

Sequential Programs

P415/515 STUDENTS: This material is fragmentary and comes from several sources. I encourage you to ask me about any missing background or confusion in notation. As I mentioned in class, many Intro-to-verification courses would carry on from this point to look at these matters. We will too, but later, after we explore model checking ...SDJ

The goal in this chapter is to take advantage of our language-specification techniques in devising a system for proving programs correct. We will do this by showing that *programs* and *assertions* interact in precise ways. In fact, we can always reduce a program to a purely logical formula in such a way that whenever the formula is true the program is correct.

Terms

We are actually going to build a framework, or family of programming languages, and leave open what kind of data they operate on. The basis is a language of terms in a given over a *data type*, \mathbb{A} , consisting of

- a set (or type) of *values* \mathbb{A}
- a collection of *constants*, c_i , denoting values in \mathbb{A} .
- a collection of *operations*, op_j , denoting functions over \mathbb{A} .
- a collection of questions, $q?_\ell$, that can be asked about elements of \mathbb{A} .

The *arity* of each operation and test is the number of arguments it requires. The language of *terms* consists of constants, variables and applications that respect operations' arities—that is, are well typed. We will call applications of questions *tests*.

$$\begin{aligned}\langle \text{TERM} \rangle &::= \langle c \rangle \\ &::= \langle \text{IDE} \rangle \\ &::= \langle \text{op} \rangle (\langle \text{TERM} \rangle_1 , \dots , \langle \text{TERM} \rangle_k) \\ \langle \text{TEST} \rangle &::= \langle \text{TEST} \rangle (\langle \text{TERM} \rangle_1 , \dots , \langle \text{TERM} \rangle_m)\end{aligned}$$

We want to extend our idea of a data type to include common composite structures like *arrays* and *records*. However, an arbitrary extension can eventually lead to problems, so for the time being let us just allow one-dimensional arrays with a (presumably simple) *index* type. Hide this extension in the definition of program variables by saying that an identifier may be either a simple name or a name applied to an “index calculation,” which for the moment we shall leave unspecified. In our examples, index calculations will all be quite simple.

$$\begin{aligned}\langle \text{IDE} \rangle &::= \langle \text{NAME} \rangle \\ &::= \langle \text{NAME} \rangle [\langle \text{INDEX} \rangle]\end{aligned}$$

Program Statements

This sequential programming languages has all the expressive power needed to describe any mathematical computation. It does not include input-output commands, however, so it cannot describe properties that are important in many applications¹

The sequential programming language of *statements* is given by

$\langle \text{STMT} \rangle ::= \langle \text{IDE} \rangle , \dots , \langle \text{IDE} \rangle := \langle \text{TERM} \rangle , \dots , \langle \text{TERM} \rangle$	Assignment
$::= \mathbf{begin} \langle \text{STMT} \rangle ; \dots ; \langle \text{STMT} \rangle \mathbf{end}$	Compound
$::= \mathbf{if} \langle \text{TEST} \rangle \mathbf{then} \langle \text{STMT} \rangle \mathbf{else} \langle \text{STMT} \rangle$	Conditional
$::= \mathbf{while} \langle \text{TEST} \rangle \mathbf{do} \langle \text{STMT} \rangle$	Loop

This language allows multiple values to be assigned in a single statement, leaving it up to the compiler to determine what order to perform the individual updates. The restriction is that all the results be computed in terms of values in place before any of the assignments were made. We shall defer until later semester specifying rigorously what these programs mean (With no I/O, you might have to think about it.)

Partial Correctness Assertions

Our immediate task will be to take a program, translate it into PVS and then prove something about the resulting model. Later on, we will try to make this a more integrated process. Having developed this much syntax, let us go a little further and introduce a way to inject some logic into our source code. We will do that simply by inserting logical predicates as comments.

To make this formal we would need to develop a grammar for predicate formulas. Let's pretend we've done that and call the resulting language $\langle \text{PRED} \rangle$. Typically, $\langle \text{PRED} \rangle$ may be any quantified logical expression whose free variables are restricted to the program variables in which the formula is embedded. A *partial correctness assertion* has the form:

$$\langle \text{PCA} \rangle ::= \{ \langle \text{PRED} \rangle \} \langle \text{STMT} \rangle \{ \langle \text{PRED} \rangle \}$$

The assertion $\{P\} S \{Q\}$ says, "if *precondition* P holds before statement S executes then *postcondition* Q will hold once S finishes executing.

Sorting Example

The program below is claimed to sort content of an N -element integer array, A . In the correctness assertion a Skolem variable is used to fix the content of A at start time, giving us something to compare A to at program point **HERE**. We often need labels like **SORT** and **HERE** to discuss programs.

¹This is not just a classroom simplification. Research in software verification has yet to looked seriously at I/O.

```

 $N$ : constant of Nat
 $A$ : array[ $N$ ] of Int
 $I, J$ : [ $N$ ] (* [ $N$ ] denoting index set  $\{i \mid 0 < i \leq N\}$  *)

{ $A = A_0$ }

SORT: begin
   $I := 0$ ;
  while  $I < N$  do
    begin
       $J := 0$ ;
      while  $J < N - I$  do
        if  $A[J + 1] < A[J]$ 
          then  $J, A[J], A[J + 1] := J + 1, A[J + 1], A[J]$ 
          else  $J := J + 1$ 
        end;
       $I := I + 1$ 
    end
  end

HERE :
{ $A = \text{sorted}(A_0)$ }

```

Figure 1: A sorting program

A PVS formulation

In a direct PVS formulation, each program statement translates to a tail-recursive function. Since the $\langle\text{STMT}\rangle$ language is “structured” our translation is strictly hierarchical. Were this not the case, we might have problems with mutual recursion. Although these can be dealt with, we don’t have to.

The formal parameters of the functions we build represent the memory of the imperative program. In other words, it is the assignment statements, not the control constructs that correspond to function calls in the PVS model. To begin the translation example, create a PVS THEORY with the appropriate declarations, including a type `IDX` for array indices.

```
translation_example: THEORY
begin

N: nat
IDX: TYPE = {k:nat | k < N}          % see type upto[n] in prelude

A: VAR ARRAY[IDX -> int]
I,J: VAR nat
```

The program `SORT` will be a function, of course. The array that program `SORT` operates on is modeled mathematically as, you guessed it, another function. There can be no concept of an effect; PVS’s `SORT` returns a function whose content models that of program `SORT`’s final array. So, `SORT` will take an index, `I`, and an array, `A`, and return an array representing the result of running the program `SORT`.

Since PVS does not allow forward references, we have to work from the inner loop outward. The inner loop in this example is actually a **for**-loop, and we can handle all the assignment statements with a single tail-recursive call.

```
inner_loop(J, I, A): RECURSIVE ARRAY[IDX -> int] =
  IF J >= N-I
    THEN A
  ELSIF A[J+1] < A[J]
    THEN inner_loop(J+1, I, A WITH [(J) := A[J+1], (J+1) := A[J]])
    ELSE inner_loop(J+1, I, A)
  ENDIF
  MEASURE N-J
```

The outer loop is another **for**-loop and is modelled in the same way.

```
outer_loop(I, A): RECURSIVE ARRAY[IDX -> int] =
  IF I >= N
    THEN A
  ELSE outer_loop(I+1, inner_loop(0, I, A))
  ENDIF
  MEASURE N-I
```

We finish the example by defining a top level expression to model the function and stating a theorem corresponding to the partial correctness assertion in Figure 1

```
sort(A): ARRAY[IDX -> int] = outer_loop(0, A)

sort_is_correct: THEOREM sorted(sort(A))

end translation_example
```

A great deal more could be said about this translation process, not to mention about verifying the model, such as, "What does it mean to be sorted?" The purpose of this illustration was twofold: to show a non-trivial example of the translation process and to show that imperative constructs like assignments and arrays are (eventually) the subject of formal analysis methods.