PROGRAM VERIFICATION I: BASIC IDEAS

CS 340, FALL 2004

PURPOSE

This document provides a short overview of some basic ideas of program verification, including the idea of program *correctness*, the notion of *implementation*, and the use of *trace tables* to extract the function of a piece of code. Several notations are introduced to deal with functions and their restrictions.

CONTENTS

Purpose 1
1. Basic Definitions 1
2. Correctness and Implementation 4
3. Trace Tables 5
4. Conditionals 7

1. BASIC DEFINITIONS

The basic verification problem consists of answering the question of whether a program (in some language) is equivalent (modulo some relation) to a specification (in some form). In order to answer this question, these notions (program, equivalent, specification) must be made sufficiently precise. The verification question is deductive; it will be answered by reasoning about the semantics of the program and its specification. This is a mathematical question; it deals with whether the program is correct *for all* inputs. Testing is a different, empirical matter. A test can only answer the question of whether the program is correct for a given input.

Definition 1.1. A *specification* is a precise (mathematical) description of a problem to be solved.

Any sufficiently precise description of the problem to be solved will necessarily be a mathematical description. A specification might be a black box (a description of a function of the form $S^* \to R$), a state box (a description of a Mealy machine), or a clear box (an algorithm). In this document a specification will be the statement of the *intended (program) function*.

In order to define what a program is, we need to define some additional ideas. Let **Var** denote a (nonempty, finite) set of variable names. Let **Val** denote a (nonempty) set of values.

Definition 1.2. A *state* t is a function of the form $t : \mathbf{Var} \to \mathbf{Val}$. Let the set of all states be **T**.

Date: 5th October 2004.

A state tells you the value of every variable. (This is sometimes called an *interpretation*.) Since a function is a set of ordered pairs, you can think of a state that way. An example state which assigns the value 5 to x, and the value 3 to y is:

$$\{\langle x,5\rangle,\langle y,3\rangle\}.$$

This state can be described by a *concurrent assignment*. This is just an assignment statement in which a vector of variable names appears on the left, and a vector of values appears on the right. Each variable is assigned the corresponding value, and all assignments happen concurrently. When using the concurrent assignment, the special symbol $:=$, pronounced "gets," will be used. The above can be written $x,y := 5,3$. This is just another way to express the same function.

Defining **T** typically involves defining a type system.[1] This is avoided here to keep things (relatively) simple. We'll just say that the initial state $t_0 \in \mathbf{T}$ is given by one or more declarations, and leave it at that.

Definition 1.3. A *declaration d* is a state.

This is somewhat non-intuitive. For example, a C language "declaration" might be as follows:

```
struct {
    char *name;
    unsigned int value;
} name_map;
```

This provides us with the type of the variable `name_map`, but does not provide us with the initial value for the variable. That is, it assigns *some* value, but we don't know which one. Still, after this declaration, `name_map` has an associated value.

There might be a sequence of declarations. We'll disallow multiple declarations for a single variable, and say that the overall effect of a sequence of declarations is the same as their union. For example, consider the following C code.

```
int x = 5;
int y = 3;
```

Applying the above rule gives the single declaration:

$$\{\langle x,5\rangle\}\cup\{\langle y,3\rangle\} = \{\langle x,5\rangle,\langle y,3\rangle\}.$$

Definition 1.4. An *expression e* is a function of the form $e : \mathbf{T} \to \mathbf{Val}$.

An expression provides an interpretation of some state. For example an expression such as $x+y$ could be interpreted for state t as $t(x)+t(y)$. For the example above, this would be $5+3$, or 8. Thus:

$$[x+y]\,(\{\langle x,5\rangle,\langle y,3\rangle\}) = 8.$$

An expression converts a state to some value, which may itself become part of a new state. Expressions are used to create statements.

Definition 1.5. A *statement s* is a function of the form $s : \mathbf{T} \to \mathbf{T}$.

---

[1] For example, you might require that there be a *type* for each variable, given by a function such as:

$$\mathbf{typ} : \mathbf{Var} \to \mathbb{P}\mathbf{Val}.$$

A statement maps one state to another. We can extend concurrent assignments to allow expressions on the right-hand side. For example, the statement $x,y := x+y,y$ would convert the state $\{\langle x,5\rangle,\langle y,3\rangle\}$ to the state $\{\langle x,8\rangle,\langle y,3\rangle\}$. Statements can thus be thought of as defining a sequence of parameterized state transformations.

Definition 1.6. A *program* is a sequence of declarations followed by a sequence of statements.

The declarations define the state and the statements then provide the transformations in state.

Example 1.7. Consider the following short C++ program.

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
    if (argc >= 2) {
        int a = atoi(argv[1]);
        int b = atoi(argv[2]);
        cout << a+b << endl;
    }
    return 0;
}
```

Inside this program is a statement of the form `int a = atoi(argv[1]);`, which contains an expression of the form `atoi(argv[1])`. This expression takes part of the state `argv[1]` and returns a value. The entire expression then modifies the state space by assigning this value to the variable `a`.

The statement `cout << a+b << endl;` does not change the values of any of the variables we've seen (`a`, `b`, `argc`, and `argv` all remain unchanged), but it does modify an output stream by appending something to it. In the world of functional programming (Lisp, Scheme, Haskell, ML) this is sometimes called a *side effect*. Here an output stream is simply included in the state, and it is modified by this statement.

Definition 1.8. For every initial state $a \in \mathbf{T}$ for which program P terminates, a final state $b \in \mathbf{T}$ is determined. The set of all such pairs $\langle a,b\rangle$ determines a function, called the *program function* of P, and denoted $[P]$.

Example 1.9. Consider the following C source code.

```
x = 5;
z = x + y;
```

The state space for this snippet consists of three variables: `x`, `y`, and `z`. The first statement, by itself, maps any state $\langle x,y,z\rangle$ to the new state $\langle 5,y,z\rangle$. Thus the program function for this statement can be written $\langle x,y,z\rangle \mapsto \langle 5,y,z\rangle$, or possibly $[\texttt{x = 5}](x,y,z) = \langle 5,y,z\rangle$.

The second line maps any state $\langle x,y,z\rangle$ to the new state $\langle x,y,x+y\rangle$. Again, this can be written $\langle x,y,z\rangle \mapsto \langle x,y,x+y\rangle$, or possibly $[\texttt{z = x + y}](x,y,z) = \langle x,y,x+y\rangle$.

The overall effect of these two lines can be expressed as:

$$\langle x,y,z\rangle \mapsto \langle 5,y,z\rangle \mapsto \langle 5,y,5+y\rangle,$$

or simply $[\texttt{x = 5; z = x + y}](x,y,z) = \langle 5,y,5+y\rangle$. Using concurrent assignment, this can be written $x,y,z := 5,y,5+y$.

---

2. Correctness and Implementation

The question of whether a program P sufficiently fulfills the constraints imposed by a specification f can be answered in a variety of ways. We can ask if P is *correct* with respect to f, or if P *implements* f. Both *correct* and *implements* are used in the literature. These are very similar notions, but they are used in slightly different ways.

It will be necessary to discuss the *restriction* of a function to a limited domain. This is the same mapping, with elements outside the restriction omitted.

Definition 2.1. Let $f : A \to B$ be a function and C a set. The *restriction* of f to C, denoted $f|C$, is defined as

$$f \cap (C \times B).$$

The function $g = \{\langle 1,2\rangle, \langle 2,4\rangle, \langle 3,6\rangle, \langle 4,8\rangle, \langle 5,10\rangle\}$ can be restricted to the subdomain $\{1,3,5\}$, giving $g|\{1,3,5\} = \{\langle 1,2\rangle, \langle 3,6\rangle, \langle 5,10\rangle\}$.

A domain restriction might also be written as a predicate prior to a mapping rule. The following defines a function which squares only positive integers:

$$f = [a \in \mathbb{Z}^+ \implies \langle a\rangle \mapsto \langle a \times a\rangle].$$

A statement of this form is called a *conditional rule*, and will become important later on. An equally good way to write the above is:

$$f = [a \in \mathbb{Z}^+ \implies a := a \times a].$$

In the following definitions, let $f : A \to B$ be a specification, and let P be a program with program function $[P] : C \to D$.

Definition 2.2. A program P is *partially correct* with respect to specification f, or *partially implements* specification f, iff $A \cap C \neq \emptyset$ and $f|A \cap C = [P]|A \cap C$.

The definition of partial correctness depends on two things. The requirement that $A \cap C \neq \emptyset$ simply states that there are *some* shared input elements; the program is, at least, relevant to the specification. The requirement that $f|A \cap C = [P]|A \cap C$ states that the two functions must agree on the shared part of the domain. That is, $a \in A \cap C$ implies $f(a) = [P](a)$. Another way to write partial correctness is:

$$f(a) = b \wedge [P](a) = c \implies b = c.$$

The program P does some, but perhaps not all, of the required behavior. It might do everything; it might do only a little.

Definition 2.3. A program P is *sufficiently correct* with respect to specification f, or *completely implements* specification f, iff $[P] \cap f = f$.

Another way to write this is $f \subseteq [P]$. That is, a program is sufficiently correct with respect to its specification if it does everything required by the specification, though it may do more. Note that sufficient correctness implies partial correctness. Another way to write sufficient correctness is:

$$f(a) = b \implies [P](a) = b.$$

The program implements everything required by the specification, but might do more.

Definition 2.4. A program P is *completely correct* with respect to specification f iff $[P] = f$.

The equality $[P] = f$ can be re-written as $f \subseteq [P]$ and $[P] \subseteq f$. That is, the program does everything required by the specification (sufficiency), and it does *only* what is required by the specification (necessity). Complete correctness implies sufficient correctness. Another way to write complete correctness is:

$$f(a) = b \iff [P](a) = b.$$

Definition 2.5. A program P *exceeds* specification f iff $C \not\subseteq A$ and P partially implements f.

A program P exceeds its specification when it performs some operations not required by f. The notion of a program exceeding its specification is closely tied to the notion of complete correctness. If a program is completely correct, then the program completely implements, but does not exceed, the specification. This can be a useful security property; the program does not provide any unspecified behavior (such as granting root privileges to unauthorized users).

Example 2.6. Let f be defined by the conditional rule:

$$x \geq 0 \wedge y \geq 0 \implies \langle x, y \rangle \mapsto \langle x + y, 0 \rangle.$$

Let $P1$ be the following C source code.

```
while (y > 0) {
    x = x + 1;
    y = y - 1;
}
```

Is $P1$ correct with respect to f? Examination of the code should convince you that it is *sufficiently correct*, but that it *exceeds* the specification. For example, $P1$ computes values when `x` is negative. A method for proving that $P1$ is sufficiently correct with respect to f will be discussed later.

---

3. Trace Tables

Given a specification f, one can *write* a program P intended to implement (perhaps only partially) the specification. Alternately, given a program P, one can *read* the program to determine $[P]$. This process is called *function abstraction*. Finally, given both f and $[P]$, one can compare these functions. This process if called *verification*.

Every program can be written using only the primes sequence, if-then-else, and while-do. If a program *isn't* constructed using these primes, a function equivalent program can be generated using the constructive proof of the Structure Theorem. The advantage of this is that one need only investigate function abstraction and program verification for just these three structures. The central ideas behind verifying each construct can be summarized in *correctness questions*.

Sequences of statements can be combined by function composition, as seen in example 1.9. The idea is that the sequence $g;h$ computes the function composition $h \circ g$.[2] For example, the sequence $g;h$ is completely correct with respect to specification f iff $f = [g;h] = h \circ g$.

> Sequence Correctness Question: $f = [g;h] = h \circ g$ for complete correctness, and $f \subseteq [g;h] = h \circ g$ for sufficient correctness.

[2]Note that, depending on what textbooks you read, you may see the order reversed. Here $h \circ g = \{\langle a, b \rangle | h(g(a)) = b\}$, and can be read "$h$ of g."

Keeping track of all the parts when computing the composition can be difficult. One method of organizing the analysis is to use a *trace table* which shows the symbolic execution of the code.

To create a trace table, number the lines of the sequence. Each line becomes a row in the trace table. One column is added for each variable present in the state space of the program. Consider the following C source code.

```
❶ x = x+y;
❷ y = x-y;
❸ x = x-y;
```

For this code segment, one constructs the following table.

Part	x	y
❶ `x = x+y;`		
❷ `y = x-y;`		
❸ `x = x-y;`		

Subscripts are used as follows: x_n denotes the value of x immediately following execution of line n, and immediately prior to execution of line $n+1$. Let x_0 denote the value of x prior to executing any code. This leads to the following entries in the table.

Part	x	y
❶ `x = x+y;`	$x_1 = x_0 + y_0$	$y_1 = y_0$
❷ `y = x-y;`	$x_2 = x_1$	$y_2 = x_1 - y_1$
❸ `x = x-y;`	$x_3 = x_2 - y_2$	$y_3 = y_2$

This gives a complete trace table for this sequence. It is now possible to solve for the final value of each variable in terms of the initial values by back substitution. In this case we have the following for x_3 and y_3.

$$\begin{aligned} x_3 &= x_2 - y_2 \\ &= x_1 - (x_1 - y_1) \\ &= x_1 - x_1 + y_1 \\ &= y_1 \\ &= y_0 \end{aligned} \qquad \begin{aligned} y_3 &= y_2 \\ &= x_1 - y_1 \\ &= x_0 + y_0 - y_0 \\ &= x_0 \end{aligned}$$

This gives the program function for the code snippet: $\langle x, y\rangle \mapsto \langle y, x\rangle$. The concurrent assignment form is $x, y := y, x$.

Example 3.1. Consider the following code snippet.

```
{ int z;
  z = x;
  x = y;
  y = z;
}
```

This code obviously implements a swap, and this is easy to prove with trace tables.

Part	x	y	z
❶ `int z;`	$x_1 = x_0$	$y_1 = y_0$	$z_1 = ?$
❷ `z = x;`	$x_2 = x_1$	$y_2 = y_1$	$z_2 = x_1$
❸ `x = y;`	$x_3 = y_2$	$y_3 = y_2$	$z_3 = z_2$
❹ `y = z;`	$x_4 = x_3$	$y_4 = z_3$	$z_4 = z_3$

$$\begin{aligned} x_4 &= x_3 \\ &= y_2 \\ &= y_1 \\ &= y_0 \end{aligned} \qquad \begin{aligned} y_4 &= z_3 \\ &= z_2 \\ &= x_1 \\ &= x_0 \end{aligned} \qquad \begin{aligned} z_4 &= z_3 \\ &= z_2 \\ &= x_1 \\ &= x_0 \end{aligned}$$

Since `z` goes out of scope when one leaves the code segment, it can be discarded. Thus the program function for the code is $\langle x,y\rangle \mapsto \langle y,x\rangle$, as expected.

---

4. CONDITIONALS

The if-then-else structure can be expressed using conditional rules. Consider the following C source code.

```
if (p) {
    g();
} else {
    h();
}
```

Let the program function of this code be f. When `p` evaluates to true, the program function of the code is the same as the program function $[\texttt{g()}] = g$. When `p` evaluates to false, the program function of the code is the same as the program function $[\texttt{h()}] = h$. This can be summarized using domain restriction:

$$f = [p \implies g] \cup [\neg p \implies h].$$

Another way to write this in a more compact form is $f = [p \implies g \,|\, h]$. Here the vertical bar can be read as "else" or "otherwise."

IF-THEN-ELSE CORRECTNESS QUESTION: $f = [\text{if } (p)\ \{g\} \text{ else } \{h\}] = [p \implies g \,|\, h]$ for complete correctness, and $f \subseteq [\text{if } (p)\ \{g\} \text{ else } \{h\}] = [p \implies g \,|\, h]$ for sufficient correctness.

Conditionals can be handled using trace tables, as well. Consider all the predicates in the code segment, and create one trace table for each possible assignment of truth values. For example, if there are two predicates, you need four tables. One table for both true, one table for both false, and two tables for one true and one false. Add a column to the trace table for conditions encountered, and write the condition using the subscripted variables. The condition can then be evaluated by back substitution.

Example 4.1. Consider the following C source code.

```
❶ x = x + y;
❷ y = y - x;
❸ if (x + y > 0) {
❹     y = x + y;
❺ } else {
❻     y = -x - y;
❼ }
```

There are several ways to approach this. The first two lines could be extracted and solved using a trace table, and replaced with a concurrent assignment (using referential transparency and the Axiom of Replacement). For the purposes of illustration, the entire code will be considered in each trace table.

First, consider the *true* case.

Part	Cond	x	y
❶ `x = x + y;`		$x_1 = x_0 + y_0$	$y_1 = y_0$
❷ `y = y - x;`		$x_2 = x_1$	$y_2 = y_1 - x_1$
❸ `if (x + y > 0) {`	$x_2 + y_2 > 0$	$x_3 = x_2$	$y_3 = y_2$
❹ `y = x + y;`		$x_4 = x_3$	$y_4 = x_3 + y_3$

$$\begin{aligned} x_4 &= x_3 \\ &= x_2 \\ &= x_1 \\ &= x_0 + y_0 \end{aligned} \qquad \begin{aligned} y_4 &= x_3 + y_3 \\ &= x_2 + y_2 \\ &= x_1 + y_1 - x_1 \\ &= y_0 \end{aligned} \qquad \begin{aligned} x_2 + y_2 &> 0 \\ x_1 + y_1 - x_1 &> 0 \\ y_0 &> 0 \end{aligned}$$

The program function for the true case can be expressed using domain restriction as the conditional rule:

$$y > 0 \implies x, y := x + y, y.$$

Thus the value of `y` is unchanged by the code fragment, and the condition, which *appears* to depend on both variables actually only depends on `y`.

Next, consider the *false* case.

Part	Cond	x	y
❶ `x = x + y;`		$x_1 = x_0 + y_0$	$y_1 = y_0$
❷ `y = y - x;`		$x_2 = x_1$	$y_2 = y_1 - x_1$
❸ `} else {`	$x_2 + y_2 \leq 0$	$x_3 = x_2$	$y_3 = y_2$
❹ `y = -x - y;`		$x_4 = x_3$	$y_4 = -x_3 - y_3$

$$\begin{aligned} x_4 &= x_3 \\ &= x_2 \\ &= x_1 \\ &= x_0 + y_0 \end{aligned} \qquad \begin{aligned} y_4 &= -x_3 - y_3 \\ &= -x_2 - y_2 \\ &= -x_1 - (y_1 - x_1) \\ &= -y_0 \end{aligned} \qquad \begin{aligned} x_2 + y_2 &\leq 0 \\ x_1 + y_1 - x_1 &\leq 0 \\ y_0 &\leq 0 \end{aligned}$$

The program function for the false case can be expressed using domain restriction as the conditional rule:

$$y \leq 0 \implies x, y := x + y, -y.$$

The program function for the entire code segment can be written using the conditional rule notation as $[y > 0 \implies x, y := x + y, y \mid x, y := x + y, -y]$.