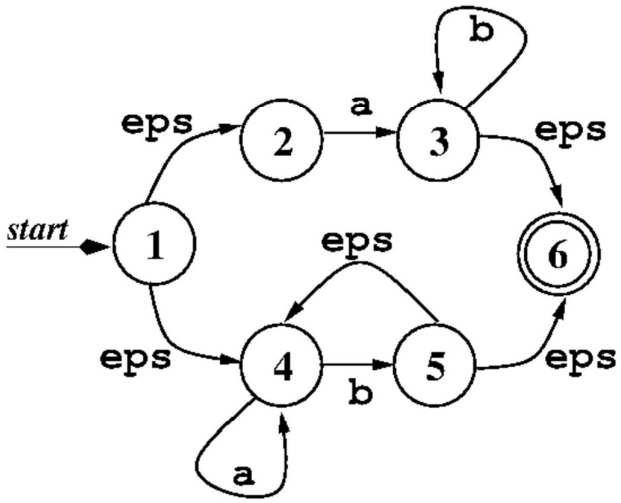
Using the systematic construction from Fig. 1.4 in the book, build the equivalent nondeterministic finite automaton (NFA). (Remember to add the final, accepting state, and to clearly indicate the start state; and make sure that each transition is labeled by either ε or a character from the alphabet.) As usual, follow the formal construction as closely as you can; do not_ attempt to hand-optimize the NFA.

(Unless you are already proficient with a suitable graphical editor or graph-drawing program, do not waste time on producing a good-looking NFA figure. As long as the result is easily readable, a hand-drawn image, scanned or photographed (in reasonable lighting!) will do fine.)

Task 4

This task refers to NFAs and DFAs (see corresponding lecture slides 8 - Lexical Analysis, and section 1.4–1.5 in the book).

Convert the nondeterministic finite automaton (NFA) below into an equivalent deterministic finite automaton (DFA) using the subset-construction algorithm:



(In the Figure, eps stands for ε, and the alphabet Σ = {a,b})

- Systematically derive the move function for all (about 10) pairs of DFA state and input character, showing how you construct each new state and transition.
- Then *draw* the resulting DFA. Again, remember to indicate starting and accepting states, and make sure that each transition is labeled by a character from the alphabet.