Designing Language Support for IEEE 754

Joseph D. Darcy
Java Floating−Point Czar
Sun Microsystems, Inc.
joe.darcy@sun.com

Outline

- Programming Language Support for Traditional Floating−point Features
  - Arithmetic operations
  - Binary $\leftrightarrow$ decimal conversion
  - Multiple precisions
  - Math library
- Issues in managing IEEE 754 State
  - Desired functionality
  - Problems with unstructured designs (SANE, C99)
  - Benefits of structured designs (Borneo)

Assumptions

- Languages considered generally from Fortran and Algol offspring
  - Includes Fortran 77, C, Java, and C99
- Focus on Java derivatives for later parts of the talk
- Will provide history and recommendations for feature support
- Will frequently reference Borneo design

Philosophical Notes

The essence of programming is control.
––W. Kahan

- You should be able to:
  Say what you mean, mean what you say
  - able to express desired semantics
  - desired semantics are implemented by compiler/runtime
- Separate mechanism from policy
  - determine what you want to support, then decide best way to support it
- Choose "good" defaults

Floating−point Support

- Earlier language had to deal with a menagerie of different floating−point systems
  - VAX, Cray(s), IBM 360, ...
- *You are in a twisty maze of little passages, all different...*
- Language standards had difficulty in being precise about floating−point semantics
  - now with IEEE 754 universality, reasonable to have language support for IEEE 754 specific features

IEEE 754 Required Features

- Floating−point values
- Arithmetic operations
- Comparison
- Conversions
  - binary ↔ decimal
  - floating−point ↔ floating−point integer
  - floating−point ↔ integer
- Formats
- Sticky flags
- Rounding modes

Prologue: General Floating−Point Support

- Some languages require (some subset of) IEEE 754 (e.g. Java)
- Other languages provide recommended/suggested IEEE 754 bindings without *requiring* IEEE 754 (e.g. Modula 3, C99)
- Many languages are rather oblivious to the details of IEEE 754 in particular and floating−point in general

Floating−point values

- All finite values usually supported
  - (may have decimal ↔ binary conversion issues)
  - signed zeros might not be supported; explicitly forbidden by Fortran 77 spec
- NaNs and infinities may have reduced support
  - often lack a literal value analogous to `"2.0"`
  - may not have defined string representations
- Recommendation: all floating−point values, including −0.0, infinities, and NaN, should have full language support
  - language literals, defined string representations, etc.

Support for Arithmetic Operations

- The four basic arithmetic operations (+, −, *, /), usually have obvious mappings between the IEEE 754 operator and the language operator
  - some languages, e.g Scheme, use prefix instead of infix notation; i.e. `(+ a b)` instead of `a + b`
- Math library provides
  - square root
    - often hardware instruction, otherwise software (early Alphas)
  - possibly IEEE 754 remainder

Support for Comparison Operations

- Apart from the <, >, and == relations, NaN is *unordered* w.r.t. all floating−point values
- IEEE 754 defines the signaling status of comparison operations
  - ==, and != should be non−signaling
  - <, >, <=, >= should be signaling
- Languages usually leave signaling status of comparisons undefined
  - C99 has macros in `math.h` for non−signaling magnitude comparisons (`is(greater|less)[equal]`)

IEEE Recommended Functions

- IEEE 754 and 854 have slightly different semantics for certain methods (`logb`)
- No fundamental implementation issue in most languages
  - convenient bitwise conversion between floating−point and integer types is helpful
    - getting bitwise conversion in C is problematic; aliasing with arrays not guaranteed to work; unions also have problems
    - Java has explicit methods for this purpose; `Float.intBitsToFloat`, `Float.floatToIntBits`, etc.
    - can (almost) implement bitwise conversions using standard floating−point and integer operations

Binary ↔ Decimal Conversion

- Often left underspecified in language specifications
  - historically runtime and compile time conversions have been different (Fortran IV)
  - conversions have not been required to be correctly rounded
- IEEE 754's requirements should be replaced by some form of correctly rounded input and output
  - C99's hex constants are an interesting way to specify the significand precisely

Other Conversion Issues

- floating−point → floating−point integer
  - rounding mode
- floating−point → integer
  - rounding mode, signaling status

Supporting multiple formats

- Mapping from language types to IEEE 754 formats
  - fixed (e.g. Java)

    `float` → {single}
    `double` → {double}
  - variable (e.g. C99)

    `float` → {single}
    `float_t` → {single, double, double extended}
    `double` → {double}
    `double_t` → {double, double extended}
    `long double` → {double, double extended, non−IEEE format}
  - I Recommend mostly fixed mapping, as in Borneo

    `float` → {single}
    `double` → {double}
    `indigenous` → {double, double extended}

Expression Evaluation Policies

- Strict evaluation (e.g. Java)
  - *float op float* $\rightarrow$ *float*
    double op double $\rightarrow$ *double*
- Widest Available, operands widened (e.g. K&R C)
  - *float op float* $\rightarrow$ [*long*] *double*
    double op double $\rightarrow$ [*long*] *double*
- Scan for widest
  - useful for intervals and variable precision computations
  - widest available more useful for computations using only hardware−supported precisions

Borneo Expression Evaluation

- Strict evaluation by default
  - compatibility with Java
- Controlled widest available with scoped language declaration

```
anonymous double;
float f, g;  double d;
// full precision product
d = f * g;
```

```
anonymous indigenous;
double d, e;  indigenous h;
// possibly increased
// precision product
h = d * e;
```

Expressions and Functions

- Languages often distinguish between expressions and functions
  - special promotion rules and conversion rules for expressions
  - possibly different handling for method arguments
- Can be challenges and non−orthogonalities mixing expression evaluation policies and method resolution

Widest Available & Method Resolution: C99

- Functions are resolved according to "syntactic" type; expressions can use a different type

```
float a, b;
double d;
d = sin(a + b);
```

```
float a, b;
double d;
// assume widest available policy
d_double = sin(a_float + b_float);
```

- Expected runtime semantics

```
float a, b;
double d;
d_double = sin_double(a_float +_double b_float);
```

Possible Runtime Semantics

- Compile time method resolution

```
float a, b;
double d;
d_double = sin_float(a_float +_float b_float);
```

- Permissible runtime semantics

```
float a, b;
double d;
d_double = sin_float(a_float +_double b_float);
```

- Explicit source semantics of runtime behavior:

```
d = (double)sinf((float)((double)a + (double)b));
```

- Very surprising behavior!

Widest Available & Method Resolution: Borneo

- Assume `float`, `double`, and `indigenous` versions of `my_sin`

```
float a, b;
double d;
{ anonymous double;
  d = my_sin(a + b);}
```

- Source semantics

```
float a, b;
double d;
{
  // anonymous double;
  d = my_sin((double)a + (double)b);
}
```

- Overloaded version of `my_sin` taking a `double` will be called

More Borneo Method Resolution

- `anonymous` widens all floating-point arguments narrower than the target type to the target type
- to suppress this behavior for method resolution, use an explicit cast

```
float f; double d; indigenous n;

anonymous double
Math.abs(f);          // calls double abs
Math.abs(d);          // calls double abs
Math.abs((float)f); // calls float abs
Math.abs(n);          // calls indigenous abs
```

- `anonymous float` specifies strict evaluation

`anonymous` Details

- `anonymous` declarations are a short−hand notation for adding explicit widening (and narrowing) casts
  - numeric operands are widened to `anonymous` type
  - results automatically narrowed across assignments (usually illegal in Java otherwise)
- Values of literals widened after initial conversion
  - `"1.0f"` is an `anonymous double` context is not equivalent to `"1.0d"`

Fused mac

- Three fused mac policies to support
  - don’t use fused mac
  - must use fused mac
  - use fused mac if there is hardware support
    - issues of defining semantics from source with fused mac transformation; e.g. does `(a * b + c * d)` ⇒ `fmac(a, b, c*d)` or `fmac(c, d, a*b)`?
- Possible mechanisms
  - forbid fmac by default
  - explicit fmac/fma method call
  - scoped declaration/pragma (C99 `FP_CONTRACT`)

Desired Functionality for Rounding Modes and Sticky Flags

- Discussed in *Lecture notes on the Status of IEEE 754 Standard 754 for Binary Floating−Point Arithmetic*, W. Kahan
- Rounding modes
  - interval arithmetic
  - debugging roundoff problems
- Exception handling
  - use faster, simpler robust algorithms
    - Demmel and Li (linear algebra), Hull et al. (complex library)
  - "count mode"/extended exponents, presubstitution, terminate computation, breakpoint, IEEE default

Comments on go to to go on

- *Go To Statement Considered Harmful*, Dijkstra

... our intellectual powers are ... geared to master master static relations and ... our powers to visualize processes evolving in time are relatively poorly developed. For that reason *we should do ... our utmost to shorten the conceptual gap between the static program and the dynamic process*, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Repercussions

- Conclusion generally accepted as true in programming language community
  - structured programming movement and backlash
    - Labeled break and continues serves most common uses of goto (early loop termination)
    - structured programs yield *reducible flow graphs* — easier for compilers to analyze and optimize
  - Static versus dynamic scoping of variables
    - static scoping generally prevailed
      (emacs lisp still dynamic, other lisps support dynamic scoping with a special declaration)
    - tricks using dynamic scope judged not to justify overall cost (debugging harder, programmer confusion)

The Need for Speed

- Amdahl's Law: the overall benefit of speeding up a particular subtask is limited by the fraction of the total time that subtask takes
- Corollary as a design rule of thumb:
  make the common case fast and the fast case common
- Beware forcing the common case to be slow!
- Common case and fast case are *not* independent variables, especially if fast case changes
  - uniform consistent speed is much simpler to model
  - *We **should** forget about small efficiencies, say about 97% of the time; premature optimization is the root of all evil.*
    ––Knuth

Good Practices

- Make writing robust programs as simple as possible
  - They are hard enough to write as it is!
- Handling of exceptional cases must be reasonable

SANE/C99 model of IEEE 754 state

- Both environments present a "hardware" model of the IEEE 754 state as a mutable global register
  - functions to get/set rounding modes and sticky flags
  - convenience procedures to get/set entire floating-point state (`ProcEntry/ProcExit` in SANE, `fe{get, set, update}env` in C99)
  - few guarantees about rounding mode when entering a function
  - few guarantees that a called function won't have modified the caller's rounding mode upon its return
  - (C99's programming conventions request structured behavior; no compiler checking of compliance)

Rounding Mode Uses

- Primarily interval arithmetic
  - static modes (fixed at compile time) are fine
- Roundoff testing
  - variable at runtime; build support into debugger
- Find max, min, of polynomial at a given value
  - re−run expression with different rounding mode
  - scope of mode change is statically determined but value of mode is variable at runtime
- Don’t often need full generality of raw model

Coding Scenario

- Want to write robust, correct version of nextAfter
  - works under all rounding modes and trapping statuses
  - returns proper sticky flags/throws proper exceptions
- Approaches
  - write *permeable* method, accepts environment of caller and behaves appropriately
    - very tedious coding, requires runtime determination of rounding mode, different expressions to raise proper exceptions under different rounding modes
    - tedious to test and debug
  - bracket method call with `ProcEntry/ProcExit`
    - adds time overhead to common case

Borneo Rounding Modes

- Programmer initiated rounding mode changes must be made using rounding declarations:

```
{
  rounding Expression;
  Statement1;
  Statement2;
}
```

- Methods may assume they are called with round to nearest in effect
- Methods which use non−default rounding modes must restore round to nearest for callees

Benefits of Borneo Rounding Modes

- Simple programming model
  - supports common functionality
- Extra expense is incurred only when non−default rounding modes are used
- Eases compiler analyses
  - rounding mode specific optimizations possible
- Dynamic rounding modes via explicit argument passing
- Rounding mode can be forcibly changed dynamically using the debugger
- (Future extension could include the ability to declare `permeable` methods to fully emulate basic arithmetic operations)

Sticky Flags

- Across method calls, used to indicate something exceptional happened
  - e.g. sin(∞)=NaN and raises invalid
- Within a method, can use flag status to determine something undesirable has happened and another algorithm needs to be used
  - linear algebra examples in [Demmel and Li]

Sticky Flag Implications

- Sticky flag signature is part of method's contract with the user
  - analogous to the type and number of arguments the method takes, the type the return value, and what exceptions the method may throw
  - sticky flag signature should have formal support in the language, not just in comments
    - which of the callers flags are visible in callee
    - which of the callee flags are visible in the caller after the callee returns
- Also need to be able to isolate and control sticky flags within a method

Consequences of Flags

- Math library and other methods should have defined flag signatures
- Implementing flag signatures at all may have non-zero cost (saving/restoring flags overhead)
  - implementing *correct* flag signatures may be costly, subtle
- Preserving flag state of expressions will limit (or complicate) optimization efforts

Borneo Sticky Flag Signatures

- *`method_name`* `admits` *Conditions*
  `yields` *Conditions*
- *Conditions* = `overflow`, `underflow`, `divideByZero`, `invalid`, `inexact`, `all`, `none`
- Semantics for condition *f*
  - `admits` *f* — caller's value of *f* is visible in callee
  - `yields` *f* — callee's value of *f* is ORed into caller's value for *f* upon callee's return

Sticky Flag Signature Semantics

- Operational meanings of flag signatures
  - `admits all, yields all`:
    callee sees caller's flag state, caller's flag state reflects execution of callee
  - `admits all, yields none`:
    callee sees caller's flag state, caller's flag state preserved across call
  - `admits none, yields all`:
    callee gets clean flag state, caller's flag state reflects execution of callee
  - `admits none, yields none`:
    callee gets clean flag state, caller's flag state preserved across call

Implementation Actions

- Assuming a single global status register, actions needed for a given flag depend on both `admits` and `yields` status for a flag
  - `admit` *f*, `yield` *f*:
    no action on method entry, no action on method return
  - `admit` *f*, no not `yield` *f*:
    save *f* on method entry, restore value on method return
  - do not `admit` *f*, `yield` *f*
    save then clear *f* on method entry, OR current value of *f* with saved value on method return
  - do not `admit` *f*, do not `yield` *f*
    save then clear *f* on method entry, restore value on method return

Implementation Example

```
// Borneo code
void example()
  admits overflow, underflow yields invalid {
  Calculation
}
```

```
// Desugaring to Java with native methods
// (not guaranteed to work)
void example()
  /* admits overflow, underflow; yields invalid */ {
  int callersFlags = getFlags();
   // clear flags that aren't admitted
  setFlags(callersFlags & (OVERFLOW|UNDERFLOW))
  try {
    Calculation; }
  finally {
     // merge callee's invalid with caller's state
    setFlags((getFlags & INVALID) | callersFlags);}
}
```

Benefits of Flag Signatures

- User does not have to deal with tedious `setFlags`/`getFlags` code
- Makes properties of method explicit
  - Allows simple compiler optimization and analyses
- Not tied to current architectural styles

Flag Signature Defaults

- For semantics, the best default is probably `admits none yields all`
  - gives callee's clean slate so they only have to worry about internal flags
- Borneo chose `admits all yields all` for compatibility with extant Java implementations
  - no flag overhead in default case
  - this default doesn't require any extra analysis for inlining
- `yields none` could allow somewhat more aggressive optimization inside a method

Borneo Exceptions

- Floating−point traps map to Java−style exceptions in Borneo
- Throwing of floating−point exceptions turned on by lexically scoped `enable` clauses
- Floating−point exceptions are checked exceptions
  - methods must declare if exceptions escape
  - compiler uses simple analysis to infer what exceptions may be thrown
- Examples in Borneo document
  - presubstitution
  - extended exponent floating−point types

Exception Example

```
static WideExpDouble multiply(WideExpDouble x, WideExpDouble y)
{
   WideExpDouble product = new WideExpDouble(0.0);
   try {
      enable overflow, underflow;
      // the exp fields are integers
      product.exp = x.exp + y.exp;

      // the sig fields are double floating point numbers
      product.sig = x.sig * y.sig;} // multiply significands
   // e.doubleValue returns correct significand bits
   catch(OverflowException e) {
      product.sig = e.doubleValue();
      product.exp += Double.BIAS_ADJUST;}
   catch(UnderflowException e) {
         product.sig = e.doubleValue();
         product.exp –= Double.BIAS_ADJUST;}
   return product;
}
```

Exception Issues

- Java exceptions are non−resumptive
  - after an exception is thrown, will go to trap handler
- Java exceptions are synchronous
  - must appear to happen where the exception is thrown; i.e. program state must be consistent
- Exceptions could put constraints on code generation
- Changing trapping status is expensive on many architectures
- Recommendation: if had to chose between flags XOR exceptions, chose flags

Complexities of `complex`

- Utility of a separate `imaginary` type
- Defining all exceptional cases (and sticky flags)
  - IEEE 754 affine infinities not the most natural
  - various possible definitions of complex infinity, comparison
- Not practical to get correctly rounded complex multiply or divide
  - what divide formula to use?
- Branch cuts for complex math library
  - hard to implement, sticky flag signatures

References

- *Apple Numerics Manual*, Second Edition, Apple, Addison−Wesley Publishing Company, Inc., 1988.
- Joseph D. Darcy, *Borneo 1.0: Adding IEEE 754 floating point support to Java*, M.S. Thesis, University of California, Berkeley 1998.
  `http://www.cs.berkeley.edu/~darcy/Borneo/`
- Joseph D. Darcy, *Writing robust IEEE recommended functions in "100 % Pure Java"*™, UC Berkeley technical report UCB//CSD−98−1009.
  `http://www.cs.berkeley.edu/~darcy/Research/ieeerecd.ps.gz`
- James W. Demmel and Xiaoye Li, Faster numerical algorithms via exception handling, IEEE Transactions on Computers, vol. 43, no. 8, August 1994, pp. 983−992.
- Edsger W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, vol. 11, no. 3, March 1968, pp. 147-148.

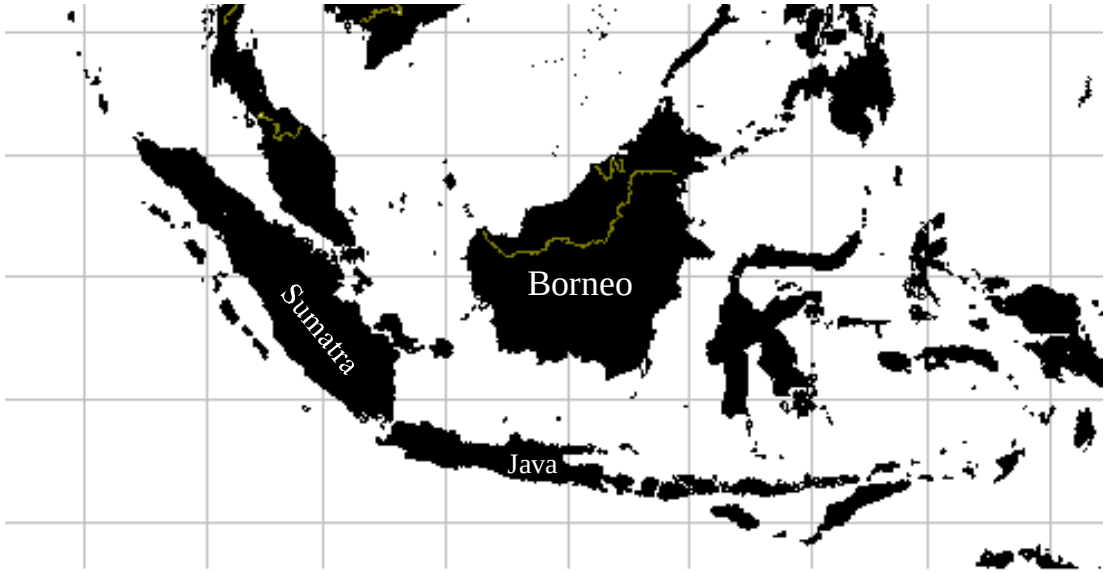
More References

- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, *Implementing Complex Elementary Functions Using Exception Handling*, ACM Transactions on Mathematical Software, vol. 20, no. 2, June 1994, pp. 215−244.
- T. E. Hull, Thomas F. Fairgrieve, Ping Tak Peter Tang, *Implementing the Complex Arcsine and Arccosine Functions Using Exception Handling*, ACM Transactions on Mathematical Software, vol. 23, no. 3, September 1997, pp. 299−335.
- ISO/IEC, Programming Languages −− C, ISO/IEC 9899(E), 1999
- William Kahan, *Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit*, Chapter 7 in The State of the Art in Numerical Analysis, ed. A. Iserles and M.J.D. Powell, Clarendon Press, Oxford, 1987.

Still More References

- W. Kahan, *Lecture notes on the Status of IEEE 754 Standard 754 for Binary Floating−Point Arithmetic*, `http://www.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`
- William Kahan and J. W. Thomas, *Augmenting a Programming Language with Complex Arithmetic*, UCB/CSD−91−667, University of California, Berkeley, 1991.
- Guy Steele, *New Models for Numerical Computing in the Java Programming Language*, Joint ACM Java Grande − ISCOPE 2001 Conference, Stanford University, California, June 2−4, 2001.
- Jim Thomas and Jerome T. Coonen, *Issues Regarding Imaginary Types for C and C++*, The Journal of C Language Translation, vol. 5, no. 3, pp. 134−138, March 1994.

Self–Promotion



`http://www.cs.berkeley.edu/~darcy/Borneo`

Borneo may be on the right track for Java.
––Guy Steele
New Models for Numerical Computing
in the Java Programming Language