NASA Contractor Report 172407

NASA-CR-172407
19850008180

FORMAL VERIFICATION OF MATHEMATICAL SOFTWARE

DAVID SUTHERLAND

ORA
ODYSSEY RESEARCH ASSOCIATES, INC. OF ITHACA
609 W. Clinton St. Ithaca, N.Y. 14850

Contract No. NAS1-17579

MAY 1984

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Table of Contents

Table of Contents

SUMMARY

The purpose of the research described in this report was to investigate methods for formally specifying and verifying the correctness of mathematical software (software which uses floating point numbers and arithmetic). The research carried out consisted primarily of the following activities:

1. Reviewing previous attempts at modelling floating point arithmetic and formally specifying/verifying mathematical software.

2. Formulating a new model of floating point arithmetic, called "the asymptotic paradigm", a language in which properties defined in the model such as "asymptotically close" can be expressed, and a formal logical system to reason about this model. Our present choice of language and logic primitves is tentative. Further experimental verifications need to be explored.

3. Investigating how the classical "Verification Condition Generation Approach" to program verification could be adapted to use the model.

4. Performing a preliminary investigation of how the more innovative "Programming Logic Approach" to program verification could be adapted.

5. Applying the model to verifying several programs under both approaches; the programs chosen were simplified versions of actual mathematical software.

Our new model of floating point computation is both intuitively clear and useful in verifying the programs we have looked at. Actual errors in floating point programs have been discovered. Interestingly, a logical error in an IMSL library routine uncovered by our techniques appears to be corrected by the FORTRAN compiler; running an interpreted BASIC transcription of the program does give bad test results. The building of verifying compilers which correct the logic of programs has always been a goal of program verification but in the present case the compiler's correction (the guard of a loop is changed from an incorrect to a correct form) probably arose from optimization considerations. The relationships between our model of correctness and optimization remains to be investigated. Our model also has proved useful in uncovering new algorithms. Progress has been made towards integrating the new model into automatic verification systems.

The research described in this report has direct relevance to aerospace applications in which correctness of software over the floating point reals is critical.

TABLE OF SYMBOLS

x <= y	x is less than or equal to y
x >= y	x is greater than or equal to y
x ~= y	x is not equal to y
x*y	x times y
x/y	x divided by y
x^y	x to the y power
|x|	absolute value of x
SQRT(x)	the square root of x
R	the real numbers
N	the natural numbers {0,1,2,...}
{p} S {q}	if p is true and S is executed and S terminates, then q is true
p & q	p and q
~p	not p
p -> q	p implies q
p iff q	p if and only if q
p <-> q	p if and only if q (same as p iff q)
all x:s [p(x)]	for all x of sort s, p(x) is true
some x:s [p(x)]	there exists x of sort s such that p(x) is true
I |= F	F is true in interpretation I
Th(I)	the set of all sentences true in interpretation I
Mod(F)	the class of all interpretations in which F is true
Cn(S)	the set of all sentences which are true in all models of S
CR	The cropping function
M-	the negative overflow threshhold

TABLE OF SYMBOLS

M+	the positive overflow threshhold
m-	the negative underflow threshhold
m+	the positive underflow threshhold
e	the relative error bound
MR(x)	x is a machine-representable real number
x ++ y	CR(x + y)
x ** y	CR(x*y)
x -- y	CR(x - y)
x//y	CR(x/y)
x^^y	CR(x^y)
{x1, ..., xn}	the set consisting of x1, ..., xn
{x : p(x)}	the set of all x such that p(x) is true
A - B	the set of all elements of A that are not elements of B (for A and B sets)
A U B	the union of A and B (for A and B sets)
<xi>	the sequence whose ith term is xi
xi --> y	the sequence <xi> converges to y
RR	the sort symbol which stands for the real numbers
NN	the sort symbol which stands for the natural numbers
NNseq	the sort symbol which stands for sequences of natural numbers
RRseq	the sort symbol which stands for sequences of real numbers
Nev	the symbol which stands for the function which takes a sequence of natural numbers <ni> and an integer j and gives nj
Rev	the symbol which stands for the function which takes a sequence of real numbers <ri> and an integer j and gives rj

Chapter 1

Overview

The aim of formal verification is to mathematically prove programs are correct. Logical techniques for carrying out such proofs are first informally developed and then emboddied in automatic verification systems. The latter provide machine support for the often tedious proofs. They keep track of the status of the verification (what has been and what remains to be proved) and aid in the deductions through the use of automatic formula simplification and logical decision procedures for parts of mathematics. For those parts of proofs which can not be machine supplied the verification system acts as a stern, humorous proof checker thus guaranteeing that no step has been omitted from the human supplied proof through negligence.

A prerequisite to proving programs correct is agreement as to how correctness should be expressed. Although we discuss an alternative approach ("The Programming Logic Approach") later we will focus on the classical Pre and Post Condition form of specification. This takes the form of Hoare sentences of the form

$$\{p\}\ S\ \{q\}$$

where S is a section of code and p, q are formulas in a mathematical language (as contrasted with a programming language). An example of such a language is first-order logic which uses the quantifiers "all" and "some" in addition to the Boolean connectives. The formulas p and q contain the variables which occur in S. The meaning of Hoare sentence is

> If the initial values of the variables satisfy p and S terminates then the final values of the variables satisfy q.

Since q may also need to refer to the initial values of the variables they are allowed to occur in q as 'x, 'y, etc. For example, if the specification of S is that it places the exponential of x and y in z and that it not change x or y, where x, y, and z are integer variables, then the correctness condition takes the form

$$\{y >= 0\}\ S\ \{z = 'x \hat{}\ 'y \ \&\ x = 'x \ \&\ y = 'y\}.$$

Here we use ^ for exponentiation and have added the Pre Condition that y be non-negative since we have decided that the program need only be correct on those values. Alternatively we could change the specification by replacing p by "true" (this means we are assuming nothing about the initial values) and replaced "z = 'x ^ 'y" in the Post Condition by

$$'y >= 0 \rightarrow z = 'x \hat{}\ 'y.$$

The two specifications express the same correctness condition but the former is to be preferred since its Pre Condition would be available during the course of the proof of correctness.

In order for the above example to be meaningful one assumes that the programming language does not contain an exponential operator, the program S computes ^ using a loop. This illustrates the need for the mathematical language to have more operators than the programming language. Quantifiers are also useful as in the following Post Condition which says that final value of y is the first prime after the input 'x

$$\text{Prime}(y) \;\&\; {}'x < y \;\&\; \neg\,\text{some } z\ (\ {}'x < z \;\&\; \text{Prime}(z) \;\&\; z < y)$$

where $\neg$ is the negation symbol and Prime(y) itself needs quantifiers

$$y > 1 \;\&\; \text{all } u, v\ (\ y = u * v \rightarrow u = y \text{ or } v = y).$$

The formal verification of mathematical software (software which uses floating point arithmetic) poses special problems not encountered in classical program verifications such as those mentioned above in which only discrete data types such as integers are considered. These problems arise from the differences between the physical representations used in machines and the ideal, mathematical entities they are based on.

When verifying integer arithmetic programs one pretends that the machine integers are exactly the same as the ideal integers.

This pretense is, strictly speaking, not valid, since there are only finitely many machine integers while there are infinitely many ideal integers. The pretense is acceptable, however, for two basic reasons:

1. The machine integer operations are the same as the corresponding ideal operations as long as neither the arguments nor the result are too large. Thus, as long as overflow does not occur, the behavior of a program which uses only integer arithmetic is the same as if it were using ideal integers.

2. The verification of programs which use only integer arithmetic is thought of as applying only when the program runs to termination without an overflow occurring. This often includes most of the uses of the program that the programmer is interested in. Thus by considering overflow as a form of non-termination one can identify the logic of the program with the logic of mathematics.

Because of the first point we are free to use the same symbol in the progamming and mathematical languages for the arithmetical operations so that

$$\{true\}\ z := x * y\ \{z = x * y\}$$

is a valid Hoare sentence. Even if the program overflows it is still a valid Hoare sentence because the latter only specifies

what happens if termination occurs. On the other hand even without overflow the above Hoare sentence is not true if x, y, and z were floating point variables. While it is true that treating integer overflow as non-termination is a kludge it is interesting to note that Ada with its user supplied exception handlers has reopened the question of how to properly write the Post Conditions of integer programs so that all possibilities are specified.

Thus "pretending" that floating point arithmetic is the same as ideal real arithmetic is not acceptable. Floating point operations deviate from ideal real number operations through roundoff and underflow as well as overflow. It is true that floating point operations are the same as the ideal real operations when roundoff, underflow and overflow do not occur, but such situations are infrequent. Thus if we verify mathematical software by "pretending" that the floating point operations are exact, and adopt the convention that the verification does not apply to runs in which roundoff, underflow or overflow occur, then the verification will not apply to most of the runs we are interested in.

For these reasons, we would prefer to verify mathematical software on the basis of a model of floating point operations which is closer to what is actually done in machines. Several

such models have been presented ([1], [2], [3]), along with lists of axioms which they satisfy. Unfortunately, these axioms systems are either too complex, in which case they are difficult to use, or they are simple but too weak to do adequate analysis of software. In addition, verification using these axiom systems usually requires the verifier to formulate and prove elaborate statements about the nature and magnitude of various sources of numerical error thus confusing a logical problem with a numerical analysis problem. While numerical analysis is important we feel that correctness is a separate issue as we will show. Even the above mentioned systems with simple axioms often involve proving theorems which are quite complicated.

The first major problem is how to express the specifications. As we pointed out above z = x * y is not a proper Post Condition for the program fragment z := x*y when x, y, and z are floating point variables. This problem becomes aggravated further when we attempt to use the Verification Condition (VC) approach to program verification on mathematical software. One of the

---

1. Mansfield, R., A Complete Axiomatization of Computer Arithmetic I to appear in the Journal of Mathematics and Computation

2. Holm, John, Floating Point Arithmetic and Program Correctness Proofs, Ph. D. thesis, Department of Computer Science, Cornell University, August 1980

3. Brown, W. S., A Simple but Realistic Model of Floating-Point Computations, Computing Science Techinical Report No. 83, Bell Laboratories, April 1981

difficulties encountered in using the VC approach on integer programs is that the VCs generated for even simple programs are logically complex. This makes it difficult to prove (or even understand) the VCs. What is perhaps even more important, it is difficult to determine what is wrong with a program when a VC is found to be false. This problem becomes even worse when complicated axiom systems like those formulated for floating point arithmetic are used. In addition, just as it is difficult to formulate appropriate specifications for mathematical software, it is also difficult to formulate the appropriate embedded assertions and loop invariants for such software required by the VC approach.

This report addresses the above the above problems in two ways:

1. A new paradigm for modelling and axiomatizing floating point arithmetic, which we will call the asymptotic paradigm, is presented. This paradigm yields a simple, intuitive axiom system which is strong enough to do non-trivial analysis of mathematical software.

2. This paradigm is applied in the context of two different approaches to program verification. One is the VC approach and the other is an alternative approach, called the Programming Logic approach, which is designed to avoid some of the problems which have arisen from the VC approach. Our discussion of the VC approach is more definitive,

reflecting the maturity of the technique; the discussion of the Programming Logic Approach is more tentative.

Chapter 2

The Denotational Semantics of the Asymptotic Paradigm

2.1 Modelling Floating Point Arithmetic: The Cropping Function

Our starting assumption is that the machine implemented floating point operations can be represented as the ideal real number operations followed by rounding. The operation of rounding is modelled by a cropping function, CR, from the real numbers (denoted by $\underline{R}$) to $\underline{R}$. The range of CR represents the machine real numbers, sometimes called the model numbers. This was the approach taken in the Mansfield and Holmes work cited previously and is consistent with the proposed IEEE standard for floating point arithmetic [1].

We will assume CR satisfies the following axioms, hereinafter referred to as "the cropping function axioms":

- Axiom 1: The range of CR is finite.

---

1. A Proposed Standard for Binary Floating Point Artihmetic, Draft 10.0 of IEEE Task P754, Dec. 1982

- Axiom 2: CR(CR(x)) = CR(x)

- Axiom 3: CR(0) = 0

- Axiom 4: [x <= y <= z & CR(x) = CR(z)] -> CR(x) = CR(y)

The first axiom expresses the fact that there are only finitely many machine real numbers. The second axiom says that the result of a rounding operation (i.e. a machine real number) is unaffected by further rounding. Note that the second axiom implies that the range of CR and the set of fixed points of CR are the same. The third axiom says that 0 is a fixed point of CR, i.e. that 0 is a machine real number. The fourth axiom says that if x and z round to the same number and y is between x and z then y rounds to the same number as x and z.

One axiom which was included by Mansfield and Holmes which we do not include is that CR is an odd function, that is, that CR(-x) = -CR(x). We do not want to require that CR be odd, since this would rule out rounding towards plus infinity and rounding towards minus infinity, two rounding modes which the proposed IEEE Standard would require to be supported.

Note that cropping function axioms 2 through 4 are expressed in first order logic, whereas the first is expressed in English. This is because the concept of "finite" cannot be expressed in first order logic without adding concepts from set theory. In order to perform truly formal program verification, we must

eventually express the first axiom more precisely. This will be dealt with later. As usual when stating axioms in first order logic there are implicit universal quantifiers in front of the formulas displayed as Axioms 2 through 4.

The cropping function axioms are consistent with the four rounding modes which the proposed IEEE Standard would require to be supported, namely rounding to the nearest machine real number, rounding towards 0, rounding towards plus infinity and rounding towards minus infinity. They are also consistent with rounding away from zero, a mode which is not mentioned in the proposed IEEE Standard.

At this point we can derive some useful consequences of the above axioms:

- Theorem 1: CR is monotone, i.e. $x \leq y \rightarrow CR(x) \leq CR(y)$

- Theorem 2: There is no machine real between x and CR(x).

The proofs of these statements are in Appendix A. They do not use Axiom 1 and the only facts about the reals which are needed is that <= is a linear order.

Note that the second statement does not imply that there is no machine real that is closer to x than CR(x). Again, we do not wish to require this because the proposed IEEE Standard would require other rounding modes than rounding to the nearest machine real.

The above cropping function axioms capture certain qualitative properties of CR. Other, quantative properties are captured by the error axioms, which are given below. These axioms are expressed in terms of five additional constants, M-, m-, m+, M+ and e. M+ and M- are the positive and negative overflow threshholds respectively; m+ and m- are the positive and negative underflow threshholds respectively. e is the relative error bound.

- Axiom 5: $M- \leq m- < 0 < m+ \leq M+$
- Axiom 6: $[x < 0 \ \& \ CR(x) = x] \rightarrow M- \leq x \leq m-$
- Axiom 7: $[x > 0 \ \& \ CR(x) = x] \rightarrow m+ \leq x \leq M+$
- Axiom 8: $0 \leq e < 1$
- Axiom 9: $[M- \leq x \leq m- \text{ or } m+ \leq x \leq M+] \rightarrow |CR(x) - x| \leq e*|x|$
- Axiom 10: x is an integer & $[M- \leq x \leq m- \text{ or } m+ \leq x \leq M+] \rightarrow CR(x) = x$

The first error axiom just states the signs and the order of the thresholds. The second and third error axioms say that the negative machine reals are bounded by M- and m- and the positive machine reals by m+ and M+. The fourth error axiom gives bounds on e, and the fifth says that e is a bound on the relative error in the cropping function for numbers between the threshholds.

The reason for having separate overflow and underflow threshholds for positive numbers and negative numbers is that it makes it easier to model rounding towards plus infinity and rounding towards minus infinity. These two rounding modes are not symmetric with respect to zero, and so we need to be able to treat the behavior on either side of 0 separately. The last axiom guarantees that integers in range round to themselves. We discovered the need for this axiom only after we began proving programs which had integer literals in the text.

In terms of CR we make the following definitions:

- Definition 1: MR(x) iff CR(x) = x, i.e. x is a machine real.

- Definition 2: x ++ y = CR(x + y).

- Definition 3: x ** y = CR(x * y).

- Definition 4: x -- y = CR(x - y).

- Definition 5: x // y = CR(x / y).

- Definition 6: x ^^ y = CR(x ^ y).

We assume that ++, **, etc. applied to machine reals model the machine operations. Previous axiom systems for floating point arithmetic were stated in terms of ++, **, etc. Unfortunately, these operations satisfy peculiar properties (e.g., ++ is commutative but not associative) so that verification in terms of

them becomes complicated. In particular, the approach through the logic of ++, **, etc. forces the verifier to state and prove complicated error statements. For example, while the Hoare statement

$$\{true\}\ z := x * y\ \{z = 'x\ **\ 'y\}$$

is true where x, y, and z are floating point variables it would take a close analysis of errors to show that

$$\{true\}\ S\ \{z = x\hat{}\hat{}n\}$$

is true where z and x are floating point variables, i and n are integer variables, and S is

```
i := 0;
z := 1.0;
DO WHILE(i < n);
   z := z * x;
   i := i + 1;
   END;.
```

The point is: when does

$$CR(x * x * .... *x) = [CR(x)\ **\ CR(x)]\ ...\ **\ CR(x)$$

where the products on both sides are n-fold? Our use of "*" in the program text for machine multiplication follows normal convention; to be precise we should really use ** (although the use of + between the integer variables is not objectionable) which from now on we will.

Actually are we really interested in Post Conditions which

contain the machine operations ++, **, ^^, etc.? These operations are implementation details and should not enter into formal specifications for Higher Order Language (HOL) programs. But then, what kinds of statements do we want to prove about mathematical software, i.e. what are the appropriate specifications?

In the next section, we begin discussing our solution to this problem. We center our discussion on the asymptotic behavior of a program, that is, the behavior as the precision of the floating point arithmetic used increases. We will be able to show that the above program correctly implements "x^n" because as precision increases the output tends to x^n in the limit. Our logic will enable us to prove this without having to actually carry out the limiting constructions. The latter are in the meta-theory which justify our axioms and need not be understood by the prover (human or machine) although such understanding would often facilitate the finding of proofs.

2.2 Appropriate Specifications for Mathematical Software

2.2.1 A Motivating Example

Suppose we wanted to write a program whose informal specification was "Add up the entries in a one-dimensional array A of machine reals with length 3". We might produce something

like the following:

```
I := 1;
SUM := 0.0;
DO WHILE(I <= 3);
   SUM := SUM ++ A(I);
   I := I + 1;
   END;.
```

Note that we have adopted the convention of using ++, **, etc. in the program text for the floating point opperations. Suppose we now wanted to formally verify that the program was correct. The first thing we would have to do is translate the above informal specification into a formal statement of what should be true when the program terminates. Since the addition in the third assignment statement is machine addition, we cannot expect to have

$$SUM = A(1) + A(2) + A(3) \tag{1}$$

on termination. We could instead say that we want

$$SUM = [A(1) \mathbin{++} A(2)] \mathbin{++} A(3) \tag{2}$$

at termination. Although this statement is true when the program in question terminates, it is not the correct formalization of the informal specification. To see why, imagine that we had written the program so that it added up the entries of A in the opposite order. Such a program would meet the informal spec as much as the above program does, but it would not necessarily meet (2). We could do an error analysis of the program to obtain some kind of specification like

$$|SUM - (A(1)+A(2)+A(3))| <= eps*(A(1)+A(2)+A(3)) \qquad (3)$$

where eps is some expression which depends on error constants associated with the machine's rounding. This has the same problem as the previous proposal: a different program might not meet the same error analysis yet still meet the informal specification. Furthermore, both (2) and (3) are examples of basing the spec on the program rather than vice versa. We need a specification which is independent of the program.

2.2.2 The Asymptotic Concept

Actually, the first of the three answers above is the closest: we wanted the program to give us the sum of the entries of A. We didn't really expect it to give us the exact sum, however, but rather something "close" to the exact sum. What do we mean by "close"? We don't really mean "as close as the machine can get", i.e. we don't mean that SUM should be the closest machine real to the actual sum. Nor should we expect SUM to be CR[A(1) + A(2) + A(3)]. What we really need is some formalization of the concept of "close" and a logic to reason about this concept.

We could take (3) as our definition of "close", but with a pre-determined eps rather than one derived from the program. There are two problems with this:

1. The above program, running on a given machine, might not

meet the condition we set for the eps we chose because the machine's arithmetic is not sufficiently precise. This is a problem with the machine, not the program, and the program really should not be called incorrect.

2. A program might have errors in it and still meet this kind of spec because the magnitude of the error on the particular machine being used was much smaller than the eps we chose. Such an error might suddenly show up if a smaller eps was used.

In the first case, the program fails to meet the spec but would if the machine were more precise. In the second case, the program meets the spec but would fail if the spec were more demanding. What we really want is that for any "degree of precision" in the Post Condition, there is a "sufficiently precise machine" such that the result of running the program on that machine meets the required degree of precision.

Another way of saying this is to say that as the precision of the machine increases, the precision of the result of running the program on that machine increases. It is our point of view that whereas numerical analysis of the program shows <u>how</u> the precision of the result increases as that of the machine increases, a logical analysis of the program can determine <u>that</u> there is such an increase and this is what we shall mean by correctness.

We formulate this concept by considering the asymptotic

behavior of the program over a series of machines of increasing precision. We will require that "correctness" be a limiting concept even though we only intend to execute the program on a single machine with fixed precision. We make these remarks precise by introducing new machines which are the "limits" of sequences of machines of increased precision. These limit machines don't operate over the ideal reals but over non-standard models of the reals. These models contain all the ideal reals together with other numbers which are "infinitesimally" close to the standard, ideal reals. What do these new numbers correspond to? Essentially to a particular sequence of machine approximations of increasing precision. A different sequence converging to the same ideal number would give rise to a different non-standard number. A fixed program P can be run with any of these inputs. Consider such a program P and a mathematical function f from $\underline{R}$ to $\underline{R}$. If it's the case that whenever x and y are infinitesmally close to the standard z we get that P(x) and P(y) are infinitesmally close to f(z) then we can say that P correctly implements f. The Post Condition will have the form {result == f(x)} where == is our symbol for "infinitesimally close".

2.3 Non-Standard Analysis

This section is a brief exposition of the relevant mathematical

notions for understanding non-standard analysis. We assume that the reader is familiar with the terminology of set theory. We begin by reviewing the language and interpretations of many-sorted first order logic.

2.3.1 Many-sorted First Order Logic

A language L of many-sorted first order logic consists of the following:

- A set of sort or type symbols, Sort.
- A set of constant symbols, Con. Each constant symbol c in Con has a sort.
- A set of function symbols, Fun. Each function symbol f in Fun has a signature <s1,.., sn> of sort symbols.
- A set of relation symbols, Rel. Each relation symbol R in Rel has a signature <s1, .., sn> of sort symbols.
- A symbol for the identity relation: =.
- For each sort s there is an infinite list of variables of that sort.
- The symbols for the Boolean connectives: &, or, ->, iff, ~ and the Boolean constants "true" and "false".
- The symbols for the quantifiers: all, some.

- Punctuation marks: ",", "(", ")", etc.

Using this datum one can define the terms and formulas. Each term has a sort. Both syntactic sets are defined recursively.

A variable or a constant is a term of the appropriate sort. If f is a function symbol of signature <s1, .. , sn> and t1, .., tn-1 are terms, ti of sort si, then f(t1, .., tn-1) is a term of sort sn.

If t1 and t2 are terms of the same sort then (t1 = t2) is an atomic formula. If R is a relation symbol of signature <s1, .., sn> and t1, .., tn are terms, ti of sort si, then R(t1, .., tn) is an atomic formula. The Boolean constants are atomic formulas. If F and G are formulas then so are (F & G), (F or G), (F -> G), (F iff G), (~F). If F is a formula and x is a variable then

some x F

all x F

are formulas. For convenience the above is frequently abbreviated and condensed. For example we won't assign sorts to variables but write

some x:s F

where s is a sort symbol and x is an unsorted variable.

The scope of a quantifier is the smallest formula containing it. An occurence of a variable is called bound if it is within the scope of a quantifier on the same variable. Otherwise the occurrence is called free. A formula F without any free occurrences of variables is called a sentence.

An interpretation of I the language consists of a non-empty set I(s) for each sort symbol s; an object I(c) in I(s) for each constant c of sort s; a function I(f) from I(s1) x ... x I(sn-1) to I(sn) for each function symbol f of signature <s1, .., sn>; a relation I(R) which is a subset of I(s1) x .. x I(sn) for each relation symbol R of signature <s1, .., sn>.

Given a sentence F and an interpretation I it is either true or false that F holds under I. We write I |= F if F is true in I. For each interpretation I we define its theory Th(I) to be

$$\{F: I \models F\}$$

] and for every sentence F we define its model class, Mod(F), to be

$$\{I: I \models F\}.$$

] More generally, if K is a class of interpretations of L then Th(K) is the intersection of all the Th(I) such that I is in K, and if S is a set of sentences then Mod(S) is the intersection of all Mod(F) such that F is in S. Th(K) is the set of all sentence true in all structures in K and Mod(S) is the set of all

structures whcih satisfy every sentence in S (i.e., they are the models of S). In terms of these we can define the logical consequence operation, Cn(S),

$$Cn(S) = Th(Mod(S)).$$

Cn(S) is the set of all sentences true in all models of S. Although our definitions are completely semantic and seem to require a great deal of set theory this is not the way mathematicians actually construct Cn(S). If S is the set of axioms for Euclidean geometry one doesn't search through all models to determine whether the Pythagorean theorem is an actual theorem. Instead one proves the latter from the former set of axioms. Fortunately, first-order logic has a complete set of proof rules which can be mechanized. If a machine can be constructed to automatically enumerate the set S then another can be constructed to automatically enumerate Cn(S). Unfortunately, this theoretical result is not often as useful as it sounds since Cn(S) may be listed in no particularly significant order. To get good results one needs interactive theorem provers to guide the generation of Cn(S).

In certain cases, Cn(S) is not only enumerable it is decidable; that is there is a program which when supplied with a sentence F determines in a finite amount of time whether F is in Cn(S). This is true for the theory of real closed fields described below.

Two interpretations, I1 and I2, are called elementary

equivalent if Th(I1) = Th(I2). This means they are indistinguishible as far as the expressive power of the language L. If F is a formula then there is no meaning to I |= F (e.g. if F is

$$x <= y$$

as opposed to something like

$$\text{all } x(\text{some } y\ (x <= y))$$

or

$$\text{all } x\ (\text{all } y\ (x <= y))$$

then it doesn't mean anything to say F is true or false in say the standard structure over the reals although in this case the first sentence above is true and the second false. On the other hand if F has free variables x1, .. , xn and a1, .., an are objects from the underlying set given by I (we are assuming for simplicity a single sort) then

$$I \models F[a1, .., an]$$

does make sense. For example if I and F are structure and formula mentioned previously then

$$I \models F[5,6]$$

is true while

$$I \models F[6,5]$$

is false.

I1 is called a subinterpretation of I2 if I1(s) is a subset of I2(s) for each sort symbol, I1(c) = I2(c) for each constant symbol, I1(f) and I1(R) and the restrictions of I2(f) and I2(R) for each function and relation symbol. FOr example, one usually things of the integers with +, *, 0, 1, and <= as a subinterpretation of the reals with the same operations (this is the basis of the standard overloading of arithmetic symbols.) A stronger relation between I1 and I2 is that of "elementary subsystem" where in addition to being a subinterpretation we have

$$I1 \models F[a1, .., an] \text{ iff } I2 \models F[a1, .., an]$$

] for all formulas F with free variables x1, .., xn and all a1, .., an from the sort sets of I1. This relation implies that I1 and I2 are elementary equivalent but is much stronger.

The basic language which talks about the reals includes the constant symbols 0 and 1, the function symbols +, *, and the relation symbols like <=. One should distinguish these syntactic objects from the actual operations given in an interpretation. Although in practise one tends not to since to do so would require a complicated meta-language. The standard model for this language is the usual interpretation. This language is referred to below as the language of real closed fields. Real closed

fields are special kinds of fields defined in algebra in a purely algebraic way. They state that 0, 1, +, *, <= form an ordered field in which every positive number has a square root and every odd degree polynomial has a zero. It is a classic result of logic that the exact same sentences are true in the standard model as are true in any real closed field, that is all real closed fields are elementary equivalent. Furthermore, the set of first order consequences of the theory of real closed fields is decidable. Real closed fields is an example of a single sorted theory. Adding a predicate N(x) to the language which singles out the integers destroys the decidability of the theory.

2.3.2 Introduction to Non-Standard Analysis

Calculus was developed in the eighteenth century based on the notion of infinitesmals. These were positive entities dx smaller than any actual postive real but not 0. Furthermore, they obeyed the laws of ordinary real arithmetic so that one could carry out ordinary algebraic manipulations like

$$y = x^2$$

$$y + dy = (x + dx)^2$$

$$(x + dx)^2 = x^2 + 2 * x * dx + (dx)^2$$

$$dy = 2 *x * dx + (dx)^2$$

$$dy/dx = 2 * x + dx.$$

In particular the deriviative, dy/dx, was the actual quotient of

two infinitesmals. In terms of our previous discussion we would say that these extended reals formed a real closed field.

Attempts in the nineteenth century to justify working with these extended reals were not successful and a different approach and proof technique in terms of limits was adopted instead (the so-called epsilon/delta method.)

In the early 60's logicians showed how to justify working with actual infinitesmals. This accomplishment consisted of two parts. First, models were constructed of domains containing infinitesmals. The proof of the existence of these models requires non-constructive techniques (the axiom of choice) and as a result although they are conceivable the models are not quite visible. This contrasts with the standard model of the reals which is always identified with the visible continuum. Owing to twentieth-century advancements in basic physics, tangibility and visiblity of models is no longer considered a necessity although it does make a subject less accessible to the non-initiated.

In addition to making models, various axiom systems reflecting how the infinitesmals in these models behave were constructed. The models prove that the axioms are consistent but all proofs using infinitesmals can be carried out completely from the axioms without any concern for the models. Again this is similar to the method used in modern physics. Students are taught how to manipulate the formalisms of quantum mechanics before they learn

(if they ever do) how to construct the underlying Hilbert spaces which justify the formalism. In the case of calculus, freshman textbooks have appeared using these axioms [2]. Freshman do not know enough mathematics (in particular, modern algebra: groups, rings, and fields) to follow the actual existence proof of the models. They just learn how to use the axioms. The axioms are accepted because the notion of infinitesmal is very intuitive (it is used in many older, "non-rigorous" engineering texts) and the student sees that the axioms presented do capture some properties of his intuitive understanding of the infinitesmally small. Furthermore, they rely on their teacher's word that the axioms will be justified in advanced courses.

Our proofs of programs can also rely on such axioms without the need to go through the construction of the models. On the hand, to understand why non-standard analysis is relevant to machine arithmetic one needs to be able to understand these constructions. After the justification is made and accepted one can just work formally using the axioms.

A first approach to building a real closed field with infinitesmals is to consider the set U of all sequences <ai> of reals. If ai is a for all i then <ai> can be identified with a and represents a standard real. Perhaps sequences <ai> of

---

2. Keisler, J., Foundations of Infinitesmal Calculus, Prindle-Weber-Schmidt, 1976

postive reals converging to zero can play the role of infinitesmals? It is not difficult to interpret the constants 0 and 1 and the function symbols + and * in this set. For example,

$$\langle ai \rangle + \langle bi \rangle = \langle ai + bi \rangle$$
$$\langle ai \rangle * \langle bi \rangle = \langle ai * bi \rangle.$$

The resulting structure is a ring but not a field. We use the term "ring" to mean a commutative ring with a 1 distinct from 0. To see why U is not a field consider the elements defined by

$$ai = \text{if } i \text{ is even then } 0 \text{ else } 1/i$$
$$bi = \text{if } i \text{ is even } 1/i \text{ else } 0$$

then $\langle ai \rangle \tilde{} = 0$ and $\langle bi \rangle \tilde{} = 0$ but $\langle ai \rangle * \langle bi \rangle = 0$ which can not happen in any field. In fact $\langle ai \rangle$ and $\langle bi \rangle$ are infinitesmals in our structure, call them dy and dx and it is thus impossible to form the quotient dy/dx.

In the above example one would like it if either $\langle ai \rangle$ or $\langle bi \rangle$ were to be considered 0. But if all sequences converging to 0 were to be considered 0 there would be no infinitesmals! What does it mean that some $\langle ci \rangle$ in U other than 0 is to be considered 0? One way to make this precise is to find an equivalence relation E on U in which $\langle ci \rangle$ and 0 are equivalent and to replace U by the collection U/E of equivalence classes. Such constructions are common in algebra. If the equivalence relation E satisfies the congruence axiom (sometimes called the

"substitution Axiom")

$$x \; E \; y \; \& \; z \; E \; w \rightarrow x + z = y + w \; \& \; x * z = y * w$$

then one can define + and * on the equivalence classes and still have a ring. It is shown in algebra that congruence relations on a ring are in one-to-one correspondence with the ideals of the ring: if E is such a congruence then J(E) = {c : c E 0} is the corresponding ideal and if J is an ideal then E(J) = { (x, y) : x - y in J} is the corresponding congruence relation.

The question thus becomes: find an ideal of U containing dx = <ai> but not containing dy = <bi> (or vice versa). Now U is a collection of real sequences that is functions from the set N of natural numbers to the set R of reals. 0 in U is that function which is always 0. Suppose we relax this condition somewhat and let J be the set of <ai> such that ai is eventually 0. J is an ideal but unfortunately this ideal doesn't solve our paroblem since the dx, dy defined previously are not in our ideal. On the other hand J does suggest an approach namely what makes J an ideal? We can state the definition of J in the following way: Let F be the collection of all cofinite subsets of N (i.e. A is in F iff N - A is finite.) Then J is the set of all <ai> such that {i : ai = 0} is in F.

More generally, let F be a collection of subsets of N and define J(F) to be the set of <ai> such that {i : ai = 0} is in F. What properties must F have to ensure that J(F) is an ideal?

Recall that a non-empty subset J of a ring is an ideal if

I1: a in J -> a * b in J, where b is any ring element

I2: a, b in J -> a + b in J.

Suppose a is in J(F) so that Z(a) = {i : ai in J} is in F. For any b in U, Z(a * b) is a superset of Z(a) so that if F has the property

F1: A in F and A subset B -> B in F

then I1 is true. Now suppose a and b are in J so that Z(a) and Z(b) are in F. Now Z(a + b) is a superset of [Z(a) intersect Z(b)] so if the non-empty F has the property

F2: A and B in F -> (A intersect B) in F

then F1 and F2 imply J(F) is an ideal. We must watch out for one case however. The ideal consisting of the whole ring will collapse everything to 0. An ideal not equal to the whole ring is called proper. The improper ideal is the only ideal containing 1. It is is given by an F containing all the subsets of $\underline{N}$. By F1 F contains all the subsets just when it contains the empty set. Thus we add the condition

F3: F does not contain the empty set.

Non-empty F satisfying F1 and F2 are called filters. If in addition F3 is satisfied the filter is called proper.

We have actually proved the following theorem.

Theorem: Suppose <V, +, *, 0, 1> is any ring and M is any set. Let U be all functions from M to V. Interpret 0 in U to be the constantly 0 function and 1 to be the constantly 1 function. Define + and * in U pointwise, i.e. f + g is that function h such that h(m) = f(m) + g(m). The resulting structure is a ring. Suppose F is a filter on M (i.e. F is a collection of subsets of M satisfying F1 and F2.) If J(F) is defined by

$$\{f : Z(f) \text{ is in } F\}$$

where Z(f) is

$$\{m : f(m) = 0\}$$

then J(F) is an ideal in the ring U. J(F) is a proper ideal if F is a proper filter.

Now when do we get a field? It is a classic theorem of ring theory that a quotient ring is a field exactly when the ideal J is maximal. J is maximal means that it is not contained in any larger proper ideal. There is a corresponding notion for filters: F is maximal if it is not contained in any larger proper filter. If F is a maximal filter will J(F) be a maximal ideal? The problem is J(F) may be contained in some ideal J' not of the form J(F') for some filter F'. (One can easily show that the mapping F --> J(F) from filters to ideals is one-to-one and preserves inclusions; the problem is is it sufficiently onto so

as to preserve maximal objects?) If the original V is a field then the mapping will preserve maximal objects.

Theorem: Suppose <V, +, *, 0, 1> is a field in the above theorem and F is a maximal filter. Then J(F) is a maximal ideal.

Proof: We will use the notation of the previous theorem. What we will first show is:

Lemma: If J' is any proper ideal of U (and V is a field) then there is a proper filter F' such that J' is contained in J(F').

Proof of Lemma: Let J' be a proper ideal of U. We will assume that V is not of characteristic 2 (i.e., $1 + 1 \neq 0$). The Lemma and Theorem are still true in this case but require a separate proof and we are really only interested in the case where V is $\underline{R}$. Let F' be the set of Z(a) for a in J'. F' will be the required proper filter. Suppose B is a superset of some Z(a). Let b be defined by

$$b(m) = \text{if } m \text{ in } B \text{ then } 0 \text{ else } 1.$$

Since J' is an ideal I1 shows that a * b is in J' but it is easy to see that Z(a * b) is B. This proves F1. Now suppose a and b are in J'. We want to show that (Z(a) intersect Z(c)) is in F' to prove F2. What we need is a c with Z(c) equal to (Z(a) intersect Z(b)). Since V is a field we can define

$$a'(i) = \text{if } a(i) = 0 \text{ then } 0 \text{ else } 1/a(i)$$

$$b'(i) = \text{if } b(i) = 0 \text{ then } 0 \text{ else } 1/b(i).$$

Since J' is an ideal a'' = a * a' and b'' = b * b' are in J'. But a'' is 0 on Z(a) and 1 elsewhere and similarly with b''. Let c = a'' + b''. Since 1 + 1 ~= 0 we have that Z(c) is the intersection of Z(a) and Z(b). So F' is a filter. Why is it proper? If the empty set were in F' then it would be Z(a) for some a in J'. Defining a' and a'' = a * a' as before shows that 1 is in J' contradicting the fact that it is a proper ideal. Now we know that J(F') is a proper ideal. But from the definition of F' and J(F') it is easy to see that J(F') contains J'. QED.

Now let us return to the Theorem. Suppose F is a maximal filter and suppose J' is a proper filter containing J(F). Construct F' as in the Lemma. Since J' extends J(F) we have that F is contained in F' (this follows from the definition of J(F) and F'). But F was maximal so F = F'. But by the lemma J(F') = J(F) extends J' which shows that J' = J(F) and J(F) has no proper extension among the proper ideals, i.e. it is maximal. QED.

Thus we see that maximal filters allow us to define extensions of the reals which are fields. What about the order relation which plays such a crucial role in analysis? If we define <= on the ring U which is a product of $\underline{R}$'s by

$$\langle ai \rangle <= \langle bi \rangle \text{ iff } \{i : ai <= bi\} \text{ in } F$$

then the congruence relation, E(F), defined by the proper filter F (this is E(J(F)) using our previous notation) satisfies the substitution axiom

$$a\ E(F)\ b\ \&\ c\ E(F)\ d\ \&\ a <= c \rightarrow b <= d.$$

Thus one can define <= on the quotient ring. Will it linearly order this ring? Not necessarily. It is easy to show that <= on U is a partial order. The problem is dichotomy. Fortunately, everything goes right if F is a maximal filter. To see why we quote without proof the following theorem on filters.

Theorem: Let M be a set and F a proper filter of subsets of M. Then the following are equivalent:

1. F is a maximal proper filter;

2. For all subsets A of M either A or M - A is in F;

3. If (A union B) is in F then either A or B is in F.

In any of these case F is called an ultrafilter.

To apply this theorem given a and b in the quotinet ring (which is a field) let

$$A = \{i : ai <= bi\}$$
$$B = \{i: bi <= ai\}.$$

Now A union B is all of M so it is in F. Since F is an ultrafilter either A is in F or B is; this means either a <= b or

b <= a. In a similar way one shows all the other axioms for linear order. In fact more is true. The quotient structure is a real closed field! To explain why we consider the general ultraproduct construction.

2.3.3 Filters and Ultrafilters

To make our discussion self contained we repeat some of our previous definitions and theorems.

Given a set I, a proper filter over I is a non-empty set F of subsets of I which satisfies the following axioms:

1. If S is an element of F, T is a subset of I and S is a subset of T, then T is an element of F.

2. If S and T are elements of F, then S intersect T is an element of F.

3. The empty set is not an element of F.

Informally, a filter is a collection of "large" subsets of I. If F is the improper filter then all subsets are "large".

An ultrafilter is a proper filter that is not a proper subset of another proper filter, i.e. a maximal filter. Ultrafilters can be characterized axiomatically by adding to the above axioms the axiom

If S is a subset of I, then either

S is an element of F or I - S is an element of F

By an argument which uses the axiom of choice in the form of Zorn's Lemma, every proper filter is a subset of some ultrafilter.

A non-empty collection G of subsets of I is said to have the finite intersection property iff

For every finite subset {S1, S2, ... , Sn} of G,
intersection(S1, S2, ... , Sn) is non-empty

G can be extended to a proper filter iff G has the finite intersection property.

For any i an element of I,

{ S : S is a subset of I and i is an element of S }

is an ultrafilter. Such ultrafilters are called principal ultrafilters. Every ultrafilter over a finite set I is principal. If I is infinite, then

{ S : S is a subset of I and I - S is finite }

is a proper filter. It is called the filter of cofinite sets. Any ultrafilter containing it must be non-principal. Further, if J is an infinite subset of I, then

{ S : S is a subset of I and I - S is infinite } U {J}

has the finite intersection property. Thus, it is a subset of some proper filter, which is a subset of some ultrafilter, and this ultrafilter must be non-principal. Thus, any infinite subset of I is an element of some non-principal ultrafilter.

2.3.4 Ultraproducts and Ultrapowers

Fix a first-order language L which for convenience we assume is single sorted and an index set I. Suppose we have a structure Mi for L for each i in I and a filter F over I. The <u>filtered product</u> of the Mi over F is a structure U for L defined as follows:

1. The universe of U is the set of equivalence classes of elements of the cartesian product of the universes of the Mi's. If <ai> and <bi> are two elements of the cartesian product, they are equivalent iff { i in I : ai = bi } is an element of F. The fact that this is an equivalence relation follows from the fact the F is a filter.

2. If k is a constant symbol of L, and ki is its interpretation in Mi, then the interpretation of k in U is [<ki>] where the square brackets indicate the equivalence class.

3. If f is an n-ary function symbol of L, and fi is its interpretation in Mi, then the interpretation of f in U is a function g such that

$$g([<a_{1i}>],\ldots,[<a_{ni}>]) = [<f(a_{1i},\ldots,a_{ni})>]$$

This can be shown to be well-defined. Well defined means:

$$[<a_{1i}>] = [<b_{1i}>] \;\&\; \ldots \;\&\; [<a_{ni}>] = [<b_{ni}>]$$

$$\rightarrow$$

$$[<f(a_{1i},\ldots,a_{ni})>] = [<f(b_{1i},\ldots,b_{ni})>]$$

4. If p is an n-ary predicate symbol of L, and pi is its interpretation in Mi, then the interpretation of p in U is a predicate q such that

$$q([<a_{1i}>],\ldots,[<a_{ni}>]) \text{ iff}$$

$$\{\, i \text{ in } I : p_i(a_{1i},\ldots,a_{ni}) \,\} \text{ is an element of } F$$

Again, this is well-defined.

The Mi from which U is constructed are called the components of U. If F is an ultrafilter then U is called an ultraproduct. An ultrapower is simply an ultraproduct in which each Mi is the same structure.

2.3.5 Properties of Ultraproducts

If the universe of each Mi is a fixed set S, then we can define a one-one function inj from S into the universe of U by

$$inj(x) = [<x>]$$

(i.e. inj(x) is the equivalence class of the I - tuple

<x,x,...>). The function inj is one-one. If each Mi is the same structure, then inj is a homomorphism of structures, and the elements in the image of inj are called the standard elements of the ultrapower. If F is principal, then inj will be an isomorphism. Otherwise, inj will not be onto, and non-standard elements will exist.

We are now ready to state a very remarkable theorem which is far from obvious.

Fundamental Theorem of Ultraproducts: If A is a sentence of L, A will be true in U iff

{ i in I : A is true in Mi }

is an element of F. If U is an ultrapower then inj is an elementary embedding, that is the image of M under inj is an elementary subsystem of the ultrapower U.

Clearly, if extra constant, function or predicate symbols are added to L, and an interpretation of the symbols is given in Mi for each i in I, these will induce a corresponding interpretation in U.

The above notions and constructions generalize in a completely straightforward way to many-sorted logics. In addition to being able to add extra constants, functions and predicates, extra sorts can be added at will, and an interpretation of each new sort in each Mi will induce a corresponding new sort in U.

2.4 Formalizing the Asymptotic Concept

We formalize the asymptotic paradigm using a certain class of ultraproducts. LO is a language with the following sorts, constants, functions and predicates:

- Four sorts:

    1. RR, whose standard interpretation will be the real numbers;

    2. NN, whose standard interpretation will be the natural numbers (regarded as disjoint from the real numbers rather than as a subset of the reals);

    3. NNseq, whose standard interpretation will be the functions from the natural numbers to the natural numbers;

    4. RRseq, whose standard interpretation will be the functions from the natural numbers into the real numbers;

- The constants, functions and predicates of the language of real closed fields, applied to the sort RR and any other symbols for real objects which we might need;

- The constants, functions and predicates of the language of integer arithmetic, applied to the sort NN and any other integer objects we might need;

- Nev, a binary function of signature <NNSeq, NN, NN> (this represents the function which takes a sequence <ni> and an integer j and returns nj);

- Rev, a binary function of signature <RRSeq, NN, RR> (this represents the function which takes a sequence <ri> and an integer j and returns rj);

- CR, a unary function of signature <RR, RR>;

- M-, m-, m+, M+ and e, constants of sort RR.

We will uniformly abbreviate Nev(s,i) as s(i) and Rev(t,j) as t(j).

Let I be the set of natural numbers. The Mi are obtained as follows: fix sequence <CRi> of functions from $\underline{R}$ to $\underline{R}$, and sequences <Mi->, <mi->, <mi+>, <Mi+> and <ei> of real numbers such that each CRi, Mi-, mi-, mi+, Mi+ and ei satisfy the cropping function and error axioms, and

1. <mi+> and <mi-> both converge to 0;

2. <Mi-> goes to minus infinity and <Mi+> goes to plus infinity;

3. <ei> goes to 0.

(i.e., CRi "converges to perfect precision". The fact that the various sequences satisfy the cropping function and error axioms implies that

CRi(x) goes to x uniformly on bounded closed intervals)

Mi is the structure for LO in which

1. RR is interpreted as $\underline{R}$;

2. NN is interpreted as $\underline{N}$;

3. NNseq is interpreted as the set of all sequences of natural numbers;

4. RRseq is interpreted as the set of all sequences of real numbers;

5. The real closed field symbols and any additional real objects are given their standard interpretations in $\underline{R}$;

6. The integer arithmetic symbols are given their standard interpretations in $\underline{N}$;

7. Nev and Rev are interpreted as indicated above;

8. CR is interpreted as CRi;

9. M-, m-, m+, M+ and e are interpreted as Mi-, mi-, mi+, Mi+ and ei respectively.

Let F be a non-principal ultrafilter over I. The resulting ultraproduct U is a combination of non-standard model of the theory of $\underline{R}$, a non-standard model of the integers $\underline{N}$, a set of "hypersequences" of non-standard integers (i.e., sequences <ni> where i ranges over both standard and non-standard integers) and a set of hypersequences of non-standard real numbers. Call the class of all such ultraproducts obtained by the above construction NSM.

2.4.1 Further Symbols

In this section we define an extension of the language LO. First, as before one can define MR(x), ++, **, --, and // in terms of CR. We also extend the language by adding a unary predicate symbol "std" of signature <RR>. For each U in NSM, interpret std in U as the standard elements of U, that is those x in U of the form inj(y) for y in $\underline{R}$. By an abuse of notation, std will be used for the standard elements of any of the sorts. In addition, add the following defined symbols:

1. fin(x) iff some y:RR [std(y) & |x| <= y] ("x is finite")

2. inf(x) iff ~fin(x) ("x is infinite")

3. diff(x) iff all y:RR [std(y) & y > 0 -> |x| < y] ("x is infinitesimal")

4. x == y iff diff(x - y) ("x is infinitely close to y")

We call the resulting language L. For certain reasons which will become apparent later, we wish to distinguish symbols and formulas which have an interpretation in each Mi (i.e. symbols and formulas of L0) from those which only have an interpretation in the U's (i.e., std and any symbol defined in terms of std). The former symbols and formulas we refer to as internal, and the latter as external.

2.4.2 Axioms

Let NSA be the set of all formulas of L which are valid for every model in NSM. These are what we wish to consider the "asymptotically true formulas". Any sentences which we adopt as axioms for the paradigm must be in NSA. The choice of what axioms to include is largely experimental; we examine what is needed in proofs. The following statements in English summarize the axioms which we have been using to date in verifying floating point programs. This list is somewhat overexhaustive, and will be cut down as much as possible as future experience in using the asymptotic paradigm indicates which are vital and which can be dispensed with.

1. The axioms of real closed fields for RR plus any axiom needed for any additional symbol for a real object (e.g. if we consider the function symbol exp of signature <RR, RR> in the language then we add axioms like

$$\text{all } x, y{:}\underline{R}R(\exp(x + y) = \exp(x) * \exp(y)$$

$$\exp(0) = 1$$

2. Axioms for integer arithmetic over $\underline{N}$.

3. If A(x) is any formula of L with x a variable of type R, the axiom which says that if S = { x in R : std(x) & A(x) } is non-empty and bounded above by a standard real, then S has a least upper bound.

4. The definitions linking defined symbols to the more primitive symbols (e.g. x == y <-> diff(x - y)).

5. The cropping function and error axioms.

6. diff(m-) and diff(m+).

7. inf(M-) and inf(M+).

8. diff(e).

9. fin(x) -> CR(x) == x.

10. Axioms which guarantee the closure of RRSeq, and NNSeq under explicit and recursive definitions.

11. The fact that std in each sort forms an elementary subsystem can be given by an axiom scheme.

Why is fin(x) -> CR(x) == x in NSA? Here is a proof. Fix an ultrafilter F and a sequence <CRi> as above. Pick an element x = [<xi>] from U such that fin(x). This implies that there exists b in $\underline{R}$ such that

$$|x| < inj(b)$$

which means

$$J = \{ i \text{ in } I : |xi| < b \} \text{ is in } F$$

which implies that J is infinite. Therefore, every term of the sequence

$$<xi : i \text{ in } J>$$

is in [-b, b]. Since CRi(x) --> x uniformly on [-b, b], the sequence

$$<CRi(xi) - xi : i \text{ in } J>$$

goes to zero, i.e. for any positive c in $\underline{R}$, there exists n such that

$$i \text{ in } J \text{ and } i > n \rightarrow |CRi(xi) - xi| < c$$

or

$$\{ i \text{ in } I : |CRi(xi) - xi| < c \} \text{ contains } J \text{ intersect } \{n + 1, n + 2, \ldots\}$$

J and {n + 1, n + 2, ...} are both in F, so their intersection is in F, so any set containing their intersection is in F, so

$$\{ i \text{ in } I : |CR(xi) - xi| < c \}$$

is in F. Therefore,

$$|CR(x) - x| < inj(c)$$

in U. Since this was true for all positive c in $\underline{R}$,

$$diff(CR(x) - x)$$

or

$$CR(x) == x \qquad \text{QED.}$$

At this point, we can give a precise statement of the first cropping function axiom. Any linearly ordered set is infinite if and only if there is neither a strictly ascending sequence nor a strictly descending sequence. This is a consequence of Ramsey's theorem. We have found in our proofs that an adequate axiom on MR is:

```
all s:RRseq [all n:NN [MR(s(n)) & s(n + 1)<=s(n)]
  -> some n:NN [all m:NN [n<m  ->  s(n)=s(m)]]]
```

Since this sentence holds in every Mi, it holds in U. The corresponding statement for ascending sequences also holds in U.

We will formalize the first cropping function axiom by the these two statements.

The following is a list of useful theorems which can be deduced from the above axioms.

1. Every standard real is finite.

2. If x is finite, there exists a standard y such that y == x.

3. The finite elements of RR form a convex [3] proper subring of U.

4. fin(x*y) -> fin(x) or fin(y)

5. The inverses of non-zero infinitesimals are infinite.

6. The inverses of infinite numbers are infinitesimal.

7. The infinitesimal elements form an ideal in the finite elements.

8. diff(x*y) -> diff(x) or diff(y)

9. The infinitesimal elements are convex.

10. == is an equivalence relation.

---

3. Given a set S with a partial ordering <= on it, a subset T of S is convex iff all x,y,z (x in T and z in T and x <= y <= z -> y in T)

11. fin(x) -> fin(CR(x))

2.4.3 Rationale

What's "asymptotic" about the above? Suppose we fix a program P which takes a floating point number as input and returns a floating point number as output, and suppose this program always terminates. Denote the function so computed by P. A given CR over R determines which numbers can be inputs to P (namely the fixed points of CR), and also determines what P(a) is for a given input (obtained by executing P on a with floating point operations being the precise operations followed by applying CR). Thus, given a sequence <CRi> as above, we get a sequence <Pi> of functions, with each Pi defined on the fixed points of CRi in range. This sequence of functions induces a single function (call it P) defined on the fixed points of CR in the ultraproduct. This function will have the first-order-expressible properties possessed by "almost all" (i.e. all but finitely many) of the Pi's.

Suppose P was intended to compute the square root of its input. How would we express the specification for a square root program mentioned above? One way to express it might be

$$a >= 0 \ \& \ MR(a) \ \& \ fin(a) \ \rightarrow \ P(a)*P(a) == a$$

Suppose we could prove the above statement about P from axioms in NSA. Now, suppose there was some sequence <CRi> going to

perfect precision, some positive (standard) real b, and some sequence of (standard) reals <ai> such that each ai is in the interval [-b, b], CRi(ai) = ai, and the sequence <Pi(ai)*Pi(ai) - ai> does not converge to zero (in other words, as the precision increases, we can choose machine representable numbers in a fixed bounded interval such that the result of running Pi on ai doesn't get closer and closer to the square root of ai). There exists a positive real number c such that for all i, there exists j > i such that

$$|Pj(aj)*Pj(aj) - aj| > c$$

Thus, the set

$$J = \{j : |Pj(aj)*Pj(aj) - aj| > c\}$$

is infinite. Let F be a non-principal ultrafilter containing J. In the following statements, [<ai>] is denoted by a, inj(b) is denoted (by an admitted abuse of notation) by b, and inj(-b) = -inj(b) by -b. Similarly, inj(c) is denoted by c.

Since

$$\{ i \text{ in } I : CRi(ai) = ai \} = I$$

which is in F,

$$CR(a) = a$$

in the ultraproduct. Since

$$\{ i \text{ in } I : -b <= ai <= b \} = I$$

which is in F,

$$-b <= a <= b$$

or

$$|a| < b$$

in the ultraproduct. Since std(b), this implies fin(a).

By choice of F,

$$\{ i \text{ in } I : |Pi(ai)*Pi(ai) - ai| > c \}$$

is in F, so

$$|P(a)*P(a) - a| > c$$

in the ultraproduct. Since std(c) and c > 0,

$$\tilde{}diff(P(a)*P(a) - a)$$

or

$$\tilde{}(P(a)*P(a) == a)$$

This contradicts our original supposition that we were able to prove P(a)*P(a) == a for all non-negative, finite fixed points of CR. Thus, if we could prove the proposed postcondition for P in our system, it would imply that for any <CRi>, any b and any <ai> as above, Pi(ai)*Pi(ai) - ai --> 0. This is in some sense what we

mean by saying "P computes the square root function asymptotically". Thus the above formalization allows to express specifications of the asymptotic behavior of programs easily and naturally. It also provides us with a natural formalization of the concepts of "large" (i.e. infinite), "small" (i.e. infinitesimal) and "close" (i.e. infinitely close).

2.4.4 Induction and Recursion in Non-Standard Models

We will need to do proofs by induction on N in the course of proving programs, and thus we need to investigate how this is done in a non-standard setting.

The set-theoretic statement of the induction principle ("Every subset of the integers containing 0 and closed under successor is the set of all integers") does not hold for non-standard models of arithmetic (the proper subset consisting of the standard integers violates the principle). The first-order formula statement of the induction principle ("For every first-order formula A(i) where i is a free variable i of sort NN,

A(0) & all i:NN [A(i) -> A(i+1)] -> all i:NN [A(i)]")

also does not hold for an arbitrary formula of L. For example, it does not hold for A = "std(i)". It does, however, hold if A is an internal formula. Why is this restriction sufficient?

If A is internal, then every symbol occurring in A has an

interpretation in each Mi. In each Mi, NN is interpreted as the standard non-negative integers, and so the above formula stating the induction principle for A is true in Mi. Since it is true in every Mi, it is true in U. The preceding argument obviously would not go through if A contained an occurrence of a symbol which has no interpretation in the Mi.

If A(i) is not a formula of LO, the following more limited statement holds:

$$A(0) \;\&\; \text{all } i{:}NN\ [A(i) \rightarrow A(i+1)] \rightarrow$$
$$\text{all } j{:}NN\ [\text{std}(j) \rightarrow A(j)]$$

that is, if A holds for 0, and A(i) implies A(i+1), then A holds for all standard integers. This is true because the set of all standard integers which satisfy A is a subset of the standard integers which contains 0 and is closed under the operation of adding 1. By the principle of set induction, which holds for the standard integers, the set of all standard integers satisfying A must contain all standard integers.

The same general principle also holds for definitions of hypersequences by recursion. That is, a definition which involves some external symbols will only define the hypersequence on the standard integers.

To summarize:

1. Proofs by induction in the non-standard case work just as in the standard case for statements expressible in L0. Proofs by induction of statements not expressible in L0 are more limited.

2. Definitions of hypersequences in the non-standard case, whether by formula or by recursion, work just as in the standard case, but only for definitions expressible in L0.

Chapter 3

The Asymptotic Paradigm and the Verification Condition Approach

3.1 Verification Condition Generation

The classical method of proving programs correct entails the use of two languages: a programming language and an assertion language. The latter is usually an extension of the Boolean expression portion of the former. The basic datum for the Verification Condition Generation (VCG) approach is an asserted program, i.e. a Hoare sentence {p} P {q} with Pre Condtion p and Post Condition q togethr with embedded assertions attached to some of the executable statements in P. The only requirement is that there be an attached assertion within each loop. The basic theorem of Floyd shows how to construct mathematical statements S1, .., Sn with the property that if all the S1 are true then so is the starting Hoare sentence. The generation of the verification conditions (VCs) S1, .., Sn is formal and schematic, that is they only contain the symbols found in the statements of P and don't depend on the meaning of the symbols. Thus one can speak of proving the Hoare sentence for programs which really

can't be executed, such as programs over our non-standard structures in NSM. What one really proves is the Verification Conditions S1, .., Sn from the chosen axioms of NSA. This is a formal exercise. As we have illustrated, the truth of the Hoare sentences over the structures U in NSM implies increasing precision over the really executable domains of actual machine reals.

How do we apply the asymptotic paradigm in the context of the VC method? We simply think of the floating point variables of the program to be verified as ranging over the machine-representable elements of some model in NSM. Since there are certain features of most programming languages which involve interaction between floating point and integer variables (such as rounding a real off to an integer, we should also think of the integer variables as ranging over a non-standard model of the integers. Pre- and postconditions and internal assertions for programs can then be written using external symbols, and the asymptotic axioms can be used to prove VCs.

The above approach can be used to do non-trivial asymptotic analysis of programs. There is a problem with it however. Consider the following asserted program:

```
{TRUE}

X := 1;
DO WHILE (X//2 < X);

   { 0 < X <= 1 & ~diff(X) & MR(X)}

   X := X//2;
   END;
END;

{FALSE}
```

X is a floating point variable.

For any finite machine real input X0 this program halts. Since that is the case the Hoare sentence {TRUE} P {FALSE} is not true. On the other hand, the VCs in the non-standard case are provable! We can prove that the loop invariant is true when the loop is entered, that it is an invariant of the loop, and that the Post-Condition follows from the negation of the loop guard and the invariant. Here are the VCs.

VC 1

TRUE

IMPLIES

$0 < 1 <= 1 \ \& \ \sim diff(1) \ \& \ MR(1)$

VC 2

$0 < X <= 1 \ \& \ \sim diff(X) \ \& \ X//2 < X \ \& \ MR(X)$

IMPLIES

$0 < X//2 <= 1 \ \& \ \neg diff(X//2) \ \& \ MR(X \ // \ 2)$

VC 3

$\neg(X \ //2 < X) \ \& \ \ 0 < X <= 1 \ \& \ \neg diff(X) \ \& \ MR(X)$

IMPLIES

FALSE

The first VC is easily proved, since 1 > 0 is an axiom of ordered fields, <= is reflexive and 1 is standard and thus not infinitesimal. We also see that we need MR(1). Our cropping axioms do not imply this and this is an oversight which we discovered through experimentation. That MR(1) holds follows from our final cropping axiom Axiom 10.

To prove the second VC, assume the hypothesis. First, we will prove ~diff(X//2). Suppose diff(X//2); then X//2 == 0. 0 < X <= 1 implies 0 < X/2 <= 1/2. Therefore, X/2 is finite, so

$$X/2 == CR(X/2) = X//2 == 0$$

so X/2 == 0. But multiplying both sides by 2 (a finite number) gives us X == 0, i.e. diff(X), a contradiction. Therefore, ~diff(X//2).

Next, we want to prove 0 < X//2 <= 1. By the hypothesis, X//2 < X <= 1, so X//2 <= 1. 0 < X implies that 0 < X/2, so by monotonicity of CR,

$$0 = CR(0) <= CR(X/2) = X//2$$

If X//2 = 0, then diff(X//2) which we have already disproved. Therefore, 0 < X//2. Finally one has MR(X // 2) since the X // 2 is CR(X / 2). This finishs the second VC.

To show the third VC we show that the hypthosis

$$\neg(X\ //2 < X)\ \&\ 0 < X <= 1\ \&\ \neg diff(X)\ \&\ MR(X)$$

is always false (so that it implies FALSE). In fact the loop guard (X // 2 < X) will always be true when 0 < X and ¬diff(X). How could X // 2 = CR(X / 2) be >= X? Since X > 0 we have 0 < X/2 < X. Applying CR we get

$$CR(0) = 0 <= X//2 <= CR(X).$$

If X // 2 >= X then

$$X <= X\ //\ 2 <= CR(X)$$

so that applying CR we would get

$$CR(X) <= X\ //2 <= CR(X)$$

which shows that X // 2 is CR(X). But MR(X) so X // 2 is X. But X // 2 == X/2 so X == X/2. Multiplying both sides by 2 and then subtracting X gives X == 0, a contradiction.

What went wrong? The Hoare sentence is proved, yet it is not true in the finite cases. What the proof of VC3 actually shows is that in non-standard models the program does not terminate. But since it doesn't terminate it must be that the sequence of values of X for the successive iterations of the loop form an infinite, strictly descending sequence of machine reals, which violates the first cropping function axiom.

Consider a finite case with a fixed CR over R. By the

monotonicity of CR, we can prove that the sequence of values of X is non-increasing. By the first cropping function axiom, this sequence must eventually reach a fixed point, at which point the loop terminates. This only happens, however, because of rounding error or underflow. At some point, X becomes so small that either a division by 2 causes it to underflow to 0, which is then the fixed point, or X//2 is rounded up to X, in which case that value of X becomes the fixed point. If we take a sequence of cropping functions <CRi> going to perfect precision, we find that it takes more and more iterations of the loop before this happens. Let Ni be the number of iterations it takes for the loop to terminate with cropping function CRi. The corresponding integer [<Ni>] in the ultraproduct is non-standard because Ni goes to infinity. Thus, the loop terminates when executing over the non-standard domain, but only in a non-standard number of steps. Nothing in our naive application of the asymptotic paradigm made allowance for a program to execute for a non-standard number of steps.

We can, however, incorporate the idea of a program executing for a non-standard number of steps into the VC approach. Given a sequence <CRi>, we can imagine running a program P with cropping function CRi for each i. Suppose P contains a floating point variable X. As P runs with CRi, X takes on various machine real values for various numbers of execution steps. This defines a sequence of machine real values, one sequence for each floating

point variable. Likewise we get a sequence of integers for each integer variable, and a sequence of "control points" which define how control passes through P as execution progresses. We get one such collection of sequences for each CRi. These sequences can be combined into a collection of hypersequences in the corresponding ultraproduct. We get a hypersequence of non-standard reals for each floating point variable, a hypersequence of integers (possibly non-standard) for each integer variable, and a hypersequence of control points in P. These hypersequences define the execution of P over the non-standard domain for non-standard numbers of steps.

How does this idea of hypersequences actually enter into the verification of a program in the VC approach? Actually, the impact is relatively minimal. The same verification conditions are generated, and they are proved in the same way as before. There is only one major difference, which occurs in the proof of loop invariants. The proof of a loop invariant is essentially a proof by induction on the number of iterations of the loop. In other words, we are essentially proving

all n:N [the loop invariant is true after n iterations]

by induction on n. However, recall that when performing induction over the non-standard integers, if the statement we are proving is external, the proof only holds for standard integers. This means that if a loop invariant is an external statement, the

usual VC method proof only proves that the loop invariant is true for a standard number of interations. Thus, if we need to use an external invariant, we must also prove that if the loop ever terminates, it terminates in a standard number of steps. Otherwise, the loop may run for a non-standard number of steps and then terminate with the loop invariant false. Notice that this is exactly what happens in the example above: the loop inavriant contains the external symbol "==", and so the invariant is only true for a standard number of iterations. As shown above, the loop does terminate after a non-standard number of steps with the loop invariant false.

The need to prove termination in a standard number of steps for external loop invariants is the only real change that must be made in the VC method in order to apply the asymptotic paradigm. For internal loop invariants, the method works exactly as before. We present an example of an external invariant and a proof of termination in a standard number of steps, in the section 4. In general we will wish to avoid using external loop invariants wherever possible, since our experience in examining programs executing over non-standard domains suggests that such programs rarely execute in a standard number of steps. In some cases we can replace an external invariant by an internal invariant which implies the original invariant. For example, we will often need to show that for every iteration of a loop, certain quantities are finite. We cannot prove this by making it

part of the loop invariant, since "fin" is an external symbol. What we generally do in such cases is to prove the appropriate quantities finite by showing that their values are bounded by certain fixed numbers which are finite. Saying that a number is between two other numbers is an internal statement, and if the bounds are fixed finite numbers, then the quantity they bound is finite by the convexity of the finite numbers.

3.2 Solution of a Differential Equation by Euler's Method

We wish to write and verify a program to compute an approximation for y where

$$\frac{dy}{dx} = y, \qquad y(0) = 1$$

by the Euler method. Consider the following asserted program:

```
{ X > 0  &  fin(X)  &  N > 0  &  fin(N) }

Y := 1;
POS := 0;
I := 0;
DO WHILE(POS < X);

   { POS == (I*X)/N  &  Y == (1 + (X/N))^I  &  I <= N }

   Y := Y**(1 ++ (X//N));
   POS := POS ++ (X//N);
   I := I + 1;
   END;
END;

{ Y == (1 + (X/N))^N }
```

X is the x for which y(x) is being computed. N is the number of steps to be performed in applying the Euler method. Y is the output of the program. POS is the current value of x in the Euler method. I is the number of times the loop has been executed, and is in the program primarily to be used in the loop invariant and in the proof of correctness. Note that we are implicitly assuming that whenever a floating point operation is performed using an integer variable, the value of the integer variable is converted to the corresponding element of R.

Let us first examine the pre- and postconditions. The postcondition simply states that the ouput value should be infinitely close to the exact value given by the Euler method. What this says in terms of asymptotic behavior is that as the precision of the cropping function increases, the output value should converge to the exact Euler method value.

The precondition requires that both X and N be positive and finite. What does the finiteness requirement mean? Recall that "finite" essentially means "bounded by a fixed number as the precision increases." Suppose we increased X without bound as the precision increased. As X becomes larger and larger, the magnitude of the error in computations like X//N also becomes larger. If we increased X fast enough, this might offset the increasing precision of CR, and so the output might not converge to the exact answer. Suppose, on the other hand, we increased N without bound as the precision increased. As N becomes larger

and larger, the number of times the loop iterates increases, and so the cumulative error in the entire computation increases. Again, if N increased fast enough, this could offset the increasing precision of CR. In fact, if we left out these restrictions on X and N we would not be able to prove the program.

The loop invariant says that the current value of POS is infinitely close to I times the step size, and Y is infinitely close to the exact Euler method value after I iterations, and that I is <= the total number of steps to be performed in the Euler method. The first thing to notice about this invariant is that it is not an internal formula because it contains the symbol == which is defined using std. Thus, the standard VC methods will only prove that the loop invariant holds for a standard number of iterations of the loop. We will have to show that the loop terminates in a standard number of steps.

Let us now examine the VCs for this program. There are three of them. The first one says that if the precondition is met, the loop invariant is true when the loop is initially entered. The second says that if the loop invariant is true at the top of the loop and the loop guard is true, then the loop invariant will be true after the loop body is executed. The third VC says that if the loop invariant is true at the top of the loop and the loop guard is false, then the postcondition is true when the program terminates.

VC 1

$$X > 0 \;\&\; \mathrm{fin}(X) \;\&\; N > 0 \;\&\; \mathrm{fin}(N)$$

IMPLIES

$$0 == (0*X)/N \;\&\; 1 == (1 + (X/N))\hat{}0 \;\&\; 0 <= N$$

VC 2

$$POS == (I*X)/N \;\&\; Y == (1 + (X/N))\hat{}I \;\&\; I <= N \;\&\; POS < X$$

IMPLIES

$$POS{+}{+}(X//N) == ((I+1)*X)/N$$

$$\&\; Y{*}{*}(1{+}{+}(X//N)) == (1+(X/N))\hat{}(I+1) \;\&\; I+1 <= N$$

VC 3

$$POS == (I*X)/N \;\&\; Y == (1 + (X/N))\hat{}I \;\&\; I <= N \;\&\; \neg(POS < X)$$

IMPLIES

$$Y == (1 + (X/N))\hat{}N$$

The proof of VC 1 is simple, since the conclusion of the implication simplifies to

$$0 == 0 \quad \& \quad 1 == 1 \quad \& \quad 0 <= N$$

The first two conjunctions are true because == is an equivalence relation. The last is true a fortiori, since 0 < N.

Next, examine VC 3. The proof breaks into two cases, the case when X is infinitesimal and the case when it is not.

If X is infinitesimal, then X == 0. N is finite and so is a standard integer. Therefore, 1/N is a standard, non-zero rational number and so we can multiply both sides of X == 0 to get

$$X/N == 0$$

From this we get

$$1 + X/N == 1$$

We now use the fact that for any standard integer J,

$$(1 + X/N)^J == 1 \qquad (1)$$

The proof of this is in Appendix A.

From the hypothesis of VC 3, we have I <= N, and N is standard,

so I is standard. By applying (1) with J = I and J = N, we get

$$Y == (1 + (X/N))^I == 1 == (1 + (X/N))^N$$

This proves VC 3 in the case where X is infinitesimal.

Now, suppose X is not infinitesimal. We have

$$I <= N \quad \& \quad Y == (1 + (X/N))^I$$

If we can prove that in fact I = N, then the conclusion of VC 3 will be proved.

Suppose I < N. Then (I*X)/N < X. Thus we have that POS >= X but POS is infinitely close to something less than X. This implies that POS == X, and so (I*X)/N == X by the transitivity of ==. We can now multiply both sides of (I*X)/N == X by N/X (note that this is finite because N is finite and X is not infinitesimal) to get I == N. But we assumed I < N, and I and N are both integers, so the difference between them must be at least 1 and so cannot be infinitesimal, a contradiction. Therefore I must be equal to N and VC 3 is proved.

Now examine VC 2. First we prove

$$POS ++ (X//N) == ((I + 1)*X)/N$$

from the hypothesis of the VC. First we prove that all the quantities we need to deal with are finite. I is a positive

integer, I <= N and N is finite, so I is finite. X is finite and N is a positive integer, so X/N is finite. Therefore,

$$X//N = CR(X/N) == X/N$$

and so X//N is finite. I finite and X/N finite implies (I*X)/N is finite. POS is infinitely close to (I*X)/N, so POS is finite.

By adding POS to both sides of X//N == X/N, we get

$$POS + (X//N) == POS + (X/N)$$

The left side is a sum of two finite numbers and so is finite. Therefore,

$$\begin{aligned} POS ++ (X//N) &= CR(POS + (X//N)) \\ &== POS + (X//N) \\ &== POS + (X/N) \\ &== ((I*X)/N) + (X/N) \\ &= ((I + 1)*X)/N \end{aligned}$$

Next, we prove

$$Y**(1 ++ (X//N)) == (1 + (X/N))^(I + 1)$$

X/N and X//N are finite, so 1 + (X/N) and 1 + (X//N) are finite. From this we get

$$
\begin{aligned}
1 \text{ ++ } (X//N) &= CR(1 + (X//N)) \\
&== 1 + (X//N) \\
&== 1 + (X/N)
\end{aligned}
$$

We now use the fact that if Z is a finite floating point number and J is a finite integer, Z^J is finite. The proof is in Appendix A. By this, (1 + (X/N))^I is finite, and Y is infinitely close to it, so Y is finite. Therefore, Y*(1 ++ (X//N)) is finite, and so

$$
\begin{aligned}
Y**(1 \text{ ++ } (X//N)) &= CR(Y*(1 \text{ ++ } (X//N)) \\
&== Y*(1 \text{ ++ } (X//N)) \\
&== Y*(1 + (X/N)) \\
&== (1 + (X/N))^I * (1 + (X/N)) \\
&= (1 + (X/N))^(I + 1)
\end{aligned}
$$

Finally, we wish to prove that I + 1 <= N, i.e. that I < N. Suppose not, i.e. suppose I >= N. We have I <= N, so I = N. By substituting into the other conjuncts of the hypothesis, we get POS == X and Y == (1+(X/N))^N. We also have POS < X. Is this a contradiction? The answer is no. It is possible to have POS < X and at the same time POS == X, as long as POS is only less than X by an infinitesimal amount. Does this mean that the program has an error in it, or do we just need to change our loop invariant to one that will give us provable VCs?

Consider the situation in which I = N, POS == X and POS < X. This occurs when the loop has executed N times, POS has been incremented by X/N each time, but due to rounding errors, POS turns out to be slightly less (i.e. infinitesimally less) than X. In this case, the loop will execute at least once more, which will result in a value for Y that is too large. Thus we see that this program is actually incorrect. The basic problem is that the loop guard cannot be trusted to terminate the loop correctly due to roundoff error in incrementing POS. The easiest way to fix the program is to change the guard to I < N. Having done this, we have no need of the variable POS, since it is not used anywhere but in the loop guard, so we can change the program to

```
{ X > 0  &  fin(X)  &  N > 0  &  fin(N) }
Y := 1;
I := 0;
DO WHILE(I < N);
   { Y == (1 + (X/N))^I  &  I <= N }
   Y := Y**(1 ++ (X//N));
   I := I + 1;
   END;
END;
{ Y == (1 + (X/N))^N }
```

The proofs of the three VCs generated for this program are proved by arguments similar to those above (in fact, the proofs are even easier). Note that for this program, there is no difficulty in proving that the loop terminates after N iterations, since we have I <= N from the invariant and I >= N from the negation of the loop guard.

The loop invariant for the fixed program is still an external

formula, so the proof of the second VC only implies that the loop invariant holds for a standard number of iterations of the loop. We must therefore show that the loop terminates in a standard number of steps. This is easy though. The quantity N - I is an integer which decreases by 1 every time the loop body is executed. Therefore the loop cannot iterate more than N times, and N is standard.

3.3 Finding a Zero of a Continuous Function by Bisection

The second example is also carried out in the VC approach. Suppose we have a continuous function f0 from $\underline{R}$ to $\underline{R}$, and two numbers A and B such that f0(A) and f0(B) are of opposite sign. We know from the Intermediate Value Theorem that f0 must have a zero between A and B. We wish to write and verify a program which finds an approximation to that zero. Before we present a candidate program, let us examine the problem to see how we might write such a program and what its pre- and postconditions should be.

First of all, to use non-standard analysis on a function we must make a non-standard extension of it. Given a non-standard model U from NSM, we can get a non-standard extension of f0 by adding a unary function symbol, say F, to L, and interpreting it as f0 in each component of U. These interpretations will induce

an interpretation of F in U which we will call f. For any x in R,

$$\begin{aligned} f(inj(x)) &= f([<x,x,...>]) \\ &= [<f0(x),f0(x),...>] \\ &= inj(f0(x)) \end{aligned}$$

That is, f is identical with f0 on the standard elements. Thus, in particular, f takes standard elements to standard elements.

Next, note that in general, we will not really be able to compute f, but rather some approximation to f given by a program. Suppose we have a program which computes a function g such that

$$\text{all } x{:}RR\ [fin(x) \rightarrow MR(g(x)) \ \&\ g(x) == f(x)]$$

In other words, g is a "machine version" of f on the finite reals. We will assume that the A and B we have are machine reals, and that g is a sufficiently good approximation to f at A and B that g(A) and g(B) are also of opposite sign (if these two assumptions do not hold we can hardly expect to be able to compute an approximate zero for f0).

How would we go about finding a zero of f0? The usual method is to use some algorithm which generates a number C between A and B which is a "guess" at the zero (we will use bisection). If g(C) = 0, the process terminates. If g(C) is not 0, then it is

either opposite in sign to g(A) or g(B). Whichever of the "old" endpoints has opposite sign, it and C form the endpoints of a new, smaller interval, and the process is repeated with the new endpoints. This process is iterated until either a zero of g is found or the endpoints become "close". In the latter case, either one of the endpoints can be taken as the approximation to the zero.

We can formulate the specifications for the program as the following pre- and postconditions. A0 and B0 are the initial values of the endpoints; to simplify writing, OPP(x,y) will be used as an abbreviation for "x < 0 < y or y < 0 < x".

PRE: fin(A0) & fin(B0) & OPP(g(A0),g(B0))

POST: fin(A) & fin (B) & fin(C)
& [g(C) = 0 or (OPP(g(A),g(B)) & A == B)]

As usual, the finiteness restrictions on the values of A0 and B0 simply signify that we do not expect the program to give us better and better approximations as the precision increases, if it is given larger and larger inputs also. The postcondition simply states that we have either found a zero of g or when the program terminates, the values of g at the endpoints are still of opposite sign and the endpoints are "close" (i.e. infinitely close).

Why does the above postcondition ensure that we have found a

number infinitely close to a zero of the original function f0? To show this, we will make use of the following fact, which is proved in Appendix A:

```
all x:RR [fin(x) ->
some y:RR [std(y) & y == x & f0(y) == g(x)]
```

In other words, for any finite real x, there is a standard real y infinitely close to x such that f0(y) is infinitely close to g(x).

Suppose the program terminates with g(C) = 0. By the above fact, there is a standard D such that D == C and f0(D) == g(C) = 0. But f0(D) is standard, and the only standard infinitesimal number is 0, therefore f0(D) = 0. Thus C is infinitely close to a zero of f0.

Suppose the program terminates with A == B and OPP(g(A),g(B)). Again, applying the above fact we get standard reals D1 and D2 such that D1 == A, D2 == B, f0(D1) == g(A) and f0(D2) == g(B). Since A == B, by transitivity D1 == D2. Since D1 and D2 are standard, D1 = D2. Therefore f0(D1) = f0(D2) == g(B). Thus f0(D1) is a standard real which is infinitely close to two numbers of opposite sign (g(A) and g(B)), and so it must be infinitely close to 0. Since it is standard, it must be 0, and so both A and B are infinitely close to a zero of f0.

How can we code the process described above as a program so

that it will stop if it finds a zero or continue until A and B are infinitely close? We cannot simply test to see if two numbers are infinitely close, because "infinitely close" is an asymptotic property which is not true or false for a given machine or precision. Thus, we must find another way to ensure that if no zero is found, the program will terminate with A and B infinitely close.

Consider the following asserted program. PRE and POST stand for the conditions given above. We have added A0 <= B0 to the precondition just to simplify the formulas slightly:

```
{ PRE & A0 <= B0 }

A := A0;
B := B0;
C := (A ++ B)//2;
DO WHILE(g(C) <> 0 & A < C < B);

   {   A0 <= A <= B0  &  A0 <= B <= B0
    &  OPP(g(A),g(B)) &  C = (A ++ B)//2 }

   IF OPP(g(A),g(C))
      THEN B := C;
      ELSE A := C;
   C := (A ++ B)//2;
   END;
END;

{POST}
```

Note that the loop invariant is an internal statement, and so it need not be proved that the loop terminates in a standard number of steps.

One thing about the above program needs explanation, namely,

why do we have the second conjunct in the loop guard? Since C is always set to the average of A and B, isn't C always between A and B? The answer is no, because roundoff error in computing the average may result in a value for C that is not strictly netween A and B. Of course, such roundoff error will only happen when A and B are very close together. We will show below that since A and B get closer and closer as the loop continues to execute, such a roundoff error eventually must happen. This is the way we ensure that if no zero is found the program will terminate with A == B.

Let us first examine the VCs for this program. There are four of them. The first one says that if the precondition is true initially then the loop invariant will be true when the loop is first entered. The second says that the loop invariant is preserved by the execution of the loop in the case when the THEN branch of the IF THEN ELSE is followed, and the third VC says the same in the case where the ELSE branch is taken. The fourth VC says that if the loop terminates with the loop invariant true then the POST is true.

VC 1

PRE & A0 <= B0

IMPLIES

A0 <= A0 <= B0 & A0 <= B0 <= B0

& OPP(g(A0),g(B0)) & (A ++ B)//2 = (A ++ B)//2

VC 2

A0 <= A <= B0 & A0 <= B <= B0

& OPP(g(A),g(B)) & C = (A ++ B)//2

& g(C) <> 0 & A < C < B

& OPP(g(A),g(C))

IMPLIES

A0 <= A <= B0 & A0 <= C <= B0

& OPP(g(A),g(C)) & (A ++ C)//2 = (A ++ C)//2

VC 3

A0 <= A <= B0 & A0 <= B <= B0

& OPP(g(A),g(B)) & C = (A ++ B)//2

& g(C) <> 0 & A < C < B

& ¯OPP(g(A),g(C))

IMPLIES

A0 <= C <= B0 & A0 <= B <= B0

& OPP(g(C),g(B)) & (C ++ B)//2 = (C ++ B)//2

VC 4

A0 <= A <= B0 & A0 <= B <= B0

& OPP(g(A),g(B)) & C = (A ++ B)//2

& (g(C) = 0 or C <= A or B <= C)

IMPLIES

POST

The proof of the first VC is trivial. The only thing which needs to be proved for the second VC is that if A and B are both between AO and BO and C is strictly between A and B, then C is between AO and BO. This is also trivial.

For the third VC, we need to prove that if g(A) and g(B) are of opposite sign and g(A) and g(C) are not of opposite sign, then g(C) and g(B) are of opposite sign. This is also trivial.

Now examine the forth VC. Assume the hypothesis. We will first prove that A, B and C are finite. AO and BO are finite and A and B are both between AO and BO. Since the finite elements are convex, A and B must be finite. Therefore, A + B is finite, so A ++ B = CR(A + B) is finite. This implies that (A ++ B)/2 is finite, and so C = (A ++ B)//2 = CR((A ++ B)/2) is finite.

If g(C) = 0, the proof is done. Otherwise, we must prove that g(A) and g(B) have opposite sign and A == B. The first is true by hypothesis. Suppose A is not infinitely close to B. By the finiteness statements proved above,

$$
\begin{aligned}
C &= (A \mathbin{++} B)//2 \\
&= CR((A \mathbin{++} B)/2) \\
&== (A \mathbin{++} B)/2 \\
&= CR(A + B)/2 \\
&== (A + B)/2
\end{aligned}
$$

Since g(A) and g(B) are of opposite sign, A is not equal to B. Therefore, (A + B)/2 is strictly between A and B. By hypothesis, C is either <= A or >= B. If C <= A, then C <= A <= (A + B)/2 and C == (A + B)/2, so A == (A + B)/2. Simplifying, we get A == B. The proof is similar in the case where B <= C. This completes the proof of the fourth VC.

How can we be sure this program terminates? Suppose it didn't terminate. Then C is always strictly between A and B when control reaches the top of the loop. At each iteration, either A is set to C, in which case the value of A increases, or B is set to C, in which case the value of B decreases. If the loop never terminates, then either A must increase infinitely often or B must decrease infinitely often (notice that "infinite" here does not mean hyperfinite, but actually hyperinfinite). If A increases infinitely often, then we can define an infinite ascending sequence of machine reals, which contradicts the first cropping function axiom. If B decreases infinitely often, then we can define an infinite descending sequence of machine reals, again a contradiction. Thus the program must terminate.

Notice that this way of ensuring termination is unlike the method usually used for programs of this type. Usually the program terminates when A and B come within a certain fixed (or sometimes user-supplied) distance of each other. When such a fixed distance is used, we cannot expect the results of the program to be closer than that distance to the actual zero. In

the above program, however, the program terminates only when the distance between A and B is very small compared to the precision of the machine's arithmetic. In fact, it only terminates when further iterations would move C further away from the zero. Not only does the above program tend to use all of the precision available on a given machine, the same program run on more and more precise machines will give more and more precise answers. Thus, the asymptotic paradigm is not only a way of analyzing programs, it is also useful for designing programs.

Published versions of the above algorithm actually contain an error! Consider again our program

```
{ PRE & A0 <= B0 }

A := A0;
B := B0;
C := (A ++ B)//2;
DO WHILE(g(C) <> 0 & A < C < B);

   {   A0 <= A <= B0  &  A0 <= B <= B0
    &  OPP(g(A),g(B)) &  C = (A ++ B)//2 }

   IF OPP(g(A),g(C))
      THEN B := C;
      ELSE A := C;
   C := (A ++ B)//2;
   END;
END;

{POST}
```

and the form similar to how it appears in IMSL

```
A := AO;
B := BO;
C := (A ++ B)//2;
DO WHILE(g(C) <> 0 & A < C < B);

   IF g(A) ** g(C) < 0
      THEN B := C;
      ELSE A := C;
   C := (A ++ B)//2;
   END;
END;
```

The program is incorrect since while g(A) * g(C) may be < 0 the machine product may round up to 0 so that g(A) ** g(B) < 0 should not be used in place of OPP(g(A), g(B)). Of course, this program will give a correct answer for many inputs so that testing might not uncover the error. To show the power of our method let us consider whether {PRE} P {POST} is a true Hoare sentence in non-standard universes where P is the above program and we use the same Pre and Post-Conditions as before namely

PRE: fin(AO) & fin(BO) & OPP(g(AO),g(BO))

POST: fin(A) & fin (B) & fin(C)

& [g(C) = 0 or (OPP(g(A),g(B)) & A == B)]

where OPP(x, y) is

$$x < 0 < y \text{ or } y < 0 < x.$$

Let f(x) = x and g(x) = CR(x) be the function which approximates it. Let AO and BO be non-infinitesmal, finite negative and

positive numbers respectively with g(A0) = A0 and g(B0) = B0 such that (A0 ++ B0) // 2 = C0 is a positive infinitesmal (note: g(C0) = C0) with A0 ** C0 = 0. It is possible to get explicit examples of this by choosing the CRi, ei, m+i, m-i, etc. appropriately. Then after the first iteration of the loop we have that A = g(A), B = g(B), and C = g(C) are all positive. Furthermore after each bisection the right hand half is chosen since A ** C >= 0. Furthermore since B0 was non-infinitesmal we always have A < C < B so that the loop never terminates. By the same argument we gave for the original program the loop does terminate in a non-standard number of steps. The Hoare sentence is then false.

Chapter 4

Applying the Asymptotic Paradigm: The Programming Logic Approach

The Verification Condition approach has been used since the late 60's to prove programs. It has generated much criticism since verification environments based on this approach generally lead to low productivity. One of the identified problems is the use of two languages; the programming and the assertional, mathematical. When an unprovable VC in the mathematical language is uncovered the corresponding error in the asserted program must be found. This error is either an error in the logic of the program or an inappropriate embedded assertion. It sometimes difficult to discern which of these alternatives is the case. If the error is in the program's logic the place where that error occurs may not correlate simply to the place where the false VC was generated. When the error is corrected the new asserted program is resubmitted to the VCG and the regenerated VCs must be proved. Slight changes in the program might change the form of several of the formerly provable VCs and these must all be reproved even if the change is slight.

Several approachs alternative to Verification Condition

Generation have been proposed. These attempt to narrow the gap between program and proof in order to avoid the above loop. In this chapter we describe some very tentative work we performed in trying to adopt one of these newer approachs, the programming logic approach, to the asymptotic paradigm.

4.1 The Programming Logic Idea

The underlying philosophy of the programming logic approach is that reasoning and correct programming are the same process. Traditionally these two activities have had their separate languages: reasoning has been done in classical first-order logic, and descriptions of algorithms in the plethora of programming languages. The ultimate goal of the programming logic approach is to find a single formal language which facilitates both logical reasoning and algorithmic description as a single activity. A programming logic is a single language to meet both demands.

Prior to the twentieth century there was less of a distinction between programming and proving since only constructive methods were allowable in proofs. The distinction between constructive and non-constructive proofs is that in the former when one claims that some exists one must actually exhibit it whereas in the latter existence can be shown through indirect means such as

reductio ad absurdum. Let us illustrate this point using the following non-constructive proof.

Theorem: There are two irrational numbers a and b such that a raised to the power of b is rational.

Proof: Consider SQRT(2). We know that SQRT(2) is irrational. Now let x=SQRT(2)^SQRT(2). Is x rational? If it is, then the theorem is proved by letting a = b = SQRT(2). Otherwise, consider x ^ SQRT(2) = SQRT(2) ^ 2 = 2. That is certainly rational, so in this case the theorem is proved by letting a = x and b = SQRT(2). This finishes the proof. In either case we have found a and b satisfying the theorem. QED.

Using contemporary standards of correctness this is a valid proof. But the naive student usually says: Where's the Beef? Where are the a and b that you promised me? In fact, the really hard question is which of the two alternatives in the proof is true (it's the second; a very deep theorem in number theory shows that a^b is transcendental when a and b are algebraic and b is not a rational.)

The above proof is not acceptible from a constructive point of view; indeed it can not even be made in a formal constructive logic. In such constructive logics one can extract from a proof of

all x (some y R(x, y))

a function f given by a term in the language such that

all x R(x, fx)

is also provable. Constructive logics are actually rather close to programming languages. The programming logic approach to verification is to formulate a constructive logic whose terms can be evaluated by an interpreter. The scenario is then the following: If one is required to program a function f with the specification

all x R(x, fx)

one proves the mathematical theorem

all x (some y R(x, y))

in the constructive logic. Since the logic is constructive the proof checker will extract automatically from the proof an algorithm for computing y from x and the interpreter will calculate this algorithm on any input supplied. We thus have a program and a proof of correctness in the same text. The algortihm extracted from the proof can be compiled and stored in a library for future use. On the other hand, the extracted algorithm may not be intelligible to humans. Programming logic enthusiasts hold the tenet that the proof from which the algorithm was extracted is the print form of the algortihm. To ask to look at anything else is like asking to look at binary code. Thus we see that in this approach programming is proving.

ML (which stands for Meta Language) is a programming language designed specifically for the purpose of developing formal systems and formal logics. It is described in the next section. ML will be the language in which we develop the ideas of a programming logic. This is done for three reasons. First, ML is designed for the very activity of developing logics. Second, fixing a particular programming language allows the discussion to present concrete examples. Finally, ML is a language which has more than a few similarities with a programming logic.

After the introduction to ML, the programming logic approach will be presented in four stages:

1. A simple fragment of constructive propositional logic is developed in ML. This is primarily to illustrate how logics are represented in ML.

2. A programming logic for integer arithmetic is described in ML.

3. An interpreter for this logic is developed which defines its semantics as a programming language.

4. Preliminary work towards incorporating the asymptotic paradigm into the programming logic approach is presented.

4.2 The Programming Language ML

) The programming language ML was originally designed by Robin Milner for use as the meta-language in the LCF verification system. The book [1] contains a detailed presentation of one variant of ML. Besides its use in LCF the language is interesting in its own right and is versatile enough to compete with other non-imparative languages like LISP and PROLOG. Although similar to LISP it contains features lacking in the former which are felt by many to be important in a modern programming language; for example, it is strongly typed and has a readable syntax. Since it is primarily a research tool, ML has not been standarized. Here we follow the syntax of the UNIX version of ML written by Luca Cardelli at Bell Laboratories.

The distinguishing characteristics of ML are

- interactive dialog;
- strong type system;
- functional style;
- exception-trap mechanism;

---

1. Gordon, Milner and Wadsworth, Edinburgh LCF, Lecture Notes in Computer Science 78, (Springer-Verlag: Berlin, 1979).

- abstract data type defining mechanism;

- separately compiled modules.

Like LISP, ML is an interactive programming lanugage. An ML session consists of a dialog between the user and the system. The user enters expressions terminated by a semicolon. Typing a carriage return sends the line to the interpreter which keeps accepting input lines until a complete expression is found. The value of the expression and its type is returned by the ML system. ML prompts the user to input an expression with "-", and responds on the following line which begins with ">". Here are some examples:

```
- (3 + 5) * 2;
> 16 : int
- "this is a string";
> "this is a string" : string
- 3,4,5;
> 3,4,5 : int * int * int
- if 1=2 then 3 else 4;
> 4 : int
- [3,true;5,false];  hd [1;2;3;4];
> [(3,true);(5,false)] : (int * bool) list
  1 : int
```

These examples give some idea how integer, boolean and string constants are used in simple expressions. The last two expressions were typed in to the ML system on the same line. Notice that elements of a list are of the same type, are separated by semicolons, and are enclosed in square brackets. The empty list is "[]". If "t" is the type of the elements then "t list" is the type of the list. What then is the type of "[]"?

It is " 'a list" where 'a is a type variable. This illustrates ML's polymorphism discussed in more detail later. Another example of a polymorphic object is "hd" used above for the head of a list. Its type is " 'a list -> 'a".

Elements of tuples like "1, 2, true" are separated by commas. They needn't be enclosed in parantheses. The type of a tuple is the cartesian product of the types of the elements. The cartesian product type operator is "*" in ML so that the former tuple has type int * int * bool. "(1, 2), true" would have type (int * int) * bool which is different.

Next we illustrate how variables are bound to values. At the top level this is done using the keyword "val". but there are various ways of making local bindings as well.

```
- val a = 3;                          { Bind the value 3 to a }
> val a = 3 : int
- val a = 4 and b = 27,true;          { Make two bindings }
> val a = 4 : int
| val b = 27,true : int * bool
- val a,b = 4,(27,true);
> val a = 4 : int
| val b = 27,true : int * true
- a + 6 where val a = 5 end;  a;   { N.B. there will
                                     be no global change in a }
> 11 : int
  4 : int
- let val a = 5 in a+6 end;
> 11 : int
```

The "let" and "where" constructs accomplish the same purpose--abbreviating a local value. Such a local binding has no effect on the global value of the variable. Notice that comments

are enclosed in curly braces.

ML is a functional programming language. Functions in ML are created using the keyword "fun". (LISP uses "LAMBDA" for the same purpose.) Functions are first class objects; they can be arguments to other functions and can be returned as values of functions. Functions can be bound to variables at the top level just like any other value: "val f = (fun x. b). Here b is some expression usually contain x. This way of defining the function f is given an alternate syntax keeping with the customary way of writing definitions: "val f(x) = b". This is completely equivalent, it only binds f using x as means of expressing the return value as a function of the input.

```
- val f (n) = n + 1;  f(2);
> val f : int -> int
  3 : int
- val g (x,y) = (x+y) div 2;  g(f(7),2+2);
> val g : (int * int) -> int
  6 : int
- val rec fact (n) = if n=0 then 1 else n*fact(n-1);
> val fact : int -> int
- val f (n: int) = g (fun x.n)  where val g (h) (n) = h(n) end;
> val f : int -> (int -> int)
- f(2)(3);
> 2:int
- val g (x, y) = f(x)(y);
> val g :int * int -> int
- g(2, 3);
> 2:int
```

The alert reader will notice that the ML interpreter not only evaluates expressions but assigns types as well. It does this usin an internal pattern matching algorithm. Consider the definition of fact given above. ML decides on the basis of the right hand expression that fact is of type int -> int (note that the variable n is not declared to be of type int in this definition of fact. The user can give type restrictions as in the binding of f where n is declared to be of type int. Note that ML will figure out that g in the local binding on this line is of type (int -> int) -> (int -> int). If one had written

```
- val f (n) = g (fun x.n) where val g (h) (n) = h(n) end;
```

then ML would return

```
> val f : 'a -> ('b -> 'a)
```

where 'a, 'b are type variables. Such type polymorphism is a unique feature of ML. Where the type of an argument does not matter, it need not be specified. Hence one need not define an identity function exclusively for integers and one for string. One identity function will do--one for any type.

```
- val f (x) = x;
> val f : 'a -> 'a
- val swap (x,y) = (y,x);
> val swap : ('a * 'b) -> ('b * 'a)
- val comp (f,g) (x) = f (g (x));
> val comp : (('a -> 'b) * ('c -> 'a)) -> ('c -> 'b)
```

Type variables always begin with a single quote in ML.

Expressions in ML can raise exceptions and then trap them. This is similar to the catch and throw mechanism in LISP. A function, instead of returning a value, may signal some abnormal condition. For example, the built-in division function signals division by zero whenever the divisor is zero. Further execution halts and ML prints a message to the user at the top level.

```
- 1 div 0;
> Exception: div
- val f (n) = if n<0 then escape "Neg arg" else fact (n);
> val f : int -> int
- f(3); f(-3);
> 6 : int
  Exception: "Neg arg"
```

Exceptions can be trapped before they reach the top level. This permits the computation of an alternate value for an expression should it raise an exception. The syntax of the trapping mechanism calls for a question mark after the expression that may signal an exception and before the alternate expression to be evaluated in the event an exception is signaled.

```
- (1 div 0) ? 45;
> 45 : int
- val g (n) = f(n) ? f(-n);  { this uses definition of f above }
> val g : int -> int
- g(3);  g(-3);
> 6 : int
  6 : int
```

ML has the capability to define new types. This can be done in two ways. The functions that construct the elements of a new type can be specified. This is a concrete type. An abstract type is defined by giving the constructors as well, but then all

the functions that will ever make use of the constructors must be defined on the spot and explicitly exported out of the type definition. The constructors are not available outside the scope of the type definition. This ensures controlled access to the type. We first give examples of a concrete types.

```
- type Color = Red | Blue | Yellow;
> con Red : Color
| con Blue : Color
| con Yellow : Color

- type rec Tree = Leaf of int | Node of Int * Tree * Tree;
> con Leaf : int -> Tree
| con Node : (int * Tree * Tree) -> Tree
```

Tree is an example of a concrete recursively defined type. This concept encapsulates the use of pointers which are not otherwise availble to the user. The functions "Leaf" and "Node" are constructors of type "Tree", since through them elements of type "Tree" are created. The tree consisting of a single leaf is created by the function "Leaf". This is the only way to create a tree without having already made other trees. "Red", "Blue" and "Yellow" are constructors as well; they require no arguments. The type "Color" would be called an enumerated type in Pascal.

An important part of the ML language is pattern matching. This is often used in conjunction with the case statement to break apart the structure of a type. An element of a type is taken apart in the case statement. It is matched against the pattern consisting of variables and constructors in each branch of the case statement. This is how destructuring is accomplished and

explains the lack of "destructors" or "selectors" in the language. We give several examples of this using the definition of the type "Tree" above.

```
- val f (t: Tree): int =                    { A silly example. }
    case t of
        Leaf n  .   n   |                          { A leaf }
        Node (n, Leaf _, Leaf _).  n     |         { A tree with two leaves }
        Node (n, Node _, Leaf _).  n     |         { A skewed tree }
        Node (n, _, _). n;                         { Everything else }
> f : Tree -> int

- val rec EqTree (t: Tree, s: Tree): bool =
    case (t,s) of
      (Leaf n, Leaf m)                        . n=m |
      (Node (n,t1,t2), Node (m,s1,s2).
            if n=m
              then if EqTree (t1,s2) then EqTree (t2,s2) else false
              else false     |
      (_,_) . false;
> val EqTree : Tree * Tree -> bool
```

The principle features to notice about pattern matching are that variables are bound and that "_" is the wildcard pattern matching any pattern.

The next example is an abstract type definition of a tree. The constructors "Leaf" and "Node" will not be available outside the scope of the abstract type definition.

```
- abstype rec Tree = Leaf of int | Node of int * Tree * Tree
  with val MakeLeaf (n) = Leaf n;
  val MakeNode (n,t1,t2) = Node (n,t1,t2);
  val Label (t) =
    case t of
      Leaf n. n |
      Node (n,t1,t2). n;
  val RightSubTree (t) =
    case t of
      Leaf n. escape "Leaf" |
      Node (n,t1,t2). t2
```

```
  end;
> abstype Tree
| val MakeLeaf : int -> Tree
| val MakeNode : (int * Tree * Tree) -> Tree
| val Label : Tree -> int
| val RightSubTree : Tree -> Tree
```

The scope of the abstract type definition extends from the ML keyword "with" to the closing keyword "end". Within this scope the constructors "Leaf" and "Node" are available and have been used to define other functions to make elements of type "Tree" and, with the help of the case statement, to take apart elements of type "Tree".

Defining a type abstractly and exporting only certain functions of the constructors is useful when one is interested in certain subtypes. For example consider balanced trees:

```
- abstype rec BalTree = Empty | Leaf of int |
                        Node of int * BalTree * BalTree
       with
            val Null = Empty;
            val MakeLeaf = Leaf;
            val MakeNode (n, t1, t2) = if height t1 = height t2 then
                            Node (n, t1, t2) else escape "Not Balanced"
                        where val rec height(t)
                          case t of
                              Empty.0 |
                              Leaf(_).1 |
                              Node(_, t1, t2). max(height(t1), height(t2);
                        end;
       end;
```

The Null, MakeLeaf, and MakeNode functions which are exported from this abstract type definition will only permit the user to construct balanced trees. Note that MakeNode will allow the user to make an unblanced tree from t1 and t2 if either one of these

were already unbalanced (since only their heights are checked not their individual symmetry) but that means the user could make an unbalanced tree if he already had an unbalanced tree. Since Null and MakeLeaf give only balanced trees we see that it is true that the user can make only balanced trees.

4.3 Representing a Logic in ML

The following ML program implements a portion of propositional logic. We assume the type Proposition has already been defined in ML. One way to do this is to introduce Proposition as an ML type using a constructor which turns identifiers into propositions:

```
-type Proposition = PropCon of string;
> type Proposition = PropCon of string
| con PropCon : string -> Proposition
```

We first define what the formulas shall be. Formulas are a data type in ML, and their definition follows the usual one in mathematical logic: Every proposition is an atomic formula, and a pair of formulas can be made into another formula using the implication connective. Of course, we might be interested in other connectives or at least in a formula to represent falsehood, but implication shall suffice for this example (although the resulting logic is not complete).

```
- type rec Formula =
    AtomicFormula of Proposition |
    Imply of Formula * Formula;
{ A Proposition is a Formula; out of 2 formulas, make implication.}

> type Formula = ...
| con AtomicFormula : Proposition -> Formula
| con Imply : (Formula * Formula) -> Formula
```

Next we need some functions to manipulate formulas: to extract the hypothesis and conclusion subformulas from an implication and a function to test for syntactic equality.

```
- val Hypothesis (f) =                { Get hypothesis of implication. }
    case f of
      AtomicFormula _      . escape "Not implication" |
      Implication (f1,f2). f1;

> Hypothesis : Formula -> Formula

- val Conclusion (f) =                { Get conclusion of implication. }
    case f of
      AtomicFormula _      . escape "Not implication" |
      Implication (f1,f2). f2;

> Conclusion : Formula -> Formula

- val rec EqFormula (f1, f2) =    { Syntactic equality of formulas. }
  case (f1,f2) of
    AtomicFormula a, AtomicFormula b .  a=b |
    Implication (h1,c1), Implication (h2,c2) .
        if EqFormula (h1,h2) then EqFormula (c1,c2) else false  |
    (_,_) . false;

> EqFormula : (Formula * Formula) -> bool
```

Finally we define the calculus of propositional logic by defining an abstract type representing proofs. An element of type theorem can be constructed only as an instance of one of the two axioms or as a result of modus ponens applied to theorems. The two axiom schemes (written using conventional notation are

$$K: p \rightarrow (p \rightarrow q)$$

$$S: (p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$$

where p, q, and r are any formulas. Modus ponens yields q when applied to p -> q and p.

In order to actually give the ML definition of the abstract data type representing proofs, a useful auxiliary function must defined along the way. This function ``ProofOf'' takes an element of type theorem and returns the formula of which the theorem is a proof.

```
- abstype rec Thm =
    AK of Formula * Formula |                          { K axiom is represented. }
    AS of Formula * Formula * Formula |  { S axiom is represented. }
    MP of Thm * Thm                          { modus ponens is represented. }
  with
    val AxiomK(p,q)=AK(p,q); val AxiomS(p,q,r)=AS(p,q,r);
    val rec ProofOf thm =
      case thm of
        AK(p,q).   Imply (p, Imply (q,p))                    |
        AS(p,q,r).
           Imply(Imply(p,Imply (q,r)), Imply(Imply(p,q), Imply(p,r)))|
        MP(t1,t2). Conclusion (ProofOf t1);

    val ModusPonens (t1,t2) =
      if EqFormula (Hypothsis (ProofOf t1), ProofOf t2)
        then MP(t1,t2) else escape "Fail"
  end;

> abstype Thm
| val AxiomK : (Formula * Formula) -> Thm
| val AxiomS : (Formula * Formula * Formula) -> Thm
| val ProofOf : Thm -> Formula
| val ModusPonens : (Thm * Thm) -> Thm
```

This completes a formalization of a fragment of propositional logic. An element of the ML data type ``Formula'' represents a formula in propositional logic. An element of ``Thm'' actually represents a proof--a bonafide proof. By virtue of the fact that ML certifies an element is of type ``Thm'', the element represents a proof in the propositional calculus, because the only way such an element can be produced is to use one of the three constructors. Each constructor represents a valid step proof in the propositional logic. An element of type ``Thm'' can be created no other way. A study of the above example will show that one is relying on the ML type encapsulating mechanism with its control of exported constructors to simplify the construction of the logic. For example, the only objects prex of type Thm

which the user can construct really are proofs of ProofOf (prex) because of the exception raised when ModusPonens is applied to arguments not of the appropriate form.

Using the data type definition given in the previous paragraph, we can now prove that a proposition P implies itself. We can be sure that it is a theorem of the propositional calculus because it is an element of the ML type ``Thm''.

The usual way P implies P is proved from these axioms is by the following informal proof

```
1. [p -> [(p ->p) -> p]] -> [[p -> (p -> p)] -> (p -> p)]   Axiom S
2. p -> [(p ->p) -> p]                                      Axiom K
3. [[p -> (p -> p)] -> (p -> p)]                            Modus ponens 1. and 2.
4. p -> p                                                   Modus Ponens 3. and 2.
```

Formally we make the following bindings in ML.

```
- val prex1 = AxiomK (P,Imply(P, P));
    { P->[(P->P)->P]}
> - : Thm
- val prex2 = AxiomS (P, Imply(P,P), P);
    {[P -> [(P ->P) -> P]] -> [[P -> (P -> P)] -> (P -> P)]}
> - : Thm
- val prex3 = ModusPonens (prex2, prex1);  { [P->(P->P)]->(P->P) }
> - : Thm
- val prex4 = ModusPonens (prex3, prex1);  { P->P }
> - : Thm
```

Now typing ProofOf (prex4) to the interpreter will yield the ML representation of the formula P->P as a response. Notice that this proof is for a particular element of ML type Proposition (which we have been denoting P). For this element of type Proposition we could use the ML object PropCon ("P"), or Imply

(PropCon ("Q"), PropCon ("R")), or any one of a number of others. The proof works for any element of type Proposition. Furthermore, if we replace all occurrences of that particular proposition P by an ML variable, say x, of type Proposition in the proof, we get a parameterized proof, call it prex4'. Then I = (fun x. prex4') is a function which maps any Proposition into a proof that it implies itself. The function I is a derived rule of inference. This illustrates how ML acts as a useful meta-language.

4.4 The Programming Logic for Arithmetic

Now we describe a formal system for reasoning and programming over the integers. Our formalism will include the reals but we will give no axioms for this sort in this section. We describe its syntax in a manner similar to what was done above for the fragment of the propositional logic. Its rules are taken directly from constructive predicate calculus and Peano arithmetic together with a finite type hierarchy.

First the sorts are defined as a recursive data type, Sort. NN, RR and Prop are basic sorts and Arrow(s1, s2), Cross(s1, s2) are sorts when s1 and s2 are. Arrow(s1, s2) is the sort of functions from sort s1 to sort s2. Thus our former sort NNSeq is Arrow(NN, NN) and RRSeq is Arrow(NN, RR). We could have used a full simple

type theory in our previous discussion but we didn't find it necessary. It is simple to do it here since ML's recursive type constructor facility strongly suggests it. Cross(s1, s2) is the Cartesian product of sorts s1 and s2.

```
- type rec Sort = NN | RR | Prop |
                  Arrow of Sort * Sort |
                  Cross of Sort * Sort;
```

Terms are defined as an abstract recursive type. The context sensitive part of the definition which corresponds to sort checking needn't be considered when declaring the basic type constructors; this semantic information is captured by having the exported functions raise an exception when their inputs are not of the right sorts. This kind of failure is detected by using the SortOf function which is defined recursively over the terms.

Informally, variables of a fixed sort are terms; if t1 is a term of sort Arrow(s1, s2) and t2 is a term of sort s1 then Application(t1, t2) is a term of sort s2, it is the result of applying the function t1 to its argument t2; if t is a term of sort s1 and x is a variable of sort s2 then Abstraction(x, t) is a term of sort Arrow(s2, s1), it is the lambda abstraction which yields a function which assigns t[t'/x] to objects t' of sort s2 where t[t'/x] is the result of replacing x in t by t'; if t1 and t2 are terms of sort s1, s2 then Pair(t1, t2) is a term of sort Cross(s1, s2); if t is a term of sort Cross(s1, s2) then First(t) is a term of sort s1 and Second(t) is a term of sort s2; Zero is

a term of sort NN; SuccFunc is a term of sort Arrow(NN, NN); FF is a term of sort Prop; Imply, Or, and And are terms of sort Arrow(Cross(Prop, Prop), Prop); if x is a variable and t is a term of sort Prop then Some(x, t) and All(x, t) are terms of sort Prop; if t1 and t2 are two terms of the same sort then Eq(t1, t2) is a term of sort Prop; if t1 is a term of sort Prop and t2, t3 are terms of the same sort s then If(t1, t2, t3) is a term of sort s; if t1 is a term of sort s and t2 is a term of sort Arrow(Cross(NN, s), s) then Rec(t1, t2) is a term of sort Arrow(NN, s). We leave out all constants, functions and relations over RR. They will play no role in this section; when they do come in later sections, they will be treated informally.

Using these basics we can introduce definitions using ML.

```
- val Succ (x) = if SortOf(x)=NN then
                          Application(SuccFunc, x) else escape
                            "Not Natural Number";

- val One = Succ(Zero);

- val Imp(x, y) = if SortOf(x)=Prop and
                             SortOf(y)=Prop then
                             Application(Imply, x, y)
                             else escape "Not Props";
- val TT = Imp(FF, FF);

- val Neg(x:Term) = Imp(x, FF);
```

The recursion operator, Rec, is used to introduce functions by recursion. If t1 is of sort s1 and t2 is of sort Arrow(Cross(NN, s1), s1)) then Rec(t1, t2) is the function g of sort Arrow(Nat, s1) given by

$$g(0) = t1$$

$$g(n + 1) = t2(n, g(n)).$$

Suppose one wants to define plus using a the Rec recursive operator. The basic equations are

$$Plus(a, 0) = a$$

$$Plus(a, n + 1) = Plus(a, n) + 1.$$

Suppose x and y are variables of sort Cross(NN, NN). Then

```
Abstraction(x, Rec(First(x), Abstraction(y,
                    Succ(Second(y)))))
```

is a term of sort Arrow(Cross(NN, NN), NN) which defines the term Plus.

Formulas are terms of sort Prop. We now want to define a constructive calculus with which we can derive true facts about arithmetic. Our calculus is a natural-deduction style calculus with introduction and elimination rules. The simplest rules are the introduction and elimination rules for "FF". Written out in standard natural deduction style, they look like this:

```
FFIntro:  A & neg A |- FF
FFElim:        FF   |- A
```

If you have a proof of the formula before the "|-", then an application of the rule yields a proof of the formula after the "|-", The formulas (there can be more than one, even no formulas at all) befre the "|-" are called the hypotheses of the rule, and the formula (there must be exactly one) after the "|-" is the conclusion. A rule with no hypothesis is call an axiom.

These natural deduction rules (like modus ponens in the propositional calculus example) are represented in ML as constructors of type "Thm". Elements of type "Thm" are called proof expression. The "FFIntro" constructor represents the false-introduction rule. It takes an argument which must be a proof of a contradiction and the result is a proof expression proving false. Surprisingly "FFElim" needs two arguments. Besides the proof of false, "FFElim" needs the formula A as an argument to indicate what the proof expression proves. Thus "FFElim (prex: FF,A)" is a proof expression proving A (where by prex: FF we mean prex is a proof expression proving FF). The

constructors "AndIntro", "AndElimR" and "AndElimL" represent the following rules of the calculus:

```
AndIntro:      A, B |- A & B
AndElimR:    A & B |- A
AndElimL:    A & B |- B
```

These constructors will fail (like modus ponens in the propositional calculus example), if the arguments are not in the form that the rule prescribes. "AndIntro" is a constructor that takes two arguments, proof expressions, and forms a proof expression of the conjunction of the arguments. "AndElimR" is a constructor that takes one argument, a proof expression proving a conjunction, and is a proof expression of the left conjunct. The constructor "AndElimL" is similar. For disjunction there are two introduction rules and one elimination rule. The introduction rules "OrIntroR" and "OrIntroL" look like:

```
OrIntroR:      A |- A or B
OrIntroL:      A |- B or A
```

The constructor "OrIntroR" has two arguments: a proof expression which proves A and a formula B. Together these supply all the information necessary to form a proof expression of the disjunction.

The "OrElim" is slightly more complicated and takes three arguments. The first must be a proof expression of a disjunction; the other two arguments must be proofs of implications with special forms.

OrElim: A or B, A -> C, B -> C |- C

The proof of an implication in a natural deduction style calculus requires assuming A, then proving B, and discharging the assumption A to conclude A -> B. In our calculus we do this using the "Assume" construct and the "ImpIntro" rule as follows:

```
- val hyp = Assume ("hyp", A);          { Assume A. }
   ...
- val prex1 = ... hyp ... ;             { Derive a proof of, say, B. }
- val prex2 = ImpIntro (hyp, prex1);    { A proof of A-> B. }
```

The assumption (hyp in the example above) is said to be discharged in the proof. Finally, there is the corresponding elimination rule "ImpElim" which is just the familiar rule of modus ponens.

ImpElim: A -> B, A |- B

With the rules governing the basic connectives out of the way, we have primarily rules for the quantifiers and arithmetic left over. We list here the rules and axioms that are most self-evident.

```
Truth                                          |- True
PeanoPostulate7   succ (n) = succ (m)          |- n=m
PeanoPostulate8                                |- ~succ (n) = 0
AllIntro                              P(x0)    |- All x . P(x)
AllElim                       All x . P(x)     |-  P(t)
```

The all-introduction rule has the usual constraint that the variable x is not free in any undischarged assumptions.

Bear in mind that although these rules and axioms are presented in their familiar mathematical form, we take them as definitions of constructors in the representation of the logic in ML. These rules all have straightforward representations in ML.

Three rules are more complicated: "some" introduction, "some" elimination, and induction. First we summarize their basic form.

SomeIntro	P(t)	|- Some x . P(x)
SomeElim	Some x . P(x), P(x0) -> Q	|- Q
Induction	P(0), P(n) -> P(Succ (n))	|- All x . P(x)

The "SomeIntro" proof expression constructor actually requires three arguments. The first argument is a "some" formula. This provides the formula to be proved, since determining it from P(x0) is not trivial. Also the formula indicates the name of the bound variable which may be convenient for renaming variables. The second argument is a proof expression. It must prove the scope of the "some" formula P for a particular term. The third argument must be the particular term x0 for which one showed in the second argument that P(x0). Put together the use of the "SomeIntro" constructor looks like this:

```
SomeIntro (Some(x,P), prex:P(x0), x0)
```

"SomeElim" also has three arguments: the proof expression of some existentially quantified formula, the variable used to refer to the term postulated, and the proof of an implication with the

appropriate hypothesis.

SomeElim (prex1: Some(x,P), x0, prex2:P(x0) -> Q)

The implementation of the "SomeElim" contructor must not overlook the usual constraint on the use of the rule: namely, x0 can't occur free in any undischarged assumptions of prex2.

"Induction" has four arguments: the "All" formula to prove, the base case P(0), the induction variable, and a proof of the implication P(n) -> P(n').

Induction (All(m,P), prex1: P(0), n, prex2: P(n) -> P(n'))

The variable n must not occur free in any undischarged assumptions of prex2.

Now come another set of rules, called the computation rules. First there is beta-reduction. The proof expression

BetaReduction (Abstraction (v, b), t)

is a proof expression proving the following formula:

Application (Abstraction (v,b),t) = b[v/t]

where t is free for v in b. Strictly speaking the formula is:

Eq (Application (Abstraction (v,b),t) , Subs (b,v,t))

The remaining rules are for the Rec and If constructs. The proof expression

BaseCase (Rec(b,i))

is a proof of

Application (Rec(b,i), Zero) = b

and

IndStep (Rec(b,i),x)

is a proof of

Application (Rec(b,i), Succ x) =
    Appliction (i, Pair (x, Application (Rec(b,i)), x))

For the If construct there are two rules. The proof expressions

TrueIf (prex: P, If(P,t,s))

FalseIf (prex: Not (P), If(P,t,s))

are proofs of

If(P,t,s) = t

If(P,t,s) = s

respectively.

Finally, the rules concerning equality should be mentioned. They are the typical rules one would expect.

Reflexivity		|- n=n
Symmetry	n=m	|- m=n
Transitivity	n=m, m=p	|- n=p
Congruence	f=g, x=y	|- f(x)=g(y)
Substitution	x=y, P(x)	|- P(y)
EqualityElim	P=Q, Q	|- P

Now that we have the syntax of a sample programming logic, it is time to given an example of a proof. We give a proof of the following theorem

All x:NN (x=0 or Some y . Succ (y) = x)

It has a very simple proof by induction which is built up as follows where Variable constructs a variable given a string and a sort.

```
- val x,y = Variable ("x", NN), Variable ("y", NN);
- val SOME (term) = Some (y, Equal (Succ y, term));
- val EQ0 (term) = Equal (term, Zero);
- val ALL = All (x, Or (EQ0 (x), SOME (x));
```

(We have surpressed the ML response to these lines since it does not add to the discussion.) This first line declares two variables for use in the proof. The reminaing lines make abbreviations used to make formulas that will come up in the proof. The theorem to be proved is expressed as the last of these formulas, "ALL". The base case of the induction requires proving the the formula "Or (EQ0 Zero, SOME (Zero))" which is

proved easily by reflexivity.

```
- val base = OrIntroL (Reflexivity Zero, SOME (Zero));
```

The induction step requires more work. The induction hypothesis is not required in any material way in the proof of "Or (EQO (Succ x), SOME (Succ Zero))".

```
- val indhyp = Assume ("indhyp", ALL);
- val prex1 = Reflexivity (Succ x);
- val prex2 = SomeIntro (SOME (Succ x)), prex1, x);
- val prex3 = OrIntroR (EQO (Succ x), prex2);
- val indstep = ImpIntro (indhyp, prex3);
- val proof = Induction (ALL, base, x, indstep);
```

In the last line, the base case and the induction step are combined to yield a proof expression corresponding to the desired proof.

4.5 Constructive Mathematics

Thus far nothing should seem very unusual. We have described a theory with its language and rules. So we know how to make proofs in the system. In a programming logic on the other hand, the goal will require proving a different kind of theorem. Typically one will want to prove a theorem of the form "for all x--there exists y". Then by the nature of a programming logic, the interpreter when supplied with the proof and a particular x will produce a y with the desired property. Thus to program the maximum function, one would just prove a theorem in a programming

logic. For instance,

All x,y . Some z . (z=x or z=y) & z>=x & z>=y

Suppose the proof of this theorem was called "MaxThm". Then by using the interpreter to evaluate the expression formed by applying MaxThm to two integers will result in integer with the desired property. In order to write this interpreter some care is needed in formulating the rules of the programming logic. Fortunately, we can draw on the experience of the constructive school of mathematics for help in devising these rules. Their criticisms of classical mathematics provide insight to the problem.

In particular we have avoided certain axioms that are taken for granted in non-constructive logics. Typically the "law of excluded middle" or the axiom A & neg A is used freely. By not including this axiom, many formulas held to be true will not be provable. For example, let F stand for the statement of Fermat's last theorem. Some mathematicians hold that F or neg F is a true formula and they appeal to the law of excluded middle. We have rejected this axiom in the programming logic, because of its lact of constructive content. Consider for the moment the possiblity of adding this axiom. Represent the axiom by a constructor "ExcludedMiddle" which requires one formula as an argument. Thus ExcludedMiddle (F) is a proof expression proving that Fermat's theorem is true or false. Consider what happens when this proof

expression is used in conjunction with or-elimination.

OrElim (ExcludedMiddle (F), case1, case2)

Suppose that case1 evaluates or reduces to 1 when given a proof of Fermat's thoerem and case2 reduces to 2 when given a proof of its negation. In either case the proof expression reduces to a natural number. But which one? The interpreter can not figure out which. In a programming logic all expressions which can reduce to a natural number, do reduce mechanically to a number in canonical form (like 45 or 17). (The interpreter or the evaluator which performs the reductions is the subject of the next section.)

For the rules as we formulated them it is easy to see how we can justify the claim that evaluation will be mechanical. Since an "or" formula can be proved only by "OrIntroL" or "OrIntroR" and thus the case is explicitly tagged, there will never be any problem deciding which case is true. It should also be apparent how a particular value can be computed, a "some" formula can be proved only by "SomeIntro" and this requires a particular term to be supplied. This value will be used by the interpreter.

4.6 An Interpreter for Arithmetic

As in the programming language ML and LISP, the interpreter for

our arithmetical calculus will take an expression (a proof expression) and return a value. In our case the value returned is a simplified proof expression. Church's lambda calculus provides the rules by which LISP expressions are evaluated. The most important of these rules is beta reduction. Since we have lambda terms in our arithmetical calculus we expect to find at least the beta reduction rule. In fact there are many such reduction rules in our arithmetical calculus, and we will go through these reduction rules now. Later we will see exactly what expressions we actually have to enter to our system in order to evaluate programs like the maximum function or the substraction function.

The rules for pairs are particularly simple.

```
First (Pair (t,s)) --> t
Second (Pair (t,s)) --> s
```

The rules dealing with conjunction are similar.

```
AndElimR (AndIntro (prex1, prex2)) --> prex1
AndElimL (AndIntro (prex1, prex2)) --> prex2
```

Here is the reduction rule for beta reduction.

```
Application (Abstract (x, b), t) --> b[x/t]
```

The notation b[x/t] means that the term t is substituted for the variable x in the term b. If any free variable of t is captured the result is an error message. There is a rule for "AllElim"

proof expression.

AllElim (AllIntro (x, prex),t) --> prex[x/t]

The rule for "ImpElim" is similar.

ImpElim (ImpIntro (hyp, prex1), prex2) --> prex1[hyp/prex2]

This is an important reduction since it comes up as a part in the remaining rules. It is worth considering this reduction a little more closely. The "ImpIntro" constructor enforces that the proof expression "hyp" is in the form "Assume (name, A)" where A is some formula. The "ImpElim" constructor enforces that whatever proof expression its first argument is, it is a proof of A->B. This much follows from the definition of the proof expression constructors. Clearly, the proof expression "ImpIntro (hyp, prex1)" is one such proof expression proving A->B for some formula B. But there are others, including some which are not necessarily implication-introduction expressions. The proof expression could be, for instance, a some-elimination or an or-elimination proof expression and still be a proof of A->B. One could say that the type of the first argument must A->B. Of course, if the evaluation is to continue these expressions of type A->B must evaluate to an implication-introduction expression so that the implication-elimination reduction rule can be applied.

The reduction rules for "or", "some" and induction complete the list of reduction rules.

```
OrElim (OrIntroR (A, prex1), prex2, prex3)
        --> ImpElim (prex2, prex1)
OrElim (OrIntroL (A, prex1), prex2, prex3)
        --> ImpElim (prex3, prex1)
SomeElim (SomeIntro (S, prex1, t), y0, prex2)
        --> ImpElim (prex2[y0/t], prex1)
AllElim (Induction (A, prex1, n, prex2), Zero)
        --> prex1
AllElim (Induction (A, prex1, n, prex2), Succ x)
        --> ImpElim (prex2, AllElim (induc, x)[n/x])
```

The proof expression "induc" in the last line is just the original induction expression:

```
Induction (A, prex1, n, prex2)
```

The role of the interpreter is to apply any of these reduction rules until none of them are applicable. We call this process normalization, simplification or reduction.

With the basic normalizing procedure in mind, consider again the substraction example. There we had a proof expression of the basic form

```
Induction (All (x, P), base, x, indstep)
```

Normalization will produce no change in this proof expression. It is already in normal form. Most often this will be the case unless by oversight a proof with needless steps was done. A reasonable implementation of normalization can, however, provide one most important service, it can guarantee that there are no free variables or undischarged assumptions in the proof. So even

if a proof expression does not simplify it is certified to represent a proof in our system.

The interpreter is not limited to the role of proof checker. Consider the following proof expression still using the substraction example.

```
AllElim (Induction (All x, P), base, x, indstep), Zero)
```

The result of evaluating this proof expression is simply base or

```
OrIntroL (Reflexivity Zero, SOME (Zero))
```

When we apply the proof by induction to "One" we get the following chain of proof expressions.

```
--> ImpElim (indstep, AllElim (Induction (...), Zero))[x/Zero]
--> ImpElim (indstep[x/Zero], OrIntroL (...))
--> ImpElim (ImpIntro (indhyp[x/Zero], prex3[x/Zero]),
             OrIntroL (...))
--> prex3[x/Zero] [indhyp/OrIntroL (...)]
--> prex3[x/Zero]
--> OrIntroR (EQ0 (Succ Zero), prex)
```

where prex is the proof expression:

```
SomeIntro (SOME (Succ Zero), Reflexivity (Succ Zero), Zero)
```

Applying the induction proof to "Two" yields a similar or-introduction proof expression; this time, however, prex is the proof expression:

SomeIntro (SOME (Succ One), Reflexivity (Succ One), One)

Thus, we have written a primitive program to subtract one from any natural number and have run the program on the first three natural numbers. Notice that all the information about the results of the subtraction are still around. We see that if subtraction does not apply (that is, subtracting one from zero), then the resulting proof expression was an or-introduction-left proof expression. If the substraction worked, the result was an or-indroduction-right proof expression. The result of subtracting one from the given quantity can be found buried in the some-introduction rule. It is the term given as a witness that there is number whose successor is the given value.

There are two parts to the some-introduction proof expression: the witness, and the proof that the witness has some property. We see from the above example that we may want to throw out the proof part as being unimportant and actually pick out the witness. Suppose we had in our language a function "Witness" that evaluated as follows:

Witness (SomeIntro (S, prex, t)) --> t

The function "Witness" could be used to ignore the proof part of a some-introduction proof expression. Finally we have all the mechanisms necessary to extract a function from a proof of the form "for all--there exists". (Actually "Witness" is definable

from the existing constructors, but for as far as we are concerned here, it can be taken as primitive.)

The substraction theorem is an interesting case, because it is an example of a partial function. The value of the function at zero is avoided. Compare the formulation of the theorem as was given originally

```
All x . (x=0 or Some y . succ (y) = x)
```

with the alternate formulation:

```
All x . Some y . ~(x=0) -> succ (y) = x)
```

This alternate formulation can be prove by induction, but the proof is slightly more difficult. But suppose we have a proof of it, call it "SubThm". The base case is vacuously true, but must be proved by some-introduction which requires a witness nevertheless. Say the base case was proved with 34 as the witness (any number will do, of course). Now consider the result of evaluating some proof expressions containing "SubThm".

```
Witness (AllElim (SubThm, Zero)) --> ThirtyFour
Witness (AllElim (SubThm, One)) --> Zero
Witness (AllElim (SubThm, Two)) --> One
```

The dicussion above should have given some idea as to how a programming logic is a programming language. One could easily make an interpreter that interacts with the user just like the ML

interpreter. There would be similar sort of dialog with the user typing in an expression and the interpreter returning the simplified form of the expression. For example, a dialog concerning "SubThm" might look like:

```
- Witness (AllElim (SubThm, Zero));
> ThirtyFour : Nat
- Witness (AllElim (SubThm, One));
> Zero : Nat
- Witness (AllElim (SubThm, Two));
> One : Nat
```

Thus we have seen that the interpreter in a programming logic is both a theorem checker (or, equivalently a program verifier) and a functional programming language evaluator.

4.7 Incorporating the Asymptotic Paradigm

In this section we raise some of the issues that arise when programming logic is extended to non-standard analysis. Our discussion is quite tentative. What we are aiming at is an extension of our previous constructive arithmetic calculus to some form of NSA. Our target system is best illustrated by the example given in the next section.

We have already encountered one technical problem in using non-standard numbers: induction. The induction rule of arithmetic must be modified to exclude the proof of external formulas (those containing the std predicate).

But there are other difficulties as well. The form of the axioms in the programming logic for the real numbers remains to be worked out. Much mathematical research has been done in the area of constructive analysis ([2], [3], [4]). This work guides the attempts to formalize the reasoning concerning real numbers. The difficulty lies in that some of the ordinary axioms for real numbers do not have constructive content. So they have problems similar to the law of excluded middle discussed in Section 4.5. One such axiom for real numbers is dichotomy; another is that odd degree polynomials have a root. The first axiom asserts one of two things happen and the second asserts the existence of some quantity. Axioms in these two forms pose the difficulty.

Consider for a moment dichotomy. We certainly want that x<=y or x>y for all real numbers. But if we take this as an axiom, then the interpreter will have to be able to decide for any real numbers which case holds. This is difficult for arbitrary real numbers. For instance, how is the interpreter to know if f(x)+2.3 is greater than g(x+y)/x? For example, consider the following proof expression:

---

2. A. S. Troelstra, Metamathematical Investigation of Intuitionistic Arithmetic and Analysis, (Springer-Verlag: Berlin, 1973).

3. Arend Heyting, Intuitionism: An Introduction, (North-Holland: Amsterdam, 1971).

4. Errett Bishop, Foundations in Constructive Analysis, (McGraw-Hill: New York, 1967).

OrElim (Dichotomy (x,y), prex1, prex2)

where "Dichotomy(x,y)" is proof expression proving x<=y or x>y. The interpreter will in general be unable to figure out which branch to take. It is unclear at this point if this causes in problem in practice.

One solution is to restrict the use of the axiom to values the interpreter can actually test. For given two floating-point numbers stored in the computer the interpreter can test them to find which is the larger. The reduction rule would then look something like this:

```
if x<=y then
  OrElim (Dichotomy(x,y), prex1, prex2)
    --> ImplElim (prex1, FACT1:x<=y)
otherwise
  OrElim (Dichotomy(x,y), prex1, prex2)
    --> ImplElim (prex2, FACT2:x>y)
```

FACT1 and FACT2 are proof expressions proving that the appropriate relationship holds between x and y. These are axioms in a sense, but cannot be invoked by the user.

Arbitrary arithmetic is also the problem with some-introduction. It may prove useful to identify a certain class of terms in NSA, call them the computable terms. This set of terms includes all the variables and constants, and is closed under the machine operations, ++, --, **, and //. Also in the list of operations which produce computable terms are Skolem

functions for existential axioms (we will encounter these shortly) and other functions defined with the Rec and If constructs as long they contain only computable terms. One can check syntactically if a term is computable.

Since some-introduction requires the interpreter to actually compute the value of the witness, some restriction on the witness is to be expected. The natural candiates for witnesses are the computable terms. The same restriction must apply to all-elimination.

Now we examine constructive content of another axiom of NSA. One of the basic axioms of NSA was that the range of the cropping function is finite. One possible way of formalizing this in the language is as follows.

FIN1: Some i . f(i+1) >= f(i)

FIN2: Some i . f(i+1) <= f(i)

We will choose a slightly more convenient form of these axioms by naming a Skolem function which computes the desired point in the sequence.

FIN1: f(FIN1(f)+1) >= f(FIN1(f))

FIN2: f(FIN2(f)+1) <= f(FIN2(f))

These axioms can be instantiated with any function "f" of the right type. We require that "f" be a computable operator.

How does this capture the fact that the range of the cropping function is finite? For one, it is a necessary condition. If the range of the cropping function is finite, then the set of machine representable numbers is finite, and thus the set of values the interpreter can return by evaluating variable-free machine terms is even smaller. On the other hand, the axioms appear sufficient for practical purposes. The axioms permit arguments of the sort that there are no infinite descending (or ascending) sequences of machine values.

If these axioms are to be understood by an interpreter, their constructive content must be understood. This is especially critical for these existential axioms, since such existential statements must actually produce the values they claim exist. Fortunately, this poses no problem here since we know no sequence of machine representable numbers can keep increasing (or decreasing) forever. We can find the place where the sequence stops increasing (or decreasing) by just examining the values in the sequence one by one. This may not be efficient, but it is guaranteed to work since the range of machine reprcsentable numbers is finite. The interpreter can compute the values of the sequence until one with the right property is found. This value cán then be used for FIN1(f) or FIN2(f). Eventually the algorithmic part of the axioms can be compiled into simple while loops. Here is the while loop for the FIN1 axiom given f.

```
i := 0;
while f(i+1) > f(i) do i := i+1 end;
return (i);
```

There is a gray area in NSA where the desire for constructive content of the axioms competes with the need for expressing idealized computations. This mixture of constructive and non-constructive rules does not in and of itself cause the interpreter any problem. What the interpreter does not understand can not be simplified. This leads to the following problem. A theorem (using non-constructive constructs) of the form "for all real numbers x-there exists a real number y" can be verified as a correct theorem in the theory, but when applied to a particular value x the proof expression may not simplify to the real number y in normal form. We can take normal form for real numbers to mean a variable-free computable term. This diminishes the usefulness of the verification. For the number-theory programming logic presented previously it is conceivable to prove a meta-theorem that all proof expressions representing natural numbers can be reduced to a series of successor functions applied to zero. Such a meta-theorem is highly desirable for NSA. Most likely it will be easier to define a subset of NSA that can be mechanically recognized which can be shown to be normalizable.

4.8 Finding Square Roots by Newton's Method

Let us now turn to an example of a floating-point program

proved correct in a programming logic version of NSA. As mentioned previously we view the example as a test case for building such a logic. The programming logic sketched in the previous section, while tentative, is adequate for carrying out the following proof which is a program. Since the logic was constructed so that all the axioms and rules of inference have constructive content, the proof can be executed by the interpreter.

The example we shall use is Newton's method for computing the square root of any real number.

Square Root Theorem. All x:real . Some r:real . x>1 => r*r==x

A proof of this formula will be a function that can take any machine representable number and if it is greater than one, this function will produce another machine representable real number whose square is infinitesimally close to the original number. The remainder of this section is devoted to showing what is involved in formalizing the proof of the Square Root Theorem.

The proof of the theorem will certainly require many of the ordinary fact about the ideal real numbers. We will use the field axioms and the order axioms without much comment. Note that we do not expect any part of the proof dealing with ideal real numbers ever to effect the evaluation of a proof expression.

To prove that the square root exists we will first define a sequence of machine representable numbers that get closer and closer to the square root of x. Here is the recursive definition of the sequence written in ML.

```
val F (i:Nat): Real =
  if i=0 then x
  else let val Next = (F(i-1)++(x//F(i-1)))//2 in
   if Next > F(i-1) then F(i-1) else Next
  end;
```

It is clear that we could have defined F using the Rec and If constructs, but such a definition would not be perspicuous. Notice that we can prove that F(i) is a machine representable for all i, since all the compuational steps (including ++, // and >) are all perfectly understandable by the interpreter.

The computation rules permit the following conclusions about the recursively defined function F:

```
F(0) = x
F(i) = if Next > F(i-1) then F(i-1) else Next
          where Next = (F(i0)++(x//F(i0)))//2
```

The computation rules for the If construct give rise to two more rules which can be used in the proof.

$$\frac{\text{Next} > F(i-1)}{F(i) = F(i-1)} \qquad \frac{\neg(\text{Next} > F(i-1))}{F(i) = \text{Next}}$$

Using these rules is the only means of proving properties about recursive definitions. We will need these particular rules to

continue the proof of the Square Root Theorem.

We can prove by induction that this sequence F has the property:

Property 1. All n:Nat . F(n) >= F(n+1)

Notice that this proof requires no reasoning about floating-point calculations whatsoever. The proof relies solely on the definition of F. Another property of the sequence F provable by induction is:

Property 2. All n: Nat . x >= F(n) >= 1

That x>=F(n) holds, follows from Property 1 about F. That F(n)>=1 holds, requires knowing something about floating-point computations. In particular, we need to know that y>=z implies y//z>=1, and that y>=1 and z>=1 imply (y++z)//2>=1. These facts follow from the monotonicity of the cropping function. We will give a more detailed proof of Property 2 later.

Now we give a sketch of the proof of the Square Root Theorem. By the finiteness axioms we know that there is some i0 for which F(i0+1)>=F(i0). This combined with the fact (Property 1) that F(n)>=F(n+1) implies that F(i0+1)=F(i0). What does this mean? If F(i0+1) equals (F(i0)++(x//F(i0)))//2, then we have what we would expect since ideally

F(i)=(F(i)+(x/F(i))/2 implies F(i)*F(i)=x

Let us set Next equal to the quantity (F(i0)++(x//F(i0)))//2. Now suppose F(i0+1) = Next, then we must show F(i0)*F(i0)==x. First we must know that Next is infinitesimally close to its ideal counterpart: (F(i0)+(x/F(i0)))/2. (This result is proved in Lemma 1 below.) Hence (F(i0)+(x/F(i0)))/2 == F(i0). From this follows (Lemma 2) the desired result. The proof is remotely similar to the ideal mathematical case, but there are many details to check. The floating-point computations do have the needed properties like their ideal counterparts do, but to verify this requires more effort and we put this off for the moment.

The proof is not yet finished. It need not be the case that F(i0+1) = Next. Recall the definition of the function F. If Next > F(i0) then F(i0+1)=F(i0). This would be the case when cropping errors in the computation of the next value in the sequence did not result in a value that was less than or equal to the previous value. But nevertheless we have F(i0)*F(i0)==x, since Next is really very close to F(i0). In fact we can prove that Next > F(i0) implies that (F(i0)+(x/F(i0)))/2 == F(i0). This is the content of Lemma 3. We defer this proof as well. F(i0)*F(i0) == x follows again from Lemma 2.

All that is needed to complete the proof is to put the two cases, Next <= F(i0) and Next > F(i0), together. The cases are exhaustive (by dichotomy) and in each case we have the desired

conclusion. This suggestes the or-elimination rule.

OrElim (Dichotomy(Next,F(i0)), case1, case2)

Finally, we pick as the square root F(i0). The proof expression for the whole proof takes on the following form.

AllIntro (x, SomeIntro (S, OrElim (...), F(i0))

We have just seen an overview of the proof of the Square Root Theorem. It is time now to go back and fill in the details.

First we prove Property 2. The proof proceeds by induction. For n=0 we must show that x>=F(0)>=1. Since F(0)=x and we assumed x>=1, this is trivial. So now we assume the induction hypothesis x>=F(n)>=1 and prove that x>=F(n+1)>=1. Set Next to be (F(n)++(x//F(n)))//2. If Next>F(n) then F(n+1)=F(n) and we are finished. Otherwise Next<=F(n) and F(n+1)=Next. Since x>=F(n), x>=F(n+1). Now comes the hard part: showing F(n+1)=Next>=1. We must analyse the floating-point operations in Next=(F(n)++(x//F(n)))//2. Since x>=F(n) and ~(F(n)=0), we expect that x//F(n)>=1. This is in fact the case. Since F(n)>=1, we expect that F(n)++(x//F(n))>=2. Finally dividing by 2 we get Next>=1, the desired conclusion.

The properties of the floating-point operations used above do follow from the axioms of NSA. Let us examine one of these facts in greater detail. From y>=z it follows that y//z. Expanding

the definition of y//z we get ~(z=0) & w=CR(y/z) & w<=1. Assuming that ~(z=0) we get that y/z>=1 by the axioms of ideal arithmetic. By the monotonicity of CR we have CR(x/y)<=CR(1)=1. Hence w<=1.

For Lemmas 1, 2 and 3 we set Next equal to (F(i0)++(x//F(i0)))//2 and IdealNext equal to (F(i0)+(x/F(i0)))/2.

Lemma 1 states that Next == IdealNext. The proof proceeds as follows.

```
x//F(i0) == x/F(i0)
F(i0)++(x//F(i0)) == F(i0)+(x/F(i0))
(F(i0)++(x//F(i0)))//2 == (F(i0)+(x/F(i0)))/2
```

Each one of these steps depends on a similar argument about floating-point calculations which makes use of the fact the fin(x) implies CR(x)==x. So, each step reduces to showing that the appropriate quantity is finite. In the first step, for example, we must show that x/F(i0) is finite. But that follows from the fact that x/F(i0)<=1<2 and that 2 is standard.

Lemma 2 states that IdealNext == F(i0) implies F(i0)*F(i0) == x. The proof proceeds as follows.

```
(F(i0)+(x/F(i0)))/2 == F(i0)
F(i0)+(x/F(i0)) == 2*F(i0)
x/F(i0) == F(i0)
x == F(i0)*F(i0)
```

Each step follows from a simlar argument about floating-point

computations. The essence of the first step is y/2==z implies y==2*z. Expanded this yields inf(y/2 - z) implies inf(y - 2*z). This follows from the fact that for all epsilon

$$|y/2 - z| < \text{epsilon}/2 \text{ implies } |y - 2*z| < \text{epsilon}$$

Recall that inf(x) is defined to be

$$\text{All epsilon . (std(epsilon) \& epsilon} > 0) \rightarrow |x| < y$$

Lemma 3 states that Next > F(i0) implies IdealNext == F(i0). By the laws of ideal arithmetic we have that IdealNext < F(i0). Hence IdealNext < F(i0) < Next. From Lemma 1 we have that Next == IdealNext. Clearly IdealNext == F(i0).

Chapter 5

Technical Feasibility

This report, although incomplete in places, shows the feasibility of our apprach to the formal specification and verification of mathematical software. Our basic concept, the use of non-standard analysis to represent the asymptotic behavior of programs, is new; there does not appear to be anything comparable to it in the literature. Further experimentation with approaches is necessary before an appropriate verification system can be designed. We believe such experimentation is best carried out using rapid prototyping. An experimental VCG can be built without an accompanying theorem prover and used to examine the forms of the VCs generated; simplification rewrite rules over the non-standard reals can be devised in order to simplify the print form of the VCs; our ML prototype should be completed in several different ways and experimented with.

Our final vision is a system in which a mathematically sophisticated programmer/mathematician could interactively verify libraries of floating-point routines or critical sections of large systems which use the floating point data type. These

verified programs might then be transfered to other machines following the Host/Target scenario familiar in embedded systems. The reason for such configurations is that environments useful for program development (including formal specification/ verification) are not necessarily optimal for run-time requirements like advanced floating point precision and efficiency. The portable programs produced by our verification environment can then be used with far greater assurance of their reliability. Indeed, our asymptotic approach to verification is consistent with and supports the use of verified programs on a variety of machines.

Our greatest departure from mainstream efforts in program verification is in using non-standard analysis. This is at once the most risky and the most innovative aspect. In the past 20 years the logical basis of non-standard analysis has been worked out but mainly by mathematical logicians as opposed to computer scientists. Thus tasks of building formal languages with their accompanying grammars and parsers which express these concepts and automated proof environments which manipulate the constructs are open research areas. The only applicable work in automated theorem proving which we are aware of is [1].

While very little precedent for our apporach is available we do

---

1. Ballantyne and Bledsoe, "Automatic Proofs of Theorems in Analysis Using Nonstandard Techniques," JACM, 24 (July 1977), pp. 353-374.

feel that based on our experience so far, such an approach appears to be conceptually simpler than conventional techniques which rely on bounding machine operations on floating-point numbers to within an "epsilon" of the actual result. The proofs using such rules are difficult and unenlightening. However, statements about the asymptotic precision of programs, like statements about the asymptotic complexity of programs, make meaningful assertions about programs and at the same time permit an intuitive theory to be developed. This is the advantage of using non-standard analysis as the theoretical underpinnings of verification.

Using non-standard analysis as the theoretical basis, we have discussed building a verification system using two different approaches with proven feasibilty. There are several well-known verifying systems based on the VC approach. There are, for example, the Stanford Pascal Verifier [2], the Gypsy Verification Environment [3], and the not yet completed Euclid Verification System [4]. None of these systems support either fixed or floating point reals. The success of the VCG approach depends on constructing a "good" theorem prover. Such a theorem prover

---

2. W. Polak, Compiler Specification and Verification, Lecture Notes in Computer Science, 124, Springer-Verlag, 1981.

3. Donald Good, et al, Using the Gypsy Methodology, University of Texas, Austin, 1981

4. D. Craigen, Ottawa Euclid and EVES:A Status Report, Proc. 1984 Symp. on Security and Privacy, IEEE Computer Society

should prove trivial theorems and simplify non-theorems automatically while supporting a user-machine interaction to prove more difficult theorems. Finding the right mix between automatic and proof checker mode is the subject of much current research. Using non-standard analysis as the underlying theory causes no additional burden, since it can be adequately axiomatized in first-order logic.

The second approach to verification using non-standard analysis that we have proposed using is the programming logic approach. A system based on this approach is presented in [5]. This is a programming logic adapted to a variant of the PL/I language (without real data types). The PRL (for Program Refinement Logic) project at Cornell University [6] is a continuing NSF sponsored reseach effort along these lines. While it is not yet clear if the formalization of non-standard analysis in this framework is flexible enough, the benefits of success would, however, be great. First of all, all verified programs terminate. The system proves total correctness and not just partial correctness. All of the VCG environments mentioned previously consider only partial correctness. Second, rapid prototyping and experimentation with the logic is possible using

---

5. Constable, R., et al, An Introduction to the PL/CV2 Programming Logic, Lecture Notes in Computer Science 135, Spring-Verlag, 1982

6. Constable and Bates, "The Nearly Ultimate Pearl", Cornell University Technical Report, January 1984.

the programming language ML. Third, meta-reasoning or more abstract reasoning would be possible as in the PRL project. Finally, decision procedures can easily be incorporated to prove the trivial details. The drawback of the programming logic approach is that it does not produce programs in the imperative form that programers are used to. This leads to a possible acceptance problem. If previously compiled library routines need only be linked and used and not modified then there is no difficulty. But if verified programs need to be modified the programming logic route would entail training in new language.

The VCG approach also leads to modifications difficulties since the verification is nullified when changes are made. It is difficult to verify a program that one hasn't written and also difficult to reverify a program which one has written but which has been modified by someone else. On the other hand, VCG based environments can be designed using data base capablities which minimize the reverification effort. If modification and non-verification expert readability is a concern then the VCG approach should be tailored to known languages like FORTRAN, HAL/S, or Ada even though they are not as well structured from a verification point of view as is Euclid or Gypsy.

Appendix A

Proofs

This Appendix contains the proofs of several theorems which were used in the course of proving VCs for examples in Section 4.

THEOREM 1: CR is monotone

Suppose not, i.e. suppose that there exist x and y such that x <= y but CR(y) < CR(x).

Case 1: CR(y) <= x

In this case, CR(y) <= x <= y, and CR(CR(y)) = CR(y) by the second cropping function axiom. Therefore, by the fourth cropping function axiom, CR(x) = CR(y), a contradiction.

Case 2: x < CR(y)

In this case, x < CR(y) < CR(x), and CR(CR(x)) = CR(x), so by the fourth cropping function axiom, CR(x) = CR(CR(y)) = CR(y), a contradiction.

THEOREM 2: There is no machine real strictly between x and CR(x)

Suppose not, i.e. that there exists x and y such that y is strictly between x and CR(x) and CR(y) = y. CR(CR(x)) = CR(x), so by the fourth cropping function axiom, CR(y) = CR(x), a contradiction.

THEOREM 3: If (1 + X/N) == 1 and J is a standard integer, then (1 + X/N)^J == 1

The proof is by induction on J. Note that the statement we are trying to prove is external, so induction will only prove it for standard J, but this is all we want.

For J = 0 the formula is trivially true. Now suppose (1 + (X/N))^J == 1. (1 + (X/N)) == 1, so (1 + (X/N)) is finite and we can multiply both sides of the inductive hypothesis to get

$$(1 + (X/N))^{(J + 1)} == (1 + (X/N)) == 1$$

and so the theorem is proved for all standard J.

THEOREM 4: If Z is a finite real and J is a finite integer, then Z^J is a finite real.

The proof is by induction on J. Again, induction will only prove the statement for J standard. Since all finite integers are standard, this will prove the theorem.

If J = 0, Z^J = 1, a finite real. Now assume Z^J is finite. Z^(J+1) = Z^J * Z. Z^J and Z are both finite, so their product is

finite. This finishes the induction.

THEOREM 5: If f0 is a continuous function from $\underline{R}$ to $\underline{R}$, f the non-standard extension of f0, and g(x) == f(x) for all finite x, then for any finite x, there exists a standard y such that y == x and f0(y) == g(x)

First of all, the non-standard analysis statement of "f0 is continuous" is

all x,y : R [std(x) & x == y -> f(x) == f(y)]

Since x is finite, there is a standard real y infinitely close to x. By the above statement of the continuity of f0, f(y) == f(x). y standard implies that f(y) = f0(y). Therefore f0(y) == f(x). x finite implies that f(x) == g(x). The theorem follows by transitivity of ==.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR 172407		

4. Title and Subtitle	5. Report Date
FORMAL VERIFICATION OF MATHEMATICAL SOFTWARE	MAY 1984
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
David Sutherland	

9. Performing Organization Name and Address	10. Work Unit No.
Odyssey Research Associates, Inc. 609 West Clinton St. Ithaca, NY 14850	11. Contract or Grant No. NAS1-17579

12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
National Aeronautics and Space Administration Washington, DC 20546	Contractor Report
	14. Sponsoring Agency Code 324-01-00-01

15. Supplementary Notes

Langley Technical Monitor: Dr. J.N. Shoosmith
Final Report

16. Abstract

The purpose of the research described in this report was to investigate methods for formally specifying and verifying the correctness of mathematical software (software which uses floating point numbers and arithmetic). Previous work in the field was reviewed. A new model of floating point arithmetic called the asymptotic paradigm was developed and formalized.

Two different conceptual approaches to program verification, the classical Verification Condition approach and the more recently developed Programming Logic approach, were adapted to use the asymptotic paradigm. These approaches were then used to verify several programs; the programs chosen were simplified versions of actual mathematical software.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Program Verification Floating Point Arithmetic	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	154	A08

N-305

For sale by the National Technical Information Service, Springfield, Virginia 22161