X3J11/94-024

17. Issues Regarding Imaginary Types for C and C++*

Jim Thomas and Jerome T. Coonen

Taligent, Inc.
10201 N. DeAnza Blvd.
Cupertino, CA 95014-2233

C-Labs
4035 Orme Street
Palo Alto, CA 94306

Abstract

This article is about the current effort to extend C and C++ to support computation in the complex domain, where values have real and imaginary parts. One approach, dating back to early versions of Fortran, adds complex data types as ordered pairs of real numbers and specifies the arithmetic of these types. A more modern approach, reflecting advances in IEEE standard real arithmetic, adds both complex and imaginary data types and specifies arithmetic on any combinations of real, imaginary, and complex operands. This simple refinement of the traditional approach captures the completeness and consistency of IEEE arithmetic at very little cost.

An Example

To contrast the two approaches, consider the product $2.0i \times (\infty + 3.0i)$. IEEE arithmetic defines infinity representations, using them for results of overflow and division by zero. With the traditional approach, $2.0i$ would have to be represented as a complex value, $0.0 + 2.0i$, so that

$$\begin{aligned} 2.0i \times (\infty + 3.0i) &\Rightarrow (0.0 + 2.0i) \times (\infty + 3.0i) \\ &\Rightarrow (0.0 \times \infty - 2.0 \times 3.0) + (0.0 \times 3.0 + 2.0 \times \infty)i \\ &\Rightarrow NaN + \infty i \end{aligned}$$

requiring four multiplications and two additions to obtain an undesirable result. IEEE arithmetic defines NaN (*Not-a-Number*) representations and uses them for results of invalid operations. Here, the invalid operation, $0.0 \times \infty$, arises solely as a byproduct of having to introduce the zero term to represent $2.0i$ as a complex value. The more modern approach, which mirrors conventional mathematical practice, allows $2.0i$ to be represented as a pure imaginary value and specifies the product without converting the imaginary operand to complex, so that

$$\begin{aligned} 2.0i \times (\infty + 3.0i) &\Rightarrow 2.0i \times \infty + 2.0i \times 3.0i \\ &\Rightarrow -6.0 + \infty i \end{aligned}$$

yielding the desired result, with just two multiplications. A similar example illustrates how the problematic zero term introduced for complex representation can pollute the sign of zero (IEEE arithmetic specifies signed zeros) and lead to subtle discrepancies in functions that are discontinuous at zero.

The decision to pursue one approach or the other must be based on costs and benefits. The increasing prevalence of IEEE hardware and recent achievements in specifying C support for IEEE real arithmetic highlight the limitations of the traditional approach, and the benefits of the modern one. Specifying and implementing imaginary types (and mixed type operations) is a relatively minor, straightforward undertaking. Notably, this natural introduction of the imaginary types offers surprisingly high leverage:

*Reprinted from *The Journal of C Language Translation*, Volume 5, Number 3, March 1994, pages 134–138. Copyright © 1994, I.E.C.C.

X3J11/94-024

The completeness and consistency, achieved for IEEE real arithmetic at a heavy cost in detailed specification, carries over in large part automatically to the complex domain.

Background

At its initial meeting in 1989, the Numerical C Extensions Group (NCEG), now incorporated into the ANSI C language committee (X3J11), targeted complex arithmetic as one of the areas requiring supporting extensions, in order for C to become suitable for general numerical programming. The first NCEG complex proposal, "Complex Extension to C" by Knaak, now revision 10 (X3J11.1/93-049), follows the traditional approach. "Augmenting a Programming Language with Complex Arithmetic" by Kahan and Thomas (NCEG/91-039) explains the problem with this approach and points to a solution based on imaginary types.

"Complex C Extensions" by Thomas (X3J11.1/93-048), CCE here for short, follows this more modern approach. This specification is designed for compatibility with "Floating-Point C Extensions" by Thomas (WG14/N319, X3J11/94-003), which is the technical report of X3J11's FP/IEEE subcommittee. Although written for C, CCE was designed with C++ compatibility in mind and has been prototyped in C++. "A Proposal for Standard C++ Complex Number Classes" by Vermeulen (X3J16/93-0165, WG21/N0372) follows the traditional approach.

The IEEE floating-point standards established a valuable level of completeness and consistency in treatment of special cases for real arithmetic. The term *special cases* here denotes exceptional operations that create infinities or NaNs, operations with infinity or NaN operands, and operations involving signed zeros. The term *completeness* refers to the property that all operations yield well-defined results, which behave in well-defined ways in subsequent operations. Completeness depends on consistent treatment of special cases. Imaginary types enable a clean solution to the problem of extending the treatment of special cases from the real domain to the complex domain. For IEEE implementations this means the special values—infinities, NaNs, and signed zeros—can behave in complex arithmetic as one reasonably would expect based on their behavior in real arithmetic.

With imaginary types, CCE is able to solve the sorts of semantic/efficiency problems illustrated in the earlier example in a natural, straightforward way, and so promote consistent treatment of special cases across the real and complex domains. On the other hand, there is no evidence that a traditional specification without imaginary types (or a commensurate mechanism) could provide consistent treatment of special cases. Despite this clear advantage of imaginary types a number of issues have been raised in their regard.

Issues

The traditional approach need not preclude consistent treatment of special cases. Indeed, infinities, NaNs, and signed zeros could be regarded as outside the specification's model, which would admit any treatment, including what CCE requires. But having an incomplete model would leave the work of determining consistency to yet another specification, which would have to be retrofitted to the deficient one. Such an approach might be reasonable provided that only a small minority of implementations supported special values, but in fact a most implementations, and almost all new ones, do support them.

Special cases are rare. Exceptional cases are rare, but robust software must account for them. Wherever a program has to test for special cases, the cost of those tests is incurred even in normal cases. If the language/arithmetic specifications cannot assure predictable, non-terminating treatment of special cases, the robust program must pretest before every operation possibly affected by special cases. The issue here is a general one about the treatment of special cases, and not particularly about complex arithmetic. IEEE arithmetic demonstrates the advantage of predictable, non-terminating treatment of special cases.

338

Although signed zeros are not rare, the sign usually doesn't matter. Still, careful treatment of the sign of zero contributes to overall consistency, and, in particular, allows certain simple algorithms to work more often (see "Augmenting a Programming Language with Complex Arithmetic" by Kahan and Thomas for examples). The imaginary types naturally provide reasonable semantics for signed zeros, as well as for infinities and NaNs; thus no additional specification is required to deal with the sign of zero.

Special cases for complex arithmetic are more complicated than for real arithmetic. Complex infinities are particularly problematic. Two topologies are used commonly in complex analysis: the complex plane with its continuum of infinities, and the Riemann sphere with its single infinity. The complex plane is better suited for transcendental functions, the Riemann sphere for algebraic functions. CCE uses the natural augmentation of the real and imaginary axes with signed infinities. This provides a useful (though admittedly imperfect) model for the complex plane. CCE prescribes a projection function mapping all infinities to one, which helps model the Riemann sphere. For complex or real arithmetic, the approach to special cases should be the same: specify behavior that is as often as possible what is desired (exceptional cases can't have universally desirable behavior, by definition) and that has a consistency to facilitate programming alternate behaviors.

The complex facility should be simple. Simplicity is a goal, most importantly for the user, and less so for the implementation and language specification. As additional types, imaginary types do complicate the specification to some degree. However, CCE demonstrates that the specification is still straightforward. No one has protested that imaginary types are difficult to implement. The modest cost of implementing imaginary types is substantially offset because they obviate certain optimizations otherwise required for acceptable performance. But, most importantly, a system with consistent treatment of special cases will be simpler for the user (programmer).

New programmers, not already trained in Fortran style complex numbers, will not be surprised to find imaginary types; textbooks commonly introduce complex numbers as sums of real and imaginary terms. Such programmers might well regard the modern approach as the more traditional one from a mathematical point of view. In the anticipated common programming model, imaginary and complex values will be expressed in the natural mathematical style, `y*I` or `x + y*I`, where `x` and `y` are real and `I` is the predefined imaginary unit constant in CCE; variables generally will be declared real or complex. (Programmers wedded to Fortran style notation can define a macro such as `complex(x,y)`.) Thus, the user gains the simplicity of consistent treatment of special cases without losing notational convenience and without even having to think about imaginary types or their representation.

The IEEE standards do not specify complex arithmetic. For expediency, the IEEE standards specify just basic arithmetic, and leave related areas such as transcendental functions and complex arithmetic to the natural extension of the standards' principles, as the state of the art permits. Application to these areas was a prime consideration of the committees in their development of basic IEEE arithmetic. The principles, including consistent handling of special cases, clearly apply to complex arithmetic.

Fortran does not support imaginary types. Fortran's complex arithmetic predates the IEEE standards, so would not be expected to accommodate their treatment of special cases. Ada does support imaginary types.

Are types the right mechanism for the problem? Values on the imaginary axis, like values on the real axis, require different treatment than for generic complex representations, in order to meet reasonable expectations of behavior and performance. The use of types to obtain desired semantics and performance for distinguished subsets of values is well established. Signed and unsigned integers, whose values are subsets of the reals, exemplify such types. So do the real floating-point types, whose values are subsets of the complex values.

Summary

The imaginary types afford complex arithmetic for C or C++ important advantages of completeness and consistency—particularly important given IEEE standard arithmetic. They align well with X3J11's proposed

X3J11/94-025

floating-point extensions and traditional mathematics, and are a natural extension to traditional Fortran style complex arithmetic. The issues that have been raised regarding imaginary types, upon consideration, either support the inclusion of imaginary types or else are insubstantial as arguments against doing so.

Jerome T. Coonen received the Ph.D. in mathematics from the University of California at Berkeley, largely for work related to the IEEE arithmetic standards. In nine years at Apple Computer and subsequently at C-Labs he has continued pursuit of his computational interests. He can be reached at jerome@apple.com.

Jim Thomas received the M.A. in mathematics from Cornell University and the M.S. in computer science from the University of California at Berkeley, and subsequently has worked on numerics and program development environments at Apple Computer and now at Taligent. He participated in IEEE arithmetic standardization, and recently authored NCEG's technical report on "Floating-Point C Extensions" and a proposal for "Complex C Extensions." He can be reached at jim_thomas@taligent.com.

∞