A myth in the modular specification of programs

K. Rustan M. Leino
7 November 1995

Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
rustan@pa.dec.com

When writing specifications of modular programs, two crucial elements are abstraction and **modifies** clauses. Without abstraction, information hiding is not possible; without **modifies** clauses, a specification must mention the variables that go unchanged explicitly, and doing so is prohibited because most of the variables are not in scope. Reasoning about modular specifications involves the difficult area of interpreting **modifies** clauses in the presence of abstraction. Such interpretation should be *sound*, meaning that program properties that are provable hold in every execution of the program.

Focusing on a commonly-thought-of-as-correct implementation of a given specification, I challenge the existence of a sound and reasonable methodology for proving such implementations correct, suggesting the legality of such implementations to be but a myth.

0 Introduction

In this section, I set the stage by presenting an example. The example is inspired by [3] and will lead us to the myth.

0.0 A set abstraction

Let $Data$ be some type, and consider the following declarations, written in the style of [1].

> **type** Set
> **spec var** $valid$: $Set \rightarrow \mathbb{B}$
> **spec var** $value$: $Set \rightarrow$ **set of** $Data$

This declares a type Set and two abstract fields (or, "specification variables") of Set objects. Field $valid$ represents the validity of a Set object, *i.e.*, some condition on the concrete representation of a Set object. Validity is established by some initialization method, and is required as a precondition by all other operations on Set objects. Field $value$ represents the abstract

value of Set objects. Since $value$ is declared as an abstract field, each implementation of Set objects decides on a concrete representation of $value$.

Here are the declarations of some of Set's methods. (The "self" parameter is shown as the first parameter to these methods.)

method $add(s\colon Set\ ;\ d\colon Data)$
  requires $valid[s]$
  modifies $value[s]$
  ensures $value_{post}[s] = value_{pre}[s] \cup \{d\}$

method $remove(s\colon Set\ ;\ d\colon Data)$
  requires $valid[s]$
  modifies $value[s]$
  ensures $value_{post}[s] = value_{pre}[s] \setminus \{d\}$

method $n\colon \mathbb{N} := size(s\colon Set)$
  requires $valid[s]$
  ensures $n = |value[s]|$

method $b\colon \mathbb{B} := member(s\colon Set\ ;\ d\colon Data)$
  requires $valid[s]$
  ensures $b \equiv d \in value[s]$

0.1 A default implementation of $member$

Consider now a default implementation of method $member$. This implementation is given in terms of the other methods and is therefore not tied to any particular implementation of Set. The idea is to attempt to remove the given element from the set to see if that affects the size of the set. If it does affect the size, the element was a member of the set, so it is inserted back into the set.

method impl $b\colon \mathbb{B} := member(s\colon Set\ ;\ d\colon Data)$
  var $oldSize$ **in**
    call $oldSize := size(s)$;
    call $remove(s, d)$;
    if $oldSize = size(s)$
      then $b := false$
      else call $add(s, d)$; $b := true$
    fi
  end

The quintessence of the proof of that this piece of code establishes its **ensures** clause, $b \equiv d \in value[s]$, is

$$|value[s] \setminus \{d\}| = |value[s]| \quad \Rightarrow \quad d \notin value[s]$$

for the **then** case and

$$d \in value[s] \cup \{d\}$$

for the **else** case, both of which can be proven from the properties of sets. However, this alone does not establish the correctness of the proposed *member* implementation—one must also prove that the implementation meets its **modifies** clause.

The specification of *member* has an empty **modifies** clause, yet the implementation contains invocations of *remove* and *add*, which may modify $value[s]$. The abstract value of $value[s]$ is restored, however, to its initial value before the method returns, as can be proven from the properties of sets.

0.2 The myth

From the above, one might conclude that the given implementation of *member* is correct. This is a myth. As alluded to above, the implementation does restore the abstract value of $value[s]$ to its initial value. But what about benevolent side effects?—The concrete representation of $value[s]$ may change as a result of invoking *remove* and *add*! This is where the particular interpretation of **modifies** clauses plays a rôle. Let me discuss a couple of possibilities.

1 Possible interpretation: Downward closure

In this section, I take the "downward closure" interpretation of **modifies** clauses (see [1]). To keep things simple, I won't worry about **new** allocations or the fancier kinds of representation dependencies described in [2] or [0]. Informally, the "downward closure" interpretation of

modifies m ,

where m is a list of possibly abstract designator expressions, is that m and its concrete representation are allowed to be modified, but nothing else.

In the *member* example, this means that *add* and *remove* are allowed to modify the abstract value and concrete representation of $value[s]$, but nothing else. For *member*, it means that no variable is allowed to change values from entry to exit of the method.

Thus, with this interpretation, $member$ should not verify—its implementation can shake up the abstract value and concrete representation of $value[s]$, but only the abstract value of $value[s]$, not the state of its concrete representation, can be proven to be restored.

To give an account of that this is a real worry, consider the following example. A possible implementation of Set keeps two fields, a and r, which keep track of the number of times an element has been added to the set and removed from the set, respectively. Initially,

$$value[s] = \{\} \wedge a[s] = 0 \wedge r[s] = 0 \quad .$$

Method $add(s, d)$ increments $a[s]$ when d is not in $value_{pre}[s]$, $remove(s, d)$ increments $r[s]$ when d is in $value_{pre}[s]$, and $size(s)$ simply returns $a[s] - r[s]$.

If $member(s, d)$ is invoked when $d \in value[s]$, the effect will be to increment both $a[s]$ and $r[s]$ by 1. However, this interpretation of **modifies** clauses allows one to prove a Hoare triple like

$$\{valid[s] \wedge a[s] = 7 \wedge d \in value[s]\} \quad \textbf{call } b := member(s, d) \quad \{a[s] = 7\} \quad , \qquad (0)$$

where b denotes a local variable, because the invocation of $member$, according to its specification, has no side effect on any program variable. A common reaction to this argument is to say that fields a and r are private to the implementation of Set and should not be accessible to general clients. Note, however, that the invocation of $member$ given in (0) may be found in the body of one of the other Set methods, in which case the fields can be accessed and the problem does arise.

We conclude that no sound modular programming methodology can both have this interpretation of **modifies** clauses and consider this implementation of $member$ to meet its specification. (I assume, for the purposes of this note, that every programming methodology works essentially like Hoare logic, modulo the translation of **modifies** clauses into postcondition contributions.)

2 Possible interpretation: Concrete variables are promiscuous

A different interpretation of **modifies** clauses is to augment the "downward closure" interpretation to always allow concrete variables to be changed. That is, a **modifies** clause tells only which *abstract* variables are and are not allowed to be changed; concrete variables are considered *promiscuous* and are implicitly allowed to modified by any method. The idea is that this would correct the problem by making it impossible to prove Hoare triples like (0).

I show that this interpretation suffers from the same problem as the interpretation above. Let $Natural$ be a type with an abstract field n: $I\!N$ and methods $increment$ and $getvalue$ with

the expected specifications. Let the fields a and r in the previous section have type $Natural$ instead of $\mathbb{N}$, and let $value[s]$ depend on $n[a[s]]$ and $n[r[s]]$ (such dependencies are called "dynamic" in [2]).

Because $value[s]$ depends on the abstract values of $n[a[s]]$ and $n[r[s]]$, methods $add(s, d)$ and $remove(s, d)$ are allowed to change the abstract values of $n[a[s]]$ and $n[r[s]]$. Yet, with this interpretation of **modifies** clauses, a Hoare triple like

$$\begin{array}{l} \{valid[s] \wedge d \in value[s]\} \\ \quad \textbf{call } x := getvalue(a[s]) ; \\ \quad \textbf{call } b := member(s, d) ; \\ \quad \textbf{call } y := getvalue(a[s]) \\ \{x = y\} \qquad , \end{array} \tag{1}$$

where x, b, y denote local variables, would verify. We conclude that no sound modular programming methodology can both have this interpretation of **modifies** clauses and consider the given implementation of $member$ to meet its specification.

One might imagine possible changes to this interpretation that would get around problems with being able to prove Hoare triples like (1). For example, one might say that, in addition to that concrete variables are always allowed to be changed, the state of these concrete variables (*e.g.*, $n[E]$ where n is a possibly abstract field and E is some concrete expression like $a[s]$) is allowed to be changed. Stated differently, we saw a problem with treating only concrete variables as promiscuous, so perhaps promiscuity should extend beyond concrete variables. Next, I show that any interpretation in which concrete variables are promiscuous rules out a common and useful programming paradigm.

Consider a type T that has two fields, x and y, and a method $transpose$, which swaps the x and y values of the object on which it is applied. The specification of $transpose$ looks like

$$\begin{array}{l} \textbf{method } transpose(t\colon T) \\ \quad \textbf{modifies } x[t], y[t] \\ \quad \textbf{ensures } x_{post}[t] = y_{pre}[t] \wedge y_{post}[t] = x_{pre}[t] \qquad . \end{array}$$

Let U be a subtype of T that has an additional concrete field, say a fixed-size matrix a, and a method m. According to its specification, m ensures some condition on the a, x, and y fields. Let's say that the most straightforward implementation of m first computes a new value of a, and then, if the determinant, say, of the new a is zero, swaps x and y. It would be nice if this implementation could invoke method $transpose$ for the swapping of x and y, especially if x and y were complex data structures. But, since a is concrete, the invocation of $transpose$ may have severe effects on the value of a, in which case m's computation of a would have been all in vain.

In this example, it is not possible for the specification of *transpose*, which appears in type T, to mention anything about field a explicitly, because a is not in scope there. Note, however, that it is possible that an implementation of *transpose* really does change the value of a because *transpose* may be implemented in some subtype of U. Therefore, the proposed programming methodology leaves only two outs: (i) compute the new value of a into a local array placed on the stack, then deliberate an invocation of *transpose*, and finally copy the local array into a, or (ii) compute a *in situ*, and then inline the swap of x and y without invoking method *transpose*. Neither alternative seems attractive. In summary, considering concrete variables to be promiscuous in the interpretation of **modifies** clauses leads to a programming methodology with undesirable restrictions on useful and common programming paradigms.

3 Conclusions

So is the quest for a sound programming methodology forever doomed? Not at all. Lacking at present any alternative, reasonable, sound interpretation of **modifies** clauses with which the given implementation of *member* verifies, we conclude simply that the given implementation of *member* ought to be considered illegal.

It would be unfair to end this note here without demonstrating a methodology that indeed detects the given implementation of *member* to be bad and without showing a specification of *member* in that methodology that does permit the given implementation. Fortunately, I know of just such a methodology, *viz.* the one in [1, 2, 0]. That methodology uses the "downward closure" interpretation of **modifies** clauses, and introduces in the generation of verification conditions so-called *residues*. The residue of an abstract variable stands for those variables on which the abstract variable depends but that are not in scope. An attempt at verifying the given implementation of *member* results in not being able to show that the residue of $value[s]$ returns to its initial value. If one wants to permit the given implementation of *member* to be valid, one must reflect that fact in the specification of *member* as follows.

$$\begin{array}{l}\textbf{method}\ b\colon \mathbb{B} := member(s\colon Set\ ;\ d\colon Data)\\ \quad \textbf{requires}\ valid[s]\\ \quad \textbf{modifies}\ value[s]\\ \quad \textbf{ensures}\ b \equiv d \in value_{post}[s] \wedge value_{post}[s] = value_{pre}[s]\end{array}$$

This specification explicitly gives *member* the right to modify the representation of $value[s]$, as long as such modification does not have any net effect on the abstract value of $value[s]$.

To recap, to specify modular programs one needs both abstraction and **modifies** clauses. By giving a precise interpretation of **modifies** clauses, like the interpretation introduced in [1], one finds out marvelous facts about programming methodology. This note revealed a myth in

the modular specification of programs and pointed out, once more, the importance of having a precise interpretation of **modifies** clauses.

4 Acknowledgements

I am grateful to Raymie Stata, Greg Nelson, and Dave Detlefs for discussions about this problem.

References

[0] D.L. Detlefs and K.R.M. Leino. Computing dependencies. KRML 61, Digital's Systems Research Center, March 1996.

[1] K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[2] K.R.M. Leino and G. Nelson. Beyond stacks. KRML 54, Digital's Systems Research Center, July 1995.

[3] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, October 1995. OOPSLA '95 conference proceedings.