Name:

Midterm

COMP 150-AVS
Program Analysis, Verification, and Synthesis
Fall 2018

October 24, 2018

Instructions

This exam contains 10 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		20
2		10
3		30
4		15
5		20
6		5
Total		100

Question 1. Short Answer (20 points).

a. (5 points) List one advantage and one disadvantage of *polymorphic variants* (i.e., a term constructed with a backtick like `` `Reg 42 ``) compared to regular OCaml *datatypes*.

> **Answer:** One advantage is that different polymorphic variant constructors can be mixed together arbitrarily (as long as they have the same contents types), whereas constructors from different datatypes cannot be mixed. On the other hand, the OCaml type system necessarily cannot detect as many errors involving polymorphic variants, e.g., it cannot easily find non-exhaustive match cases.

b. (5 points) Briefly describe what an *intermediate representation* is.

> **Answer:** An intermediate representation is a program representation that is at a level of abstraction in between the program source code/abstract syntax tree and the compiler output (typically bytecode or machine code).

c. (5 points) Briefly explain the relationship between a *partial order* and a *lattice*.

Answer: A lattice is a partial order such that for every pair of elements x and y, the meet $x \sqcap y$ and join $x \sqcup y$ exist.

d. (5 points) For α (an abstraction function) and γ (a concretization function) to form a Galois insertion, both must be monotonic, and two other properties must hold. What are the other two properties?

Answer:

$S \subseteq \gamma(\alpha(S))$ for any concrete set S and

$\alpha(\gamma(A)) = A$ for any abstract element A.

Question 2. Boolean Expressions and Bitvectors (10 points). Recall the types for boolean expressions, assignments, and bitvectors from Project 1:

```
type bexpr =                          | ENot of bexpr
    EFalse                            | EForall of string * bexpr
  | ETrue                             | EExists of string * bexpr
  | EVar of string
  | EAnd of bexpr * bexpr          type asst = (string * bool) list
  | EOr of bexpr * bexpr           type bvec = bexpr list (* low order bit at head of list *)
```

a. (5 points) Write a function bexpr_if (b1:bexpr) (b2:bexpr) (b3:bexpr):bexpr that returns a boolean expression that evaluates to the value of b2 if b1 evaluates to *true* and to b3 otherwise. For example, the following expressions both evaluate to *true*.

eval ["x", EFalse] (bexpr_if ETrue ETrue EFalse)
eval ["x", EFalse; "y", ETrue] (bexpr_if (EVar "x") EFalse (EVar "y"))

Answer:

```
let bexpr_if b1 b2 b3 = EOr (EAnd(b1, b2), EAnd(ENot(b1), b2))
```

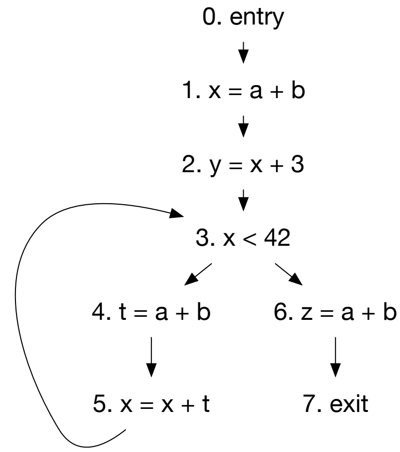
b. (5 points) Write a function bvec_if (b1:bexpr) (b2:bvec) (b3:bvec):bvec that returns a bitvector where the bit at position i evaluates to the value of the ith bit of b2 if b1 evaluates to true, and to the ith bit of b3 otherwise. You can call bexpr_if as a subroutine, and you can assume b2 and b3 have the same length.

Answer:

```
List.map2 (bexpr_if b1) b2 b3
```

Question 3. Data flow analysis (30 points).

a. (25 points) In the following table, show each iteration of *very busy expressions* for the control-flow graph on the right. For each iteration, list the statement taken from the worklist in that step, the value of *in* computed for that statement, and the new worklist at the end of the iteration. You may or may not need all the iterations; you may also add more iterations if needed. Do not add the exit node to the worklist. Assume that x<42 is *not* a possible very busy expression.



Use $\emptyset$ for the set of no expressions, and $\top$ for the set of all expressions. What is $\top$?

$\top$ = a+b, x+3, x+t

What are the initial *in*'s for each statement?

Stmt	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Initial in	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\emptyset$

Iteration	**0**	**1**	**2**	**3**	**4**
Stmt taken from worklist	N/A	6	3	5	4
in of taken stmt	N/A	a+b	a+b	a+b, x+t	a+b
New worklist	0,1,2,3,4,5,6	0,1,2,3,4,5	0,1,2,4,5	0,1,2,4	0,1,2,3

Iteration	**5**	**6**	**7**	**8**	**9**
Stmt taken from worklist	3	2	1	0	
in of taken stmt	a+b	x+3, a+b	a+b	a+b	
New worklist	0,1,2	0,1	0	$\emptyset$	

Iteration	**10**	**11**	**12**	**13**	**14**
Stmt taken from worklist					
in of taken stmt					
New worklist					

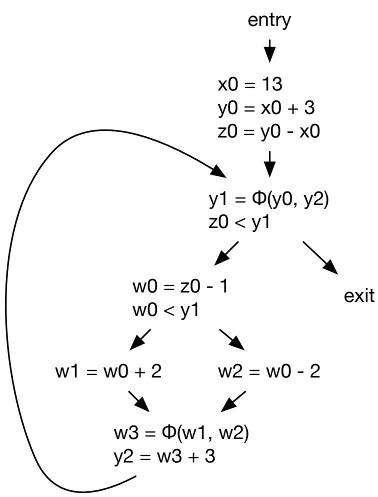
b. (5 points) Write the gen and kill sets for *available expressions* for the statements shown below. Assume the set of expressions is {a+b, a+1, x+y, x+1}. Write ∅ for an empty gen or kill set.

stmt	gen	kill
x := a + b	a+b	x+y, x+1
x := y	∅	x+y, x+1
a := a + 1	∅	a+b, a+1

Question 4. Static Single Assignment Form (15 points). Draw a static single assignment form control-flow graph for the following program. **Put all statements in the largest possible basic blocks.** Be sure to include the entry and exit nodes. Don't worry about whether Φ nodes go in their own basic blocks or get added to existing ones—do whichever is convenient.

```
x = 13;
y = x + 3;
z = y − x;
while (z < y) {
  w = z − 1;
  if (w < y) {
    w = w + 2
  } else {
    w = w − 2
  }
  y = w + 3
}
```

Answer:



Question 5. Abstract Interpretation (20 points). In this problem, you will develop part of an abstract interpreter for RubeVM. The abstract domain will be the domain of *intervals*, where an interval $[x, y]$ represents all integers n such that $x \leq n \leq y$. An interval is empty if $x > y$. We will represent an interval as a type inter = int × int, where the left component is the lower bound and the right component is the upper bound.

Below is part of the RubeVM instruction set. We've modified the type of the register file to map registers to abstract values.

```
type reg = [`Reg of int]  type value = [`Int of int]
type instr =
  | I_const of reg * value (* dst, src *)
  | I_add of reg * reg * reg (* dst, src1, src2 *)
  | I_sub of reg * reg * reg (* dst, src1, src2 *)

type inter = int * int
type regs = (int, inter) hash
```

a. (5 points) Implement an abstraction function alpha : int → inter that returns the interval representing a single integer, and a concretization function gamma : inter → int list that returns a list of the integers (in any order) corresponding to an interval.

Answer:

```
let alpha n = (n, n)
let rec gamma (x, y) = if x>y then [] else x::(gamma (x+1, y))
```

b. (10 points) Write a function arun_inst : regs → instr → unit that abstractly executes the three instructions given in the figure above. *Hint: Subtraction can be implemented as negation followed by addition.* You will probably want to use Hashtbl.add and Hashtbl.find, with signatures:

```
val add : ('a, 'b) t -> 'a -> 'b -> unit        val find : ('a, 'b) t -> 'a -> 'b

open Hashtbl
```

Answer:

```
let empty (x, y) = y < x

let arun_inst rs = function
| I_const (`Reg r, `Int n) -> add(r, alpha n)
| I_add (`Reg r1, `Reg r2, `Reg r3) ->
  let (x1, x2) = find rs r2 in
  let (y1, y2) = find rs r3 in
  if (empty (x1, x2)) || (empty (y1, y2)) then (1, 0)
  else add rs r1 (x1+y1, x2+y2)
| I_sub (`Reg r1, `Reg r2, `Reg r3) ->
  let (x1, x2) = find rs r2 in
  let (y1, y2) = find rs r3 in
  if (empty (x1, x2)) || (empty (y1, y2)) then (1, 0)
  add rs r1 (x1-y2, x2-y1)
```

c. (5 points) As a step toward abstractly executing branches, implement a function join : inter → inter → inter that returns the smallest interval containing both of its arguments.

Answer:

```
let join (x1, x2) (y1, y2) =
  if x2 < x1 then (y1, y2)
  else if y2 < y1 then (x1, x2)
  else (min x1 y1, max x2 y2)
```

Question 6. Subtyping (5 points). In class we discussed a subtyping system for the following language, which is the simply typed lambda calculus extended with tainted (n^t with type int^t) and untainted (n^u with type int^u) integers, where $int^u \leq int^t$:

$$\begin{array}{rcl} e & ::= & n^t \mid n^u \mid x \mid \lambda x{:}t.e \mid e\ e \\ t & ::= & int^t \mid int^u \mid t \rightarrow t \end{array}$$

Suppose we extend the language with *pair types* $t \times t$ with the following subtyping rule:

$$\frac{t_1 \leq t_2 \qquad t_1' \leq t_2'}{(t_1 \times t_1') \leq (t_2 \times t_2')}$$

Give a list of all types t such that t is a subtype of $int^t \times (int^t \rightarrow int^u)$. *Hint: It might be easiest to write down all potential subtypes and then cross out the ones that aren't actually subtypes.*

Answer:

$$\begin{array}{c} int^u \times (int^t \rightarrow int^u) \\ int^t \times (int^t \rightarrow int^u) \end{array}$$