Available online at [www.sciencedirect.com](http://www.sciencedirect.com/)



[Electronic Notes in Theoretical Computer Science 288 (2012) 15–23](http://dx.doi.org/10.1016/j.entcs.2012.10.004)

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Checking Compatibility of Bit Sizes in Floating Point Comparison Operations

Manuel F¨ahndrich[1](#_bookmark0) Francesco Logozzo[2](#_bookmark0)

*Microsoft Research, Redmond*

**Abstract**

We motivate, define and design a simple static analysis to check that comparisons of floating point values use compatible bit widths and thus compatible precision ranges. Precision mismatches arise due to the difference in bit widths of processor internal floating point registers (typically 80 or 64 bits) and their corresponding widths when stored in memory (64 or 32 bits). The analysis guarantees that floating point values from memory (*i.e.* array elements, instance and static fields) are not compared against floating point numbers in registers (*i.e.* arguments or locals).

Without such an analysis, static symbolic verification is unsound and hence may report false negatives. The static analysis is fully implemented in Clousot, our static contract checker based on abstract interpre- tation.

*Keywords:* Abstract Interpretation, Design by Contracts, Floating points, Numerical Abstract Domains, Static Analysis, .NET.

# Introduction

Comparing floating point values in programs can introduce subtle errors due to precision mismatches of the compared values. Precision mismatches arise as a result of truncating typically larger register internal floating point widths (80 or 64 bits) to the floating point width used when storing the value into main memory (64 or 32 bits). Such mismatches may produce unexpected program behavior, resulting in programmer confusion and—if ignored—unsound static program analysis.

We introduce the problem with the code snippet in Fig. [1](#_bookmark1), extracted from the *“classical”* bank account example annotated with contracts [[8](#_bookmark15)]. In this paper, we use C# as our language and the .NET runtime. However, the general problem ad- dressed in this paper is present in numerous programming languages and runtimes. We address these other contexts in Sect. [5](#_bookmark7).

1 Email:[maf@microsoft.com](mailto:maf@microsoft.com)

2 Email:[logozzo@microsoft.com](mailto:logozzo@microsoft.com)

1571-0661 © 2012 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

[doi:10.1016/j.entcs.2012.10.004](http://dx.doi.org/10.1016/j.entcs.2012.10.004)

public class Account

{

private float balance;

public Account(float initial)

{

Contract.Requires(initial >= 0.0f);

balance = initial;

}

public void Deposit(float amount)

{

Contract.Requires(amount >= 0.0);

Contract.Ensures(balance == Contract.OldValue(balance) + amount);

balance = balance + amount;

}

*// Other methods here*

}

Fig. 1. A C# code snippet for the classical Bank account example. Contract*.*Requires specifies the precon- dition, Contract*.*Ensures specify the postcondition, Contract*.*OldValue denotes the value of the argument expression at the entry point (not directly expressible in C#). It turns out that, as it is, the postcondition is incorrect, and it may fail at runtime for opportune values of amount.

The class Account represents a bank account. The method Deposit updates the balance by a given non-negative amount. The postcondition for Deposit states that on method exit the balance has been correctly updated. The current balance is stored in an instance field of type float. The ECMA standard requires *.*NET implementations of floating point types to follow the IEC:60559:1989 standard.

At a first glance, one expects the postcondition to hold and any static an- alyzer to easily prove it. In fact, a simple reasoning by symbolic propagation (balance0 denotes the value of the field balance at method entry) could be:

assert balance == balance0 + amount

*⇐⇒ {* by the assignment: balance = balance0 + amount *}*

assert balance0 + amount == balance0 + amount

*⇐⇒ {* by equality *}*

true

Unfortunately, a static analyzer for *.*NET performing this reasoning would be un- sound! For instance, the following two lines of C# code:

var account = new Account(6.28318548f); account.Deposit(3.14159274f);

cause a postcondition violation in the method Deposit (cf. Fig. [2](#_bookmark2)). What is wrong here? Let’s first rule out causes that are *not* the problem:

* Overflow can be excluded, as floating point numbers cannot overflow (at worst, operations result in special values *±∞* or NaN).
* Non-determinism is ruled out by the IEEE754 standard, and by the fact that the code in the example is single-threaded.
* Cancellation is to be ruled out too: *e.g.* the numerical quantities are positive and of the same order of magnitude.
* Floating point addition is commutative, so this is not the cause of the problem

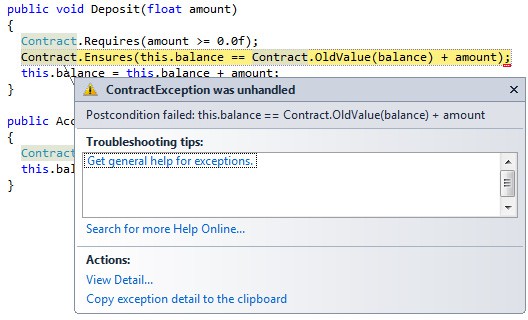


Fig. 2. A failure at runtime of the postcondition for Deposit.

either.

* + Addition is *not* necessarly associative, but we do not need associativity here (we are adding only two numbers).

The real culprit here is the equality test. In general all comparisons of floating point values are problematic. However it is still unclear at first sight why the comparison is a source of problems here: after all we are adding up the same two quantities and then comparing them for equality. If some rounding error occurs, then the same error should occur in both additions, or won’t it?

The reason for the unexpected behavior is to be found deeper in the specification of the Common Language Runtime (Partition I, Sect. 12.1.3 of [[1](#_bookmark8)]):

*Storage locations for floating-point numbers (statics, array elements, and fields of classes) are of fixed size. The supported storage sizes are* float32 *and* float64 *. Everywhere else (on the evaluation stack, as arguments, as return types, and as local variables)* ***floating-point numbers are represented using an internal floating-point type****. In each such instance, the nominal type of the variable or expression is either* float32 *or* float64 *, but its value can be represented internally with additional range and/or precision. The size of the internal floating-point representation is implementation-dependent, can vary, and shall have precision at least as great as that of the variable or expression being represented. An*

*implicit widening conversion to the internal representation from* float32 *or* float64 *is performed when those types are loaded from storage. [...]* ***When a floating-point value whose internal representation***

***has greater range and/or precision than its nominal type is put in a storage location, it is automatically coerced to the type of the storage location****.*

The standard allows exploiting the maximum precision available from the floating point hardware for operations on values in registers *despite* of their nominal type, *provided* that on memory stores the internal value is truncated to the nominal size. It is now easy to see why we get the postcondition violation at runtime.

The result of the evaluation of the expression this*.*balance + amount is in- ternally stored at the maximum precision available from the hardware (on Intel processors 80 bits registers, on ARM architectures 64 bits). In the example, the

|  |  |  |  |
| --- | --- | --- | --- |
| l = *k* | *(load const)* |  | |
| l = l1  l = l1 *op* l2 l = o*.*f | *(copy)*  *(binary op) (load field)* | l = (type) l1  o*.*f = l | *(cast)*  *(store field)* |
| l = a[li] | *(load array)* | a[li] = l | *(store array)* |

Fig. 3. The simplified bytecode language we consider for the analysis. A constant *k* can only be a constant belonging to the float32 or float64 ranges. Casting is allowed only to type *∈ {*float32 *,* float64 *}*.

result of the addition is 9*.*42477822, a value that cannot be precisely represented in 32 bits.

The successive field store forces the value to be truncated to 32 bits, thereby changing the value. In the example, 9*.*42477822 is coerced to a float, causing a loss of precision resulting in the value 9*.*424778 being stored in the field this*.*balance. When the postcondition is evaluated, the truncated value of balance is re- loaded from memory, but the addition in the postcondition is re-computed with the internal precision. Comparing these two values causes the postcondition to fail,

since 9*.*424778 */*= 9*.*42477822.

*Contribution*

We present a simple static analysis to check that floating point comparisons (equal- ities, inequalities) use operands of *compatible* types. When they are not compatible, the analysis reports a warning message to the user, so that all successive valida- tions should be understood as conditional. We fully implemented the analysis in Clousot, our static contract checker based on abstract interpretation for .NET [[2](#_bookmark9)]. We validated the analysis by running it on the base class library of .NET where it emitted 5 real warnings.

# The Language

We illustrate our analysis on a minimalistic bytecode language. We make some simplyfing assumptions. There are two kinds of variables: store variables (f*,* a *∈* S) and locals (l*,* p *∈* L). *Store* variables are instance fields, static fields and arrays. *Local* variables are locals, parameters, and the return value. Variables belong to the set Vars = S *∪* L. Aliasing is not allowed.

The language has only two nominal floating point types (float32 *,* float64 *∈* TN ) and one internal floating point type (floatX ) such that 64 *≤* X. On x86, floatX is float80, allowing extended floating point precision. Please note that the

*.*NET standard does not include a long double as for instance C [[3](#_bookmark10)], so application programmers have no access to floatX types.

All variables have a nominal floating point type. At runtime, the nominal float- ing point type for locals may be widened but not that of store variables. We say “may be widened”, as it depends on register allocation choices by the compiler. We think it is reasonable to force the code to compute values independent of floating point register allocation choices.

The simplified bytecode language is presented in Fig. [3](#_bookmark3). Floating point constants

|  |  |  |  |
| --- | --- | --- | --- |
| 0 : | l1 | = | this*.*balance |
| 1 : | l2 | = | amount + l1 |
| 2 : | this*.*balance | = | l2 |
| 3 : | l1 | = | this*.*balance |
| 4 : | lcomp | = | l1 == l2 |

Fig. 4. The (partial) compilation of the running example in our simple bytecode instruction set.

are loaded into locals *(load const)*. Admissible constant values are only those ad- mitted by the nominal types, *i.e.* floating point constants in 32 or 64 bits including special values as *±∞* and NaN (Not-a-number).

Values can be copied to locals, retaining their internal value *(copy)*. Casting is allowed only to nominal types, with values narrowed or widened as needed *(cast)*. In general it is not true that if l1 and l have the same *nominal* type then *(cast)* is semantically equivalent to *(copy)* as their *internal* type may differ.

Binary operations are the usual floating point arithmetic ones (+*, −, ∗, /*) and (unordered) comparison operations (==*, <, ≤,*) *(binary op)*. The result of a com- parison is 0*.*0 if the comparison is false, 1*.*0 otherwise.

Values are loaded from and stored to fields *([load/store] ﬁeld)*. We do not dis- tinguish between static and instance fields. Fields only contain values of nominal types: therefore, when storing a local into a field, its value is automatically nar- rowed to the field nominal type value. If the value of l is too large or too small, then it is approximated to *±∞* or to 0. Similarly, values read from arrays have a nominal type value and values written into arrays are narrowed to the nominal type of the array type. Arrays are indexed by local values, and in addition to the usual out-of-bounds checking, we assume that the computation stops also when li is a floating point number with non-zero decimal part or it is NaN.

**Example 2.1** The compilation to simplified bytecode of the body of method deposit (without contracts) of Fig. [1](#_bookmark1) is in Fig. [4](#_bookmark4). Please note that the store and load field operations are now made explicit in the bytecode.

# The Abstract Semantics

* 1. *Abstract Domain*

The abstract domain T we use captures the potential runtime floating point width a variable may have, which may be more precise than its nominal type. Therefore, the elements of T belong to the set Vars *−→* TX where 64 *≤* X and TX is the abstract domain:

7*T*,7¸,¸

float3¸2 ¸

floa,t6,4

fl7oa7tX

*⊥*

|  |  |  |
| --- | --- | --- |
| Jl = *k* )(*τ* ) | = | *τ* [l *'→ η*(l)] |
| Jl = l1)(*τ* ) | = | *τ* [l *'→ τ* (l1)] |
| Jl = (type)l1)(*τ* ) | = | *τ* [l *'→* type] |
| Jl = l1*op*l2)(*τ* ) | = | *τ* [l *'→* floatX ] |
| Jl = o*.*f)(*τ* ) | = | *τ* [l *'→ η*(o*.*f)] |
| Jo*.*f = l)(*τ* ) | = | *τ* |
| Jl = a[li])(*τ* ) | = | *τ* [l *'→ η*(a[*·*]) |
| Ja[li]= l)(*τ* ) | = | *τ* |

Fig. 5. The abstract semantics. The function *η* returns the nominal type of a variable.

If X = 64, *i.e.* the hardware does not provide any wider floating point register, then float64 and floatX co-incide. This is the case on ARM architectures, but not for x86 architectures which provide extra precision registers.

The operations of the abstract domain T (order, join, meet) are the functional pointwise extensions of those on the lattice above. No widening is required as the lattice is of finite height.

* 1. *Abstract Semantics*

The abstract semantics J*·*) *∈* P *×* T *−→* T statically determines, at each program point an internal type for each local variable. Store variables are known, by the ECMA standard, to have their nominal type coincide with the internal type.

The abstract transfer function is defined in Fig. [5](#_bookmark5). The only constant values that can be explicitly represented are those admissible as float32 or float64 values: the internal type of a local after a load constant is its nominal type. Variable copy retains the internal type. The ECMA standard guarantees that casting a value *v* to type truncates the value *v* to one in the type range. If *v* is too large or too small for type then it is rounded to *±∞* or 0. The result of a binary operation is a value of maximum hardware precision, which we denote by floatX . Reading from a field or an array location provides a value of the nominal type (no extra precision can be stored in fields). Writing into a field or an array location causes the truncation of the value to the corresponding nominal type.

**Example 3.1** For the bytecode in Fig. [4](#_bookmark4), with *τ*0 = [amount *'→* floatX ], the inferred internal types after each program point are:

0 : *τ*1 = *τ*0[l1 *'→* float32 ]

* + 1. : *τ*2 = *τ*1[l2 *'→* floatX ]
    2. : *τ*3 = *τ*2
    3. : *τ*4 = *τ*3[l1 *'→* float32 ]
    4. : *τ*5 = *τ*4[lcomp *'→* floatX ]
  1. *Checking*

Checking a program P for precision mismatch in floating point comparisons is now quite easy. First run the analysis JP) to collect an over-approximation of the internal types for each program point. Then, for each *(binary op)* in P

pp : l = l1 *op* l2

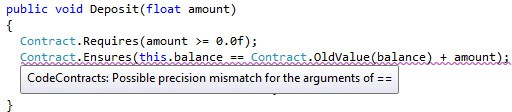


Fig. 6. The warning emitted by Clousot.

such that *op* is one of ==*, ≤,<* get *τpp*, which is the abstract pre-state for pp.

If *τpp*(l1) or *τpp*(l2) are different from *T*, and *τpp*(l1) = *τpp*(l2) then the com- parison is on variables with the same internal type. Otherwise, the comparison may happen on floating point values of different width, and hence a warning to the user should be emitted.

**Example 3.2** In our running example, *τ*5(l2) = floatX and *τ*5(l1) = float32 . So a warning is emitted to the user (cf. Fig [6](#_bookmark6)).

* 1. *Fixing the warnings*

When the analysis cannot prove that the operands of a comparison are of the same type, the user can fix it by adding explicit casts. For instance in our running example, the right postcondition is one where the type coercion is made explicit:

Contract.Ensures(balance == (float)Contract.OldValue(balance) + amount);

From a programmer’s point of view, this coercion seems redundant, as the expres- sion Contract*.*OldValue(balance)+ amount already has nominal type float32 . He/she may expect that the explicit cast to float32 would be a no-operation. But for the reasons explained in this paper, the cast may be a truncation.

# Implementation and Experiment

We have implemeted the analysis described in this paper in Clousot, our static analyzer for CodeContracts [[2](#_bookmark9)]. The analyzer first reads the IL from disk, then con- structs for each method the control flow graph, inserting contracts at the necessary points. Then it simplifies the program by getting rid of the evaluation stack and the heap, reconstructs expressions lost during compilation, and finally produces a *scalar* program.

Several analyses are run on the top of the scalar program, *i.e.* non-null, numeri- cal, array, or arithmetic. We added the detection of precision mismatches in floating point comparisons described in this paper to the arithmetic analysis. The analy- sis is fast and accurate: on the core library of the *.*NET framework, mscorlib*.*dll, constisting of 25089 methods, it adds less than 10 seconds to the total time, and it reports 5 warnings. We manually inspected those, and they all represent real warnings similar to the following example:

bool IsNonZero(float f) {

return f != 0.0F;

}

According to the ECMA standard, the parameter f of callers may be passed in registers and thus use more bits than float32 in memory. Therefore, the inequality comparison against 0*.*0F may result in true, even though f truncated to float32 is equal to 0*.*0F. As an example of where this could result in problems, consider:

class C {

float a; *// should never be 0.0F*

void Update(float f) { if (IsNonZero(f)) {

this.a = f;

}

...

In the above code, the programmer expects to guard the assignment to the field to make sure the value stored in the field is never 0*.*0F. However, due to register allocation of f, a value represented using more than 32 bits, close to 0*.*0F but not equal to 0*.*0F can pass the test and be truncated to 0*.*0F when stored into this*.*a.

# Discussion

Some languages and compilers have addressed the problem described in this paper using other means. C compilers typically offer compile-time switches to control whether the compiler should emit code that adheres strictly to the bit-widths de- clared in types, or whether it is allowed to use extra precision for computations in registers.

Using strict adherence to the declared types requires compilers to emit trun- cating casts after each floating point operation that could result in more precision, or putting the floating point unit into a mode that automatically performs trunca- tion [[10](#_bookmark17)]. The Java standard provides the strictfp keyword to enforce this trun- cating behavior and thereby avoids the problem described in this paper entirely at the cost of precision and speed.

We decided to detect problems with precision mismatches only when comparing floating point numbers. Another view would be to write an analysis that warns users whenever some higher precision float is implicitly cast to the nominal type width (effectively whenever a result is stored into memory). The problem with such an alternative analysis is that it would a) warn about most memory writes, b) miss implicit narrowings arising due to register spilling into memory introduced by the compiler. We found that warning about comparisons addresses the problem in a more actionable way.

# Related Work

Previous work in this area focuses on enhancing static analysis techniques to soundly analyze programs with floating points. For instance [[4](#_bookmark11),[6](#_bookmark13),[5](#_bookmark12)] present static analyses to spot the source of imprecision in floating point computations. [[9](#_bookmark16)] introduces ideas to

extend numerical abstract domains to the analysis of IEEE 754-compliant floating point values and [[7](#_bookmark14)] introduces program transformation techniques to reduce the error in floating point computations.

Our work is orthogonal to those in that we are not interested in the actual *values* of the computation but only in detecting situations that may cause unex- pected comparison results. The need for such detection is particularly important to avoid runtime failures of contracts that seemingly have been validated via symbolic execution.

# Conclusions

We described a simple static analysis to detect the absence of “surprising” behaviors of comparisons involving floating point numbers. The analysis is motivated by feeedback from users of our Code Contract tools. They reported false negatives similar to the example described in the introduction. We integrated the analysis in the static checker tool for Code Contracts, and experience shows it to be accurate enough.

# Acknowledgement

Many thanks to Matthias Jauernig, who was the first to report the unsoundess in Clousot illustrated here. Thanks also to Juan Chen for the detailed explanation of the floating point subsystem in the ARM architecture.

# References

1. ECMA. Common Language Runtime specification.
2. M. F¨ahndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS’10*, LNCS. Springer-Verlag, 2010.
3. American National Standard for Programming Languages. The C language specification, 1990.
4. E. Goubault and S. Putot. Static analysis of finite precision computations. In *VMCAI*, pages 232–247, 2011.
5. M. Martel. Propagation of roundoff errors in finite precision computations: A semantics approach. In

*ESOP*, pages 194–208, 2002.

1. M. Martel. Static analysis of the numerical stability of loops. In *SAS*, pages 133–150, 2002.
2. M. Martel. Program transformation for numerical precision. In *PEPM*, pages 101–110, 2009.
3. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.
4. A. Min´e. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP’04*.
5. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.