Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping

Krishna Kishore Dhara
Iowa State University

Gary T. Leavens
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Programming Languages and Compilers Commons, and the Software Engineering Commons

Recommended Citation
http://lib.dr.iastate.edu/cs_techreports/360

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping
Krishna Kishore Dhara and Gary T. Leavens
TR #01-02
March 2001

Keywords: Weak behavioral subtyping, strong behavioral subtyping, viewpoint, aliasing, mutation, modularity, specification, verification, Java language, JML language.


Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping

Krishna Kishore Dhara
Avaya Labs Research
1D-632A, 101 Crawfords Corner Rd.
Holmdel, New Jersey 07733 USA
+1 732 817 2160
dhara@avaya.com

Gary T. Leavens
Department of Computer Science
Iowa State University
226 Atanasoff Hall, Ames, IA 50011 USA
+1 515 294 1580
leavens@cs.iastate.edu

ABSTRACT
Existing work on behavioral subtyping either ignores aliasing or restricts the behavior of additional methods in a subtype and only allows one to use invariants and history constraints in reasoning. This prevents many useful subtype relationships; for example, a type with immutable objects (e.g., immutable sequences), cannot have a behavioral subtype with mutable objects (e.g., mutable arrays). Furthermore, the associated reasoning principle is not very useful, since one cannot use the pre- and postconditions of methods. Weak behavioral subtyping permits more behavioral subtype relationships, does not restrict the behavior of additional methods in subtypes, and allows the use of pre- and postconditions in reasoning. The only cost is the need to restrict aliases so that objects cannot be manipulated through the view of more than one type.

Keywords
Weak behavioral subtyping, strong behavioral subtyping, viewpoint, aliasing, mutation, modularity, specification, verification, Java language, JML language

1. INTRODUCTION
When enhancing existing object-oriented (OO) software, one commonly adds new subtypes to existing types. The type system permits one to operate on objects of these new subtypes using the protocol of their supertypes. For this reuse of the supertype’s protocol to achieve its desired effect, there must be some connection between the specifications of the expected supertype methods and those of the new subtypes. Without such a connection, one would have to rethink (retest, and reverify) all such reused code whenever new types are added, which is not practical.

The notion of subtyping that is enforced by statically-typed OO languages, like C++ [44] and Java [4], does not take the semantics of types into account; that is, the type system of such languages merely checks that declared subtype relationships satisfy certain restrictions that guarantee the lack of runtime type errors [8]. These checks are modular in the sense that the type checker only needs access to type information for the supertypes and the code of the new subtypes; it does not need to re-typecheck unchanged code that is reused. Modularity allows type checking to be practical. However, such checks are not enough to prevent surprising behavior; that is, they are only part of what is needed to reason about the behavior of a program when new subtypes are added.

A well-known technique for preventing such surprising behavior is “behavioral subtyping.” Behavioral subtyping ensures that objects of the new subtypes “act like” objects of their supertypes, when manipulated as if they were supertype objects, by comparing the specifications of the supertypes and the new subtypes. Various authors have proposed conditions that ensure behavioral subtyping, some using proof theory [2, 3, 12, 30, 33, 34, 46, 45], and others using model theory [7, 11, 22, 23, 24, 30]. One common feature of these definitions is that there should be a way to map the state of subtype objects to the state space of their supertypes; another common feature is that the behavior of the subtype’s methods that are also present in the supertype must simulate the behavior of the corresponding supertype methods (modulo the mapping) [2, 3, 11, 12, 24, 30, 33, 34, 46, 45].

However, when mutable objects are considered, there is some room for disagreement on what restrictions need to be placed on the additional methods that the subtype may have, but which are not present in the supertype. Two different restrictions on these additional methods result in the notions

---

*This work was supported in part by the US National Science Foundation under Grant CCR-9803843.
of strong behavioral subtyping [30] and weak behavioral subtyping [10, 11, 12].

In this paper, we focus on weak behavioral subtyping, which supports better reasoning and which permits more behavioral subtype relations than Liskov and Wing’s notion of strong behavioral subtyping [30]. For example, a type with immutable objects (e.g., immutable sequences), cannot have strong behavioral subtypes whose objects are mutable (e.g., mutable arrays), although such types can be weak behavioral subtypes. Similarly, non-const objects of a type $T$ are weak behavioral subtypes of const objects of type $T$, but are not strong behavioral subtypes.

This contribution of this paper are twofold:

- It provides a comprehensive discussion of various issues in modular reasoning for OO languages related to mutation and aliasing.
- It describes viewpoints and weak behavioral subtyping, and their advantages for modular reasoning.

This paper attempts to convey the ideas of weak behavioral subtyping without getting bogged down in semantic details; thus it is outside its scope to give formal proofs of the soundness of the associated reasoning techniques. However, a formal proof that gives some indication of the soundness the reasoning technique has been given elsewhere [10].

In the next section, we present the modular reasoning problem and discuss the choices for modular reasoning in the context of mutation and aliasing. Section 3 gives the definition of weak behavioral subtyping. In section 4 we discuss one way to enforce restrictions on aliasing necessary for sound modular reasoning with weak behavioral subtyping. Section 5 presents examples of behavioral subtype hierarchies that are commonly used in object-oriented languages. Finally, we present related work and conclusions.

2. MODULAR REASONING CHOICES

In this section, we describe the problem in more detail, and lay out various choices for modular reasoning techniques. We start with a discussion of the basic problem before focusing on the problems caused by combination of mutation and aliasing.

2.1 The Basic Problem

To illustrate the effect on existing code of adding new subtypes, consider the function given in Figure 1, which can observe the behavior of its two arguments $p1$ and $p2$. Such observation functions are (extreme) abstractions of the client code that uses various supertypes. For example, the code

```java
public class ObservePairFI {
    public static boolean obsFunc1(PairFI p1, PairFI p2) {
        if (p1 == null || p2 == null) {
            return true;
        } else {
            int first = p1.getFirst();
            p2.incSecond();
            return first == p1.getFirst();
        }
    }
}
```

Figure 1: An observation function, obsFunc1.

in Figure 1 views its arguments through the type PairFI. In this paper, we will follow the convention that such observation functions are always expected to terminate normally and return true for any inputs of their declared argument types. Therefore, such observation functions constitute a partial specification of the sequential behavior of objects of this type. (A more declarative specification of this type will be discussed below, see Figure 2.) The relation of such observation functions to test cases should be obvious.

Now suppose a new subtype of PairFI, for example, Triple, is created. Since Triple is a subtype of PairFI, we expect that existing code still works, in approximately the same way, for objects of the new types.

That existing code “still works” on objects of a subtype is captured by type theory. For example, in a language like Java, the objects of Triple must support all the methods of PairFI, with the same argument types, and the same result type; furthermore, each such common method must throw no more (checked) exceptions than those that can be thrown by the supertype, PairFI. Such rules are a specialization to Java of the general theory of subtyping for object-oriented languages [1]. These syntactic properties allow one to assign objects of a subtype to variables whose static type is a supertype, and to invoke methods on such variables without encountering runtime type errors.

However, more than just the type-theoretic notion of syntactic subtyping is needed to ensure that existing code still works in “approximately the same way” on objects of new subtypes. Let us sharpen this notion by considering observation functions, such as obsFunc1 in Figure 1. We say that Triple objects can be viewed as PairFI objects with respect to obsFunc1 if all possible combinations of arguments of the types Triple and PairFI return the expected value of true. For example, the following should return true.

```java
ObservePairFI.obsFunc1( new Triple(3, 4, 5), new PairFI(7, 9, 11));
```

3We do not consider multithreading or other forms of parallelism in this paper.
If the result of such an observation is not true, we say it is unexpected, and thus that the arguments exhibit unexpected behavior.

One way to ensure that such observations do not encounter unexpected results when new subtypes are added to a program, is to reason directly about the observations, arguing that objects of the new subtypes cannot cause unexpected behavior. However, this reasoning technique is not modular, because it has to be repeated for each potential observation function, and because it depends on the size of the observation functions.

By contrast, modular reasoning does not depend on how the new subtypes are reused, and the amount of reasoning required is independent of the size of the reused code. The basic strategy for any such modular reasoning technique is to directly compare the behavior of the subtype and its super-types. Such a technique is sound if it prevents unexpected behavior.

2.2 Behavior of Types

One way to formally specify the behavior of a type is to describe the abstract values of its objects, and to specify the behavior of its methods in terms of these abstract values [15, 29, 51]. In addition one can specify other properties, such as invariants and history constraints [30].

For example, Figure 2 specifies the behavior of class PairFI, using the Java Modeling Language (JML) [19, 20, 41]. JML is a behavioral interface specification language for Java; it is based on Larch [14] and Eiffel [34].

Annotations in JML are found in comments that either start with //@ and extend to the end of a line, or that start with /*@ and end with @*/; at-signs (@) found on the beginning of annotation lines are ignored. The first two annotations declare that the protected fields first and second, are to be considered to be public for specification purposes. The second annotation is an invariant; the invariant property defaults to true when this clause is omitted, which allows for more succinct specifications. The third annotation is a history constraint [30], which states that the value of first may not be changed once initialized.

In JML, the specification of a method or constructor precedes its code. In Figure 2, the constructor’s specification illustrates a fairly general form of method specification, using behavior, which allows one to specify a precondition (following requires), a frame axiom (following assignable), which says what fields the method may assign, a normal postcondition (following ensures), and an exceptional postcondition (following signals). If the requires is omitted, the precondition defaults to true, which means that the method can always be called. If the assignable clause is omitted, the list of assignable variables defaults to nothing, which means that the method may not assign to any variables. The specification form that uses normal_behavior is sugar for a behavior specification where the exceptional postcondition is false (for all exceptions); hence normal_behavior specifies executions that cannot throw any exceptions (when the

```java
public class PairFI {
    protected /*@ spec_public @*/ int first;
    protected /*@ spec_public @*/ int second;

    //@ public invariant true;
    //@ public constraint first == \old(first);

    /*@ public behavior
    @ requires true;
    @ assignable first, second;
    @ ensures first == fst && second == snd;
    @ signals (Exception) false;
    @*/
    public PairFI(int fst, int snd) {
        first = fst;
        second = snd;
    }

    /*@ public normal_behavior
    @ ensures \result == first;
    @*/
    public int getFirst() {
        return first;
    }

    /*@ public normal_behavior
    @ ensures \result == second;
    @*/
    public int getSecond() {
        return second;
    }

    /*@ public normal_behavior
    @ assignable second;
    @ ensures second == \old(second + 1);
    @*/
    public void incSecond() {
        second++;
    }
}
```

Figure 2: A JML specification and Java implementation of the class PairFI.
public class BadPairSubtype extends PairFI {
    //@ public normal_behavior
    @ assignable first, second;
    @ ensures first == fst & second == snd;
    @[/*
    public BadPairSubtype(int fst, int snd) {
        super(fst, snd);
    }
    @[/*
    also public normal_behavior
    @ assignable first;
    @ ensures first == 0;
    @*/
    public void incSecond() {
        first = 0;
    }
}

Figure 3: Java code for the class BadPairSubtype.

precondition holds). An expression of the form \texttt{\textbackslash old}(E) denotes E's value in the state at the start of the method's execution\footnote{The use of old expressions in history constraints has the same semantics, since history constraints can be thought of as a way of abbreviating assertions that go in all postconditions \cite{30}.} \cite{34}. Thus, for example, the incSecond method may be called in any state, is only allowed to assign to second, must return normally, and when it does so, it must make the the model field second have a value that is one more than the value it had when the method started. For a non-void method, the expression \texttt{\textbackslash result} means the value or object returned by the method, as in the specifications of first and second.

2.3 Behavior of Common Methods

As noted above, an important part of most definitions of behavioral subtyping is that the behavior of the supertype's methods must be matched by the behavior of those methods in the subtype.

To see why this is so, consider the subtype BadPairSubtype, whose code is given in Figure 3. This subtype overrides the method incSecond in a way that does not conform to the specification of incSecond given by its supertype PairFI; thus this implementation leads to unexpected behavior. This unexpected behavior can been seen using obsFunc1.

2.4 Problems Caused by Multiple Viewpoints

To see the problems caused by multiple viewpoints on objects and additional methods in subtypes, consider Figure 4. In this figure obsFunc2 takes an argument of type IncFirst, specified in Figure 5. The first JML annotation in this figure declares a model instance field fst; such a field is used for specification purposes only (hence model), and is imagined to be present in each object that implements the interface (hence instance).

Why do we expect that obsFunc2 will always return true? The main reason is because the viewpoint that obsFunc2 has on p is given by the static type of \texttt{p}, PairFI. Hence we wish to reason using the specification of PairFI. The specification given in Figure 2 has a history constraint which says that the field \texttt{first} cannot change. Thus, using this viewpoint, we do not expect \texttt{p.first} to change. Additionally, since IncFirst is unrelated to PairFI, there is no reason to expect any interaction between \texttt{p} and \texttt{t}. At least, it is easy to see how one could overlook such possible interactions.

The problem is that, because of the aliasing possible between the two arguments to obsFunc2, when new types are introduced into the program, another viewpoint on \texttt{p} is possible, which can lead to an unexpected result.

Suppose one adds a new type, Triple, that is a subtype of both PairFI and IncFirst. This type is specified in Figure 6. The connection between the model instance field \texttt{fst} and the field \texttt{first} inherited from PairFI is given by the depends and represents clauses in Figure 6 \cite{25, 26}. The depends clause says that the value of \texttt{fst} is determined by \texttt{first}, and hence that whenever \texttt{fst} is assignable, so is \texttt{first}. The represents clause says how the value of \texttt{fst} is recovered from \texttt{first}, in this case they are the same.
public class Triple /*@ weakly */ implements IncFirst {
    protected /*@ spec_public @*/ int third;
   //@ public depends fst <- first;
   //@ public represents fst <- first;
   /*@ public normal_behavior
    @ assignable first, second, third;
    @ ensures first == fst && second == snd
    @     && third == thd;
    @*/
    public Triple(int fst, int snd, int thd) {
        super(fst, snd);
        third = thd;
    }
    // specification inherited from IncFirst
    public void incFirst() { first++; }
   /*@ public normal_behavior
    @ ensures \result == third;
    @*/
    public int getThird() { return third; }
   /*@ public normal_behavior
    @ assignable third;
    @ ensures third == \old(third + 1);
    @*/
    public void incThird() { third++; }
}

Figure 6: Java code for the class Triple.

In JML specifications, public and protected model fields, invariants and specifications for non-static public methods are inherited from supertypes [12, 40, 34, 48, 49, 50] For example, the method specification of incFirst is inherited from the interface IncFirst, and the depends clause allows it to assign a new value to the field first inherited from PairFI. Since Triple is specified to extend its supertype PairFI weakly, the history constraint of PairFI does not apply to the additional methods of Triple.\textsuperscript{5} In particular, the inherited history constraint does not apply to incFirst, which allows it to be implementable.

Returning to the observation in Figure 4, consider what happens when the same triple is passed in both arguments, as in the following.

```
Triple t = new Triple(3, 4, 5);
Observe2.obsFunc2(t, t)
```

When the above code is executed, an alias is created within the obsFunc2, between p and t and the observation returns false. This is unexpected, and thus indicates unsoundness in the reasoning we used to conclude that obsFunc2 would always return true.

One way to reason soundly about examples like obsFunc2 is to analyze every possible aliasing pattern, even between unrelated types, such as PairFI and IncFirst. One problem is that there are an exponential number of such aliasing patterns among variables and fields. Worse, this technique seems error prone if applied informally, since without a tool to force one to consider all possible aliasing patterns, one is likely to overlook some of them. Psychologically this is bad, because most of these patterns will seem utterly pointless, since the types are not related. Hence we would like to do better.

Another possibility for sound reasoning about such examples is to rethink (retest, and reverify) the client code whenever new subtypes are introduced into the program that could cause new aliasing patterns. In terms of the example, one would have to go back to client code when Triple was introduced as a subtype of PairFI and IncFirst. However, this approach is not modular, so we reject it as well.

There are thus two options for sound modular reasoning:

- declaring that Triple is not a behavioral subtype of PairFI, or
- ruling out observations where such multiple viewpoints on objects are possible.

We explore these ideas below.

\textbf{2.5 Mutation, Aliasing, and Subtyping}

\textsuperscript{5}If weakly was omitted, the history constraint would also apply to all the additional methods of Triple.
In this subsection, we first look at what kinds of mutation can occur, and then use this analysis to explore the choices for sound modular reasoning.

Looking at an object’s state and the methods of a subtype and its supertypes, we can classify mutation of an object as follows.

**CS-CM** Mutation of common state by common methods. Example: the common function `incSecond` of `PairFI` mutates common state `second`.

**AS-AM** Mutation of additional state by additional methods. Example: the additional function `incThird` mutates additional state `third`.

**CS-AM** Mutation of common state by additional methods. Example: the additional function `incFirst` mutates the common state `first`.

**AS-CM** Mutation of additional state by common methods. Example: an override of common function `incSecond` could also mutate `third` as a side-effect, although this is not shown in the example.

When an object of the subtype is held in a variable of the supertype, if there is no aliasing, then only mutations of the forms CS-CM and AS-CM can occur, since a strongly-typed language will only allow the common methods to be invoked on the object. However, in the presence of aliasing, the additional methods can be called on an alias, and thus, mutations of the forms CS-AM, and AS-AM can occur.

Only mutations of the form CS-AM have the potential to cause problems for modular reasoning [30]. Mutations of the form CS-CM and AS-CM must obey the supertype’s specification of the common methods, and thus cannot cause problems. Mutations of the form AS-AM affect the additional state of the subtype, and are not visible from the supertype’s viewpoint. However mutations of the form CS-AM are observable through the supertype’s methods, and thus may produce unexpected results. This analysis leads to the following choices that permit sound modular reasoning.

**Aliasing Choice 1:** Restrict the behavior of the subtype’s additional methods so that they manipulate the common state in ways that are not surprising. This renders mutations of the form CS-AM harmless. Because such mutations are harmless, cross-type aliasing (and downcasting) presents no problems, and does not need to be prohibited.

**Aliasing Choice 2:** Prohibit multiple viewpoints on objects. Since a subtype object held in a supertype variable can only be manipulated from the supertype’s viewpoint, this prohibits additional methods of the subtype from being invoked on supertype variables, and so harmful mutations of the form CS-AM cannot occur. Because such mutations cannot occur on supertype variables, the behavior of the subtype’s additional methods need not be restricted.

We consider two ways of prohibiting multiple viewpoints on objects.

a. Prohibit cross-type aliasing. Unfortunately, with dynamic dispatch, the implicit receiver in an overridden method (this in Java) is a variable of a subtype that may have been viewed as a supertype object. Since it seems hard to prevent this statically, we consider the following.

b. Prohibit cross-type aliasing for all variables and fields except the implicit receiver of a method (this in Java). The multiple viewpoints that this may have do not cause harmful CS-AM mutations, even when this is used in message passing to call additional methods, because common methods in in a subtype must obey the specifications of the corresponding supertype methods they override. Thus clients cannot directly call additional methods, and implementations of common methods cannot cause unexpected behavior by calling such methods. So mutations of the form CS-AM either cannot occur, or are harmless.

These different aliasing choices lead to different reasoning principles, which we discuss below.

### 2.6 Modular Reasoning and Subtyping

One way to reason about OO programs modularly is to use supertype abstraction [24]. Supertype abstraction allows one to reason about client code using the static types of variables and the specifications of the these types [3, 30, 33, 34, 39, 37, 46, 45].

When is supertype abstraction sound? The analysis described above leads us to the following two choices for sound modular reasoning in the context of mutation and aliasing.

**Reasoning Choice 1:** Restrict the behavior of additional methods in subtypes following aliasing choice 1, and allow clients to reason using the invariants and history constraints stated for the static types of expressions. Clients are not allowed to reason using the pre- and postconditions of methods [30, p. 1812].

**Reasoning Choice 2:** Prohibit multiple viewpoints on objects, using aliasing choice 2(b), and allow clients to reason using the pre- and postconditions of methods taken from the specifications of the static types of expressions, as well as the invariants and history constraints of these types.

---

6For the moment, we ignore downcasting, which can also allow a program to manipulate an object from more than one viewpoint, in much the same way as aliasing. Our techniques for preventing multiple viewpoints on objects will also prevent downcasting from causing problems.

7We make no claims that these are the only available choices for reasoning.
These reasoning choices determine the notions of behavioral subtyping that are sound. For example, with reasoning choice 1, and aliasing choice 1, all additional methods of behavioral subtypes must satisfy the history constraint (and invariant) of their supertypes. This leads to Liskov and Wing’s history constraint definition of behavioral subtyping [32, 30]. Liskov and Wing’s other definition of behavioral subtyping, based on explaining how additional methods could be programmed using the supertype’s methods [31, 30], also uses reasoning choice 1, except that the history constraints available for use with this definition are not specified directly, but are all history constraints that are valid. We refer to Liskov and Wing’s definitions using the name strong behavioral subtyping.

In the remainder of this paper we focus on reasoning choice 2, which prohibits multiple viewpoints on objects as in aliasing choice 2(b). This has advantages for reasoning, since clients can reason about pre- and postconditions, and are not restricted to only reasoning about safety properties using just the invariant and history constraints of types, as they are with strong behavioral subtyping [30, p. 1812].

It also has the advantage of allowing more subtype relationships than strong behavioral subtyping. The additional subtype relationships are allowed because the behavior of the additional methods of subtypes are not constrained. Hence we call it weak behavioral subtyping [10, 12].

3. WEAK BEHAVIORAL SUBTYING

To define weak behavioral subtyping, we use the following notation. We use $\leq_w$ to refer to a weak behavioral subtype relation. Type symbols are represented by $S$ and $T$, with $S$ the subtype, and type vectors by $\vec{U}$ and $\vec{V}$. An invariant of a type $T$, by $I_T$, and a history constraint of $T$, by $C_T$. The notation $pre^T_m(\texttt{this}, \vec{x})$ denotes the precondition predicate specified for method $m$ in $T$, with receiver $\texttt{this}$ and additional parameters $\vec{x}$. Substituting $z$ for $y$ in predicate $p(y)$ is written as $p(z)$. By normal postcondition we mean the postcondition where no exceptions can be thrown. Similarly, the notation $\texttt{expost}_T^T(\texttt{this}, \vec{x})$ means the postcondition of method $m$ in $T$ that applies when exceptions can be thrown; such a postcondition can refer to the result object but the exact notation is unimportant. In relating pre- and postconditions, we use $\texttt{old}(e)$ for the value of $e$ in the pre-state.

The definition of weak behavioral subtyping given below is for single dispatching languages, like Java and C++, that do not support contravariance of arguments. It is also adapted to specification languages that, like JML, model objects as records (i.e., as a collection of named fields). In JML, the model of an object of a subtype inherits all of the fields used to model objects of its supertypes; this allows assertions used in the specification of its supertypes to be interpreted on subtype objects, without the need of an abstraction function. Alternatively, one can imagine that the abstraction function that maps values of the subtype to the supertype is always a projection, which forgets the subtype’s extra fields. A more general version of the definition below, which supports contravariance of arguments and specified abstraction functions is presented in [12]. The definition below can also be extended easily to multiple dispatching languages [10]. This definition uses ideas from [3, 30, 12, 10].

**Definition 3.1 (Weak Behavioral Subtyping).** A type $S$ is a weak behavioral subtype of $T$ with respect to a binary relation $\leq_w$ on types if and only if the following properties are satisfied.

**Syntactic:** For each non-static method $m$ of $T$, $S$ also has a method $m$ such that:

- Invariance of argument types. If the types of the additional arguments of $m$ in $S$ and $T$ are $\vec{U}$ and $\vec{V}$ respectively, then $\vec{U} = \vec{V}$.
- Covariance of result types. If the result types of $m$ in $S$ and $T$ are $U_r$ and $V_r$ respectively, then $U_r \leq_w V_r$.
- Covariance of exception result types. For each declared exception result type $E_r$ of $m$ in $S$, $m$ in $T$ has an exception result type $E_T$ such that $E_S \leq_w E_T$.

**Semantic:** The following implications have to hold the theory of the $S$’s specification.

- Invariant rule. For all objects $\texttt{this} : S$,
  $$I_S(\texttt{this}) \Rightarrow I_T(\texttt{this}).$$
- Constraint rule. For all objects $\texttt{this} : S$,
  $$C_S(\texttt{this}) \Rightarrow C_T(\texttt{this}).$$
- Methods rule. For all non-static methods $m$ of $T$, if the types of the additional arguments types of $m$ are $\vec{V}$ and if the result types of $m$ in $S$ and $T$ are $U_r$ and $V_r$ respectively, then for all objects $\texttt{this} : S$ and $\vec{y} : \vec{V}$, the following hold:
  - Precondition rule.
    $$pre^T_m(\texttt{this}, \vec{y}) \Rightarrow pre^S_m(\texttt{this}, \vec{y})$$
  - Normal postcondition rule.
    $$\texttt{old}(\texttt{pre}^T_m(\texttt{this}, \vec{y})) \Rightarrow \texttt{normpost}^S_m(\texttt{this}, \vec{y})$$
  - Exceptional postcondition rule.
    $$\texttt{old}(\texttt{pre}^T_m(\texttt{this}, \vec{y})) \Rightarrow \texttt{expost}^S_m(\texttt{this}, \vec{y})$$
The main distinction between strong and weak behavioral subtyping is in the interpretation of the history constraint. In JML, one specifies that the weak behavioral subtype interpretation of history constraints is desired by using the keyword `weakenly`, as in Figure 6; omitting this keyword gives a strong behavioral subtype. For strong behavioral subtyping, the history constraint applies to all non-static public methods of the subtype, including the additional methods; however, for weak behavioral subtyping the history constraint is only applied to the common non-static public methods. That is, for weak behavioral subtypes, the history constraint of the supertype only has to be valid for computations that do not invoke the additional methods of the subtype. Thus, although the constraint rule in the definition of weak behavioral subtyping is similar to the constraint rule in Liskov and Wing’s history constraint definition of strong behavioral subtyping [32, 30], the different meaning of these constraints explains the different effects they have on permitted subtype relationships.

Another way of interpreting the difference in the interpretation of history constraints is by viewing the supertype’s history constraint as part of the postcondition of each of its non-static public methods. In that case, when subtype methods are specified, because of the postcondition rule, the supertype’s history constraint should be satisfied by the common methods; however, for weak behavioral subtypes, it need not be satisfied by the subtype’s additional methods. Violation of the supertype’s constraints by the subtype’s additional methods will not be surprising because, according to aliasing choice 2(b), multiple viewpoints are not permitted.

As described above, `Triple` in Figure 6 is a weak behavioral subtype of the type `PairFI`. However, since the additional method `incFirst` of `Triple` does not satisfy the history constraint of `PairFI`, `Triple` is not a strong behavioral subtype of `PairFI`.

On the other hand, the type `BadPairSubtype` in Figure 3, is not a weak behavioral subtype of `PairFI` as the overloaded method `incSecond` in `BadPairSubtype` violates the postcondition rule. In section 5, we discuss more examples of weak behavioral subtype hierarchies.

4. ALIASING

In this section, we describe the alias restrictions of aliasing choice 2(b), and then we sketch a type system that enforces these alias restrictions. While the type system presented is somewhat restrictive, it does demonstrate that the aliasing choice 2(b) can be statically enforced, and that the enforcement is not so restrictive as to be unusable. Since the type system is not the main point of this paper, we only sketch it here.

4.1 Alias Restrictions

Aliasing in OO programs can be either aliasing between variables or aliasing between fields of objects. Figure 7 part (a) illustrates allowed and prohibited aliasing between variables, and between variables and fields of objects. As shown in part (a), a variable `x` of type `T` may refer to the same object as a variable `y` or a field `f` of the same declared type, but (unless `x` is “this”) it may not refer to the same object as a variable or field of a different type. Figure 7 part (b) illustrates aliasing between fields of two objects `a1.f : T` and `a2.g : T`. Again, a field may be aliased with a field of the same type, but not with fields of different types. In short these restrictions prevent cross-type aliases that lead to multiple viewpoints on objects (aside from those the viewpoints of the implicit receiver, “this”).

4.2 Enforcing Alias Restrictions

A type system that prevents cross-type aliases for client code in a multiple-dispatch language is presented in [10, 11]. In the remainder of this subsection we present a variant of that type system that is adapted to single dispatch languages.

Each expression has a static type and a viewpoint set. A viewpoint set is a conservative approximation to the set of
viewpoints through which an object may be manipulated. More concretely, a viewpoint set is a conservative approximation to the set of static types of fields and variables, other than “this”, which may reference the object. The notation \( E : T :: r \) means that \( E \) has static type \( T \) and viewpoint set \( r \).

In a language like Java and C++, some expressions are primitive values, not objects. For an expression of such a primitive value type, such as \( \text{int} \), the viewpoint set is empty, because there cannot be any observable aliasing of primitive values.

To prevent multiple viewpoints of an object, the viewpoint set of each field or variable, other than “this”, must be either \{ \}, when it is not assigned or a singleton set of its static type, when it is assigned. Hence there is no need to declare the viewpoint set of a variable or field. When used as an expression, the viewpoint set of field or variable reference, other than “this”, is the singleton set containing its static type.

The pseudo-variable “this”\(^ {10} \) is assigned by dynamic dispatch, and thus may reference objects that are viewed through multiple types. For example, suppose \( x \) has static type \( T \) and thus \( x \)'s viewpoint set is \( \{ T \} \); then when \( x \) denotes an object of dynamic type \( S \), where \( S \) is a subtype of \( T \), a call of the form \( x.m() \) may invoke a method of \( S \). Within the code of method \( m \) from type \( S \), \( \text{this} \) has static type \( S \), and hence is viewed through type \( S \), but the object also has the viewpoint \( T \) via \( x \).

Therefore, to be conservative, the type system must assume that the viewpoint set of \( \text{this} \) occurring in a method of a type \( S \) consists of \( S \), all supertypes of \( S \), and all potential subtypes of \( S \). We represent the potential subtypes of \( S \) by the special viewpoint \( \text{SubtypesOf}(S) \), which we specify as distinct from all other types. The important point to note for the discussion below is that the alias type set of the expression \( \text{this} \) is thus a set with at least 2 elements, the static type of \( \text{this} \), say \( S \), and \( \text{SubtypesOf}(S) \).

The most basic type checking rule is the assignment statement’s. If \( x \) is a variable of static type \( T \), an assignment of the form \( x = E \) is allowed only if \( E : S :: r, S \subseteq w:T, \) and \( r \subseteq \{ T \} \). Thus the viewpoint set, \( r \) of the expression, \( E \), can be either empty, or it may contain \( T \). Because the viewpoint set of \( \text{this} \) is a set with more than one element, the rule prevents one from assigning \( \text{this} \) to a variable or field.

To illustrate this rule, consider the Java code in Figure 8. On line 2, the expression \( \text{new Triple}(10, 20, 30) \) has an viewpoint set of \{ \}, and after the assignment on line 3, the viewpoint set of \( t \) is \{Triple\}. The assignment to \( p1 \) on line 3 is illegal because the object \( t \) would, if this assignment were permitted, be aliased by \( t \) and \( p1 \), so its viewpoint set would be \{PairFI, Triple\}; such non-singleton sets are prohibited because they indicate multiple viewpoints (i.e., cross-type aliasing). However, the next two assignments in lines 4 and 5 are valid, because the viewpoint sets of the expressions being assigned are \{ \} (on line 4) and \{PairFI\} (on line 5).

The arguments of a method are implicitly assigned to the formal parameters of the method. Thus the same considerations apply as for assignment. That is, when passing an actual parameter expression \( E \), to a formal of static type \( T \), the viewpoint set of \( E \) must be a subset of \( \{ T \} \). Again, since \( \text{this} \) has a viewpoint set containing at least two elements, it cannot be passed as an additional argument to a method. Overriding methods must, as in Java or C++, have the same parameter types; this invariance of method argument types allows the type checker to use the static type of the receiver in a method call to determine the formal argument types, which are the same in all overriding methods.

To obtain the viewpoint set for the result of a method call, each non-void method must declare the viewpoint set for its result. For this purpose, method declarations have an added \texttt{may alias} clause, which declares an upper bound on the viewpoint set of the method’s result.\(^ {11} \) Type checking ensures that the results that a method may return are a subset of its declared viewpoint set. For example, if a method has a clause of the form “\texttt{may alias} \{ \}” then at runtime it cannot return an object that is aliased. The declared viewpoint set must either be empty, or be a singleton type, because a result with multiple viewpoints could never be assigned to a variable or field. Thus returning \( \text{this} \) as a result is also prohibited, since it’s viewpoint set has at least two elements.

When a method overrides a method in a supertype, the viewpoint set declared for the overriding method must be a subset of the alias type set of the method it overrides. This allows the type checker to conservatively approximate the viewpoint set of the call using the declared viewpoint set for the method found in the declaration that corresponds to static type of the receiver. This rule does not allow a type mentioned in the \texttt{may alias} clause of an overriding method to be a subtype of a type mentioned in the \texttt{may alias} clause of the supertype. This restriction is necessary, because the purpose of the alias type set is to prevent cross-type aliasing.

\(^{10}\) The name \texttt{this} is used in Java, and is \texttt{*this} in C++, and \texttt{self} in Smalltalk. In languages, like Smalltalk, where \texttt{super} can be used as a synonym for \texttt{self} in some contexts, the remarks we make about \texttt{this} also apply to \texttt{super}.

\(^{11}\) In a language like C++, function declarations also need a \texttt{may alias} clause. Constructors also need such a clause, since they may cause aliases by assigning \texttt{this} to variables or fields.
ing, which could arise if overriding methods could change the viewpoint on the method’s result.

The same considerations described above for normal results also apply to exception results, that is, for the objects that are used in throwing exceptions. However, to simplify the type system, instead of adding may alias declarations for exception results, the type system should just require that the viewpoint sets for all exception results are empty. This corresponds to the usual practice of creating a new object when throwing an exception.

The type system also has rules for other expressions and statements. Casts are particularly interesting. Casts must be type-safe, as in Java, but the viewpoint set is not changed by a cast, since no new viewpoints are introduced; that is, 

\[(T:E): T::r\] if for some \(S, E: S::r\). Casts cannot change the viewpoint set of an existing object. In programming, one may need to clone the object, making a copy which has an empty viewpoint set. The clone can then acquire a different viewpoint, by assignment or parameter binding. For example, one could fix line 3 of Figure 8 by changing it to the following.

\[pl = new\ Triple(t.getFirst(), t.getSecond(), t.getThird());\]

The conservative nature of the type system can be seen in rules such as the one for conditional expressions, where the viewpoint set of the entire expression is the union of viewpoint sets of the alternatives; that is, 

\[(E_0 ? E_1 : E_2): T::r\] if for some \(b, r_1,\) and \(r_2, E_0; E_1; T::r_1, E_2; T::r_2,\) and \(r = r_1 \cup r_2.\)

The implicit receiver, this, cannot, by the rules described above, be assigned to any other variable or field, passed as an argument to a method, or returned as a result. These restrictions prevent the multiple viewpoints associated with different occurrences of this from escaping to other variables or fields in the program. This rules out certain linked data structures, which require this to be assigned to various fields. It also rules out double dispatching [17], which passes this as argument, and hence prevents the use of certain design patterns such as the visitor pattern [13].

It would thus be desirable to weaken our aliasing restrictions in such a way that the single viewpoint restriction is enforced, but which might permit linked data structures and passing this as an argument. One approach might be to use restrictions that are more semantic, such as those proposed by Leino and Stata [27]. Leino and Stata specify pivot objects, which cannot be aliased. To guarantee that these pivot objects are not aliased, they prohibit assigning arguments to these pivot fields and restrict assignments to these pivot object unless the result is not aliased. Our approach has some similarities, but we do not make a distinction among variables based on specifications, and we do not attempt to prevent aliasing, just cross-type aliasing. In other work, such as that by Müller and Poetzsch-Heffter [39, 37, 36], or the work of Noble, et al. [38], the aim of these type systems is not to prevent multiple viewpoints on objects, but rather to prevent certain kinds of aliasing, such as representation exposure. We leave combining our type system with such sophisticated alias control systems as future work, and hope that it may lead to less more flexible rules that are sufficient for weak behavioral subtyping.

Another avenue for future work on this type is system is ways of combining it with strong behavioral subtyping, which does not require aliasing restrictions. This may be another avenue to flexibility in practice.

5. BEHAVIORAL SUBTYPE EXAMPLES

In this section we present examples of behavioral subtype hierarchies. Though we classify these types into different categories based on our discussion in Section 2.5, one can have subtype hierarchies that fall under more than one category. Since every strong behavioral subtype is a weak behavioral subtype, we use less formal examples from [10] to illustrate CS-CM, AS-AM subtypes, which are examples of strong (and hence also weak) behavioral subtyping. For the CS-AM subtypes, which are weak behavioral subtypes that are not strong behavioral subtypes, we use formal examples.

5.1 CS-CM Subtypes

CS-CM subtypes are subtypes that have a common state with the supertype and only common methods that mutate the state.

Constraint subtypes Constraint subtypes are subtypes that restrict the set of abstract values that the model fields of the supertype can hold. Figure 9 part (a) shows a simple example of a constraint subtype. A Car is a Vehicle with a constraint on its size, weight, and number of wheels.

Deterministic Types An incompletely specified supertype can have more deterministic subtypes. Figure 9 part (b) shows an example of such deterministic subtype. The choose method of Bag is specified to be nondeterministic and can return any element that belongs to the Bag. However, a subtype List of Bag has a deterministic choose method that returns the last element inserted.

Bounded subtypes An unbounded supertype can have bounded subtypes. Figure 9(c) shows a VaryingSequence.
whose bound can be set when an instance is created and a subtype \textit{BoundedSequence}, whose bound is fixed. The type \textit{BoundedSequence} is a weak behavioral subtype of a \textit{Sequence}.

5.2 \textbf{AS-AM Subtypes}

Subtypes with additional methods that operate only on the additional state in the subtypes can also be thought of as extension subtypes. Figure 10 shows an example of such a subtype relationship. The subtype \textit{ImmutableTriple} extends \textit{ImmutablePair} with an additional third component and an observer for the third component. Hence, \textit{ImmutableTriple} is both a strong and a weak behavioral subtype of the type \textit{ImmutablePair}.

5.3 \textbf{AS-CM Subtypes}

An AS-CM subtype has an overridden common method that mutates its additional state. In Figure 11 when \textit{negate} is invoked on a \textit{TwoDPoint} only the $x$-coordinate and $y$-coordinate are mutated. However if one calls \textit{negate} on a \textit{ThreeDPoint}, it mutates the $z$-coordinate also.

5.4 \textbf{CS-AM Subtypes}

For weak behavioral subtyping, the additional methods in the subtype need not preserve the history constraint of the supertype. Hence weak behavioral subtyping allows subtypes that can mutate the common state with their additional methods. CS-AM subtypes are examples of subtypes that are weak behavioral subtypes and not strong behavioral subtypes.

Figure 12 shows an example of a CS-AM subtype hierarchy. Figures 13, 14, 15, and 16 give the formal specifications of these types in JML. The first four JML annotations in Figure 13 declare the model instance fields, that is the state, of an \textit{ImmutableStudentRecord}. The history constraint of \textit{ImmutableStudentRecord} states that none of its fields is mutable. The subtypes \textit{AdmissionsStudentRecord}, \textit{FinancialStudentRecord}, and \textit{StudentRecord} each provide a different view of \textit{ImmutableStudentRecord} with their additional methods. For example, the additional methods for \textit{AdmissionsStudentRecord} in Figure 14, \textit{changeAddress}, \textit{setHighSchoolGPA}, and \textit{admit}, mutate the fields \textit{address}, \textit{highSchoolGPA}, and \textit{admitted} respectively, provide a way an admissions office can observe and mutate these model fields. Note that the additional methods in the subtype \textit{AdmissionsStudentRecord} do not preserve the history constraint of \textit{ImmutableStudentRecord}. This is a weak behavioral subtype relation that is not a strong behavioral subtype relation. Similarly, \textit{FinancialStudentRecord} is a weak behavioral subtype of \textit{ImmutableStudentRecord} and \textit{StudentRecord} is a weak behavioral subtype of both types \textit{AdmissionsStudentRecord} and \textit{FinancialStudentRecord}.

Fig 10 illustrates a subtype hierarchy between tuple types with varying degrees of mutability. A \textit{MutablePair} and an \textit{ImmutablePair} share a common state but the \textit{MutablePair} object has additional methods that can mutate its state. Similarly, in Figure 10, \textit{MutableTriples} is a weak behavioral subtype of \textit{SemiMutableTriples} and \textit{MutableTriples} is a weak behavioral subtype of \textit{SemiMutableTriples}.

Another example is the \textit{const} modifier, which, as in C++, takes the methods that have side-effects on objects out of a type’s interface. So \textit{T} is a weak behavioral subtype \textit{const T}, but \textit{T} is not a strong behavioral subtype of \textit{const T}.

6. \textbf{RELATED WORK}

Liskov and Wing [30] were the first to point out the key problem of aliasing (or concurrency), which allows the ad-
public interface ImmutableStudentRecord {
    //@ public model instance double acctBalance;
    //@ public model instance String address;
    //@ public model instance float highSchoolGPA;
    //@ public model instance boolean admitted;
    
    /*@ public invariant address != null
     *   && 0.0 <= highSchoolGPA
     *   && highSchoolGPA <= 4.5;
     */
    
    /*@ public constraint
     *   acctBalance == \old(acctBalance);
     *   address == \old(address);
     *   highSchoolGPA == \old(highSchoolGPA);
     *   admitted == \old(admitted);
     */
    
    /*@ public normal_behavior
     *   ensures \result == acctBalance;
     */
    public int getAcctBalance();
    
    /*@ public normal_behavior
     *   ensures \result == address;
     */
    public String getAddress();
    
    /*@ public normal_behavior
     *   ensures \result == highSchoolGPA;
     */
    public int getHighSchoolGPA();
    
    /*@ public normal_behavior
     *   ensures \result == admitted;
     */
    public boolean getAdmitted();
}

Figure 13: A JML specification for the Java interface ImmutableStudentRecord.

public interface AdmissionsStudentRecord extends ImmutableStudentRecord /*@ weakly */ {
    /*@ public constraint
     *   acctBalance == \old(acctBalance);
     */
    
    /*@ public normal_behavior
     *   requires addr != null;
     *   assignable address;
     *   ensures address.equals(addr);
     */
    public void changeAddress(String addr);
    
    /*@ public normal_behavior
     *   requires 0.0 <= gpa && gpa <= 4.5;
     *   assignable highSchoolGPA;
     *   ensures highSchoolGPA == gpa;
     */
    public void setHighSchoolGPA(float gpa);
    
    /*@ public normal_behavior
     *   requires !admitted;
     *   assignable admitted;
     *   ensures admitted;
     */
    public void admit();
}

Figure 14: A JML specification for the Java interface AdmissionsStudentRecord.
public interface FinancialStudentRecord
extends ImmutableStudentRecord /*@ weakly @*/ {
    /*@ public constraint
    @  highSchoolGPA == \old(highSchoolGPA)
    @  && admitted == \old(admitted);
    @*/
    /*@ public normal_behavior
    @  requires amt >= 0.0;
    @  assignable acctBalance;
    @  ensures acctBalance
    @    == \old(acctBalance + amt);
    @*/
    public void credit(double amt);
    /*@ public normal_behavior
    @  requires addr != null;
    @  assignable address;
    @  ensures address.equals(addr);
    @*/
    public void changeAddress(String addr);
}

Figure 15: A JML specification for the Java interface FinancialStudentRecord.

public interface StudentRecord
extends FinancialStudentRecord /*@ weakly @*/,
AdmissionsStudentRecord /*@ weakly @*/ {
}

Figure 16: A JML specification for the Java interface StudentRecord.

public class Observe3 {
public static boolean obsFunc3(PairFI p,
IncSecond s) {
    if (p == null || s == null) {
      return true;
    } else {
      int second = p.getSecond();
      s.incSnd();
      return second == p.getSecond();
    }
}
}

Figure 17: An observation function, obsFunc3.

public interface IncSecond {
    //@ model instance int snd;
    /*@ public normal_behavior
    @  requires snd >= 0.0;
    @  assignable snd;
    @  ensures snd == \old(snd + 1);
    @*/
    public void incSnd();
}

Figure 18: The interface IncSecond.java
ditional methods of the subtype to cause observable state changes in a supertype object. Since their notion of strong behavioral subtyping is discussed throughout the paper, in this section we restrict ourselves to showing that with strong behavioral subtyping and no restrictions on aliasing, clients cannot use pre- and postconditions for modular reasoning.

To see why clients cannot use pre- and postconditions for modular reasoning with strong behavioral subtyping, consider the observation in Figure 17. This uses the types PairFI, in Figure 2, and IncSecond in Figure 18. Reasoning using the pre- and postconditions of the unrelated types PairFI and IncSecond, one can conclude that obsFunc3 always returns true.

However, consider what happens when a subtype TripleFI is added. This type is a strong behavioral subtype of both PairFI and IncSecond. Invoke the observation as follows.

TripleFI t = new TripleFI(3, 4, 5);
Observe3.obsFunc3(t, t)

When the above code is executed, an alias is created within obsFunc3, between p and s, and the observation returns false. Again, one could reason about all such possible aliasing patterns, but that would not be modular. Thus, with only strong behavioral subtyping and no restrictions on aliasing, reasoning based on the pre- and postconditions is not

14
public class TripleF1 extends PairF1
  implements IncSecond {
  protected /*@ spec_public @*/ int third;
  //@ public depends snd <- second;
  //@ public represents snd <- second;
  
  /**@ public normal_behavior
    * assignable first, second, third;
    * ensures first == fst && second == snd
    * && third == thd;
    * @*/
  public TripleF1(int fst, int snd, int thd) {
    super(fst, snd);
    third = thd;
  }

  /**@ public normal_behavior
    * ensures \result == third;
    * @*/
  public int getThird() {
    return third;
  }

  /**@ also
    * assignable third;
    * ensures third == \old(third + 1);
    * @*/
  public void incThird() {
    third++;
  }

  /**@ public normal_behavior
    * assignable second;
    * ensures second == \old(second + 1);
    * @*/
  public void incSnd() {
    second++;
  }
}

Figure 19: Java code for TripleF1.java.

valid. Of course, Liskov and Wing do not claim that such reasoning is valid, and make it clear that they only consider safety properties guaranteed by invariants and history constraints [30, p. 1812]. Note that our conclusions about the expected result of obsFunc3 were invalid precisely because they tried to use more than just the invariant and history constraints of the types involved.

Lewerentz and his colleagues [28] use refinement calculus to define simulations on programs that are observations on types. They do not consider aliasing or interference. Mikha-
jlova and her coauthors [35] present sound verification of OO programs in a refinement calculus framework. However, their work is based on class refinement and treating classes as types restricts both subclasses and subtypes [42].

Abadi and Leino [2] extend the work of Cardelli’s [8] structural subtyping rules on records to include behavior. They present an axiomatic semantics and provide guidance on reasoning about OO programs. However, their approach is not modular.

Recently, Huisman [16] and Oheimb [47] have given sound (and in the case of Oheimb, relatively-complete) verification logics for Java. However, these do not allow one to verify code in a way that is modular with respect to patterns of potential aliases among different types, as we do. Furthermore, Oheimb’s work does concern itself with modularity.

The work of Müller and Poetzsch-Heffter [39, 37, 36] has a verification logic for Java that has also been proved to be sound. The focus of this work is on modularity, in particular for checking frame axioms (like JML’s assignable clause) and invariants. They control aliasing through a “universe type system.” They use supertype abstraction in reasoning about code using the pre- and postconditions of methods, but do not consider history constraints, and hence are not concerned with the effect of such constraints on additional methods of a subtype. Although their reasoning technique allow modular verification with what are effectively weak behavioral subtypes, their alias control techniques do not allow one to limit multiple viewpoints. Hence their notion of modularity does not extend to patterns of potential aliases among different types.

For a more comprehensive discussion on behavioral subtyping in general, refer to [21].

7. CONCLUSIONS

The main contributions of this paper are a comprehensive discussion of issues related to mutation, aliasing, subtyping, and modular reasoning, and a more flexible notion of behavioral subtyping—weak behavioral subtyping. Weak behavioral subtyping permits more subtype relations and allows modular reasoning based on the pre- and post condi-
tions of the types. However, for weak behavioral subtyping cross-type or multiple viewpoint aliasing should be re-
stricted. We have demonstrated one way to enforce these restrictions statically. Weak behavioral subtyping permits several useful subtype relations, such as types with mutable objects as subtypes to types with immutable objects.
Acknowledgments
Thanks to Art Fleck, Ken Slonneger, and Cesare Tinelli at the University of Iowa for comments on a draft of this paper. Thanks to Peter Müller for comments on his work.

8. REFERENCES


APPENDIX

A. EQUIVALENT RULES FOR POSTCONDITIONS

The equivalence of our postcondition rule and Formula (1) is an immediate consequence of the following lemma.

**Lemma A.1.** For all $S_{pre}$, $S_{post}$, $T_{pre}$, and $T_{post}$,

$$(S_{pre} \Rightarrow S_{post}) \Rightarrow (T_{pre} \Rightarrow T_{post})$$

is equivalent to

$$T_{pre} \Rightarrow ((S_{pre} \Rightarrow S_{post}) \Rightarrow T_{post}).$$

**Proof:** Let the predicates be given. We calculate as follows.

\[
\begin{align*}
(S_{pre} \Rightarrow S_{post}) \Rightarrow (T_{pre} \Rightarrow T_{post}) & = (by \ P \Rightarrow (Q \Rightarrow R) \equiv (P \land Q) \Rightarrow R) \\
((S_{pre} \Rightarrow S_{post}) \land T_{pre}) \Rightarrow T_{post} & = (by \ symmetry \ of \ conjunction) \\
(T_{pre} \land (S_{pre} \Rightarrow S_{post})) \Rightarrow T_{post} & = (by \ (P \land Q) \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)) \\
T_{pre} \Rightarrow ((S_{pre} \Rightarrow S_{post}) \Rightarrow T_{post}) &
\end{align*}
\]