I. PROGRAM VERIFICATION

A perennial problem with all software is getting it to be correct. This is a particularly severe problem with software that is to be reused, because the consequences of any residual errors will be so widespread. Accordingly much software development effort is devoted to such pragmatic error detection and prevention measures as design review, code walk through, independent testing, etc.

In theory, at least, if we have been careful to specify formally what a piece of software is supposed to do, we should actually be able to prove mathematically that it is correct (if it really is). In fact, this proof process turns out to be reasonably straightforward, but tedious, to carry out for normal programs.

It is perhaps easiest to understand the program proof process by looking at a simple example. Suppose that we want to verify a piece of code whose purpose is to reverse a stack *S*.

Assume S_Reversed = Λ **and** |S| ≤ Max_Depth;
Remember
    While Depth_of(S) ≠ 0
        maintaining @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth;
    do
        Pop(Next_Entry, S);
        Push(Next_Entry, S_Reversed);
    end;
    S :=: S_Reversed;
    Confirm S = @S^{Rev} **and** S_Reversed = Λ;
forget;

Here we presuppose that the stacks *S* and *S_Reversed* and the entry *Next_Entry* have been declared to be of the appropriate types. The **Assume** statement tells us what we know about the situation when our code starts, and the **Confirm** statement tells us what should be true when it finishes. The **Remember** part of the **Remember-forget** pair just indicates the point at which *@S* gets its value.

A program with such **Assume** and **Confirm** statements is called an assertive program and, in a case such as this, these assertive statements would typically come from the **requires** and **ensures** clauses of some operation.

The objective of proof process is to establish that a particular assertive program is correct. As in a normal mathematical proof, this is done by developing a sequence of progressively more complex assertions each of whose correctness follows by some obvious principle (proof rule) from the correctness of earlier assertions in the sequence. As a practical matter, it is usually easier to discover such a sequence of assertions by working backwards, starting from the assertive program that we wish to prove correct and discovering simpler assertive programs which, if they could be proved correct, would allow us to deduce the correctness of subsequent assertive programs.

In the case of our example, we might try to reformulate a simpler version of our assertive program by attempting to remove the swap statement *S* :=: *S_Reversed*, but if we did so, we would have to be careful to modify the **Confirm** statement appropriately. The result would be:

Assume S_Reversed = Λ **and** |S| ≤ Max_Depth;
Remember
    While Depth_of(S) ≠ 0
        maintaining @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth;
    do
        Pop(Next_Entry, S);
        Push(Next_Entry, S_Reversed);
    end;
    Confirm S_Reversed = (@S)^{Rev} **and** S = Λ;
forget;

It is clear that if this slightly shorter program is correct, then our original would also be correct. Technically, we would justify that deduction by an application of a general proof rule for swap statements which is written schematically as:

code; **Confirm** Q[x ⇝ y, y ⇝ x]
---
code; x :=: y; **Confirm** Q;

This rule simply says that the correctness of a program which ends with a swap statement *x* :=: *y* followed by a **Confirm** statement for an assertion *Q* follows from the correctness of the shorter assertive program which omits the swap statement but replaces all occurrences in *Q* of *x* by *y* and of *y* by *x*.

There are similar general proof rules covering each different kind of programming statement. The rule for the **If** statement, for example is:

code; **Assume** B; code1; **Confirm** Q;
code; **Assume** ¬ B; code2; **Confirm** Q;
---
code; **If** B **then** code1 **else** code2 **end_if**; **Confirm** Q;

It just says that, in order to prove the correctness of a program whose next to last statement is an **If** statement, you can prove the correctness of two shorter programs, one of which follows the "**then**" branch, while the other follows the "**else**" branch.

The rule for **while** statements is:

code; **Confirm** I;
Assume I **and** B; body; **Confirm** I;
(I **and** ¬ B) ⇒ Q
---
code; **While** B **maintaining** I **do** body **end**; **Confirm** Q;

It says that when the next to last statement is a **While** statement, you need to prove three hypotheses are correct. The first hypothesis says that the proposed loop invariant *I* is true when

the loop is reached. The second hypothesis says that executing the body of the **While** loop does not invalidate the loop invariant *I*. (i.e., *I* is really an invariant for this loop). The third hypothesis guarantees that the post condition *Q* will always be true whenever the loop terminates.

For our example the **While** rule gives us the three hypotheses:
(1) **Assume** S_Reversed = Λ **and** |S| ≤ Max_Depth;
Remember
    Confirm @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth;
forget

(2) **Assume** @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** |S| ≠ 0);
Pop(Next_Entry, S);
Push(Next_Entry, S_Reversed);
Confirm @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth;

(3) (@S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** |S| = 0) ⇒
S_Reversed = (@S)^{Rev} **and** |S| = 0

The third hypothesis is easy to prove correct using an ordinary mathematical proof in string theory. (|S| = 0, so S = Λ, so @S = S_Reversed^{Rev}. S_Reversed = (S_Reversed^{Rev})^{Rev} = (@S)^{Rev}.)

To continue with the first hypothesis, we need the proof rule for the **Remember-forget** construct.

code; **Confirm Remove@**[Q];
---
code; **Remember**; **Confirm** Q; **forget**;

It simply says that, since the **Remember** sets the values of all *@x*'s to agree with the corresponding values of the ordinary variables *x*, we can just treat the *@x*'s as synonyms for *x*'s at this point by using the **Remove@** operator to strip @'s out of *Q*. We are left with:
(1) **Assume** S_Reversed = Λ **and** |S| ≤ Max_Depth;
Confirm S = S_Reversed^{Rev}∘S **and** |S| ≤ Max_Depth;

Now we need the **Assume** rule:

code; **Confirm** P ⇒ Q;
---
code; **Assume** P; **Confirm** Q;

In our case it produces:
(1) **Confirm** (S_Reversed = Λ **and** |S| ≤ Max_Depth) ⇒
S = S_Reversed^{Rev}∘S **and** |S| ≤ Max_Depth;

Technically we now need the **Confirm** rule to get us back into an ordinary mathematical proof mode:

Q

Confirm Q;

This rule just says that, if a program consists only of a **Confirm** statement, then you must prove in ordinary mathematics that its assertion *Q* is valid. In this case we must prove the easy string theory result:
(1) (S_Reversed = Λ **and** |S| ≤ Max_Depth) ⇒ S = $S\_Reversed^{Rev}$∘S **and** |S| ≤ Max_Depth.

The second hypothesis involves us with the pair of rules that cover procedure calls and declarations, and it is probably more logical to start by considering the rule for declarations.

The extra complication here is that procedure declarations occur at a place in the program text which is remote from the various places where the procedure is actually used (i.e. called). The specification for a procedure will be needed in both the places where it is called and where it is declared. We will use a mechanism called a context to link together a procedure's declaration with its various uses. In general, a *context* $\mathscr{C}$ will consist of the specifications (headings) of all of the procedures that are available to a given piece of code, and we will use a "\" symbol to separate it from the code it applies to, that is, we will write "$\mathscr{C}$ \ code".

With this extra mechanism, then, the rule for procedure declaration becomes:

$\mathscr{C}$∪{**Oper** p(x, y); **req** pre; **ens** post;} \ **Assume** pre; **Remember**; body;
Confirm post **and** y = @y; **forget**;
$\mathscr{C}$∪{**Oper** p(x, y); **req** pre; **ens** post;} \ code; **Confirm** Q;

$\mathscr{C}$ \ **Oper** p(**var** x: T; **pres** y: T);
requires pre/x, y\;
ensures post/x, @x, y\;
procedure
body;
end p; code; **Confirm** Q;

The first hypothesis requires that the *p* procedure meet its specification (the postcondition *post*) assuming that its input requirement (the precondition *pre*) is met. The second hypothesis requires that the rest of the program *code* work correctly, assuming, of course, that *p* works properly. Note that the assumption that *p* works properly is also available in the proof of correctness of the body of *p* so that recursive procedures can also be verified.

In our example, we can assume that the procedure declaration rule has already been applied to *push* and *pop* so that their specifications would already be available in the context. Technically, all of the proof rules that we have seen so far should be upgraded so that they carry the context along properly. For example, the swap rule would become:

$\mathscr{C}$ \ code; **Confirm** Q[x ⇝ y, y ⇝ x];

$\mathscr{C}$ \ code; x :=: y; **Confirm** Q;

It should be clear that this poses no problem.

All we need now to complete our example program is the proof rule for procedure calls:

$\mathscr{C}$∪{**Oper** p(**upd** x, **rest** y); **req** pre; **ens** post} \ code;
Confirm pre[x ⇝ a, y ⇝ b] **and** ∀ ?a: T, **if** post[x ⇝ ?a, @x ⇝ a, y ⇝ b], **then** Q[a ⇝ ?a];

$\mathscr{C}$∪{**Oper** p(**var** x, **pres** y); **req** pre; **ens** post} \ code; p(a, b); **Confirm** Q;

The first part of the **Confirm** statement in the hypothesis requires that the precondition *pre* for *p* be met when *p* is called. Note that the formal parameters *x* and *y* are replaced by the actual parameters *a* and *b*. The next part requires that the postcondition *post* tell enough about what *p* does that the conclusion *Q* can be established. The new variable *?a* stands for the value of *a* after the procedure *p* has been executed.

For our example we may assume that our context $\mathscr{C}$ contains the heading:

Oper Push(**upd** E: Entry; **upd** S: Stack);
req |S| < Max_Depth;
ens S = ⟨@E⟩∘@S **and** Entry.**Is_Initial**(E);

Applying the call rule then gives us:

(2) $\mathscr{C}$ \ **Assume** @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** ¬(S = Λ);
Pop(Next_Entry, S);
Confirm |S_Reversed| < Max_Depth **and**
∀ ?Next_Entry: Entry, ∀ ?S_Reversed: Stack,
if ?S_Reversed = ⟨Next_Entry⟩∘S_Reversed **and** Entry.**Is_Initial**(?Next_Entry)
then @S = ?S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth;

Here the final **Confirm** statement simplifies somewhat to:

Confirm |S_Reversed| < Max_Depth **and**
@S = (⟨Next_Entry⟩∘S_Reversed)^{Rev}∘S **and** |@S| ≤ Max_Depth;

Next we need the call rule as applied to *Pop*. From the context we get the heading:

Oper Pop(**rpl** R: Entry; **upd** S: Stack);
req S ≠ Λ;
ens @S = ⟨R⟩∘S;

Our example then becomes:

(2) $\mathscr{C}$ \ **Assume** @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** ¬(S = Λ);
Confirm S ≠ Λ **and** ∀ ?Next_Entry: Entry, ∀ ?S: Stack,
if S = ⟨?Next_Entry⟩∘?S **then** |S_Reversed| < Max_Depth **and**
@S = (⟨?Next_Entry⟩∘S_Reversed)^{Rev}∘?S **and** |@S| ≤ Max_Depth;

An application of the **Assume** rule leaves us to verify:

(2) $\mathscr{C}$ \ **Confirm if** @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** S ≠ Λ **and**
S = ⟨?Next_Entry⟩∘?S, **then** S ≠ Λ **and** |S_Reversed| < Max_Depth **and**
@S = (⟨?Next_Entry⟩∘S_Reversed)^{Rev}∘?S **and** |@S| ≤ Max_Depth;

Simplifying and applying the **Confirm** rule leaves the proposition:

(2) **If** @S = S_Reversed^{Rev}∘S **and** |@S| ≤ Max_Depth **and** S = ⟨?Next_Entry⟩∘?S,
then |S_Reversed| < Max_Depth **and** @S = (⟨?Next_Entry⟩∘S_Reversed)^{Rev}∘?S.

The second conclusion here follows by substituting for *S* in the equation:

@S = S_Reversed^{Rev}∘S
= S_Reversed^{Rev}∘(⟨?Next_Entry⟩∘?S)
= (S_Reversed^{Rev}∘⟨?Next_Entry⟩)∘?S
= (⟨?Next_Entry⟩∘S_Reversed)^{Rev}∘?S

The first conclusion follows by substituting for *@S* and *S* in the inequality:

|@S| ≤ Max_Depth
|S_Reversed^{Rev}∘S| =
|S_Reversed^{Rev}∘(⟨?Next_Entry⟩∘?S)| =
|S_Reversed^{Rev}| + |⟨?Next_Entry⟩∘?S| =
|S_Reversed| + |⟨?Next_Entry⟩| + |?S| =
|S_Reversed| + |⟨?Next_Entry⟩| ≤
|S_Reversed| + 1 =
|S_Reversed| <