```
theory Examples
imports "vcg/VCG"
begin

section ‹Examples›
lemmas nat_distribs = nat_add_distrib nat_diff_distrib Suc_diff_le nat_mult_distrib nat_div_distrib

subsection ‹Common Loop Patterns›

subsubsection ‹Count Up›
text ‹
  Counter ‹c› counts from ‹0› to ‹n›, such that loop is executed ‹n› times.
  The result is computed in an accumulator ‹a›.
›
text ‹The invariant states that we have computed the function for the counter value ‹c››
text ‹The variant is the difference between ‹n› and ‹c›, i.e., the number of
  loop iterations that we still have to do›


program_spec exp_count_up
  assumes "0≤n"
  ensures "a = 2^nat n_0"
  defines ‹
    a = 1;
    c = 0;
    while (c<n)
      @variant ‹nat (n-c)›
      @invariant ‹n=n_0 ∧ 0≤c ∧ c≤n ∧ a=2^nat c›
    {
      a=2*a;
      c=c+1
    }
  ›
  apply vcg
  by (auto simp: algebra_simps nat_distribs)
  thm exp_count_up_def

subsubsection ‹Count down›
text ‹Essentially the same as count up, but we (ab)use the input variable as counter›

text ‹The invariant is the same as for count-up.
  Only that we have to compute the actual number
  of loop iterations by ‹n_0 - n›. We locally introduce the name ‹c› for that.
›

```

```
program_spec exp_count_down
  assumes "0≤n"
  ensures "a = 2^nat n₀"
  defines ‹
    a = 1;
    while (n>0)
      @variant ‹nat n›
      @invariant ‹let c = n₀-n in 0≤n ∧ n≤n₀ ∧ a=2^nat c›
    {
      a=2*a;
      n=n-1
    }
  ›
  apply vcg_cs
  by (auto simp: algebra_simps nat_distribs)


subsubsection ‹Approximate from Below›
text ‹Used to invert a monotonic function.
  We count up, until we overshoot the desired result,
  then we subtract one.
›
text ‹The invariant states that the ‹r-1› is not too big.
  When the loop terminates, ‹r-1› is not too big, but ‹r› is already too big,
  so ‹r-1› is the desired value (rounding down).
›
text ‹The variant measures the gap that we have to the correct result.
  Note that the loop will do a final iteration, when the result has been reached
  exactly. We account for that by adding one, such that the measure also decreases in this case.
›

program_spec sqr_approx_below
  assumes "0≤n"
  ensures "0≤r ∧ r² ≤ n₀ ∧ n₀ < (r+1)²"
  defines ‹
    r=1;
    while (r*r ≤ n)
     @invariant ‹n=n₀ ∧ 0≤r ∧ (r-1)² ≤ n₀›
     @variant ‹nat (n+1-r*r)›
      { r = r + 1 };
    r = r - 1
  ›
  apply vcg
  apply (auto simp: algebra_simps power2_eq_square)
  done
```

```



      @variant ‹nat (n + 1 - r*r)›
      @invariant ‹n=n₀ ∧ 0≤r ∧ (r-1)² ≤ n₀›
subsubsection ‹Bisection›
text ‹A more efficient way of inverting monotonic functions is by bisection,
  that is, keep track of a possible interval for the solution, and half the
  interval in each step. The program will need $O(\log n)$ iterations, and is
  thus very efficient in practice.

  Although the final algorithm looks quite simple, getting it right can be
  quite tricky.
›
text ‹The invariant is surprisingly easy, just stating that the solution
  is in the interval ‹l..<h›. ›

program_spec sqr_bisect
  assumes "0≤n" ensures "r²≤n ∧ n<(r+1)²"
  defines ‹
    l=0; h=n+1;
    while (l+1 < h)
      @variant ‹nat (h-l)›
      @invariant ‹n=n₀ ∧ 0≤l ∧ l<h ∧ l²≤n ∧ n < h²›
    {
      m = (l + h) / 2;
      if m*m ≤ n then l=m else h=m
    };
    r=l
  ›
  apply vcg

  text ‹We use quick-and-dirty apply style proof to discharge the VCs›
  apply (auto simp: power2_eq_square algebra_simps add_pos_pos)
  apply (smt not_sum_squares_lt_zero)
  by (smt mult.commute semiring_normalization_rules(3))

subsection ‹More Numeric Algorithms›

subsubsection ‹Euclid's Algorithm (with subtraction)›

(* Crucial Lemmas *)
thm gcd.commute gcd_diff1

program_spec euclid1
  assumes "a>0 ∧ b>0"
```

```
  ensures "a = gcd a_0 b_0"
  defines ‹
    while (a≠b)
      @invariant ‹gcd a b = gcd a_0 b_0 ∧ (a>0 ∧ b>0)›
      @variant ‹nat ( a+b )›
    {
      if a<b then b = b-a
      else a = a-b
    }
  ›
  apply vcg_cs
  apply (metis gcd.commute gcd_diff1)
  apply (metis gcd.commute gcd_diff1)
  done


subsubsection ‹Euclid's Algorithm (with mod)›

(* Crucial Lemmas *)
thm gcd_red_int[symmetric]

program_spec euclid2
  assumes "a>0 ∧ b>0"
  ensures "a = gcd a_0 b_0"
  defines ‹
    while (b≠0)
      @invariant ‹gcd a b = gcd a_0 b_0 ∧ b≥0 ∧ a>0›
      @variant ‹nat ( b )›
    {
      t = a;
      a = b;
      b = t mod b
    }
  ›
  apply vcg_cs
  apply (simp add: gcd_red_int[symmetric])
  done

subsubsection ‹Extended Euclid's Algorithm›

text ‹TBD. Homework?›


subsection ‹Debugging›

subsubsection ‹Testing Programs›
```

```

text ‹Stepwise›
schematic_goal "(sqr_approx_below,<''n'':=4>) ⇒ ?s"
  unfolding sqr_approx_below_def
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  apply big_step
  done

text ‹Or all steps at once›
schematic_goal "(sqr_bisect,<''n'':=4900000001>) ⇒ ?s"
  unfolding sqr_bisect_def
  by big_step+


end

```