Interactive Formal Verification
Course Handouts

Lawrence C Paulson

January 12, 2010

Interactive Formal Verification consists of 12 lectures and 4 practical sessions, each held on Wednesday mornings in SW02. The dates of the practical sessions for 2010 are 27 January, 10 February, 24 February, 10 March.

The handouts for the first two practical sessions will not be assessed in any way. Both handouts contain much more work than can be completed in an hour. You are not required to do all (indeed any) of the problems on these handouts, but I hope that you will do as many of them as you find beneficial for learning. Many more exercises can be found on the Internet, at http://isabelle.in.tum.de/exercises/. You may use the terminals in SW02 whenever the room has not been booked for another course.

The handouts for the last two practical sessions *will be assessed* to determine your final mark. For each assessed exercise, please complete the indicated tasks and write a brief document explaining your work. You may prepare these documents using Isabelle's theory presentation facility, but this is not required. A very simple way to print a theory file legibly is to use the Proof General command `Isabelle > Commands > Display draft`. You can combine the resulting output with a document produced using your favourite word processing package. A clear write-up describing elegant, clearly structured proofs of all tasks will receive maximum credit.

The first assessed exercise will be due on Friday, 12 March 2010 and the second assessed exercise will be due on Tuesday, 20 April 2010, both at 12 noon.

Please deliver a printed copy of each completed exercise to student administration by that deadline, and also send the corresponding theory file to me using the address lp15@cam.ac.uk.

1 Replace, Reverse and Delete

Define a function `replace`, such that `replace x y zs` yields `zs` with every occurrence of `x` replaced by `y`.

```
consts replace :: "'a ⇒ 'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems. You may have to prove some lemmas first.

```
theorem "rev(replace x y zs) = replace x y (rev zs)"
theorem "replace x y (replace u v zs) = replace u v (replace x y zs)"
theorem "replace y z (replace x y zs) = replace x z zs"
```

Define two functions for removing elements from a list: `del1 x xs` deletes the first occurrence (from the left) of `x` in `xs`, `delall x xs` all of them.

```
consts del1   :: "'a ⇒ 'a list ⇒ 'a list"
       delall :: "'a ⇒ 'a list ⇒ 'a list"
```

Prove or disprove (by counterexample) the following theorems.

```
theorem "del1 x (delall x xs) = delall x xs"
theorem "delall x (delall x xs) = delall x xs"
theorem "delall x (del1 x xs) = delall x xs"
theorem "del1 x (del1 y zs) = del1 y (del1 x zs)"
theorem "delall x (del1 y zs) = del1 y (delall x zs)"
theorem "delall x (delall y zs) = delall y (delall x zs)"
theorem "del1 y (replace x y xs) = del1 x xs"
theorem "delall y (replace x y xs) = delall x xs"
theorem "replace x y (delall x zs) = delall x zs"
theorem "replace x y (delall z zs) = delall z (replace x y zs)"
theorem "rev(del1 x xs) = del1 x (rev xs)"
theorem "rev(delall x xs) = delall x (rev xs)"
```

2 Power, Sum

2.1 Power

Define a primitive recursive function *pow x n* that computes x^n on natural numbers.

consts
```
  pow :: "nat => nat => nat"
```

Prove the well known equation $x^{m \cdot n} = (x^m)^n$:

theorem `pow_mult: "pow x (m * n) = pow (pow x m) n"`

Hint: prove a suitable lemma first. If you need to appeal to associativity and commutativity of multiplication: the corresponding simplification rules are named `mult_ac`.

2.2 Summation

Define a (primitive recursive) function *sum ns* that sums a list of natural numbers: $sum[n_1, \ldots, n_k] = n_1 + \cdots + n_k$.

consts
```
  sum :: "nat list => nat"
```

Show that *sum* is compatible with *rev*. You may need a lemma.

theorem `sum_rev: "sum (rev ns) = sum ns"`

Define a function *Sum f k* that sums *f* from 0 up to $k-1$: $Sum\ f\ k = f\ 0 + \cdots + f(k-1)$.

consts
```
  Sum :: "(nat => nat) => nat => nat"
```

Show the following equations for the pointwise summation of functions. Determine first what the expression `whatever` should be.

theorem `"Sum (%i. f i + g i) k = Sum f k + Sum g k"`
theorem `"Sum f (k + l) = Sum f k + Sum whatever l"`

What is the relationship between `sum` and `Sum`? Prove the following equation, suitably instantiated.

theorem `"Sum f k = sum whatever"`

Hint: familiarize yourself with the predefined functions `map` and `[i..<j]` on lists in theory List.

3 Assessed Exercise I: Greatest Common Divisors

The greatest common divisor of two natural numbers can be computed by a binary version of Euclid's algorithm:

- The GCD of x and 0 is x.
- If the GCD of x and y is z, then the GCD of $2x$ and $2y$ is $2z$.
- The GCD of $2x$ and y is the same as that of x and y if y is odd.
- The GCD of x and y is the same as that of $x - y$ and y if $y \leq x$.
- The GCD of x and y is the same as the GCD of y and x.

Note that frequently more than one of these cases is applicable, so it is not immediately obvious that they express a function.

Task 1 *Define inductively the set* `GCD` *such that* `(x,y,g) ∈ GCD` *means* `g` *is the greatest common divisor of* `x` *und* `y`*, as specified by the description above.*

Task 2 *Show that the GCD of* `x` *und* `y` *is really a divisor of both numbers:*

```
lemma GCD_divides: "(x,y,g) ∈ GCD ⟹ g dvd x ∧ g dvd y"
```

Task 3 *Show that the GCD of* `x` *und* `y` *is really the greatest common divisor of both numbers, with respect to the divides relation. Hint: consider using the predicate* `coprime`*, which belongs to the theory* `GCD`*. This theory will be present in your session because it is an ancestor of theory* `Prime`*.*

```
lemma GCD_greatest_dvd:
  "(x,y,g) ∈ GCD ⟹ d dvd x ⟹ d dvd y ⟹ d dvd g"
```

Task 4 *Show that, despite its apparent non-determinism, the relation* `GCD` *is deterministic and therefore defines a function:*

```
lemma GCD_unique:
  "(x,y,g) ∈ GCD ⟹ (x,y,g') ∈ GCD ⟹ g = g'"
```

Hint: first, prove a lemma establishing a connection between the relation `GCD` and the function `gcd`, which belongs to the theory `GCD`. This theory provides many lemmas that can help you complete this exercise.

4 Assessed Exercise II: Semantics

This assessed exercise continues the proofs concerning operational semantics that were outlined in the lectures. Please deliver the completed exercise and theory file by the appropriate deadline.

4.1 Syntax and Semantics of Commands

As in the lectures, the theory begins by specifying the types of locations, values (here we use the natural numbers), states, and finally arithmetic and boolean expressions.

```
typedecl loc
  — an unspecified (arbitrary) type of locations (adresses/names) for variables

types
  val   = nat — or anything else, nat used in examples
  state = "loc ⇒ val"
  aexp  = "state ⇒ val"
  bexp  = "state ⇒ bool"
  — arithmetic and boolean expressions are not modelled explicitly here,
  — they are just functions on states
```

The commands include SKIP, which does nothing, assignments, sequencing, conditionals and repetition. Note: our use of the semicolon character for sequencing could cause syntactic ambiguities if you attempt to use the semicolons to separate the preconditions of theorems. You can instead express properties using the symbol $\Longrightarrow$.

```
datatype
  com = SKIP
      | Assign loc aexp        (infixr ":==" 80)
      | Semi   com com         (infixr ";" 70)
      | Cond   bexp com com    ("IF _ THEN _ ELSE _"  [0, 90, 90] 91)
      | While  bexp com        ("WHILE _ DO _" [0, 91] 90)
```

The big-step execution relation `evalc` is defined inductively, as in the lectures.

```
inductive
  evalc :: "[com,state,state] ⇒ bool" ("⟨_, _⟩/ ⇝ _" [0,0,60] 60)
where
  Skip:    "⟨SKIP,s⟩ ⇝ s"
| Assign:  "⟨x :== a,s⟩ ⇝ s(x := a s)"

| Semi:    "⟨c0,s⟩ ⇝ s'' ⟹ ⟨c1,s''⟩ ⇝ s' ⟹ ⟨c0; c1, s⟩ ⇝ s'"
```

```
| IfTrue:  "b s ⟹ ⟨c0,s⟩ ⇝ s' ⟹ ⟨IF b THEN c0 ELSE c1, s⟩ ⇝ s'"
| IfFalse: "¬b s ⟹ ⟨c1,s⟩ ⇝ s' ⟹ ⟨IF b THEN c0 ELSE c1, s⟩ ⇝ s'"

| WhileFalse: "¬b s ⟹ ⟨WHILE b DO c,s⟩ ⇝ s"
| WhileTrue:  "b s ⟹ ⟨c,s⟩ ⇝ s'' ⟹ ⟨WHILE b DO c, s''⟩ ⇝ s'
               ⟹ ⟨WHILE b DO c, s⟩ ⇝ s'"
```

Next come commands that set up Isabelle's automation. The rules that make up the inductive definition can be used as introduction rules, and rule inversion from the definition supplies us with elimination rules. This again is very similar to the material in the lectures.

lemmas `evalc.intros [intro]` — use those rules in automatic proofs

```
inductive_cases skipE [elim!]:   "⟨SKIP,s⟩ ⇝ s'"
inductive_cases semiE [elim!]:   "⟨c0; c1, s⟩ ⇝ s'"
inductive_cases assignE [elim!]: "⟨x :== a,s⟩ ⇝ s'"
inductive_cases ifE [elim!]:     "⟨IF b THEN c0 ELSE c1, s⟩ ⇝ s'"
inductive_cases whileE [elim]:   "⟨WHILE b DO c,s⟩ ⇝ s'"
```

4.2 Equivalence of commands

Two commands are equivalent if they allow the same transitions.

```
definition
  equiv_c :: "com ⇒ com ⇒ bool" (infixr "∼" 60)
where
  "(c ∼ c') = (∀s s'. (⟨c, s⟩ ⇝ s') = (⟨c', s⟩ ⇝ s'))"
```

The following rule of inference, made available to Isabelle's automatic methods as an introduction rule, allows us to prove semantic equivalence statements. This again was covered in the lectures.

```
lemma equivI [intro!]:
  "(⋀s s'. ⟨c, s⟩ ⇝ s' = ⟨c', s⟩ ⇝ s') ⟹ c ∼ c'"
```

Task 1 *Prove the following theorem.*

```
lemma equiv_if3:
  "c1 ∼ c2 ⟹
   (IF b1 THEN c1 ELSE IF b2 THEN c2 ELSE c3) ∼
   (IF b2 THEN c2 ELSE IF b1 THEN c1 ELSE c3)"
```

Task 2 *Prove the following theorem, which establishes that semantic equivalence is a congruence relation with respect to* `While`. *Unlike analogous proofs for other constructors, this proof requires a lemma proved by induction.*

lemma `equiv_while`:

```
"c ∼ c' ⟹ (WHILE b DO c) ∼ (WHILE b DO c')"
```

Task 3 *Prove the following theorem, which expresses that the Boolean expression guarding the loop holds at the start of the loop body.*

lemma `equiv_while_if`:

```
"(WHILE b1 DO IF b2 THEN c1 ELSE c2) ∼
 (WHILE b1 DO IF (λs. b1 s & b2 s) THEN c1 ELSE c2)"
```

4.3 A Command Preserves a Boolean Expression

The following two properties allow a command `c` to be moved out of a conditional command. One of them can be proved as shown. The other one can be proved subject to the precondition `preserves c b`, which expresses that the command `c` preserves the value of the Boolean expression `b`.

Task 4 *Formalise the concept* `preserves c b` *in Isabelle, and prove both properties in the appropriate form.*

lemma `equiv_if1`:

```
"(IF b THEN (c; c1) ELSE (c; c2)) ∼ (c; (IF b THEN c1 ELSE c2))"
```

lemma `equiv_if2`:

```
"(IF b THEN (c1; c) ELSE (c2; c)) ∼ ((IF b THEN c1 ELSE c2); c)"
```