On an Incompatibility of Functional Programming and Functional Semantics

Bruce W. Weide

26 March 2008

In this short technical note—which admittedly will require considerable elaboration, discussion of related work, and references to create an acceptable scholarly work—I want to demonstrate that a functional programming system cannot simultaneously support both abstract data types and relational specifications.

Two Things Software Engineers Want

Two features of programming systems that software engineers have (or, by now, should have) found useful are abstract data types (ADTs) and relational specifications. I assume the former needs no explanation or elaboration. The value of the latter may be controversial, especially to functional programming advocates, so let me give a simple example to show why it is potentially important.

With a relational specification, you define an operation's desired behavior by giving an input-output relation that must be satisfied by any correct implementation—as opposed to defining the desired output as a function of the input. There are many interesting examples that better illustrate the advantages of allowing relational specifications, e.g., optimization problems that admit multiple solutions that are tied for best. The simple example used here is intended to show that, even without postulating a serious software engineering situation, it is possible to encounter a fundamental expressiveness problem that arises if you insist on a functional specification.

Consider an operation to compute the minimum of two objects a and b of arbitrary type T based on a total pre-ordering $\preccurlyeq$ of T. For current purposes, the salient property of such an ordering is that it is not necessarily antisymmetric (i.e., $a \preccurlyeq b$ and $b \preccurlyeq a$ does not entail $a = b$). For instance, a and b might be points in the plane with coordinates ($a.x$, $a.y$) and ($b.x$, $b.y$), respectively; and $a \preccurlyeq b$ might be defined as $a.x \leq b.x$, capturing the property "a is not to the right of b". If a and b lie on a vertical line, so that $a.x = b.x$, then both $a \preccurlyeq b$ and $b \preccurlyeq a$ are true; but $a = b$ does not follow because it is possible to have $a.y \neq b.y$.

Now, how can you write a formal specification for this operation? In the Resolve notation used in this report—a hypothetical functional-style Resolve 0.1—relational specifications of program functions are allowed. Given $\preccurlyeq$, you can write this relatively simple specification for *Min*:

```
function Min (a: T, b: T): T
    ensures
        Min ≼ a and Min ≼ b and (Min = a or Min = b)
```

There is no way to express the same behavior using a specification that is functional in form. To make it functional, you would have to decide which of a or b is "the" answer, when there is no rational basis for choosing and no way to express the choice other than relying on the accident of the order of the parameters. A relational specification, on the other hand, defers to an implementer to make such a decision based upon information about the (hidden) structural form of type T objects and about performance considerations.

One Thing a Functional Programming System Must Provide

A functional programming advocate might be willing to accept the idea that relational specifications are useful, and insist only that programs (as opposed to specifications) be functional. That is, you could allow as a correct implementation of a relational specification any function that is consistent with the specification in the usual sense of being "within the envelope" of the relation. Interpreting the relational specification as a set of allowable input-output pairs and the functional program purported to implement it as another set of input-output pairs (which happens to have the function property of being single-valued), you could declare that the function is a correct implementation of the specification if and only if, for each input where the relation is defined, the function is also defined and its output there results in an input-output pair that is among the pairs in the relation.

The main reason to insist nonetheless that programs be functional is simply this: there is an attractive simplicity that comes from treating the mathematical function "defined by" a program function *F* as a mathematical function *f*.[1] This is, indeed, how you normally define the semantics of a program function *F* in a functional language.

Now, here is the key property of any mathematical function *f* that you might use when reasoning about a client program that uses the corresponding program function *F*:

$$x = y \Rightarrow f(x) = f(y)$$

In other words, *f* really *is* a mathematical function. Every program function acts just like a mathematical function, so program functions can be composed with conventional (or, in some languages, funky) syntax and have intuitively conventional meaning *as if they were* the corresponding mathematical functions.

I plan to show that this fundamental and critical property is not valid if a functional programming system supports both (a) ADTs with abstraction functions that are not necessarily one-to-one (after all, requiring abstraction functions to be one-to-one means there is little point in allowing ADTs in the first place), and (b) relational specifications. This claim has nothing to do with total *vs.* partial functions, so suppose all the functions involved are total; and it has nothing to do with abstraction functions *vs.* abstraction relations, so suppose only abstraction functions are allowed.

An Illustration of the Problem Reconciling the Requirements

For purposes of exposition, imagine a "functional Resolve" language in which all program operations are functions, but where the **`ensures`** clause for a program function need not be given as an explicit functional expression as in the current versions of the actual Resolve language, but rather may be defined relationally as with *Min* above.

Consider this specification—which is certainly useless except to illustrate a point, so please do not try to figure out where one might use it in a real program:

```
contract Z4Facility
```

---

[1] I will use an identifier with an initial upper-case character for a program function name, and the same identifier with all lower-case characters for the mathematical function it "defines". I enclose the term "define" in quotes for reasons that will become clear later.

```
    type Z4 is modeled by integer[2]
        exemplar i
        constraint
            0 <= i  and  i < 4

    function C0 (): Z4
        ensures
            C0 = 0

    function C1 (): Z4
        ensures
            C1 = 1

    function C2 (): Z4
        ensures
            C2 = 2

    function C3 (): Z4
        ensures
            C3 = 3

    function F (i: Z4): Z4
        ensures
            i = (2 * F) mod 4  or
            i = (2 * F + 1) mod 4

    function G (i: Z4, j: Z4): Z4
        ensures
            G = (i + j) mod 4

end Z4Facility
```

For example, given this specification and supposing $i = 2$ and $j = 3$ (i.e., these are their abstract values of type *Z4*), one should be able to reason like this about a program function call $G(i, j)$:

$$G(i, j) = g(i, j) = g(2, 3) = (2 + 3) \; mod \; 4 = 1$$

Then what is $F(G(i, j))$? Well, it is $F(g(i, j)) = f(g(i, j)) = f(1)$, which the specification of F says may be either 0 or 2. (Alternatively, you can think of $F(G(i, j))$ as $f(G(i, j)) = f(g(i, j)) = f(1)$, giving the same result.) The proposed correctness criterion for a functional program implementing a relational specification such as that for F says any correct implementation of *Z4Facility* must be consistent and always evaluate $F(1)$ to be a fixed one of 0 or 2, because the language semantics is that f is a mathematical function.

There are, of course, many ways to implement this contract. For definiteness, consider layering it on top of the next contract, which incidentally includes only functional specifications.[3]

---

[2] In the current Resolve 0.1 syntax, this is written as **integer**. In the more mathematical style using Unicode rather than ASCII, which Bill and Murali prefer, it is $\mathbb{Z}$. Similarly with <= *vs.* $\leq$, etc.

[3] Note the specification of *Half* happens to be functional, but it is not manifestly so like the others, which are explicitly and syntactically functional. If you assume you have an integer division operator, then you can write it as an explicit function. Syntactically, though, it is virtually identical to the specification of *F* in *Z4Facility*, which is truly relational.

```
contract Z8Facility

    type Z8 is modeled by integer
        exemplar i
        constraint
            0 <= i  and  i < 8

    function Zero (): Z8
        ensures
            Zero = 0

    function One (): Z8
        ensures
            One = 1

    function Sum (i: Z8, j: Z8): Z8
        ensures
            Sum = (i + j) mod 8

    function Half (i: Z8): Z8
        ensures
            i = 2 * Half  or
            i = 2 * Half + 1

end Z8Facility
```

The proposed implementation strategy for *Z4Facility* is to represent an object of type *Z4* by using a single *Z8* object in which, if you will, the high-order bit is ignored. The implementations of *C0*, ..., *C3*, and *G* are obvious; the implementation of *F* "shifts the bits" of the representation object to the right, i.e., it divides the representation object by 2. Here's the code, in which the functional program expression for an operation returning type *Z4* (i.e., the new ADT being implemented in this realization) designates the returned value of the representation of the result, which is of type *Z8*:

```
realization Proposed for Z4Facility

    uses Z8Facility

    type Z4 is represented by Z8
        correspondence
            i = i.rep mod 4

    function C0 (): Z4 =
        Zero()

    function C1 (): Z4 =
        One()

    function C2 (): Z4 =
        Sum (One(), One())

    function C3(): Z4 =
        Sum (One (), Sum (One(), One()))

    function F(i: Z4): Z4 =
```

```
            Half (i.rep)

        function G (i: Z4, j: Z4): Z4 =
            Sum (i.rep, j.rep)

    end Proposed
```

I claim this realization is correct for any reasonable notion of the correctness of an ADT representation, and for the definition of the correctness of a functional programming implementation of a relational specification outlined earlier. That is, each function's implementation is self-evidently correct when considered in isolation.

Let's trace evaluation of the expression $F(G(i, j))$ through the concrete state space for *Z4*, assuming that $i.rep = 2$ (so $i = 2$) and $j.rep = 3$ (so $j = 3$). How can we have gotten to this situation? Simply, *i* can be (the result of evaluating) *C2*() and *j* can be (the result of evaluating) *C3*().

- $G(i, j)$ is evaluated, resulting in $G(i, j).rep = 5$, so $G(i, j) = 1$.
- $F(G(i, j))$ is then evaluated, resulting in $F(G(i, j)).rep = 2$, so $F(G(i, j)) = 2$.

This is fine as far as it goes: reasoning from the specification says the value of this expression may be either 0 or 2, and 2 it is.

Now, let's trace evaluation of the expression $F(G(i, j))$ through the concrete state space for *Z4*, assuming that $i.rep = 6$ (so $i = 2$) and $j.rep = 3$ (so $j = 3$). How can we have gotten to this situation, the only difference from above being that $i.rep = 6$ rather than $i.rep = 2$? Simply, *i* can be (the result of evaluating) $G(C3(), C3())$.

- $G(i, j)$ is evaluated, resulting in $G(i, j).rep = 1$, so $G(i, j) = 1$.
- $F(G(i, j))$ is then evaluated, resulting in $F(G(i, j)).rep = 0$, so $F(G(i, j)) = 0$.

Here's the problem we now face: $F(1)$ evaluates to 2 in one case, to 0 in the other. In other words, program function *F* does not actually "define" a mathematical function *f*. The fundamental characteristic of functional programming, that the semantics of the program is a mathematical function, is violated in this case.

A more realistic ADT example of the same phenomenon results from layering a functional-programming version of *SetTemplate* over a functional-programming version of *QueueTemplate* (where the former has a function with a relational specification, e.g., *Select*(*s*) evaluates to some abitrary element in *s*). The same thing can happen in many other situations with ADTs and relational specifications, so this is not a mere curiosity.