Final Exam

Concrete Semantics with Isabelle/HOL

10 August 2015

First name:	______________
Last name:	______________
Student ID (Matrikelnummer):	______________
Signature:	______________

1. You may only use a pen/pencil, eraser, and two A4 sheets of notes to solve the exam. Switch off your mobile phones.
2. Please write on the sheets of this exam. At the end of the exam, there are two extra sheets. If you need more sheets, ask the supervisors during the exam.
3. You have 120 minutes to solve the exam.
4. Please put your student ID and ID card or driver's license on the table until we have checked it.
5. Please do not leave the room in the last 20 minutes of the exam—you may disturb other students who need this time.
6. There are in total 60 points, distributed over 6 questions.

Proof Guidelines: We expect detailed, rigorous, mathematical proofs—but we do not ask you to write Isabelle proof scripts. You are welcome to use standard mathematical notation; you do not need to follow Isabelle syntax. Proof steps should be explained in ordinary language like a typical mathematical proof.

Major proof steps, especially inductions, need to be stated explicitly. For each case of a proof by induction, you must give the **inductive hypotheses** assumed (if any) and the **goal** to be proved.

Minor proof steps (corresponding to *by simp*, *by blast*, etc.) need not be justified if you think they are obvious, but you should say which facts they follow from. You should be explicit whenever you use a function definition or an introduction rule for an inductive predicate—especially for functions and predicates that are specific to an exam question. (You need not reference individual lemmas for standard concepts like integer arithmetic, however, and in any case we do not ask you to recall lemma names from any Isabelle theories.)

1 Induction on Lists (7 points)

The goal of this question is to prove a simple property on lists, which is included as part of `List.thy` in the Isabelle distribution.

The *map* function applies a function (given as first argument) to each element of a list (given as second argument) and returns the resulting list. Its recursive specification is as follows:

fun *map* :: $('a \Rightarrow 'b) \Rightarrow 'a\ list \Rightarrow 'b\ list$ **where**
  map f [] = []
| *map f* (*x* # *xs*) = *f x* # *map f xs*

Recall that the notations [] and # are syntactic sugar for the constructors *Nil* :: *'a list* and *Cons* :: $'a \Rightarrow 'a\ list \Rightarrow 'a\ list$.

Question Prove the following property by induction:

lemma
  assumes *map f xs* = *map g ys*
  shows *length xs* = *length ys*

2 Very Busy Expressions Analysis (14 points)

An expression is *very busy* at the end of a command if, no matter what path is taken after executing the command, the expression must always be used before any of the variables occurring in it are assigned to. Consider the program

```
if a < b then
 (x := b + a;
  y := a + x)
else
 (x := a + x;
  y := (b + a) + 1)
```

or, in Isabelle syntax,

```
definition prog1 :: com where
  prog1 =
   IF Less (V ''a'') (V ''b'') THEN
    (''x'' ::= Plus (V ''b'') (V ''a'');;
     ''y'' ::= Plus (V ''a'') (V ''x''))
   ELSE
    (''x'' ::= Plus (V ''a'') (V ''x'');;
     ''y'' ::= Plus (Plus (V ''b'') (V ''a'')) (N 1))
```

The (sub)expression `b + a` is very busy at the beginning of the program because it is used in *both* branches of the `if`, without any intervening assignment to `a` or `b`. By contrast, `a + x` is not very busy, due to the first assignment in the `then` branch, which *may* alter the value of `x` (and hence of `a + x`).

The identification of very busy expressions enables a compiler optimization: The expression can be evaluated before its use and the result cached, avoiding some code duplication.

When considering very busy expressions, we restrict our attention to **arithmetic** expressions (*aexp*) occurring either in the condition of an `if` or `while` command or in the right-hand side of an assignment. Only nontrivial expressions are of interest (i.e., expressions other than a constant or variable). In the last assignment of the example above, the interesting expressions are `(b + a) + 1` and `b + a`.

Question 1) Define functions that extract the nontrivial arithmetic expressions from an arithmetic or Boolean expression:

```
fun suba :: aexp ⇒ aexp set where
  suba (N n) =
| suba (V x) =
| suba (Plus a b) =
fun $subb$ :: $bexp \Rightarrow aexp\ set$ where
  $subb\ (Bc\ c) =$
| $subb\ (Not\ b) =$
| $subb\ (And\ b_1\ b_2) =$
| $subb\ (Less\ a_1\ a_2) =$
```

2) The *very busy expressions analysis* computes the very busy expressions of a command. It is an instance of the gen/kill framework. Fill in the holes in the definition below:

```
fun
  $gen\_vb$ :: $com \Rightarrow aexp\ set$ and
  $kill\_vb$ :: $com \Rightarrow aexp\ set$
where
  $gen\_vb\ SKIP =$
| $gen\_vb\ (x ::= a) = suba\ a$
| $gen\_vb\ (c1\ ;;\ c2) = gen\_vb\ c1 \cup (gen\_vb\ c2 - kill\_vb\ c1)$
| $gen\_vb\ (IF\ b\ THEN\ c1\ ELSE\ c2) = subb\ b \cup (gen\_vb\ c1 \cap gen\_vb\ c2)$
| $gen\_vb\ (WHILE\ b\ DO\ c) = subb\ b$

| $kill\_vb\ SKIP =$
| $kill\_vb\ (x ::= a) =$
| $kill\_vb\ (c1\ ;;\ c2) =$
| $kill\_vb\ (IF\ b\ THEN\ c1\ ELSE\ c2) = kill\_vb\ c1 \cup kill\_vb\ c2$
| $kill\_vb\ (WHILE\ b\ DO\ c) = kill\_vb\ c$
```

```
definition $VB$ :: $com \Rightarrow aexp\ set \Rightarrow aexp\ set$ where
  $VB\ c\ A = (A - kill\_vb\ c) \cup gen\_vb\ c$
```

3) Annotate each command in *prog1* above with the very busy expressions before the command, assuming that the set of very busy expressions at the end of *prog1* is {}:

```
{                                                           }
if a < b then
 ({                                                          }
  x := b + a;
  {                                                          }
  y := a + x
  {                                                          })
else
 ({                                                          }
  x := a + x;
  {                                                          }
  y := (b + a) + 1
  {                                                          })
{}
```

3 Quickies (6 points)

(Wrong answers give 0 points.)

1) The live variable analysis is a
  (a) forward, may analysis;
  (b) forward, must analysis;
  (c) backward, may analysis;
  (d) backward, must analysis.

2) The very busy analysis from Question 2 is a
  (a) forward, may analysis;
  (b) forward, must analysis;
  (c) backward, may analysis;
  (d) backward, must analysis.

3) Which of the following statements are true?
  (a) A denotational semantics is a mapping from semantics to syntax.
  (b) Hoare logic is called an axiomatic semantics because it cannot be proved correct.
  (c) Small-step semantics are generally considered the most convenient way to reason about concrete programs.
  (d) Small-step semantics allow to reason about intermediate steps in a computation, hence their name.

4) Concerning the verification of a large fragment of Java with arrays and threads, based on the existing Jinja formalization by Klein and Nipkow, which of the following statements more accurately describes Lochbihler's experience?
  (a) Adding arrays was difficult, adding threads was relatively easy.
  (b) Adding threads was difficult, adding arrays was relatively easy.

5) List two features you found useful in Isabelle for formalizing semantics and/or your project, and two (mis)features that are lacking or missing. Justify very briefly.

4 Hoare Proof of Factorial (10 points)

The goal of this question is to prove partial correctness of a program that computes factorials $n! = 1 \cdot \ldots \cdot n$. For this question, we assume that our arithmetic expressions comprise a *Times* :: *aexp* ⇒ *aexp* ⇒ *bool* constructor, denoting multiplication (×).

1) Define a *functional* implementation of the factorial function on integers. Make sure that it returns 1 for nonpositive arguments.

fun *fact* :: *int* ⇒ *int* **where**

2) Write an IMP program that computes the factorial of the initial value of variable n imperatively, leaving the result in r. The program is allowed to modify n and any other variables. The program may be written in Isabelle syntax (e.g., *''x''* ::= *N 0*) or in the more convenient concrete syntax sometimes used in the course (e.g., `x := 0`).

definition *FACT* :: *com* **where**

3) Come up with a meaningful (syntactic) Hoare triple for *FACT*, capturing the requirement stated in 2. In particular, the triple should hold not only for your implementation of *FACT* but also for any other partially correct implementation.

```
{                                                           }
FACT
{                                                           }
```

4) Annotate the *FACT* program systematically with (syntactic) Hoare logic assertions, including especially loop invariants, and discharge any nontrivial proof obligation with a proof.

5 Recursion on Arithmetic Expressions (8 points)

Given the type of arithmetic expressions

```
datatype aexp =
  N int
| V vname
| Plus aexp aexp
```

1) Implement a simultaneous substitution function, which replaces occurrences of given variables by arbitrary expressions. The substitution is specified as a function of type *vname* ⇒ *aexp option*, which returns *None* for variables that are not affected by the substitution.

fun *subst* :: (*vname* ⇒ *aexp option*) ⇒ *aexp* ⇒ *aexp* **where**

2) Assume the existence of an evaluation function *aval* :: *aexp* ⇒ (*vname* ⇒ *value*) ⇒ *value*. Prove the substitution lemma, which states that evaluating an expression after substitution in a state *s* is the same as evaluating the original expression in a suitable state:

lemma *aval* (*subst* ϱ *a*) *s* = *aval a* (λx. *aval* (*subst* ϱ (*V x*)) *s*)

6 Deadlock Freedom (15 points)

We extend the IMP language with a synchronization mechanism inspired by Java's `synchronized` block. Unlike in Java, locks are identified statically by a number.

```
type_synonym lock = nat
datatype com =
    SKIP
  | Assign vname aexp               (_ ::= _ [1000, 61] 61)
  | Seq com com                     (_;;/ _  [60, 61] 60)
  | If bexp com com                 ((IF _/ THEN _/ ELSE _)  [0, 0, 61] 61)
  | While bexp com                  ((WHILE _/ DO _)  [0, 61] 61)
  | SYNCHRONIZED lock com
```

In our extended language, *SYNCHRONIZED l c* acquires lock *l*, executes command *c*, and finally releases lock *l*. Different processes may run at the same time and act on the same global state (memory), but each process holds its own set of locks (which it has acquired and not yet released).

```
type_synonym process = com × lock set
```

A running process is modeled as some code to execute next (or *SKIP*) together with the set of locks it holds. A lock can be held by at most one process at any point in time.

1) Complete the following definition a small-step semantics for lists of processes. The notation *ps* @ (*c*, *L*) # *qs* stands for an arbitrary list of processes with a process (*c*, *L*) somewhere in it. *Hint:* You might need an auxiliary function definition to handle the *SYNCHRONIZED* case.

```
inductive
  small_step :: process list × state ⇒ process list × state ⇒ bool (infix → 55)
where
  (ps @ (x ::= a, L) # qs, s) → (ps @ (SKIP, L) # qs, s(x := aval a s))
| (ps @ (SKIP ;; c₂, L) # qs, s) → (ps @ (c₂, L) # qs, s)
| (ps @ (c₁, L) # qs, s) → (ps @ (c₁', L') # qs, s') ⟹
    (ps @ (c₁ ;; c₂, L) # qs, s) →

| bval b s ⟹
                                                          →

| ¬ bval b s ⟹
                                                          →

| (ps @ (WHILE b DO c, L) # qs, s) →
    (ps @ (IF b THEN c ;; WHILE b DO c ELSE SKIP, L) # qs, s)
|                                                                          ⟹
    (ps @ (SYNCHRONIZED l c, L) # qs, s) →
```

2) A *deadlock* occurs when two or more processes are trying to acquire locks held by each other, and cannot make progress. The following situation, in which process *proc1* holds lock 1 and *proc2* holds lock 2, is a deadlock:

definition *proc1* :: *process* **where** *proc1* = (*SYNCHRONIZED 2 SKIP*, {*1*})
definition *proc2* :: *process* **where** *proc2* = (*SYNCHRONIZED 1 SKIP*, {*2*})

Introduce an inductive predicate *waits_for ps i j* that returns true if and only if process *i* is trying to acquire a lock held by process *j*. The predicate must return false if the indices are out of bound. The auxiliary function *hd_com* may be useful to retrieve the next command to execute.

fun *hd_com* :: *com* ⇒ *com* **where**
  hd_com (c_1 ;; c_2) = *hd_com* c_1
| *hd_com c* = *c*

inductive *waits_for* :: *process list* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**

$\Longrightarrow$

waits_for ps i j

Use *waits_for* to define a *deadlock ps* predicate that returns true if and only if some of the processes belonging to *ps* are involved in a deadlock.

inductive *deadlock* :: *process list* ⇒ *bool* **where**

$\Longrightarrow$

deadlock ps

3) One approach to avoid deadlocks is to enforce an order on the acquisition of the locks. A simple policy is to require locks to be acquired in increasing numeric order. Following this policy, command *com1* is acceptable, whereas *com2* and *com3* are not:

definition *com1* = (*SYNCHRONIZED 2* (*SYNCHRONIZED 4 SKIP*))
definition *com2* = (*SYNCHRONIZED 2* (*SYNCHRONIZED 1 SKIP*))
definition *com3* = (*SYNCHRONIZED 1* (*SYNCHRONIZED 1 SKIP*))

Define a type system to check this property. A judgment $l \vdash c$ indicates that command c respects the locking policy and acquires only locks with numbers l or above.

inductive *lock_type* :: *nat* ⇒ *com* ⇒ *bool* ((_/ ⊢ _) [*0*,*0*] *50*) **where**
  $l \vdash$ *SKIP*
|

Extra Sheet 1

Extra Sheet 2