Technische Universität München
Fakultät für Informatik
Dr. Peter Lammich
Simon Wimmer

WS 2018/19
27.11.2018

Semantics of Programming Languages

Exercise Sheet 7

Exercise 7.1 While Invariants

We have pre-defined a while-combinator

$$\textit{while} :: ('a \Rightarrow \textit{bool}) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ \textit{option}$$

such that the following unfolding property holds:

$$\textit{while}\ b\ f\ s = (\textit{if}\ b\ s\ \textit{then}\ \textit{while}\ b\ f\ (f\ s)\ \textit{else}\ \textit{Some}\ s)$$

To prove anything about the computation result of *while* we need to use a proof rule with an invariant (similarly to what you have seen for the weakest precondition calculus). Prove that the following rule is correct:

theorem *while_invariant*:
  assumes "*wf R*" **and** "*I s*"
    and "$\bigwedge s.\ I\ s \Longrightarrow b\ s \Longrightarrow I\ (f\ s) \wedge (f\ s,\ s) \in R$"
  shows "$\exists s'.\ I\ s' \wedge \neg\ b\ s' \wedge \textit{while}\ b\ f\ s = \textit{Some}\ s'$"
  using *assms(1,2)*
proof *induction*
  case (*less s*)

Here is an example of how we can use this rule:

definition
  "$\textit{list\_sum}\ xs \equiv \textit{fst}\ (\textit{the}\ (\textit{while}\ (\lambda(s,\ xs).\ xs \neq [])\ (\lambda(s,\ xs).\ (s + \textit{hd}\ xs,\ \textit{tl}\ xs))\ (0,\ xs)))$"

lemma *list_sum_list_sum*:
  "$\textit{list\_sum}\ xs = \textit{sum\_list}\ xs$"
proof −
  let $?I$ =
  let $?R$ = "$\{((s,\ as),\ (s',\ bs)).\ \textit{length}\ as < \textit{length}\ bs \wedge \textit{length}\ bs \leq \textit{length}\ xs\}$"
  have "*wf ?R*"
    by (*rule wf_bounded_measure*[**where**
        ub = "$\lambda\_.\ \textit{length}\ xs$" **and** f = "$\lambda(\_,\ ys).\ \textit{length}\ xs - \textit{length}\ ys$"]) *auto*
  have "$\exists\ s'.\ ?I\ s' \wedge \neg\ (\lambda(s,\ xs).\ xs \neq [])\ s' \wedge$
    $\textit{while}\ (\lambda(s,\ xs).\ xs \neq [])\ (\lambda(s,\ xs).\ (s + \textit{hd}\ xs,\ \textit{tl}\ xs))\ (0,\ xs) = \textit{Some}\ s'$"
    apply (*rule while_invariant*[*OF* ‹*wf ?R*›])

```
    apply simp
   apply clarsimp
   subgoal for zs ys
     apply (rule exI[where x = "ys @ [hd zs]"])
     apply auto
     done
   done
 then show ?thesis
   unfolding list_sum_def by auto
qed
```

Fill in a suitable invariant!

Exercise 7.2 Weakest Preconditions

You have seen this definition of sum of the first n natural numbers before:

```
fun sum :: "int ⇒ int" where
"sum i = (if i ≤ 0 then 0 else sum (i − 1) + i)"

lemma sum_simps[simp]:
  "0 < i ⟹ sum i = sum (i − 1) + i"
  "i ≤ 0 ⟹ sum i = 0"
  by simp+

lemmas [simp del] = sum.simps
```

Consider the following program to calculate the sum of the first n natural numbers. Find suitable variants and invariants and prove that it fulfills the specification!

```
program_spec sum_prog
  assumes "n ≥ 0" ensures "s = sum n₀"
  defines ‹
    s = 0;
    i = 0;
    while (i < n)
      @variant ‹nat undefined›
      @invariant ‹undefined :: bool›
    {
      i = i + 1;
      s = s + i
    }
  ›
```

Recall the following scheme for squaring a non-negative integer:

```
1 2 3 4
2 2 3 4
3 3 3 4
4 4 4 4
```

Write down a program that implements this algorithm following the "count up scheme" and prove that it is correct!

Homework 7.1 Factorial

Submission until Tuesday, November 27, 2018, 10:00am.

In this homework you will prove the correctness of a program for computing the factorial of a positive integer. Consider the following definition of the factorial:

fun *factorial* :: *"int ⇒ int"* **where**
  "factorial i = (if i ≤ 0 then 1 else i ∗ factorial (i − 1))"

In the following program fill in suitable variants and invariants, and prove that the program fulfills the specification:

```
program_spec factorial_prog
  assumes "n ≥ 0" ensures "a = factorial n₀"
  defines ‹
    a = 1;
    i = 1;
    while (i ≤ n)
      @variant ‹nat undefined›
      @invariant ‹undefined :: bool›
    {
      a = a * i;
      i = i + 1
    }
  ›
```

Homework 7.2 Fibonacci Sequence

Submission until Tuesday, November 27, 2018, 10:00am.

Consider the following definition of the Fibonacci numbers:

fun *fib* :: *"int ⇒ int"* **where**
  "fib i = (if i ≤ 0 then 0 else if i = 1 then 1 else fib (i − 2) + fib (i − 1))"

lemma *fib_simps*[*simp*]:
  "$i \leq 0 \Longrightarrow fib\ i = 0$"
  "$i = 1 \Longrightarrow fib\ i = 1$"
  "$i > 1 \Longrightarrow fib\ i = fib\ (i - 2) + fib\ (i - 1)$"
  by *simp+*

lemmas [*simp del*] = *fib.simps*

Prove that the following program implements the Fibonacci numbers correctly:

```
program_spec fib_prog
  assumes "n ≥ 0" ensures "a = fib n"
  defines ‹
    a = 0; b = 1;
    i = 0;
    while (i < n)
      @variant ‹nat undefined›
      @invariant ‹undefined :: bool›
    {
      c = b;
      b = a + b;
      a = c;
      i = i + 1
    }
  ›
```

You can get two bonus points if you also manage to prove that the program fulfills the post-condition if the pre-condition is vacuous:

```
program_spec fib_prog'
  assumes True ensures "a = fib n₀"
  defines ‹
    a = 0; b = 1;
    i = 0;
    while (i < n)
      @variant ‹nat undefined›
      @invariant ‹undefined :: bool›
    {
      c = b;
      b = a + b;
      a = c;
      i = i + 1
    }
  ›
```

Homework 7.3 Unmodified Variables

Submission until Tuesday, November 27, 2018, 10:00am.

In this exercise we want to prove that variables that do not occur on the left hand side of an assignment are never modified. First define the set of all variables that occur on the left hand side of an assignment:

fun *lhsv* :: *"com ⇒ vname set"* **where**

Now show that we can always strengthen a weakest precondition with the knowledge that the variables that do not occur in *lhsv c* remain unmodified:

theorem *wp_strengthen_modset*:
  $\text{“}wp\ c\ Q\ s \Longrightarrow wp\ c\ (\lambda s'.\ Q\ s' \wedge (\forall x.\ x \notin lhsv\ c \longrightarrow s'\ x = s\ x))\ s\text{”}$

Hint: You do not want to prove this on *wp* directly. Instead come up with a lemma that captures the essence of why we can do this first. The proof for *wp* should follow trivially from it.