Final Exam

Semantics of Programming Languages

15. 2. 2023

First name:	
Last name:	
Student-Id (Matrikelnummer):	
Signature:	

1. The exam is to be solved in Isabelle on your own Laptop.
2. Submit your solutions to do.proof.in.tum.de, in the **Semantics Exam 2022/2023** contest. You may submit any number of times, only your last submission will be counted.
3. You may use all lecture material (including exercises and your homework) to solve the exam.
4. Using the internet is not allowed for any other reason than submitting your solution to the submission system.
5. You have 120 minutes to solve the exam and submit your solutions.
6. Please put your student ID and ID-card or driver's license on the table until we have checked it.
7. You may not leave the room in the last 15 minutes of the exam — you may disturb other students who need this time.

Proof Guidelines: We expect valid Isabelle proofs. `sledgehammer` may be used. The use of `sorry` may lead to the deduction of points but is preferable to spending a lot of time on individual proof steps. Unfinished proofs should be well written and easy to understand!

1 Induction

Consider the following recursive function *count* and inductive predicate *ok*:

```
fun count :: "bool list ⇒ int" where
"count [] = 0" |
"count (True # bs) = count bs + 1" |
"count (False # bs) = count bs − 1"

inductive ok :: "bool list ⇒ bool" where
"ok []" |
"⟦ ok as;  ok bs ⟧ ⟹ ok (True # as @ False # bs)"
```

Prove the following lemma:

```
lemma ok_count_Zero: "ok bs ⟹ count bs = 0"
```

2 Bounds on Small-Step Execution

Regard commands *without WHILE*. Define a function

fun *bound* :: *"com ⇒ nat"*

such that

$(c, s) \to (c', s') \Longrightarrow \mathit{bound}\ c' < \mathit{bound}\ c$

Prove this property!

Note: Intuitively, *bound* computes an upper bound on the number of small steps required to completely execute the command.

3 Hoare Logic

Consider an extension of IMP with a *CHANGE x ST b* command, which finds a value for the variable x such that b holds, i.e., its big-step semantics is:
$bval\ b\ (s(x := n)) \Longrightarrow (CHANGE\ x\ ST\ b,\ s) \Rightarrow s(x := n)$.
Define a backwards rule for our Hoare calculus. Define it in the following abbreviation:

abbreviation (*input*) *change_rule* :: *"vname ⇒ bexp ⇒ assn ⇒ assn"*

such that the new rule is:

$"\vdash \{change\_rule\ x\ b\ P\}\ CHANGE\ x\ ST\ b\ \{P\}"\ |$

Show your new rule sound and complete:

lemma *sound*: $"\models \{change\_rule\ x\ b\ P\}\ CHANGE\ x\ ST\ b\ \{P\}"$
lemma *complete*: $"\models \{P\}\ (CHANGE\ x\ ST\ b)\ \{Q\} \Longrightarrow \vdash \{P\}\ CHANGE\ x\ ST\ b\ \{Q\}"$

Define an IMP command *MINUS* for an unary minus operation without using a loop, and show it correct:

lemma *MINUS_correct*:
$"\vdash \{\lambda s.\ s{=}s_0\}\ MINUS\ \{\lambda s.\ s\ ''y'' = -\ s\ ''x'' \wedge (\forall v \neq ''y''.\ s\ v = s_0\ v)\}"$

The *CHANGE x ST b* command is hard to compute on a real machine. Assume we have a *SWAP x y* command instead, which swaps the values of the variables x and y. The following program *SIM_CHG* simulates the change command:

```
x := 0;
WHILE not b DO (
  SWAP x y;
  IF b THEN SKIP
  ELSE (
    SWAP x y;
    x := x + 1;
    y := y - 1
  )
);
y := x + y
```

Find an invariant to prove the following specification for partial correctness (i.e., the arising verification conditions should be easily provable). *No proof required!*

lemma
  assumes $"y \notin vars\ b"$
  shows $"\vdash \{\lambda s.\ s{=}s_0 \wedge s\ x < s\ y\}\ SIM\_CHG\ x\ b\ y\ \{\lambda s.\ bval\ b\ s \wedge (\forall v.\ v \neq x \longrightarrow s\ v = s_0\ v)\}"$

4 Abstract Proof

Let f :: *nat* $\Rightarrow$ A such that f is increasing ($\forall\, n.\ n \leq f\ n$) and $A = \{m.\ \exists\, n.\ m = f\ n\}$ has a maximal element. Then f has a fixed point. Prove the following formalization of this fact:

lemma *fixp*:
  fixes f :: *"nat ⇒ nat"*
  assumes *incr*: "$\forall\, n.\ n \leq f\ n$"
  assumes A: "$A = \{m.\ \exists\, n.\ m = f\ n\}$"
  assumes *max*: "$\exists\, m \in A.\ \forall\, n \in A.\ n \leq m$"
  shows "$\exists\, k \in A.\ f\ k = k$"

The proof must be structured into small steps. That is, every proof step (*have* etc) must use at most one of the four assumptions and at most two propositions you proved in previous steps. No *apply*.

5 Inverse Analysis

An interesting analysis for abstract interpretation is whether a program could have an arithmetic overflow. For this analysis, we consider the abstract domain of a bounded number of bits:

datatype *bits* = *B nat* | *Any*

A value of *B n* represents all numbers whose binary representation needs at most n bits (only positive numbers, assuming a version of IMP with *nat*s instead of *int*). The number *0* needs at least one bit to be represented. As usual, *Any* represents any number.

Define $\leq$ and $\sqcup$ on this domain, for any n:

fun *less_eq_bits* :: *"bits ⇒ bits ⇒ bool"* (**infix** *"≤" 50*)
fun *sup_bits* :: *"bits ⇒ bits ⇒ bits"* (**infix** *"⊔" 50*)

Is this a complete lattice?

Next, define an abstract plus operation for a given bound:

fun *plus'_bits* :: *"nat ⇒ bits ⇒ bits ⇒ bits"*

Run the abstract interpreter on the following program, with $n{=}1$, as seen in the exercises: Document its state at every step for the variable x, until a fixed point is reached. Write down a list of steps for each annotation, using _____________ when a value did not change w.r.t. the previous iteration.

```
x := 1;
WHILE b DO (
  x := x+1;
)
```

Define the inverse analysis. Always return the smallest abstract values possible. Start as follows:

fun *inv_plus'_bits* :: *"bits ⇒ bits ⇒ bits ⇒ (bits * bits)"* **where**
  "inv_plus'_bits _ (B 0) _ = (B 0, B 0)" |
  "inv_plus'_bits _ _ (B 0) = (B 0, B 0)"

fun *inv_less'_bits* :: *"bool ⇒ bits ⇒ bits ⇒ (bits * bits)"* **where**
  "inv_less'_bits _ (B 0) _ = (B 0, B 0)" |
  "inv_less'_bits _ _ (B 0) = (B 0, B 0)"