Technical Report OSU-CISRC-5/08-TR26, May 2008

On Soundness of Verification for Software with Functional Semantics and Abstract Data Types

Derek Bronish, Jason Kirschenbaum, Bruce Adcock, Bruce W. Weide

The Ohio State University, Columbus, OH 43210, USA
{bronish,kirschen,adcockb,weide}@cse.ohio-state.edu
http://www.cse.ohio-state.edu

Abstract. A threat to the soundness of modular verification systems arises from the following combination of specification and programming language features: a semantics in which the denotation of every program operation is a mathematical function, the opportunity to write relational specifications for program operations, and support for abstract data types. There is no apparent practical workaround for this problem short of changing one of these features. After accounting for software engineering considerations, the recommendation is to relax the first one and to restrict the second, i.e., (1) to partition program operations into "function" operations and "procedure" operations; (2) to define the language semantics so the meaning of each function operation is a mathematical function, and to permit only a functional specification for a function operation; and (3) to define the language semantics so the meaning of each procedure operation is a mathematical relation, and to permit either a functional or a relational specification for a procedure operation.

1 Introduction

If a program is verified as correct relative to its specification, then—assuming the compiler and run-time system properly implement the language semantics—it should operate correctly when executed. A method for software verification that has this property is *sound*. Of course, a proof system that can never prove anything is sound; in practice, we are only interested in systems that are both sound and useful, preferably relatively complete.

The primary contributions of this paper are the identification, explanation, and analysis of a soundness problem for useful verification systems, which arises from the confluence of three standard features of modern specification and programming languages: denoting every program operation by a mathematical function, the opportunity to use relational specifications in describing the behavior to be verified, and support for abstract data types.

Section 2 begins with an overview of a modular verification of a simple program. In the interest of presenting real examples, we write the code in pure Lisp [1], but any (functional *or* imperative) language with functional semantics would do. Our choice of Lisp is advantageous because of its clear functional semantics [2] and because its later extension, CLOS [3], supports user-defined data types. We emphasize that we do not know of a full-strength verification system that has actually been proposed for CLOS, and note that the language would have to be augmented with a formal specification language for verification to be meaningful. Regardless of its somewhat hypothetical nature, though, the verification approach described here involves nothing that the reader familiar with other software verification methods is likely to find surprising. A rigorous but informal argument for a parameterized program's correctness is presented; in principle, it could be fully formalized, even mechanized. The reader is urged to check this argument carefully. The functions in the program have single-character names, making it less likely that some step in the argument should sneak by on the basis of "wishful naming". The reader is, in other words, thrust into a role similar to that of a mechanized proof-checker. In this section, one Lisp and one CLOS program are presented as instances of this verified parameterized program. The reader who accepts the arguments of Section 2 should have no doubt that these two programs are correct—and indeed, when the programs are executed, there are no surprises.[1]

Section 3 shows that the proof argument of Section 2 is invalid in the presence of abstract data types with the standard method for proving correctness of data representation [4]. It does this by exhibiting another instance of the parameterized program that meets all the requirements imposed during the argument that the code is correct, yet does not meet its specification when executed.

Section 4 traces the source of the difficulty to the confluence of features mentioned above, and—because it stands out as a relative late-comer historically—specifically to the introduction of user-defined abstract data types into a mix with the other two features. In short, an accepted mechanism for "scaling up" to deal with software engineering concerns [5] is the culprit. This section also argues that there is no practical workaround that can salvage all three language features.

Section 5 comments on related work, and Section 6 concludes with a recommendation for how to deal with this issue in a way that respects software engineering considerations.

2 Modular Verification of a Simple Program Function

2.1 Framework for Modular Verification

The goal is to prove that a program unit `E` (e.g., code for an operation or a class) implements its specification. By *modular verification*, we mean a system in which such a proof has two properties: (1) it is based only on the specifications of the program units that `E` directly depends on, and (2) the proof is done only once, not once for every context where `E` is used. It is widely agreed that a verification system must be modular in order to "scale up" to large programs so they can be proved correct one piece at a time; this is the only way to manage the intellectual complexity involved [5].

[1] The actual code comprising these examples is available at `http://www.cse.ohio-state.edu/~bronish/public/rsrg/Soundness.html`.

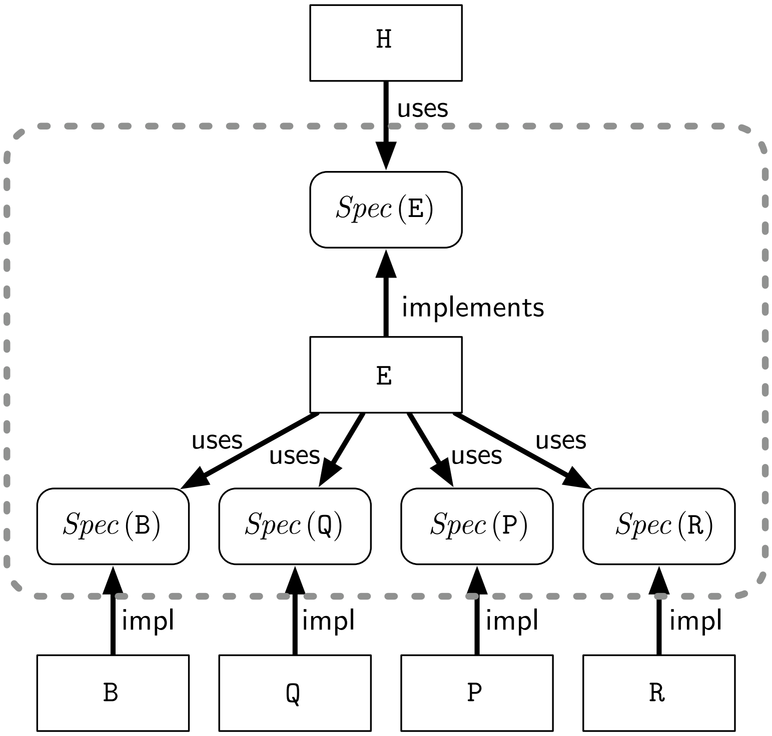

Fig. 1. Modular reasoning about a program unit's correctness

Figure 1 illustrates this situation for the Lisp program to be verified later in this section. Here, `E` is a Lisp function that can invoke `B`, `Q`, `P`, and `R`; a client of `E` might be another function `H`. The proof of correctness of `E` must not involve the code for `B`, `Q`, `P`, `R`, or `H`. It must rely only on the specifications of `B`, `Q`, `P`, `R`, and `E` itself—and it must be independent of `H`. The proof is relative: if `B`, `Q`, `P`, and `R` implement their respective specifications, and if `E` has been proved to implement its specification, then `E` should behave correctly in the context of any `H` that relies only on the specification of `E`. In short, modular verification that `E` implements *Spec*(`E`) involves only the part of Fig. 1 inside the dotted boundary.

2.2 Example Program to be Verified

Consider the following Lisp function, which tests equality of two lists using the natural recursive approach:

```
(defun lists-are-equal (list1 list2)
  (cond
    ((null list1) (null list2))
    ((null list2) nil)
    (t (and (equal (car list1) (car list2))
            (lists-are-equal (cdr list1) (cdr list2))))))
```

The strategy evinced by this code generalizes to a "project and reduce" approach for testing equality of any two "complex" values by projecting to "simple" values which themselves are testable for equality. We can formulate this as a Lisp function schema stripped of semantically-loaded function names:

```
(defun E (x1 x2)
  (cond
    ((B x1) (B x2))
    ((B x2) nil)
    (t (and (Q (P x1) (P x2))
            (E (R x1) (R x2))))))
```

The reader should recognize the connection between this code and Figure 1, and should realize that *Spec*(`E`) and the other specifications are not yet known; the code above is just the box labeled `E`.

2.3 Semantics, Specifications, and What Needs to be Proved

As noted by McCarthy [6], the key feature of Lisp from the standpoint of "proving the correctness of Lisp programs" is that "Lisp functions are functions." That is, for purposes of mathematical reasoning about Lisp programs, the language semantics ***Sem*** assigns as the meaning of each program function a corresponding mathematical function—the obvious one [7]. For convenience, we use e as shorthand for ***Sem***(`E`), so e denotes that corresponding mathematical function; similarly for the other program functions `B`, `Q`, `P`, and `R`. Specifically, this is the mathematical function that the above program computes, according to the language semantics:

$$e(x1, x2) = \begin{cases} b(x2), & \text{if } b(x1) \\ \mathit{false}, & \text{if } \neg b(x1) \wedge b(x2) \\ q(p(x1), p(x2)) \wedge e(r(x1), r(x2)), & \text{otherwise} \end{cases} \qquad (\star)$$

Let us suppose the specifications of all these program functions stipulate that they are restricted to their intended domains, as outlined above for the equality-testing task that the code is intended to solve. In other words, the programs in question are not required to work as advertised unless invoked on arguments that make sense. The signatures of the mathematical functions computed are then as follows, where C is the mathematical type (with $=$) of the "complex" things and S is the mathematical type (also with $=$) of the "simple" ones:

$$\begin{aligned} e &: \; C \times C \to \mathbb{B} \\ b &: \; C \to \mathbb{B} \\ q &: \; S \times S \to \mathbb{B} \\ p &: \; C \to S \\ r &: \; C \to C \end{aligned}$$

The formalization of the requirement that `E` should test equality of two C values is:

$$\mathit{Spec}(\mathtt{E}) \overset{\text{def}}{\equiv} e(x1, x2) \iff x1 = x2$$

The verification task is to show that the semantics for `E`, i.e., the mathematical function shown in equation ($\star$) above, satisfies the property demanded by $\mathit{Spec}(\mathtt{E})$. The proof that it does may rely only on the specifications of the functions involved in the code. The only one of these we know for sure is:

$$\mathit{Spec}(\mathtt{Q}) \overset{\text{def}}{\equiv} q(z1, z2) \iff z1 = z2$$

The unknown specifications will be important in the proof as well, as one would expect. In particular, to complete the proof it will be sufficient that b, p, and r should together satisfy the following properties:

$$\exists\, !\, x : C \;\text{ s.t. } (b(x)) \tag{P1}$$

$$\begin{aligned} &\forall\, x, y : C \; (\neg b(x) \land \neg b(y) \implies \\ &\qquad\qquad (p(x) = p(y) \land r(x) = r(y) \implies x = y)) \end{aligned} \tag{P2}$$

$$\exists\, m : C \to \mathbb{N} \;\text{ s.t. } (\forall x : C \; (\neg b(x) \implies m(r(x)) < m(x))) \tag{P3}$$

According to the definition of modular verification, these properties must be deducible from the specifications of $\mathit{Spec}(\mathtt{B})$, $\mathit{Spec}(\mathtt{P})$, and $\mathit{Spec}(\mathtt{R})$. The reasons these particular properties are needed will become evident in the proof that follows.

2.4 Proof that `E` Implements *Spec*(`E`)

The proof of correctness is, not surprisingly, by mathematical induction, as dictated by the structure of the code with its recursive call to `E` [8]. The proof is divided into three parts. First, we show that $x1 = x2 \implies e(x1, x2)$, i.e., that if its arguments are equal then `E` reports this. Next, we show that $e(x1, x2) \implies x1 = x2$, i.e., that if `E` reports its arguments are equal then indeed they are. Finally, we show that the code terminates, or equivalently, that the induction is well-founded. The inductive hypothesis states that the recursive call does in fact perform an equality test, that is: $e(r(x1), r(x2)) \iff r(x1) = r(x2)$; this comes directly from $\mathit{Spec}(\mathtt{E})$.

Proof that $\boldsymbol{x1 = x2 \implies e(x1, x2)}$: Assume that $x1 = x2$. Note that since b is a function, $b(x1) = b(x2)$. We proceed by case analysis to establish $e(x1, x2)$.

Case 1: $b(x1) = b(x2) =$ *true:* By the first condition of $(\star)$, we see that $e(x1, x2)$ is true.

Case 2: $b(x1) = b(x2) =$ *false:* Since p is a function, $p(x1) = p(x2)$. So by *Spec*(`Q`) we know $q(p(x1), p(x2))$ is true. Since r is a function, $r(x1) = r(x2)$. The inductive hypothesis then gives $e(r(x1), r(x2))$. We see now that the result in the third case of $(\star)$ is true, so $e(x1, x2)$ is true.

Proof that $\boldsymbol{e(x1, x2) \implies x1 = x2}$: Assume that $e(x1, x2)$ is true. Notice from $(\star)$ that there are only two ways for this to occur: $b(x1) \wedge b(x2)$ or $\neg b(x1) \wedge \neg b(x2) \wedge q(p(x1), p(x2)) \wedge e(r(x1), r(x2))$. Again we proceed by cases.

Case 1: $b(x1) \wedge b(x2)$*:* In this case property (P1) implies that $x1 = x2$.

Case 2: $\neg b(x1) \wedge \neg b(x2) \wedge q(p(x1), p(x2)) \wedge e(r(x1), r(x2))$*:* By *Spec*(`Q`), we know that $p(x1) = p(x2)$. By the inductive hypothesis, we have $r(x1) = r(x2)$, and so by property (P2) we see that $x1 = x2$.

Termination/well-foundedness: Given the inductive definition $(\star)$ for `E`, property (P3) is sufficient to show there are no infinite descending chains in the series of arguments to e: there is a progress metric m that maps the first argument $r(x1)$ in the recursive invocation of e to a natural number strictly smaller than $m(x1)$. The existence of m essentially ensures that invoking `R` "reduces the problem."

Both directions of the bi-implication having been established, the proof that $e(x1, x2) \iff x1 = x2$ is completed. This coupled with the termination argument implies that `E` does indeed implement *Spec*(`E`). ■

We can now return to the `lists-are-equal` code and convince ourselves that the proof is applicable. The functions match up as follows: `E`=`lists-are-equal`, `B`=`null`, `P`=`car`, and `R`=`cdr`. Property (P1) is satisfied because there is exactly one list for which `null` returns *true*, namely the empty list. Property (P2) is satisfied because, by the semantics of Lisp, two lists with equal `car`s and equal `cdr`s are equal. Property (P3) is satisfied with m as the length of the argument (a list). This program is evidently verified, then, as an instance of the parameterized program `E`.

2.5 An Example With Abstract Data Types

Now consider the following function, presented using CLOS syntax, i.e., `defmethod` and type restrictions on `n1` and `n2` in the parameter list. CLOS is used because there are user-defined abstract data types involved. This code checks the equality of two `nat`s, where the mathematical model of a `nat` is a natural number, the mathematical model of a `digit` is a natural number between

0 and 9 inclusive, and the function names accurately reflect their behavioral specifications (shown below).

```
(defmethod nats-are-equal ((n1 nat) (n2 nat))
  (cond
    ((is-zero n1) (is-zero n2))
    ((is-zero n2) nil)
    (t (and (digits-are-equal (mod-by-10 n1) (mod-by-10 n2))
            (nats-are-equal (div-by-10 n1) (div-by-10 n2))))))
```

The significance of modular verification is that the correctness of this code should be determined independently of the representations used for nat and digit and of the algorithms used for the functions is-zero, digits-are-equal, mod-by-10, and div-by-10; all that matters is that these should satisfy their specifications. The functions match up with E=nats-are-equal, B=is-zero, Q=digits-are-equal, P=mod-by-10, and R=div-by-10; the domains of interest are $C = S = \mathbb{N}$. The specifications of B, P, and R are:

$$\begin{aligned}
&\mathit{Spec}(\texttt{is-zero}) \overset{\text{def}}{\equiv} \mathit{is\text{-}zero}(n) \iff n = 0\\
&\mathit{Spec}(\texttt{mod-by-10}) \overset{\text{def}}{\equiv} \exists k : \mathbb{N} \;\text{ s.t. } (n = 10 * k + \mathit{mod\text{-}by\text{-}10}(n)\\
&\qquad\qquad\qquad\qquad\quad \wedge 0 \leq \mathit{mod\text{-}by\text{-}10}(n) < 10)\\
&\mathit{Spec}(\texttt{div-by-10}) \overset{\text{def}}{\equiv} \exists k : \mathbb{N} \;\text{ s.t. } (n = 10 * \mathit{div\text{-}by\text{-}10}(n) + k \wedge 0 \leq k < 10)
\end{aligned}$$

Property (P1) is satisfied because there is exactly one natural number for which is-zero returns *true*, i.e., 0. Property (P2) is satisfied because if two natural numbers are divided by 10 and have equal quotients and equal remainders, then they are equal. Property (P3) is satisfied with m as the mathematical value of its argument (a natural number). This program is evidently verified, as another instance of the parameterized program E.

3 Unsoundness

Consider one final CLOS function. This code checks the equality of two sets, where the mathematical model of a set is a finite set of elements, and the function names accurately reflect their behavioral specifications (shown below).

```
(defmethod sets-are-equal ((s1 set) (s2 set))
  (cond
    ((is-empty s1) (is-empty s2))
    ((is-empty s2) nil)
    (t (and (elements-are-equal (member-from s1) (member-from s2))
            (sets-are-equal (remove-one s1)
                            (remove-one s2))))))
```

The functions match up to the schema, with E=sets-are-equal, B=is-empty, Q=elements-are-equal, P=member-from, and R=remove-one; the domains of interest are S (the type of the set elements), and $C = \mathcal{P}(S)$. The specifications are as follows:

$$Spec(\texttt{is-empty}) \overset{\text{def}}{\equiv} \textit{is-empty}(s) \iff s = \{\}$$

$$Spec(\texttt{member-from}) \overset{\text{def}}{\equiv} s \neq \{\} \implies \textit{member-from}(s) \in s$$

$$Spec(\texttt{remove-one}) \overset{\text{def}}{\equiv} s \neq \{\} \implies \textit{remove-one}(s) = s \setminus \textit{member-from}(s)$$

Property (P1) is satisfied because there is exactly one set for which is-empty returns *true*, i.e., the empty set. Property (P2) is satisfied because if equal elements are removed from each of two non-empty sets, and the resulting sets are equal, then the original sets are equal. Property (P3) is satisfied with m as the cardinality of its argument (a finite set). The program is evidently verified, then, as another instance of the parameterized program E.

The problem is that executing this program when $s1 = s2$ may return *false*, even though the function implementations satisfy the above specifications. The following class and method definitions comprise one realization that leads to the faulty behavior. The idea of this code is to use a list to represent an abstract set. The principle of information hiding dictates that clients of set will not manipulate or reason about the list directly, and indeed this could be enforced via Lisp's packaging and exporting constructs, but the syntax is omitted for brevity. The empty-set, singleton, and set-union functions provide ways of constructing new sets.

```
(defclass set () ((rep :initarg :init-val
                       :reader my-set-rep)))

(defmethod is-empty ((x set))
   (null (my-set-rep x)))

(defmethod empty-set ()
  (make-instance 'set :init-val nil))

(defmethod singleton (x)
  (make-instance 'set :init-val (cons x nil)))

(defmethod set-union ((x set) (y set))
  (make-instance 'set :init-val
    (union (my-set-rep x) (my-set-rep y))))
    ; union is a built-in list function that combines
    ; its argument lists and throws away duplicates

(defmethod member-from ((x set))
  (car (my-set-rep x)))
```

```
(defmethod remove-one ((x set))
  (make-instance 'set :init-val (cdr (my-set-rep x))))

(defmethod elements-are-equal ((s1 set) (s2 set))
  (sets-are-equal s1 s2))

(defmethod elements-are-equal (e1 e2)
  (equal e1 e2))
```

One potentially confusing aspect of this implementation is the overloaded `elements-are-equal` method. The first version, with parameters specialized to be `set`s, allows `sets-are-equal` to handle a recursive `set` data structure (e.g., a set of sets of integers). The second version will be invoked in all other cases, and relies on the built-in Lisp `equal` function. Once more, due to the relative nature of modular reasoning, the argument of Section 2.4 only establishes the correctness of `sets-are-equal` when the `set` elements are themselves `set`s, or are objects for which `equal` correctly implements the equality predicate. This limitation could be surmounted by parameterizing the `set` class with an equality operation for its elements rather than trying to build one from scratch inside the implementation, but the details are tangential to this work, and thus are omitted.

We can see that `sets-are-equal` is defective by testing the equality of two `set`s that contain the same elements, and hence are equal according to the specification, but whose internal list representations differ:

```
[1]> (sets-are-equal
       (set-union (singleton 1) (singleton 2))
       (set-union (singleton 2) (singleton 1)))
NIL
```

4 Analysis

What went wrong? A key part of the argument in Subsection 2.4 first appears in Case 2 of the first part of the proof: "Assume that $x1 = x2$ ... Since p is a function, $p(x1) = p(x2)$." The subtle problem with this argument is that, when $x1$ and $x2$ are variables of an abstract data type (in the problematic case, `set`s), the conclusion $p(x1) = p(x2)$ may not be valid even though the language semantics says "each program function denotes a mathematical function." There has been a failure to distinguish between (a) the semantics of each program function being a mathematical function *on the concrete (representation) domain*—where, indeed, it is, by construction; and (b) the semantics of each program function being a mathematical function *on the abstract domain of the ADT*—where evidently it may not be. Specifically, if the mathematical set $x = \{1, 2\}$ is represented by the list `(1 2)` then `(member-from x)`, as implemented by the code in Section 3, returns `1`. If that same abstract value is represented as `(2 1)` instead, `(member-from x)` returns `2`. Figure 2 illustrates this situation.

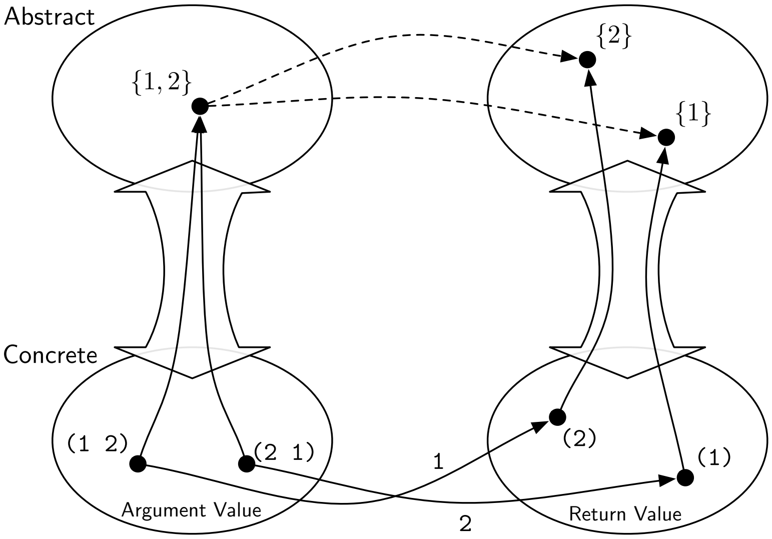

Fig. 2. Correspondence between abstract set values and concrete representations undergoing a call to `remove-one`. The labels on the arrows from the arguments to the return values indicate the element that is removed by `remove-one`. Notice that although we see two representations of the *same* abstract value as arguments, the call produces different abstract results. The dotted arrows indicate non-functional behavior at the abstract level, which is incompatible with the assumption that the language semantics assigns a mathematical function as the meaning of every program function.

An initial response to this difficulty might be that the implementation of `set` should be considered defective. However, by standard arguments used in showing the correctness of data representation, it is evident that all these methods meet their specifications. With a Lisp list as the data representation, the `null` test in `is-empty` clearly satisfies *Spec*(`is-empty`). Likewise, using `car` to return an element from the `set` representation in `member-from` is a legitimate realization of that method's specification. Finally, simply constructing a new `set` whose representation is the `cdr` of the argument `set`'s representation fulfills *Spec*(`remove-one`). Each of the latter two functions meets its specification by remaining within the envelope of behavior that the relational specification allows. Note that `member-from` never returns a value from the argument's representation list *other* than its `car`, even though the specification licenses it to return any element of the list. No matter which `set` element happens to be at the front of the list that represents it, `member-from` meets its specification; similarly, `remove-one`. Even if the implementation of `member-from` instead made a random choice of which element from the list to return, it could still be reasoned about in a functional manner [9]. Non-determinism is not the issue; the fact that a functional semantics is not preserved across abstraction boundaries is the issue.

One might suggest that the rules for proof of correctness of data representation should be changed, in order to guarantee that functional behavior in the concrete domain always leads to functional behavior in the abstract domain (even for relational specifications). The problem with this path is that performance must be seriously compromised to achieve it. In this situation, each of `member-from` and `remove-one` would have to pick the same element of the `set` no matter where that element happened to be in the list representing the `set`. How could they do this, knowing nothing else about the elements? Even if there were enough information to select a canonical element (e.g., a `set` client could be required to supply a total ordering on the elements that could be computed during execution of `member-from` and `remove-one`), these two program functions, which should have taken constant time, would instead take linear time.

One might then be tempted to disallow the aforementioned relational specifications, on the grounds that they seem in some sense discordant with the code's presumed functional semantics. On further reflection, however, `set` itself constitutes one example of this position's untenability. In general, any unordered collection that may be filled with arbitrary data requires a way for clients to iterate through its contents. `Set`s that do not offer such a mechanism are cumbersome in practice, as [10] demonstrates. In that work, the author specifies a `set` ADT in a purely functional manner, but must immediately resort to ungainly workarounds to compensate for its lack of relational behavior. For example, in a proposed solution to the set-covering problem, the design forces the collection of subsets to be passed in as a list of sets—because if they were passed in as a set of sets (which would make logical sense), then there would be no way to actually examine or process the subsets as the solution requires. In addition, the proposed implementations of the ADT include an extra operation not specified in the interface: `showset`. This operation evidently is intended to have relational behavior: the two proposed implementations do not produce the same results for the same abstract set value, yet both are claimed to have been "verified."

A common idiom, particularly in functional languages, is to use `mapcar` (or an analog of it) as an iterator to compute over a collection in an apparently holistic rather than piecewise manner. This is not a solution to the problem, however, since the specification of such a construct will again involve relational rather than functional behavior in order not to tie down the implementation to one particular order of application. Java's `java.util` package serves as an instructive example here: the interface for its `Set` collection specifies (albeit informally) that the result of the `iterator` method is not guaranteed to return elements in any particular order.

Another attempt to repudiate problematic cases such as `set` might be to demand that we choose data representations that correspond to abstract values in a one-to-one fashion, thus collapsing functions on abstract domains and functions on concrete domains into a single notion. Note, however, that this amounts to a relinquishment of true abstraction. The ability to represent a single abstract value in multiple ways is essential, not only for elegant design, but also for efficiency [11].

5 Related Work

Hoare [4] first proposed inductive proofs for verifying the correctness of ADTs. His primary example is indeed a set, but his formulation offers no relational behavior. Early subsequent work along these lines focused on ADTs that were specified algebraically rather than by a model [12], leaving the issue of relational specification mostly obscured. However, Hoare has remarked that "the complexities of application seem to require the model-based approach" [13]. It is therefore no surprise that virtually all modern specification methods and languages (e.g., Larch [14], Z [15], JML [16], Resolve [17]) are model-based [18].

Schweizer and Denzler [19] verify a set ADT by automatically generating algebraic specifications from a proposed implementation, and showing that these semantically match the desired specifications. As in [10], the verified set suffers from low utility, allowing only a limited range of small integer elements due to the chosen implementation: an array of Boolean values.

Tucker and Zucker [20] develop a language-independent theoretical framework for formal reasoning about abstract data types, and specifically observe that a relational approach is necessary for specification, even when the programs themselves are purely functional. Other work on specifications with functional languages includes EML [21].

More generally, the message from related work on verification is that soundness matters. History has demonstrated that unsoundness is not merely an unfortunate technical shortcoming of some verification systems, but also a potential public-relations problem that has led some to question the very idea of program verification [22]. It should, therefore, be viewed as a challenge to the software verification community when any new or even slightly modified verification system is proposed: determine whether the system is sound, and if not, identify the factors that contribute to its unsoundness.

For example, thirty years ago, Cook [23] explained how repeated arguments to calls (e.g., $P(x, x)$), and similarly global variables as arguments to calls, can render Hoare logic unsound. This was a valuable observation because it served as a clear warning about specific pitfalls to be avoided by those proposing modular verification systems.

This is hardly the only hazard, though. Two specific documented threats to soundness are potentially aliased pointers/references to mutable objects (a variant on Cook's observation about repeated arguments), and the failure to distinguish fully between mathematical and programming entities [24]. Heym [25] illustrates how it is easy to introduce unsoundness when generating verification conditions on the basis of very plausible but informal arguments, and cites examples of such mistakes in earlier approaches. He then proceeds to prove the soundness of his own proposed method for verification.

Soundness can be compromised as well by the absence of clear semantics for specifications and code. A new paper [26] claims to have achieved "full functional [and modular] verification of linked data structures" in Java. The paper makes a number of interesting observations, but it does not address the soundness of the proposed verification method in any formal sense. Indeed, this would be impossible without considering the formal semantics of both the specification language and Java. Yet there are two plausible meanings for the specifications in the paper. The paper says its specifications “completely capture the desired behavior of the Java data structure implementations”, and argues that “there is a broad consensus on how they should behave.” With one obvious interpretation of the specifications, the components are of limited utility to clients and are quite different in behavior from their counterparts in the Java libraries, so this interpretation seems dubious. However, with the other obvious interpretation—the one in which the specifications really do “capture the desired behavior”—there is a more serious problem. The proof rules’ assumption that the built-in Java “==” operator coincides with mathematical equality in the specifications, combined with this more likely interpretation of the specifications, subverts the soundness of the verification method.

Repeated arguments, potentially aliased references to mutable objects, failure to make all appropriate distinctions between programming and mathematical entities and operators, and vague semantics are potential sources of unsoundness in modular verification systems. This paper shows that there is another—certainly not the last one.

6 Conclusion and Recommendation

A threat to sound verification arising from the confluence of purely functional semantics, relational specifications, and abstract data types has been identified and explained. The latter two properties have been argued to be indispensable on what amount to software engineering grounds. We therefore conclude that purely functional semantics must be relaxed. This is not to say that no program entities may correspond to mathematical functions; indeed this is quite a useful attribute in many cases. It is recommended instead that purely functional code be distinguished, both semantically and syntactically, from code that may exhibit relational behavior in the abstract domain, even when it perforce acts functionally on a concrete data representation.

Specifically, we recommend the following approach to work around this soundness problem: (1) partition program operations into “function” operations and “procedure” operations; (2) define the language semantics so that the meaning of each function operation is a mathematical function, and permit only a functional specification for a function operation; and (3) define the language semantics so the meaning of each procedure operation is a mathematical relation, and permit either a functional or a relational specification for a procedure operation. A sound and relatively complete verification system that supports both specification of relational behavior and efficiently implementable ADTs seems to be possible under this scenario [25].

Finally, we observe that the natural desire to have purely functional rather than relational semantics is similar to the desire to have only abstraction functions rather than abstraction relations as the basis for verifying data representations for ADTs. Functions are somewhat simpler to deal with in both situations;

the problem is that they are just a little too simple in both situations. The need for abstraction relations arises from an expressiveness problem [11]. The need for program operations whose meanings are relations arises from a soundness problem that apparently cannot be addressed in any other acceptable way.

Acknowledgments

The authors are grateful for healthy skepticism and constructive feedback from Paolo Bucci, Steve Edwards, Harvey M. Friedman, Wayne Heym, Gary Leavens, Brandon Mintern, Bill Ogden, Nasko Rountev, Don Sannella, Murali Sitaraman, Paul Sivilotti, Hampton Smith, Neelam Soundarajan, and Anna Wolf. This work was supported in part by the National Science Foundation under grant DMS-0701260.

References

1. McCarthy, J.: LISP 1.5 Programmer's Manual. The MIT Press (1962)
2. McCarthy, J.: A Basis for a Mathematical Theory of Computation. In Braffort, P., Hirschberg, D., eds.: Computer Programming and Formal Systems, North-Holland, Amsterdam (1963) 33–70
3. Bobrow, D.G., DeMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A.: Common lisp object system specification. SIGPLAN Not. **23**(SI) (1988) 1–142
4. Hoare, C.A.R.: Proof of correctness of data representations. Acta Informatica **1**(4) (1972) 271–281
5. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12) (1972) 1053–1058
6. McCarthy, J.: Lisp - notes on its past and future. In: LFP '80: Proceedings of the 1980 ACM conference on LISP and functional programming, New York, NY, USA, ACM (1980) .5–viii
7. Boyer, R.S., Moore, J.S.: Program verification. Journal of Automated Reasoning **1**(1) (1985) 17–23
8. Boyer, R.S., Moore, J.S.: Proving theorems about lisp functions. J. ACM **22**(1) (1975) 129–144
9. Burton, F.W.: Nondeterminism with referential transparency in functional programming languages. Comput. J. **31**(3) (1988) 243–247
10. Harrison, R.: Abstract data types in standard ML. John Wiley & Sons, Inc., New York, NY, USA (1993)
11. Sitaraman, M., Weide, B.W., Ogden, W.F.: On the practical need for abstraction relations to verify abstract data type representations. IEEE Trans. Softw. Eng. **23**(3) (1997) 157–170
12. Guttag, J.V., Horowitz, E., Musser, D.R.: Abstract data types and software validation. Commun. ACM **21**(12) (1978) 1048–1064
13. Hoare, C.A.R., Jones, C.B., eds.: Essays in computing science. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
14. Guttag, J.V., Horning, J.J.: Larch: languages and tools for formal specification. Springer-Verlag New York, Inc., New York, NY, USA (1993)

15. Spivey, J.M.: The Z notation: a reference manual. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1989)
16. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota. (2000) 105–106
17. Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., Weide, B.W.: Specifying components in resolve. SIGSOFT Softw. Eng. Notes **19**(4) (1994) 29–39
18. Wing, J.: A specifier's introduction to formal methods. Computer **23**(9) (Sep 1990) 8, 10–22, 24
19. Schweizer, D., Denzler, C.: Verifying the specification-to-code correspondence for abstract data types. In: Proc. 6th Conf. on Dependable Computing for Critical Applications, Garmisch-Partenkirchen, Germany (1997)
20. Tucker, J.V., Zucker, J.I.: Toward a general theory of computation and specification over abstract data types. In Akl, S.G., Fiala, F., Koczkodaj, W.W., eds.: Advances in Computing and Information ICCI '90, International Conference on Computing and Information, Niagara Falls, Canada, May 1990. Volume 468. Springer-Verlag, New York, N.Y. (1991) 129–133
21. Kahrs, S., Sannella, D., Tarlecki, A.: The definition of Extended ML: A gentle introduction. Theoretical Computer Science **173**(2) (1997) 445–484
22. Fetzer, J.H.: Program verification: the very idea. Commun. ACM **31**(9) (1988) 1048–1063
23. Cook, S.A.: Soundness and completeness of an axiom system for program verification. SIAM J. Comput. **7**(1) (1978) 70–90
24. Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W.D., Pike, S.M., Hollingsworth, J.E.: Reasoning about software-component behavior. In: ICSR-6: Proceedings of the 6th International Conerence on Software Reuse (LNCS). Volume 1844., Springer-Verlag (2000) 266–283
25. Heym, W.D.: Computer Program Verification: Improvements for Human Reasoning. PhD thesis, Department of Computer and Information Science, The Ohio State University, Columbus, OH (December 1995)
26. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. In: ACM Conf. Programming Language Design and Implementation (to appear), `http://lara.epfl.ch/~kuncak/papers/ZeeETAL08FullFunctionalVerificationofLinkedDataStructures.pdf`. Accessed 4/24 (2008)