Reasoning About Programs

Panagiotis Manolios
Northeastern University

April 3, 2019
Version: 107

1

Introduction

These lecture notes were developed for Logic and Computation, a freshman-level class taught at the College of Computer and Information Science of Northeastern University. Starting in Spring 2008, this is a class that all students in the college are required to take.

The goals of the Logic and Computation course are to provide an introduction to formal logic and its deep connections to computing. Logic is presented from a computational perspective using the ACL2 Sedan theorem proving system. The goal of the course is to introduce fundamental, foundational methods for modeling, designing, specifying and reasoning about computation. The topics covered include propositional logic, recursion, contracts, testing, induction, equational reasoning, termination analysis, term rewriting, and various proof techniques. We show how to use logic to formalize the syntax and semantics of the core ACL2s language, a simple LISP-based language with contracts. We then use the ACL2s language to formally reason about programs, to model systems at various levels of abstraction, to design and specify interfaces between systems and to reason about such composed systems. We also examine decision procedures for fragments of first-order logic and how such decision procedures can be used to analyze models of systems.

The students taking the Logic and Computation class have already taken a programming class in the previous semester, in Racket. The course starts by reviewing some basic programming concepts. The review is useful because at the freshman level students benefit from seeing multiple presentations of key concepts; this helps them to internalize these concepts. For example, in past semesters I have asked students to write very simple programs (such as a program to append two lists together) during the first week of classes and a surprisingly large number of students produce incorrect code.

During the programming review, we introduce the ACL2s language. This is the language we use throughout the semester and it is similar to Racket. The syntax and semantics of the core ACL2s language are presented in a mathematical way. We provide enough information so that students can determine what sequence of glyphs form a well-formed expression and how to formally evaluate well-formed expressions potentially containing user-defined functions with constants as arguments (this is always in a first-order setting). This is a pretty big jump in rigor for students and is advanced material for freshmen students, but they already have great intuitions about evaluation from their previous programming class. This intuition helps them understand the rigorous presentation of the syntax and semantics, which in turns helps strengthen their programming abilities.

The lecture notes are sparse. It would be great to add more exercises, but I have not done that yet. Over the course of many years, we have amassed a large collection of homework problems, so students see lots of exercises, and working through these exercises is a great way for them to absorb the material, but the exercises are not in the notes. You can think of the lecture notes as condensed notes for the course that are appropriate for someone who knows the material as a study guide. The notes can also be used as a starting point by

students, who should mark them up with clarifications as needed when they attend lectures. I advise students to read the lecture notes before class. This way, during class they can focus on the lecture instead of taking notes and they are better prepared to ask for clarifications.

When I started teaching the class, I used the ACL2 book, *Computer-Aided Reasoning, An Approach* by Kaufmann, Manolios and Moore. However, over the years I became convinced that using an untyped first-order logic was not the optimal way of introducing logic and computation to students because they come in with a typed view of the world. That's not to say they have seen type theory; they have not. But, they are surprised when a programming language allows them to subtract a string from a rational number. Therefore, with the help of my Ph.D. student Harsh Chamarthi, I have focused on adding type-like capabilities to ACL2s. Most notably, we added a new data definition framework to ACL2s that supports enumeration, union, product, record, map, (mutually) recursive and custom types, as well as limited forms of parametric polymorphism. We also introduced the `defunc` macro, which allows us to formally specify input and output contracts for functions. These contracts are very general, *e.g.*, we can specify that `/` is given two rationals as input, and that the second rational is not 0, we can specify that `zip` is given two lists of the same length as input and returns a list of the same length as output and so on. Contracts are also checked statically, so ACL2s will not accept a function definition unless it can prove that the function satisfies its contracts and that for every legal input and every possible computation, it is not possible during the evaluation of the function being defined to be in a state where some other function is poised to be evaluated on a value that violates its input contract. I have found that a significant fraction of erroneous programs written by students have contract violations in them, and one of the key things I emphasize is that when writing code, one needs to think carefully about the contracts of the functions used and why the arguments to every function call satisfy the function's contract. Contracts are the first step towards learning how to specify interfaces between systems. With the move to contracts, the ACL2 book became less and less appropriate, which led me to write these notes.

I have distributed these notes to the students in Logic and Computation for several years and they have found lots of typos and have made many suggestions for improvement. Thanks and keep the comments coming!

9

Reasoning about Imperative Code

We have seen how to reason about programs written in the ACL2s language using the ACL2s logic and the ACL2s theorem prover. How do reason about programs written in other languages? Can we use what he have learned so far or do we have to define a logic for the language and then a theorem prover for that language and logic? We will explore one approach to reasoning about programs in other languages in this chapter. We start by introducing a simple imperative language (SIP) and define its semantics using ACL2s. We then discuss how to reason about SIP programs using invariants, loop invariants, pre-conditions, post-conditions, assumptions and guarantees.

9.1 SIP: A Simple Imperative Language

We introduce SIP, a simple imperative language, via an example. Consider the following simple SIP program.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while  (cnt < m)
  { cnt  := cnt+1;
    res  := res+n;
  }
  return res;
}
```

SIP programs have a name; in the above example it is `multiply`. The SIP programs we will consider have one procedure called `main`. In the program above, main has two inputs, `n` and `m`. Both are of type `nat`. In our language the type, we have two types. The type `int` corresponds to integers of arbitrary precision, *i.e.*, there is no minimal integer and there is no maximal integer. We also support the type `nat`, which corresponds to integers greater than or equal to 0.

All non-input variables need declarations. The variables `res` and `cnt` are declared to be of type `int`. We use the assignment operator (`:=`) to assign initial values to the variables.

Next we have a while loop, which include the loop condition `(cnt < m)` and the body of the loop, which updates the variables `cnt` and `res`.

Finally we have a `return` statement that indicates what the prodecure returns. In `multiply`, the value of `res` is returned.

Before we formalize the semantics of such programs, let us start by considering program traces. We will do that with the simple approach of adding `print` statements as follows.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while  (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
  }
  print(n, m, cnt, res);
  return res;
}
```

Now, imagine running the program with the inputs `n = 7` and `m = 4`. The trace our program generates is the following (without the header).

```
n  m  res  cnt
7  4    0    0
7  4    7    1
7  4   14    2
7  4   21    3
7  4   28    4
```

The trace makes it clear that the program computes `n * m` by repeatedly adding `m` to `res`.

9.2 Semantics of SIP

To define the semantics of SIP, we will write a compiler that takes a SIP program and generates an ACL2s representation. This compiler can be written in any language. We will not define this compiler, but we will show what it generates for the `multiply` program.

```
(sip-program
 multiply
 ((natp n) (natp m))
 ((intp cnt) (intp res))
 ((res 0) (cnt 0))
 (< cnt m)
 ((print n m cnt res)
  (cnt . (+ 1 cnt))
  (res . (+ res n)))
 ((print n m cnt res))
 (res))
```

The above is a call of the ACL2s macro `sip-program` that include the name of the SIP program (`multiply`), the input variables and their types, the declared variables and their

types, the initialization code, the while loop condition, the while loop body, where we either have a `print` statement or an assignment operation. Note that we use alists to denote how a variable gets updated. Then we have any post-loop statements and finally what, if anything, gets returned.

Notice that the ACL2s representation is very close to the original SIP program and that is by design because we want this step to be as simple and transparent as possible.

Now, the `sip-program` macro will wind up generating a list of definitions that provide the semantics of `multiply`. The macro generates a list of utilities. One of those utilities allows us to generate traces. More of these utilities will be introduced later. We will not define the macro, but we will reveal what it generates.

First, the macro generates `sip-multiply`, an ACL2s version of `multiply`. In order to define such a function, we need to deal with the while loop and this is done via the use of a recursive helper function, `sip-multiply-loop`, as follows.

```
:program

(definec sip-multiply-loop (n :nat m :nat cnt :int res :int) :int
  (declare (ignorable n m cnt res))
  (if (< cnt m)
      (let ((cnt (+ 1 cnt))
            (res (+ res n)))
        (sip-multiply-loop n m cnt res))
    res))

(definec sip-multiply (n :nat m :nat) :int
  (declare (ignorable n m))
  (let ((res 0)
        (cnt 0))
    (sip-multiply-loop n m cnt res)))
```

Now, we can run the program as the following examples show.

```
(sip-multiply 7 4)
(sip-multiply 127 671)
```

Not only do we now have executable versions of SIP programs such as `multiply`, but we also have access to all ACL2s capabilities. For example, we can use `check=` and `test?` forms as shown below.

```
(check= (sip-multiply 7 4) (* 7 4))

(test? (implies (and (natp n) (natp m))
                (equal (sip-multiply n m) (* n m))))
```

The `sip-program` macro also generates utility functions for generating traces.

```
(definec sip-multiply-loop-trace (n :nat m :nat cnt :int res :int) :tl
  (declare (ignorable n m cnt res))
  (if (< cnt m)
      (cons (list n m cnt res)
            (let ((cnt (+ 1 cnt))
                  (res (+ res n)))
```

```
              (sip-multiply-loop-trace n m cnt res)))
    `((,n ,m ,cnt ,res))))

(definec sip-multiply-trace (n :nat m :nat) :tl
  (declare (ignorable n m))
  (let ((res 0)
        (cnt 0))
    (app '((multiply-trace)
           (n m cnt res))
         (sip-multiply-loop-trace n m cnt res))))
```

Here is an example of the trace utility functions in action.

```
(sip-multiply-trace 7 4)
(sip-multiply-trace 21 18)
```

In order to develop robust, usable tools, we have to deal with syntax checking, type checking, error reporting, termination analysis, etc. during the compilation step. We will ignore these issues as considering them in any depth will take us too far afield. We will therefore assume that SIP programs are free of errors and that they are terminating.

Exercise 9.1 *Define a compiler that given a SIP program generates the corresponding* `sip-program` *form. You can use your favorite language to do that.*

Exercise 9.2 *Define a version of the* `sip-program` *macro that generates the forms shown above.*

Exercise 9.3 *You may wonder why the functions generated by* `sip-program` *are defined in* `:program` *mode. Come up with a SIP program that is terminating, but for which one of the ACL2s functions generated by* `sip-program` *is non-terminating.*

9.3 Reasoning About SIP Programs

Now that we can compile SIP programs to ACL2s programs, we can reason about SIP programs using ACL2s. Try the following exercise before continuing.

Exercise 9.4 *Prove the following conjecture using paper and pencil. Then prove it using ACL2s. Assume that* `sip-program` *generated* `:logic` *mode functions.*

```
(implies (and (natp n)
              (natp m))
         (equal (sip-multiply n m) (* n m)))
```

Notice that since `sip-multiply` is a non-recursive program that calls `sip-multiply-loop`, we really need a lemma about `sip-multiply-loop`. Also, for reasons we have already discussed in the context of tail recursive functions, we will not be able to prove a lemma about `sip-multiply-loop` that includes constants: we will have to generalize. Once we have the appropriate lemma, then the main theorem follows via equational reasoning.

There are many lemmas that can be used to prove the main theorem. We know we have to generalize, so one idea is to prove a lemma that before contract completion is as close

as possible to this, *i.e.*, we are trying to determine that `sip-multiply-loop` does given arbitrary inputs.

```
(equal (sip-multiply-loop n m cnt res)
       ...)
```

We can use ACL2s to help us discover an appropriate lemma (of course we need to perform contract completion). Since we keep adding to `res` during the loop, we expect an expression of the form `(+ ... res)`. We are adding `n` each time through the loop, so we should get something like `(+ (* n ...) res)`. How many times do we add `n`? At the beginning of the loop it is `m`, but at an arbitrary point during the loop we have already gone through `cnt` iterations, so there should be `(- m cnt)` iterations left. So, we can try the following.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res))
          (equal (sip-multiply-loop n m cnt res)
                 (+ (* n (- m cnt)) res))))
```

We get counterexamples, but they involve assignments where `cnt` is greater than `m`, which never happens. We can add an extra hypothesis to account for that and we get.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m))
          (equal (sip-multiply-loop n m cnt res)
                 (+ (* n (- m cnt)) res))))
```

No counterexamples are reported, so let us see if ACL2s can prove it.

```
(defthm mult-lemma
  (implies (and (natp n) (natp m) (intp cnt) (intp res)
                (<= cnt m))
           (equal (sip-multiply-loop n m cnt res)
                  (+ (* n (- m cnt)) res))))
```

That worked, so we can try the main theorem, which now goes through using only equational reasoning.

```
(thm
 (implies (and (natp n) (natp m))
          (equal (sip-multiply n m) (* n m))))
```

Here is another way to go about proving the main theorem. We will try to prove a lemma of the following form.

```
(implies ...
         (equal (sip-multiply-loop n m cnt res)
                (* n m)))
```

The idea here is that, in addition to contract completion hypotheses, we want to identify all the invariants that hold before and after `sip-multiply-loop`, as these invariants should imply that the final result we get is `(* n m)`. Again, we can use ACL2s to help us find these invariants. We start with `(<= cnt m)` since we needed that in our first attempt.

```
(test?
```

```
(implies (and (natp n) (natp m) (intp cnt) (intp res)
              (<= cnt m))
         (equal (sip-multiply-loop n m cnt res)
                (* n m))))
```

ACL2s generates counterexamples that include negative numbers. We can rule such examples by observing that `cnt` and `res` are always non-negative.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m) (<= 0 cnt) (<= 0 res))
          (equal (sip-multiply-loop n m cnt res)
                 (* n m))))
```

ACL2s still generates counterexamples. There are counterexamples where `res` is not even a multiple of `n`. Actually, our hypotheses do not relate `res` with the input variables at all, so of course we are going to get counterexamples! What is the relationship between `res` and the other variables? The answer to the question leads us to the following conjecture.

```
(test?
 (implies (and (natp n) (natp m) (intp cnt) (intp res)
               (<= cnt m) (<= 0 cnt) (<= 0 res) (= res (* cnt n)))
          (equal (sip-multiply-loop n m cnt res)
                 (* n m))))
```

ACL2s did not find any counterexamples and it can prove the conjecture. In fact, we do not even need all of the hypotheses, as the following theorem shows.

```
(defthm mult-lemma
  (implies (and (natp n) (natp m) (intp cnt) (intp res)
                (<= cnt m) (= res (* cnt n)))
           (equal (sip-multiply-loop n m cnt res)
                  (* n m))))
```

Again, the main theorem goes through using only equational reasoning.

```
(thm
 (implies (and (natp n) (natp m))
          (equal (sip-multiply n m) (* n m))))
```

So, we can reason about SIP programs. Are we done? No because this state of affairs is not entirely satisfatory, for several reasons. One technical problem is that, as Exercise 9.3 shows, we cannot expect `sip-program` to generate `:logic` mode functions, unless we do more work to ensure we can prove termination. But, the main objection to our current approach is that a SIP programmer should not have to learn ACL2s in order to reason about SIP programs.

We need another approach, one where the reasoning is done directly on SIP programs. We use our running example to introduce the key ideas.

First, we need a mechanism by which we can express what it is that `main` is supposed to do. We will use *`Guarantee`* statements at the end of `main` to specify what `main` is supposed to do. For our example, `main` is supposed to set `res` to the product of `n` and `m`, so we have the following *`Guarantee`* statement.

`Guarantee:` ⟦ `res = n*m` ⟧

The brackets are there to make it clear that this is not program text; it is an *invariant*, a Boolean-valued statement that holds whenever program execution reaches the statement (in this case, when `main` ends).

There is a dual statement that expresses what we can assume about the inputs, beyond their types. That statement is the ***Assume*** statement. For example, if `n` and `m` were declared to be of type `int`, then we could have added the following assumption at the beginning of the program.

```
Assume: ⟦ n >=0 & m >= 0⟧
```

So, the specification of a procedure includes both the assumption and guarantee and a procedure is correct if whenever it is given inputs that satisfy the types and the assumption, then when the procedure finishes execution, the guarantee holds. Notice the similarity between this and contracts in ACL2s.

In many verification systems, the terms ***Precondition*** and ***Postcondition*** are used instead of ***Assume*** and ***Guarantee***.

Now that we know how to specify what SIP programs are supposed to do, how do we prove that they are correct? We are going to design a system where programmers do not have to provide proofs. Instead, they only have to provide key insights. This is similar to what we did when reasoning about the ACL2s version of the code using ACL2s. Once we identified the main lemma, ACL2s took care of all the tedious equational reasoning for us. SIP programmers will not even have to write out lemmas. Instead, they will only provide *loop invariants*: invariants that hold right before the loop condition is evaluated. Suppose that the user has provided loop invariant I, then they are claiming that I holds before and after the loop, *i.e.*, that I holds whenever program execution reaches the program locations indicated below.

```
program multiply
main(n, m: nat)
{ var res, cnt: int;
  res := 0;
  cnt := 0;
  while ⟦I⟧ (cnt < m)
  { print(n, m, cnt, res);
    cnt  := cnt+1;
    res  := res+n;
    ⟦I⟧
  }
  print(n, m, cnt, res);
  return res;
  Guarantee: ⟦ res = n*m ⟧
}
```

We already saw that we can prove that the ACL2s version of the `multiply` program is correct using `mult-lemma`. The same idea works for when reasoning about the SIP program directly. Instead of requiring the user to specify types, we will infer the types from the variable declarations, so we will only require the user to set I to `cnt <= m & res = cnt * n` and the solver (ACL2s, but this is not something the programmers needs to know) can prove correctness. In order to prove correctness, we first need to show that I is a *loop invariant*: that it holds when program execution reaches the loop the first time around and

that if we are in any state that satisfies the invariant before the loop, then the invariant holds after the loop as well. Once we prove that I is a loop invariant, we use it to show the guarantee. Our proof obligation here is that whenever program execution reaches the end of the program, the guarantee holds. To get to the end of the program, we have to get past the loop, which means we satisfy I and the loop condition fails. If any such state satisfies the guarantee, then we have proven program correctness.

This process of generating proof obligations is often called *verification condition generation*. Many tools for reasoning about various programming languages operate in this way. While we will not define such a verification condition generator here, it is not hard to define such a tool using ACL2s; in fact, the `sip-program` macro is what generates all the code needed to generate verification conditions. We will show the proof obligaitons generated when I is `cnt <= m & res = cnt * n`.

```
; Show that the invariant holds initially.
(thm
 (let ((res 0) (cnt 0))
      (implies (and (intp cnt)
                    (intp res)
                    (natp n)
                    (natp m))
               (and (<= cnt m) (= res (* cnt n))))))

; Show that the invariant is inductive
(thm
 (implies (and (and (intp cnt)
                    (intp res)
                    (natp n)
                    (natp m))
               (< cnt m)
               (and (<= cnt m) (= res (* cnt n))))
          (and (<= (+ 1 cnt) m)
               (= (+ res n) (* (+ 1 cnt) n)))))

; Show that the loop invariant implies the guarantee
(thm
 (implies (and (and (intp cnt)
                    (intp res)
                    (natp n)
                    (natp m))
               (not (< cnt m))
               (<= cnt m)
               (= res (* cnt n)))
          (equal res (* n m))))
```

Exercise 9.5 *Provide a paper and pencil proof of the above proof obligations.*

Exercise 9.6 *Play the invariant discovery game* `http://invgame.atwalter.com/` *to gain experience with loop invariants.*

Exercise 9.7 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program multiply-by-1000
main(k: int)
{ var res, i: int;
  i := 0;
  res := 10;
  while ⟦I⟧  (i < 1000)
  { print(k, i, res);
    res := res + k;
    i := i + 1;
    ⟦I⟧
  }
  print(k, i, res);
  Guarantee: ⟦ res=10+k*1000 ⟧
}
```

Exercise 9.8 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program mult-of-6
main(n: nat)
{ var cnt, res: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < n)
  { print(n, cnt, res);
    cnt := cnt + 1;
    res := res + (cnt * cnt);
   ⟦I⟧
  }
  print(n, cnt, res);
  Guarantee: ⟦ res = (n * (n + 1) * (1 + (2 * n))) / 6 ⟧
}
```

Exercise 9.9 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program summation
main(n: nat)
{ var i, sum: int;
  sum := 0;
  i   := 1;
  while ⟦I⟧ (i <= n)
  { print(n, i, sum);
    sum := sum + i;
    i   := i + 1;
    ⟦I⟧
```

```
  }
  print(n, i, sum);
  Guarantee: ⟦ sum = n*(n+1)/2 ⟧
}
```

Exercise 9.10 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program summation2
main(n: nat)
{ var sum, cnt, mys: int;
  sum := 0;
  mys := 0;
  cnt := 1;
  while ⟦I⟧ (cnt <= n)
  { print(n, mys, cnt, sum);
    sum := sum + cnt;
    mys := cnt;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(n, mys, cnt, sum);
  Guarantee: ⟦ sum = n*(n+1)/2  ⟧
}
```

Exercise 9.11 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program mult-by-add
main (j, k: nat)
{ var i: int;
  i := 0
  while ⟦I⟧ (i < j*k)
  { print(j, k, i);
    i := i + 1;
    ⟦I⟧
  }
  print(j, k, i);
  Guarantee: ⟦ i = j*k  ⟧
}
```

Exercise 9.12 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program square-times-2
main(k: nat)
{ var res, cnt: int;
  cnt := 0;
  res := 0;
```

```
  while ⟦I⟧ (cnt < k)
  { print(k, cnt, res);
    res := res + 2*k;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(k, cnt, res);
  Guarantee: ⟦ res=2*k ^2 ⟧
}
```

Exercise 9.13 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program square-times-const
main(k: nat, j: int)
{ var res, cnt: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < k)
  { print(k, j, cnt, res);
    res := res + j*k;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print(k, j, cnt, res);
  Guarantee: ⟦ res=j*k ^2 ⟧
}
```

Exercise 9.14 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program cube
main(n: nat)
{ var cnt, res: int;
  cnt := 0;
  res := 0;
  while ⟦I⟧ (cnt < n)
  { print (n, cnt, res);
    res := res + (3 * ((cnt+1) ^2)) + ((cnt+1) * -3) + 1;
    cnt := cnt + 1;
    ⟦I⟧
  }
  print (n, cnt, res);
  Guarantee: ⟦ res = n ^3 ⟧
}
```

Exercise 9.15 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program cube2
main(n: nat)
{ var cnt, res, a, b: int;
  cnt := 0;
  res := 0;
  a := 1;
  b := 6;
  while ⟦I⟧ (cnt < n)
  { print(n, cnt, a, b, res);
    cnt := cnt + 1;
    res := res + a;
    a := a + b;
    b := b + 6;
    ⟦I⟧
  }
  print(n, cnt, a, b, res);
  Guarantee: ⟦ res = n ^3 ⟧
}
```

Exercise 9.16 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program int-square-root
main(n: nat)
{ var cnt, sqr, odd: int;
  cnt := 0;
  sqr := 1;
  odd := 1;
  while ⟦I⟧ (sqr <= n)
  { print (n, sqr, odd, cnt);
    cnt := cnt+1;
    odd := odd+2;
    sqr := sqr+odd;
    ⟦I⟧
  }
  print (n, sqr, odd, cnt);
  Guarantee: ⟦ cnt ^2 <= n & n < (cnt+1) ^2 ⟧
}
```

Exercise 9.17 *Exhibit a loop invariant, prove that it is a loop invariant and prove that it implies the guarantee.*

```
program binary-product
main(a, b: nat)
{ var dig, cnt, res: int;
  dig := a;
```

```
  cnt := b;
  res := 0;
  while ⟦I⟧ (cnt != 0)
  { print(a, b, dig, cnt, res);
    if (cnt       res := res + dig;
      cnt := cnt - 1;
    } else {
      dig := 2*dig;
      cnt := cnt / 2;
    }
    ⟦I⟧
  }
  print(a, b, dig, cnt, res);
  Guarantee: ⟦ res = a*b ⟧
}
```