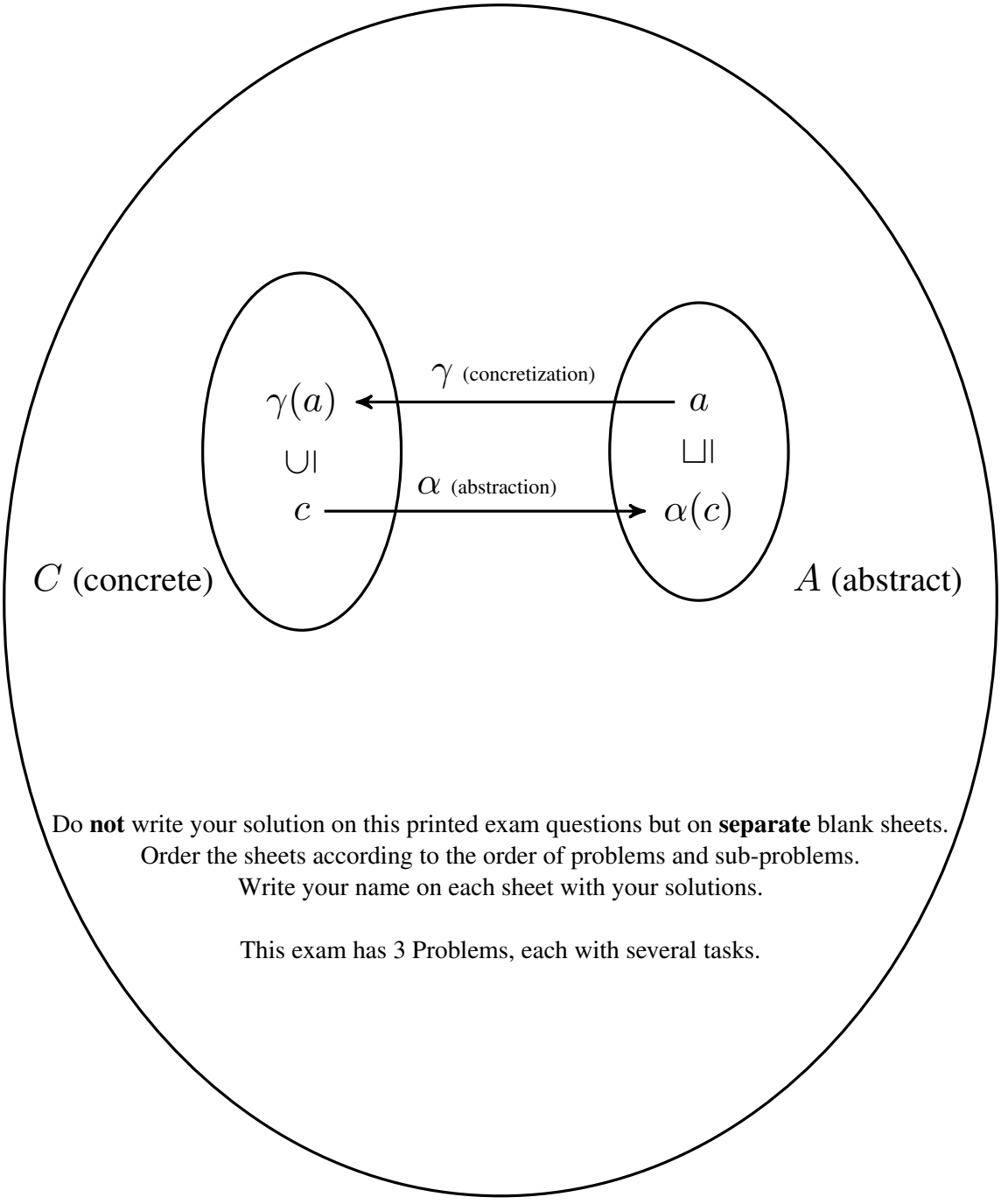
Quiz

Synthesis, Analysis, and Verification 2017

Monday, 10 April 2017



Do **not** write your solution on this printed exam questions but on **separate** blank sheets.
Order the sheets according to the order of problems and sub-problems.
Write your name on each sheet with your solutions.

This exam has 3 Problems, each with several tasks.

Problem 1: Quantifier Elimination ([20 points])

Consider the formula $F(x)$ given by

$$F(x) = \bigwedge_{i=1} a_i < x \wedge \bigwedge_{j=1} x < b_j \wedge \bigwedge_{i=1} K_i | (x + t_i).$$

Recall that the terms a_i, b_j, t_i may in general contain other variables than x.

Task 1) (*5 points*)
Assume all a_i, b_j, t_i are integer constants. Give an algorithm that, given any formula of the form above, returns:

- a value for x, if such value exists, and
- "UNSAT" if no such value exists

Task 2) (*15 points*)
Give a recursive algorithm that, given a formula in the above form returns

- one map from variables to integers for which formula evaluates to true, if such a map exists, and
- "UNSAT" if no such map exists.

Problem 2: Galois Connections ([20 points])

Let $(C, \subseteq)$ and $(A, \sqsubseteq)$ be two lattices. Let $\bot$ be the least element of $(A, \sqsubseteq)$. Let $\cup$ be the binary least upper bound operator in $(C, \subseteq)$ and $\sqcup$ the binary least upper bound operator in $(A, \sqsubseteq)$. Let $\alpha : C \to A$ and $\gamma : A \to C$ be two monotonic functions such that (α, γ) is a Galois **connection**. Remember that this is defined as:

$$\forall a \in A.\, \forall c \in C.\quad c \subseteq \gamma(a) \iff \alpha(c) \sqsubseteq a$$

We assume the above conditions throughout Problem 2.
We will say that (α, γ) is a **Galois insertion** if and only if it is a Galois connection and additionally

$$\forall a.\, \alpha(\gamma(a)) = a \tag{1}$$

We have already seen several alternative definitions for Galois insertion in the lecture (you can use those in your solution if you think it helps). We start this problem by showing two more alternative conditions.
Task 1) (*4 points*)
Show that (**??**) **if and only if**

$$\forall a_1, a_2.\ a_1 \sqsubseteq a_2 \iff \gamma(a_1) \subseteq \gamma(a_2) \tag{2}$$

(This is interesting because it says that the ordering in A is uniquely given by γ and the ordering in C.)

Task 2) (*6 points*)
Show that (**??**) **if and only if**

$$\forall a_1, a_2.\ a_1 \sqcup a_2 = \alpha(\gamma(a_1) \cup \gamma(a_2)) \tag{3}$$

(This is interesting because it says that the least upper bound in A is uniquely given by γ and the least upper bound in C.)

Task 3) (*10 points*) Let us now assume that (α, γ) is a Galois **insertion**. We explore approximating the fixed-points of monotonic functions. Let $F : C \to C$ be a monotonic function (you can think of this function as defining the system recursively, but there is no need to assume continuity of F in this problem).
Define its approximate version $F^{\#}$ by

$$F^{\#}(a) = \alpha(F(\gamma(a))) \tag{4}$$

Again, this choice is uniquely given by F and (α, γ). We will be considering iterating function $F^{\#}$ to compute a fixed-point.
Let us assume that there exists a non-negative integer n (so, n is finite!) such that

$$(F^{\#})^{n+1}(\bot) = (F^{\#})^{n}(\bot)$$

Define $a_* \in A$ by $a_* = (F^{\#})^{n}(\bot)$. (We can think of a_*, for example, as an invariant computed to approximate reachable states of the system given by F.)

Task 3.1) (*4 points*) Show that $\alpha(F^{n}(\gamma(\bot))) \sqsubseteq a_*$.

Task 3.2) (*3 points*) Show that

$$F(\gamma(a_*)) \subseteq \gamma(a_*) \tag{5}$$

Task 3.3) (*3 points*)
Show that if c_* is the least such that $F(c_*) \subseteq c_*$, then $\alpha(c_*) \sqsubseteq a_*$.

Problem 3: Inductive Proofs ([20 points])

It is possible to guide the Stainless verification system through proofs by manually providing the induction schema to the system. This schema can be naturally encoded into a recursive boolean function. Consider for example the following property:

```
def lemma[T,U,V](xs: List[T], f: T ⇒ U, g: U ⇒ V): Boolean = {
  xs.map((x: T) ⇒ g(f(x))) == xs.map(f).map(g)
}.holds
```

The unfolding procedure underlying Stainless verification will try to verify this property by progressively unfolding the definition of map on larger and larger lists. The expression xs.map((x: T) ⇒ g(f(x))) will thus lead to the (infinite) sequence of clauses:

```
(xs == Nil()) ⟹
  (xs.map((x: T) ⇒ g(f(x))) == Nil())
(xs == Cons(y, ys)) ⟹
  (xs.map((x: T) ⇒ g(f(x))) == Cons(g(f(y)), ys.map((x: T) ⇒ g(f(x)))))
(xs == Cons(y, Cons(z, zs))) ⟹
  (xs.map((x: T) ⇒ g(f(x))) == Cons(g(f(y)), Cons(g(f(z)), zs.map((x: T) ⇒ g(f(x))))))
...
```

and the expression xs.map(f).map(g) will lead to

```
(xs == Nil()) ⟹
  (xs.map(f).map(g) == Nil())
(xs == Cons(y, ys)) ⟹
  (xs.map(f).map(g) == Cons(g(f(y)), ys.map(f).map(g)))
(xs == Cons(y, Cons(z, zs))) ⟹
  (xs.map(f).map(g) == Cons(g(f(y)), Cons(g(f(z)), ys.map(f).map(g))))
...
```

It is obvious here that in an inductive proof on xs, we could simply assume the inductive hypothesis ys.map((x: T) ⇒ g(f(x))) == ys.map(f).map(g) and we would have a proof. Interestingly, it is in fact possible to help Stainless along here by making lemma recursive:

```
def lemma_induct[T,U,V](xs: List[T], f: T ⇒ U, g: U ⇒ V): Boolean = {
  val prop = xs.map((x: T) ⇒ g(f(x))) == xs.map(f).map(g)
  val induct = xs match {
    case Cons(y, ys) ⇒ lemma_induct(ys, f, g)
    case Nil() ⇒ true
  }
  prop && induct
}.holds
```

It is clear that lemma_induct holds iff lemma holds (the properties are equivalent). As the unfolding procedure assumes the postcondition of recursive calls, when verifying this second definition, Stainless will be able to prove that lemma_induct holds as it can assume that lemma_induct(ys, f, g) is true (namely, that ys.map((x: T) ⇒ g(f(x))) == ys.map(f).map(g)).

Task 1) (*3 points*)
Provide a function definition because such that the above definition of lemma_induct can be stated as follows:

```
def lemma_induct[T,U,V](xs: List[T], f: T ⇒ U, g: U ⇒ V): Boolean = {
  (xs.map((x: T) ⇒ g(f(x))) == xs.map(f).map(g)) because (xs match {
    case Cons(y, ys) ⇒ lemma_induct(ys, f, g)
    case Nil() ⇒ true
  })
}.holds
```

Task 2) (*9 points*)
Consider the following definitions:

```
def size[T](list: List[T]): BigInt = list match {
  case Cons(x, xs) ⇒ 1 + size(xs)
  case Nil() ⇒ 0
}

def append[T](l1: List[T], l2: List[T]): List[T] = l1 match {
  case Cons(x, xs) ⇒ Cons(x, append(xs, l2))
  case Nil() ⇒ l2
}

def get[T](list: List[T], i: BigInt): T = {
  require(i >= 0 && i < size(list))
  list match {
    case Cons(x, xs) if i == 0 ⇒ x
    case Cons(x, xs) ⇒ get(xs, i − 1)
    // case Nil() ⇒ impossible case!
  }
}
```

Which of the following functions will Stainless manage to verify automatically? For the properties on which Stainless would fail, provide an equivalent version of the property that will verify.

Task 2.1) (*3 points*)

```
def rightUnitAppend[T](list: List[T]): Boolean = {
  append(list, Nil()) == list
}.holds
```

Task 2.2) (*3 points*)

```
def leftUnitAppend[T](list: List[T]): Boolean = {
  append(Nil(), list) == list
}.holds
```

Task 2.3) (*3 points*)

```
def appendGet[T](l1: List[T], l2: List[T], i: BigInt): Boolean = {
  require(0 <= i && i < size(l1) + size(l2))
  get(append(l1, l2), i) == (if (i < size(l1)) get(l1, i) else get(l2, i − size(l1)))
}.holds
```

Task 3) (*8 points*)
Consider the following definitions:

```
def snoc[T](xs: List[T], x: T): List[T] = xs match {
  case Cons(y, ys) ⇒ Cons(y, snoc(ys, x))
  case Nil() ⇒ Cons(x, Nil())
}

def reverse[T](list: List[T]): List[T] = list match {
  case Cons(x, xs) ⇒ snoc(reverse(xs), x)
  case Nil() ⇒ Nil()
}
```

Provide an equivalent statement of the following property that Stainless will manage to verify:

```
def lemma[T](list: List[T]): Boolean = {
  reverse(reverse(list)) == list
}.holds
```

Note that you will need to introduce an auxiliary lemma in order to prove this property. Make sure both lemmas verify! In order to invoke a lemma in Stainless, it suffices to `&&` it with the poperty you are trying to prove (*for example:* `lemma(x, y, z) && property`).