Specification of the IEEE-854 Floating-Point Standard in HOL and PVS

(To be presented at the 1995 International Workshop on Higher Order Logic Theorem Proving and its Applications, September 11-14, Aspen Grove, Utah, USA, as a track B paper and included in supplemental proceedings)

Victor A. Carreño Paul S. Miner
NASA Langley Research Center
Hampton, VA 23681-0001
{v.a.carreno,p.s.miner}@LaRC.NASA.GOV

Abstract

The IEEE-854 Standard for radix-independent floating-point arithmetic has been partially defined within two mechanical verification systems. We present the specification of key parts of the standard in both HOL and PVS. This effort to formalize IEEE-854 has given the opportunity to compare the styles imposed by the two verification systems on the specification.

1 Introduction

The HOL [3] and PVS [7] systems are general purpose mechanical verification systems whose specification languages are based on higher-order logic. We have partially defined the ANSI/IEEE-854 standard for radix-independent floating-point arithmetic [5] in both of these verification systems [2, 6]. This effort to formalize IEEE-854 has given the opportunity to compare the styles imposed by the two verification systems on the specification. This is not the first formalization of floating-point arithmetic. Geoff Barrett [1] describes the Z formalization of IEEE-754 used in the development of the INMOS T800 Transputer. The work reported here is different in two respects. Z is primarily a specification language with limited mechanical support. Both HOL and PVS provide substantial support for machine checked theorem proving. Also, IEEE-854 is a generalization of the ANSI/IEEE-754 [4] standard for binary floating-point arithmetic.

This paper will compare portions of the two specifications. Section 2 will describe the aspects of IEEE-854 addressed by this paper. Sections 3 and 4 will present the HOL and PVS formalizations, respectively. Section 5 will discuss the differences between the two specifications. We believe that modern verification systems are advancing to the point where it may become practical to formally define standards using a mechanized logic.

2 IEEE-854 Standard

IEEE-854 is a general standard for floating-point arithmetic. In contrast to the IEEE-754 standard for binary floating-point arithmetic [4], IEEE-854

- does not define formats for storage of floating-point numbers,
- does not fully specify the required number of digits,
- does not fully specify the exponent range, and
- allows for decimal as well as binary arithmetic.

However, some constraints are still required to ensure that the number system is well behaved. Thus, IEEE-854 is parameterized with constraints placed on the formal parameters. Section 3.1 of the standard defines the parameters:

> *Four integer parameters specify each precision:*
>
> $$\begin{aligned} b &= \textit{the radix} \\ p &= \textit{the number of base-}b\textit{ digits in the significand} \\ E_{max} &= \textit{the maximum exponent} \\ E_{min} &= \textit{the minimum exponent} \end{aligned}$$
>
> *The parameters are subject to the following constraints:*
>
> 1. *b shall be either 2 or 10 and shall be the same for all supported precisions*
> 2. *$(E_{max} - E_{min})/p$ shall exceed 5 and should exceed 10*
> 3. *$b^{p-1} \geq 10^5$*
>
> *The balance between the overflow threshold ($b^{E_{max}+1}$) and the underflow threshold ($b^{E_{min}}$) is characterized by their product ($b^{E_{max}+E_{min}+1}$), which should be the smallest integral power of b that is ≥ 4.* [5, page 8]

The precisions defined are single, double, single extended, and double extended. In addition to satisfying the above constraints, the relationship between the supported precisions is also constrained. Since the examples in this paper do not deal with multiple precision, we will not present these constraints here.

2.1 Floating-Point Numbers

The standard defines operations using the following definitions of floating-point numbers:

Each precision allows for the representation of just the following entities:

1. *Numbers of the form* $(-1)^s b^E (d_0.d_1 d_2 \cdots d_{p-1})$ *where*

$$\begin{aligned} s &= \text{an algebraic sign} \\ E &= \text{any integer between } E_{min} \text{ and } E_{max}\text{, inclusive} \\ d_i &= \text{a base-}b\text{ digit } (0 \le d_i \le b-1) \end{aligned}$$

2. *Two infinities,* $+\infty$ *and* $-\infty$
3. *At least one signaling NaN*
4. *At least one quiet NaN* [5, page 8]

We will illustrate how we represent these entities in both HOL and PVS.

2.2 Rounding

Since floating-point numbers are a finite approximation of the real numbers, the standard defines how real numbers are mapped into a floating-point representation:

> *An implementation of this standard shall provide round to nearest as the default rounding mode. In this mode the representable value nearest to the infinitely precise result shall be delivered; if the two nearest representable values are equally near, the one with its least significant digit even shall be delivered.* [5, Section 4.1, page 9]

In addition, the standard continues:

> *An implementation of this standard shall also provide three user-selectable directed rounding modes: round towards* $+\infty$*, round towards* $-\infty$*, and round towards 0.* [5, Section 4.2, page 9]

There are many different ways to specify rounding. The HOL specification gives abstract definitions that satisfy the required properties. The PVS specification uses more concrete definitions of the rounding functions. These are proven to satisfy the necessary abstract properties. Rounding was the most difficult part of the standard to define formally.

2.3 Arithmetic Operations

The standard states the following requirements for arithmetic operations:

> *All conforming implementations of this standard shall provide operations to add, subtract, multiply, divide, extract the square root, find the remainder, ...*

..., each of the operations shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then coerced this intermediate result to fit in the destination's precision. [5, Section 5, page 10]

We will illustrate the definition of arithmetic functions using definitions of floating-point addition.

3 HOL system specification

This section gives a partial specification of IEEE-854 for illustration purposes and for comparison with the PVS specification. The full IEEE-854 specification in HOL is given in [2]. The definition in HOL of the parameter restrictions, floating-point type, rounding, and the add operation are discussed in this section. John Harrison's reals library and Elsa Gunter's integer library are used for the definition of the real and integer type. The natural, real, and integer numbers are separate types in the HOL system with different functions defining arithmetic and other operations. A sample of the arithmetic operators for the naturals, integers and reals is:

natural	integer	real	standard symbol
`-`	minus	real_sub	$-$
`<`	below	real_lt	$<$
`<=`	below_or_e	real_le	$\leq$
0	INT 0	& 0	0
		abs(x)	$\|x\|$

In the HOL system the symbol `?` represents $\exists$, `!` represents $\forall$, and `@` is the choice or Hilbert operator.

The four parameters defining a precision, b, p, E_{max}, and E_{min}, are defined in the HOL system by declaring b as a constant and placing constraints on the values of p, E_{max}, and E_{min}. `b` and `p` are of type `":num"`; `Emax` and `Emin` are of type `":integer"`.

```
new_definition(`b`,"b = @n.(n=2)\/(n=10)");;

new_definition(`single`,
"single p emax emin = (INT(5*p) below (emax minus emin))");;

new_definition(`sig`,
"sig p = ((b = 2) ==> (17 < p)) /\ ((b = 10) ==> (5 < p))");;
```

The definition of `b`, the radix, is in accordance with the first constraint, Section 2. The second constraint is addressed by the predicate `single`. The definition of `sig` is an algebraic simplification of the third constraint. In order to be complaint with

IEEE-854, it is necessary to show that the predicates `sig` and `single` are true for the corresponding instantiation of `p`, `Emax` and `Emin`. We have proven in HOL that the values for single precision in Standard IEEE-754 comply with the predicates `sig` and `single`.

3.1 Floating-point numbers

The meaning of a floating-point number, a value, positive and negative infinite, and signaling and quiet NaNs, is represented in the HOL system respectively by,

"`finite (sign,Exp,dig)`"[1] with value $-1^{sign} * b^{Exp} * d_0.d_1d_2...d_{p-1}$,

"`infinite 0`" and "`infinite 1`"

"`NaN (signal,n)`" and "`NaN (quiet,n)`"

where `n` is an arbitrary natural number and `finite`, `infinite`, and `NaN` are type constructors that when applied to a triple of type `(num#integer#(num -> num))`, an element of type `num`, and a pair of type `(NaN_type#num)`, respectively, will return an element of type `fp_num`.
Before the floating-point type is defined, a type for signaling and quiet NaNs is defined,

```
define_type `NaN_type` `NaN_type = signal | quiet`;;
```

The floating-point type is defined by,

```
define_type `fp_num`
`fp_num = finite (num#integer#(num -> num)) |
          infinite num                      |
          NaN (NaN_type#num)`;;
```

The following definitions for identifying and manipulating floating-point(fp) numbers are used in the specification of floating-point operations.

```
new_definition(`is_finite`, "is_finite fp = (?X.fp = (finite X))");;

new_definition(`is_infinite`,
               "is_infinite fp = (?X.fp = (infinite X))");;

new_definition(`is_NaN`, "is_NaN fp = (?X.fp = (NaN X))");;

new_definition(`i_finite`, "i_finite fp = (@X.fp = (finite X))");;
```

[1] *dig* is a function which takes the position of the digit and returns the digit itself. For example $dig(2) = d_2$.

```
new_definition(`i_infinite`,
                "i_infinite fp = (@X.fp = (infinite X))");;

new_definition(`i_NaN`, "i_NaN fp = (@X.fp = (NaN X))");;
```

The first three definitions are predicates which return *true* when applied to a finite, infinite, and NaN fp number, respectively, and false otherwise. The last three definitions are the inverse of the respective type constructors and will return the argument of the constructor when applied to the appropriate fp number. The theorems,

```
|- !z.i_finite (finite z) = z
|- !z.i_infinite (infinite z) = z
|- !z.i_NaN (NaN z) = z
```

have been proven on these functions and illustrate the action of the inverse functions.

Floating-point numbers in HOL are restricted in accordance with IEEE-854 by the predicate,

```
new_definition (`precis_c`,
"precis_c emax emin fp =
(emin below_or_e (exponent fp))/\
((exponent fp) below_or_e emax)/\
(!n.(digits fp)n < b)");;
```

The predicate `precis_c` restricts the exponent value of an fp number to be within the minimum and maximum exponent values and the value of each digit in the significand to be between 0 and b-1 inclusive.

3.2 Rounding

Rounding is defined in the HOL system in an assertional style. The rounding function takes a real number, a rounding precision, and a destination precision predicate and returns a finite floating-point representation which complies with the IEEE-854 requirements. A function is used for each rounding mode. Round to near is defined by,

```
"round2near r p precis =
  (?fp1.precis fp1  /\
    (!fp.(precis fp) /\~(fp_value fp p  = fp_value fp1 p) ==>
     abs(fp_value fp1 p real_sub r) real_lt
              abs(fp_value fp p real_sub r))) =>
   (@fp1.precis fp1  /\
    (!fp.(precis fp) /\~(fp_value fp p = fp_value fp1 p) ==>
     abs(fp_value fp1 p real_sub r) real_lt
```

```
        abs(fp_value fp p real_sub r))) |
   @fp1.(precis fp1) /\
    (!fp.(precis fp) ==>
     abs(fp_value fp1 p real_sub r) real_le
        abs(fp_value fp p real_sub r)) /\
            (EVEN ((digits fp1)(p-1)))");;
```

The function `round2near` states that if there exists an *fp* with a unique value nearest to r, then return that *fp*. If there are more than one *fp* with values nearest to r then return an *fp* with value nearest to r and last digit even. `round2near` uses the function `fp_value` which extracts the value of a floating-point number returning a real number. We have shown in HOL that `|- is_zero (round2near 0 p precis)` and `|- is_zero fp ==> (fp_value (fp) = &0)`.

3.3 Arithmetic operations

Arithmetic operations on floating-point numbers are defined by performing the operations on the floating-point number values in the real numbers domain, and converting the result to a floating-point representation using the rounding definition.

The add definition is selected to represent the definition of arithmetic operations on floating-point.

```
new_definition (`fp_add`,
"fp_add fp1 fp2 p pr traps mode tiny acc emax emin =
(is_infinite fp1 /\ is_infinite fp2 /\ ~(fp_sign fp1 = fp_sign fp2))
                                   => (NaN(quiet,cn),invalid)      |
(is_infinite fp1)                  => (fp1,no_excep)               |
(is_infinite fp2)                  => (fp2,no_excep)               |
((fp_is_zero fp1)/\(fp_is_zero fp2)/\(fp_sign fp1 = fp_sign fp2))
                                   => (fp1,no_excep)               |
round ((fp_value (i_finite fp1) p) real_add
       (fp_value (i_finite fp2) p))
      pr traps mode tiny acc emax emin");;
```

Function `fp_add` takes two floating-point numbers, `fp1` and `fp2`, a rounding mode `mode`, and several other parameters[2]. It returns a floating-point and an exception flag. When both operands in the fp add operation are infinite and their algebraic signs are not equal the operation produces a quiet NaN, an invalid exception, and possibly invokes a trap handler. When both operands are infinite and their algebraic signs are

[2] `p` is the operand's number of significant digits, `pr` is the rounding precision, `traps` are the enabling and disabling flags for trap handlers, `tiny` and `acc` are the methods to detect underflow, and `emax` and `emin` are the destination precision maximum and minimum exponent. For a full explanation of these arguments see reference [2].

equal, or when one of the operand is infinite, the add operation produces an infinite fp number of the appropriate sign with no exceptions. When the operands are both finite, the values of the fp numbers are converted to reals, added with infinite precision using the `real_add` function and rounded to the destination precision according to the rounding mode.

The operands passed on to `fp_add` are always finite or infinite. NaNs are filtered by the function invoking `fp_add`.

4 PVS Specification

This section illustrates parts of the PVS specification of IEEE-854 [6]. The PVS prelude defines the real numbers as a base type that satisfies a standard set of axioms. The basic arithmetic operations are built-in and a number of pre-proven theorems about the real numbers are available in the PVS prelude. Many of these properties are also known by the PVS decision procedures. Other numeric types are defined as progressively smaller sub-types of the reals. For example, the rationals are defined as a sub-type of the reals that does not satisfy the Completeness Axiom. Similarly, the integers are defined as a sub-type of the rationals that is not closed under division.

In PVS, the parameters required by IEEE-854 [5] can be defined as parameters to the formal theory. Within a theory, the parameters are treated as constants of the appropriate type. By instantiating the following theory multiple times with different values for the parameters, we can readily define the different precisions allowed by the standard.

```
IEEE_854 [b,p:above(1),E_max,E_min:integer]: THEORY
  BEGIN

  ASSUMING
    Base_values: ASSUMPTION b=2 or b=10
    Exponent_range: ASSUMPTION (E_max - E_min)/p > 5 %10
    Significand_size: ASSUMPTION b^(p-1)>=10^5
%    E_balance: ASSUMPTION
%        IF b < 4 THEN E_max + E_min = 1 ELSE E_max + E_min = 0 ENDIF
  ENDASSUMING

%  Exponent_balance: LEMMA b^(E_max+E_min) <4 & 4<=b^(E_max+E_min+1)

  E_max_gt_E_min: LEMMA E_max > E_min

  IMPORTING IEEE_854_defs[b,p,E_max,E_min]

  END IEEE_854
```

This PVS theory has four formal parameters: b and p are constrained to be of type `above(1)`, i.e. $b, p \in \{i : int | i > 1\}$; E_{max} and E_{min} are unconstrained integers. The assuming section allows us to define constraints on the formal parameters. The above assumptions correspond directly to the constraints given by the standard. Any PVS theory that imports `IEEE_854` incurs proof obligations during typechecking to show that the actual parameters satisfy these assumptions. In the case of the instantiations required for IEEE-754 [4][3], PVS automatically verifies these assumptions for both single and double precision. The assumption `E_balance` and lemma `Exponent_balance` are commented out. These constraints are not strictly required by the standard. They are included here to indicate how the optional portions of the standard may be addressed within PVS.

Theory `IEEE_854_defs` imports all of the underlying theories containing the definitions of floating-point numbers and operations.

4.1 Definition of Floating-Point Numbers

Floating-point numbers are defined using the PVS abstract datatype mechanism [8]. The following PVS theory is parameterized as above, except that E_{min} is constrained via the dependent type mechanism to be strictly less than E_{max}.

```
IEEE_854_values
  [b,p:above(1),
   E_max:integer,
   E_min:{i:integer | E_max > i}]: THEORY

  BEGIN

  sign_rep: type = {n:nat | n = 0 or n = 1}
  Exponent: type = {i:int | E_min <= i & i <= E_max}
  digits: type = [below(p)->below(b)]

  NaN_type: type = {signal, quiet}
  NaN_data: NONEMPTY_TYPE

  fp_num: datatype
   begin
    finite(sign:sign_rep,Exp:Exponent,d:digits):finite?
    infinite(i_sign:sign_rep): infinite?
    NaN(status:NaN_type, data:NaN_data): NaN?
   end fp_num
```

[3]For single precision, the IEEE-754 parameters are: $b = 2$, $p = 24$, $E_{max} = 127$, and $E_{min} = -126$.

We use the predicate subtype mechanism to constrain our representation to the set of values required by the standard. Type `sign_rep` is the set $\{0, 1\}$. Type `Exponent` is defined to be the collection of integers between E_{min} and E_{max} inclusive; the restriction on E_{min} in the formal parameter list ensures that this type is non-empty. Type `digits` is the collection of functions from `below(p)` to `below(b)`, where the PVS prelude defines $\texttt{below}(n) : \text{TYPE} \ = \{i : nat | i < n\}$.

The definition of datatype `fp_num` states that the type of floating-point numbers is the disjoint union of three sets: finite numbers, infinite numbers, and Not a Numbers (NaNs). A finite number can be constructed (using constructor `finite`) from an algebraic sign, an integer exponent (in the appropriate range), and a significand; an infinity can be constructed from an algebraic sign; and a NaN can be constructed from a status flag (i.e. signal or quiet) and data undetermined by the standard.

The valuation function implied by the standard is defined in PVS by:

```
value_digit(d:digits)(n:nat):nonneg_real =
   IF n < p THEN d(n) * b ^ (-n) ELSE 0 ENDIF

value(fin:(finite?)): real =
  (-1) ^ sign(fin) * b ^ Exp(fin) * Sum(p, value_digit(d(fin)))
```

Where type `(finite?)` is a predicate sub-type of datatype `fp_num`, and `sign`, `Exp`, and `d` are the accessors for the sign, exponent, and significand fields of finite floating-point numbers. We initially verified that a few instances of finite floating-point numbers have the expected value to provide evidence that this definition is correct. The first proof attempt uncovered an error in our first definition of `value`.

We have since proven that the range of function `value` is correct. Namely, that the value of a floating-point number representing zero is 0 and that for every nonzero finite floating-point number, `fin`:

$$b^{(E_{min}-(p-1))} \quad \leq \quad |\texttt{value(fin)}| \quad \leq \quad b^{(E_{max}+1)} - b^{(E_{max}-(p-1))}$$

4.2 Rounding

The PVS specification first defines rounding an arbitrary real to an integer for each of the rounding modes required by the standard. This allows us to take advantage of the functions `floor` and `ceiling` to control the direction of rounding.

```
sgn(r:real): integer =
     IF r >= 0 THEN 1 ELSE -1 ENDIF

round_to_even(r:real): integer =
  IF r - floor(r) < ceiling(r) - r THEN floor(r)
  ELSIF ceiling(r) - r < r - floor(r) THEN ceiling(r)
  ELSIF floor(r) = ceiling(r) THEN floor(r)
```

```
    ELSE 2 * floor(ceiling(r) / 2)
    ENDIF

  round(r:real,mode:rounding_mode): integer =
    CASES mode of
      to_nearest: round_to_even(r),
      to_zero:    sgn(r) * floor(abs(r)),
      to_pos:     ceiling(r),
      to_neg:     floor(r)
    ENDCASES
```

The built-in PVS strategy[4] (grind) is able to prove that $|r - \texttt{round}(r\texttt{,mode})| < 1$ and that $|r - \texttt{round\_to\_even}(r)| \leq \frac{1}{2}$ [6]. We can use these definitions to round reals to p significant base-b digits by scaling the argument appropriately.

```
  scale_correct: LEMMA
      b ^ (p-1) <= px * b ^ (-scale(px)) &
       px * b ^ (-scale(px)) <  b ^ p

  fp_round(r, mode): real =
    IF r = 0 THEN 0
    ELSIF over_under?(r) then
      round_exceptions(r,mode)
    ELSE LET E = scale(abs(r)) IN
      b ^ E * round(r * b ^ (-E), mode)
    ENDIF
```

Analogous to the integer cases, we have proven in PVS that fp_round(r,mode) is within one least significant base-b digit of r (within $\frac{1}{2}$ for mode = to_nearest). We have also shown that the direction of rounding is correct [6]. These functions are combined with function real_to_fp which maps an appropriately rounded real to a corresponding floating-point representation. The inexact exception is signaled whenever $r \neq \texttt{fp\_round}(r\texttt{,mode})$. Function round_exceptions handles those cases where overflow or underflow may have occurred [6].

4.3 Arithmetic Operations

The basic definition for an arithmetic operation is illustrated by the following definition for fp_add; the definitions for fp_sub and fp_mult are nearly identical. Operation fp_div requires special treatment when fp2 denotes 0.

```
  fp_add(fp1, fp2, mode): fp_num =
```

[4] PVS strategies are analogous to HOL tactics.

```
IF finite?(fp1) & finite?(fp2) THEN fp_op(add, fp1, fp2, mode)
ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan(add, fp1, fp2)
ELSE fp_add_inf(fp1, fp2)
ENDIF
```

The function definition invokes one of three functions depending on the arguments. If both arguments are finite, then this function invokes the corresponding real function applied to the values of the arguments. If one argument is a NaN, then the rules for operations on NaNs are invoked. When one of the arguments is infinite, the result required by the standard is returned.

The definition of `fp_op` is given below. The function definitions for the other cases, including appropriate exception handling, are in [6].

```
apply(op,fin1,(fin2:fin| div?(op) => not zero?(fin))): real =
  cases op of
    add:  value(fin1) + value(fin2),
    sub:  value(fin1) - value(fin2),
    mult: value(fin1) * value(fin2),
    div:  value(fin1) / value(fin2)
  endcases

fp_op(op,fin1,(fin2:{fin| div?(op) => not zero?(fin)}),mode): fp_num
        = real_to_fp(fp_round(apply(op,fin1,fin2),mode))
```

Function `fp_op` calls function `apply` to perform the arithmetic operation and then rounds the result prior to converting the result to a floating-point number. Function `apply` uses the dependent type and predicate subtype mechanisms of PVS to restrict the domain of its third argument to nonzero numbers when the operation is division[5].

5 Comparison

This section discusses the most notable differences between the HOL and PVS systems encountered during the specification of IEEE-854. We made errors in both the HOL and PVS formalizations of IEEE-854. Our experience has been that we find such errors quickly within PVS. This is due to two main factors: (1) The PVS specification language is more expressive (primarily due to subtyping); and (2) The PVS prover is more effective (due to decision procedures).

5.1 Theories and abstract theories

The HOL system does not explicitly support abstract theories. A mechanism has been implemented in HOL by the library *Abstract Theory* [9] which permits the pa-

[5]This restriction was added to the original definition. PVS generated a TCC requiring that `fin2` be non-zero when the operator was `div`. This TCC was unprovable without the restriction.

rameterization of theories. However, instead of using the *Abstract Theory* library in HOL, the parameter b was declared to be a constant and the parameters p, E_{max}, and E_{min}, are arguments to functions. Making the parameters arguments to functions permits the definition of functions that operate on mixed precisions within the same theory.

Theories in PVS can be parameterized so that the formal parameters to the theory are treated as constants in the theory body. In the definition of IEEE-854, the values for b, p, E_{max}, and E_{min}, are used as the parameters to the PVS theory. The theory can be instantiated for any supported precision.

5.1.1 Assumptions

Restricting the value of parameters in the HOL system can be accomplished by using axioms or by defining predicates which are then used as assumptions in theory proofs. The second alternative was used in the HOL definition of IEEE-854.

A PVS theory may contain assumptions that restrict the parameter values. Within a theory, assumptions are similar to axioms defining properties of the theory parameters. When a theory with assumptions is imported by another theory, the assumptions generate proof obligations in the importing theory.

5.2 Subtyping

Subtyping is not supported in the HOL system. Real numbers, integers and naturals are all of different type. In order to perform comparisons and arithmetic operations between numbers of different type, explicit conversions must be made.

PVS defines the real numbers as a primitive type with the rationals, integers, and natural numbers as subtypes. Thus, for example, functions defined on rationals can accept integral arguments. While the subtyping mechanism adds a great deal of flexibility to the PVS specification language, it also makes the type system undecidable. When PVS is unable to determine if a declaration is well-typed, it generates proof obligations called Type Correctness Conditions (TCCs). These proof obligations must be discharged for PVS to fully admit a proof involving the corresponding function. Often, an error in a function definition results in an unprovable TCC. Inspection of such TCCs provides useful diagnostic information for debugging a specification.

5.2.1 Predicate Subtypes

HOL supports a limited form of predicate subtyping via restricted quantification. This is syntactic sugar, and we did not use it in the definition of IEEE-854.

In PVS, any predicate defined on type α may be used to declare a subtype of α. If PVS is unable to determine if a function argument is of the appropriate type, the system will generate a TCC to ensure that the arguments conform to the restrictions of the subtype.

5.2.2 Dependent Subtypes

PVS allows definitions to restrict the type of functions and arguments based on the value of other arguments. This allows a finer touch in defining functions. An example is the definition of `apply` in Section 4.3.

5.3 Abstract Datatypes

A new type declaration in the HOL system produces a theorem which can then be used to manually construct recognizers and function extractors. Recognizers and function extractor are automatically generated when using HOL with TkHolWorkbench. TkHolWorkbench is a graphical user interface being developed by Donald Syme.

An abstract datatype declaration in the PVS system produces several supporting definitions. Predicates recognizing the range of each constructor are automatically generated. Similarly, accessor functions, which extract the components of each constructor, are also defined. Many properties of abstract datatypes are automatically added to PVS' decision procedures.

5.4 Proofs

We have verified a number of properties of both specifications. In general, it was more difficult to prove these properties using HOL. PVS automates much of the proof effort, and a significant number of properties can be proven using just a few PVS prover commands. In addition, the PVS decision procedures automate the verification of most simple arithmetic properties.

As a simple illustration, consider the second constraint from Section 2, that $(E_{max} - E_{min})/p > 5$. A simple consequence is that $E_{max} > E_{min}$.

Using the HOL formulation of this constraint, a proof of this fact is:

```
g "(INT(5 * p)) below (emax minus emin) ==> emin below emax";;

e (ASM_CASES_TAC "INT 0 = INT(5 * p)");;
e (UNDISCH_TAC "INT 0 = INT(5 * p)" THEN
  DISCH_THEN ((\th.REWRITE_TAC[th]) o ONCE_REWRITE_RULE[EQ_SYM_EQ]));;
e STRIP_TAC;;
e (REWRITE_TAC [BELOW_DEF]);;
e (REWRITE_TAC [POS_IS_ZERO_BELOW;]);;
e (ASM_REWRITE_TAC[]);;
e (UNDISCH_TAC "~(INT 0 = INT(5 * p))" THEN
   DISCH_THEN (ASSUME_TAC o REWRITE_RULE [INT_ONE_ONE]));;
e (IMP_RES_TAC (DISJ_IMP (SPEC "5*p" LESS_0_CASES)));;
e (UNDISCH_TAC "0 < (5 * p)" THEN
```

```
   REWRITE_TAC[NUM_LESS_IS_INT_BELOW]);;
e (REPEAT STRIP_TAC);;
e (IMP_RES_TAC TRANSIT);;
e (REWRITE_TAC [BELOW_DEF]);;
e (REWRITE_TAC [POS_IS_ZERO_BELOW;]);;
e (ASM_REWRITE_TAC[]);;
```

This is probably not the most efficient HOL proof of this goal, but it is representative of an initial proof attempt.

In PVS, the proof consists of a single command:

```
  |-------
{1}   5 * p < E_max - E_min => E_min < E_max

Rule? (ground)
Applying propositional simplification and decision procedures,
Q.E.D.
```

In PVS, type information is available to the decision procedures. In this case, the critical information is that $p > 1$. This sequent also illustrates the ease of using different numeric types within the same PVS expression. E_{max} and E_{min} have type integer, p has type `above(1)`, and the arithmetic operators are defined over the reals.

As a result of the difference in proof effort, more properties have been proved of the PVS specification than of the HOL specification.

6 Concluding Remarks

We have presented a portion of our formalization of the IEEE-854 standard for radix-independent floating-point arithmetic in both HOL and PVS. This effort enabled us to highlight some differences between the two verification systems. In general, the specification language on each system has the expressiveness to represent all properties of IEEE-854. However, we found that PVS was a more natural environment for defining the standard. The specification language provided a rich collection of constructs that allowed for a straightforward definition of most aspects of the standard. The capabilities of the PVS prover greatly simplified the task of verifying putative challenges to our specification. The type system of PVS also aided in debugging the specification. We frequently encountered unprovable type correctness conditions (TCCs) in the course of defining IEEE-854 in PVS. In some cases these were due to typographical errors, but in other cases they illustrated conceptual oversights in our specification.

Thus far, the only proofs attempted have been to demonstrate that the specifications possess certain properties. There has been no attempt to verify the correctness of algorithms with respect to these specifications. Also, the specifications have only been reviewed by a few people. It is possible that some features of the standard have been overlooked. Prior to using these specifications in a serious verification effort, it will be necessary to subject them to a more rigorous review. Writing a specification in a formal language makes it easier for us to catch errors; it does not prevent us from making errors.

References

[1] Geoff Barrett. Formal methods applied to a floating-point number system. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[2] Victor A. Carreño. Interpretation of IEEE-854 floating-point standard and definition in the HOL system. To appear as NASA Technical Memorandum 110189, 1995.

[3] M. J. C. Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P. A. Subramanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer Academic Publishers, 1988.

[4] IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Std 754-1985.

[5] IEEE. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*, 1987. ANSI/IEEE Std 854-1987.

[6] Paul S. Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Memorandum 110167, NASA, Langley Research Center, Hampton, VA, July 1995.

[7] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[8] N. Shankar. Abstract datatypes in PVS. Technical Report CSL-93-9, Computer Science Laboratory, SRI International, Menlo Park, CA, December 1993.

[9] Phillip J. Windley. Abstract theories in HOL. Research Report LAL-92-07, Laboratory for Applied Logic, University of Idaho, June 1992.