Preventing Cross-Type Aliasing for More Practical Reasoning

Krishna Kishore Dhara and Gary T. Leavens

TR #01-02a
March 2001, Revised November 2001

Keywords: Cross-type aliasing, viewpoint restriction, weak behavioral subtyping, strong behavioral subtyping, aliasing, mutation, modularity, specification, verification, Java language, JML language.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — Object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented langauges; D.3.3 [*Programming Languages*] Language Constructs and Features — classes and objects, inheritance; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques;

Submitted for publication

An earlier version of this paper was titled "Mutation, Aliasing, Viewpoints, Modular Reasoning, and Weak Behavioral Subtyping." This paper is also Avaya Labs Research Technical Report ALR-2001-025.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Preventing Cross-Type Aliasing for More Practical Reasoning

Krishna Kishore Dhara[1] and Gary T. Leavens[2]

[1] Avaya Labs Research, 1D-632A,
101 Crawfords Corner Rd., Holmdel, NJ 07733 USA
dhara@avaya.com, +1 732 817 2160
[2] Department of Computer Science, Iowa State University,
226 Atanasoff Hall, Ames, IA 50011 USA,
leavens@cs.iastate.edu, +1 515 294 1580

Abstract. To reason about the correctness of a method when cross-type aliases are possible, one must not only consider all possible patterns of aliasing among the method's arguments, but all possible ways in which these types' abstract (specification-only) fields may be aliased. Because of the large number of such aliasing possibilities, and because of the complications they cause for reasoning, cross-type aliases make the use of method specifications impractical in reasoning about correctness. Hence, existing work on behavioral subtyping either ignores aliasing or prohibits the use of method specifications in reasoning We present a simple type system that prohibits cross-type aliases, and thus eliminates these problems. The "viewpoint restriction" enforced by this type system supports a less restrictive notion of behavioral subtyping — weak behavioral subtyping. Weak behavioral subtyping allows types with immutable objects (e.g., immutable sequences), to have behavioral subtypes with mutable objects (e.g., mutable arrays). Thus, besides permitting one to reason with method specifications, the viewpoint restriction also permits a more flexible and useful notion of behavioral subtyping.

Key words: Cross-type aliasing, viewpoint restriction, weak behavioral subtyping, strong behavioral subtyping, 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's client code to operate on objects of these new subtypes using the protocol of their supertypes. However, the type systems of OO languages, like C++ [46] and Java [3], do not take the semantics of types into account; that is, the type system only guarantees the lack of runtime type errors [7], not the lack of behavior that would be surprising, given a specification of the supertype's behavior.

A well-known technique for preventing such surprising behavior in a modular way is behavioral subtyping [1, 2, 6, 11, 12, 21–24, 31, 34, 35, 47, 48]. Behavioral

subtyping ensures that objects of the new subtypes "act like" objects of their supertypes when manipulated as if they were supertype objects.

However, when mutable objects are considered, different assumptions about aliasing[1] lead to different expectations about the behavior of objects, which lead to different reasoning principles and different notions of behavioral subtyping.

For example, one practical and modular technique for reasoning in the presence of unrestricted aliasing is to only expect that objects obey various safety properties [31]. For purposes of this paper, we define a *safety property* of an object as either an *invariant* property, which is true in all its publicly-visible states, or a *history constraint* [31], which is a property that relates earlier and later publicly-visible states. History constraints are necessarily monotonic; for example, a history constraint might say that the `age` field of a `Person` object can only increase. When one only expects safety properties to hold, surprising behavior means violation of an invariant or history constraint. Liskov and Wing's notion of strong behavioral subtyping [31] can be used to modularly prevent such surprising behavior.

In this paper we consider how reasoning is aided by a slightly stronger assumption about aliasing. This assumption is a viewpoint restriction. A *viewpoint* on an object is the (minimal) static type of a variable (other than "`this`") or a field that refers to that object, i.e., the set of all types through which one can refer to the object. At any given point in a program's execution, an object's *viewpoint set* is the set of all its viewpoints. For example, after executing the assignment `Collection c = new LinkedList()` the viewpoint set of the object created is {`Collection`}. The viewpoint restriction that we consider in this paper restricts the viewpoint set of each object to be either empty or a singleton; that is, an object may not be viewed through variables (other than "`this`") or fields of different types. We discuss modular ways to enforce this viewpoint restriction in Section 5.

When the viewpoint restriction is enforced by a programming language, one only has to reason about possible aliases among variables and fields of the same type. In this case it is easier to prove the absence of certain aliases. Furthermore, one does not have to consider how the abstract fields used in the specifications of different types may be aliased, as discussed in Section 3.3 below. This makes modular reasoning about properties other than safety properties practical. We call properties other than safety properties *liveness properties*; liveness properties include method specifications. Method specifications include pre- and postconditions, which are assertions about the states before and after a method call.

Because the viewpoint restriction is a stronger assumption about aliasing, it makes weaker, i.e., less restrictive demands on behavioral subtyping. So even though one may expect both safety and liveness properties to hold, what we call weak behavioral subtyping [10–12] permits more behavioral subtype relations than Liskov and Wing's notion of strong behavioral subtyping [31]. For example, a type with immutable objects (e.g., immutable sequences), cannot have strong

[1] Concurrency can also lead to similar considerations [31], which we ignore in this paper.

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. On the other hand, when the viewpoint restriction holds, every strong behavioral subtype is also a weak behavioral subtype.

The main contribution of this paper is its discussion of various issues in modular reasoning for OO languages related to mutation and aliasing; in particular it describes the advantages of the viewpoint restriction. The paper also describes the notion of weak behavioral subtyping, and how it compares to strong behavioral subtyping; although some discussion of weak behavioral subtyping has appeared in print before [10–12], what is new here is a clear description of the advantages for reasoning that weak behavioral subtyping has over strong behavioral subtyping, due to the viewpoint restriction. Furthermore, this paper specializes the treatment of weak behavioral subtyping to single dispatch languages (as opposed to multiple dispatch languages like CLOS and Cecil), unlike our previous work on weak behavioral subtyping. Finally, the type system that enforces the viewpoint restriction is new with this work.

This paper attempts to convey the ideas of viewpoints and 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 of the soundness of reasoning with the viewpoint restriction and weak behavioral subtyping, in a model-theoretic setting, has been given elsewhere [10].

In the next section, we present a brief introduction to JML, which is used to discuss formal reasoning in this paper. Section 3 discusses the reasoning problem and discusses the choices for modular reasoning in the context of mutation and aliasing. Section 4 discusses weak behavioral subtyping and gives some examples. In section 5 we discuss one way to enforce restrictions on aliasing necessary for sound modular reasoning with weak behavioral subtyping. Finally, we present related work and conclusions.

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 abstract values [15, 30, 54]. In addition one can specify other properties, such as invariants and history constraints [31].

This paper uses the Java Modeling Language (JML) [19, 20, 43] for such specifications. JML is a behavioral interface specification language tailored to Java; it is based on Larch [14] and Eiffel [35]. An example of a JML specification is given in Fig. 1. This specifies the behavior of class `PairFI`, which we will use as a running example of a supertype in the paper.

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

```
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++;
    }
}
```

Fig. 1. A JML specification and Java implementation of the class `PairFI`.

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 [31], which states that the value of `first` may not be changed once initialized.

In JML, the specification of a method or a constructor can precede its code. In Fig. 1, the constructor's specification illustrates a fairly general form of method specification, introduced by the keyword `behavior`. A `behavior` specification 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` clause 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 precondition holds). An expression of the form `\old(`E`)` denotes E's value in the state at the start of the method's execution[2] [35]. 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 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 `\result` means the value or object returned by the method, as in the specifications of `first` and `second`.

3 Choices for Modular and Practical Reasoning

In this section, we describe the problem in more detail, and lay out various choices for modular and practical reasoning techniques. We start with a discussion of the problems caused by multiple viewpoints, which motivates the viewpoint restriction. We then further motivate the viewpoint restriction by noting that it permits more behavioral subtype relations. Finally, with the viewpoint restrictions and with different notions of behavioral subtyping we enumerate several reasoning choices for OO software.

3.1 Problems Caused by Multiple Viewpoints

To see the problems caused by multiple viewpoints on objects consider Fig. 2. In this figure `obsFunc1` takes an argument of type `IncFirst`, specified in Fig. 3. The first JML annotation in Fig. 3 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`).

[2] 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 method postconditions [31].

```
public class Observe1 {
    public static boolean obsFunc1(PairFI p, IncFirst t) {
        if (p == null || t == null) {
            return true;
        } else {
            int first = p.getFirst();
            t.incFirst();
            return first == p.getFirst();
        }
    }
}
```

Fig. 2. An observation on `PairFI` and `IncFirst`.

```
public interface IncFirst {
    //@ model instance int fst;

    /*@ public normal_behavior
      @   assignable fst;
      @   ensures fst == \old(fst + 1);
      @*/
    public void incFirst();
}
```

Fig. 3. The interface `IncFirst`.

Treating `obsFunc1` in Fig. 2 as a partial specification involves the safety property that the first component of the object `p`, which is viewed through static type `PairFI` is immutable. Treating `obsFunc1` as client code, we want to be able to reason about it using the specification of `PairFI` given in Fig. 1, since the static type of `p` is `PairFI`. `PairFI`'s specification has a history constraint that says that the field `first` cannot change. Thus, using this viewpoint, we do not expect `p.first` to change. Additionally, since `IncFirst` is unrelated to `PairFI`, there is no reason to expect any interaction between `p` and `t`. At least, it is easy to see how one could overlook such possible interactions.

The problem is that, because of the possible aliasing between the two arguments to `obsFunc1`, when new types are introduced into the program, another viewpoint on `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 Fig. 4. The connection between the model instance field `fst` and the field `first` inherited from `PairFI` is given by the

`depends` and `represents` clauses in Fig. 4 [20, 27, 25, 26, 37]. The depends clause says that the value of `fst` is determined by `first`, and hence that whenever `fst` is assignable, so is `first`. The represents clause says how the value of `fst` is recovered from `first`, in this case they are the same.

```
public class Triple extends PairFI /*@ 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++; }

}
```

Fig. 4. 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, 42, 35, 51–53]. 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`.

How can `Triple` be a behavioral subtype of `PairFI`, when it has a method, `incFirst`, that violates a safety property, the immutability of `first`, specified by `PairFI`'s history constraint? The answer depends on the assumptions one makes about aliasing when reasoning about client code, such as these observation functions.

Consider what happens when the same triple is passed in both arguments to `obsFunc1`, as in the following.

```
Triple t = new Triple(3, 4, 5);
Observe1.obsFunc1(t, t);
```

When the above code is executed, an alias is created, within `obsFunc1`, between `p` and `t` and the observation returns `false`. Depending on what assumptions one has made about aliasing, either:

- the above code is legal, and the unexpected result indicates that there is something wrong our reasoning or with the types involved, or
- the above code is not legal, in which case there is no unexpected result to worry about, and this example does not show any problem with our reasoning or with the types involved, but instead shows a problem with the programming language.

Suppose the above code is legal, i.e., that the programming language allows aliasing between `p` and `t` in client code like `obsFunc1`. Then we can look for a problem in either our reasoning or in the types involved.

If our reasoning about `obsFunc1` tells us that it should always return `true`, when in fact it returns `false` for the above code, then our reasoning technique is unsound. The unsoundness arises because we did not analyze every aliasing possibility in the code of `obsFunc1`. In particular we ignored aliasing between variables of unrelated types, such as `PairFI` and `IncFirst`. Although considering all aliasing possibilities is modular, there are an exponential number of such aliasing possibilities among variables and fields.[3] Worse, this technique seems error prone if applied informally. There are two reasons for this. First, if there are a large number of aliasing possibilities, one is likely to overlook some of them. (However, a tool could force one to consider all aliasing possibilities.) Second, most of these aliasing possibilities, such as those in `obsFunc1`, will seem utterly pointless, since the types involved are not related. So in this case it would be helpful if there were some way to cut down the number of aliasing possibilities; furthermore, if cross-type aliases are permitted, tool support also seems essential.

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 possibilities. In terms of the example, one would have to go back to client code when `Triple` was introduced as a

[3] Although there are an unbounded number of possible types, like `Triple` that might be subtypes of such unrelated types, one only needs to consider the possibility of any combination of variables being aliased through a common subtype; the exact subtype does not matter.

subtype of `PairFI` and `IncFirst`. However, this approach is not modular, so we reject it.

Still assuming that the above code is legal, we could retain sound modular reasoning by finding a problem with the types involved. The most reasonable possibility seems to be to say that `Triple` is not a behavioral subtype of `PairFI`.

If, instead, we wish to have sound modular reasoning without considering cross-type aliasing in our reasoning, and if we wish `Triple` to be a behavioral subtype of `PairFI`, we are forced to the other possibility—rejecting the above observation code. In this case we again have two choices. We can either say that the call to `obsFunc1` is illegal, or we can reject `obsFunc1` itself.

We might reject the call to `obsFunc1` in the above code because we restrict aliasing in various ways. The weakest such restriction that works for this example is to prohibit cross-type aliasing; that is to enforce the restriction that every object is viewed through variables of at most one static type. Since `p` and `t` in `obsFunc1` have different types, this viewpoint restriction would prohibit the call in the above code. Stronger restrictions, for example prohibiting aliasing completely, would also have the effect of rejecting the call.

If we wish the call in the above code to be legal, then we make the code of `obsFunc1` illegal. For example, one could say that `p` and `t` should have been declared using `arg` types in the sense of flexible alias protection [40], and hence that `obsFunc1` cannot legally call methods that access mutable state (such as `incFirst`).

Therefore there are four options for sound modular reasoning about the above code. These same four options apply in general, and fall into two general categories as follows.

1. Allowing unrestricted aliasing and:
   (a) providing tool support that limits the number of aliasing possibilities (and preferably allows programmers to ignore cross-type aliases), or
   (b) restricting behavioral subtyping so that, for example, `Triple` is not a behavioral subtype of `PairFI`.
2. Restricting aliasing so that:
   (a) objects are referred to through at most one viewpoint, or
   (b) observation of an object's mutable state is not possible when it is aliased.

In this paper we pursue only options 1(b) and 2(a), because we wish to contrast the effects of different reasoning choices on the different notions (strong and weak) of behavioral subtyping. We thus leave the other options, and various combinations as future work.

3.2 Mutation, Subtyping, and Aliasing

In this subsection we refine the statement of our chosen options for modular reasoning by looking at whether the methods that mutate an object's state are in the supertype or not, and whether the state being mutated is in the supertype or not.

Looking at an object's state and the methods of a subtype and its supertypes, we can classify the possible mutations of an object into the following four types.

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 the 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 whose type is a 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.[4] 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 a client's modular reasoning [31]. 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, but are not observable through the supertype's methods. However mutations of the form CS-AM are observable through the supertype's methods, and thus may produce unexpected results. This allows us to refine our decisions for sound modular reasoning into the following.

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 variable of the supertype 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.

How can a language prohibit multiple viewpoints on objects? The direct approach seems to be to just prohibit cross-type aliasing. Unfortunately, with

[4] For 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.

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 multiple viewpoints for the implicit receiver of a method, we only prohibit cross-type aliasing for other variables and fields. That is we 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 a subtype must obey the specification of the corresponding supertype methods they override. (Here we are assuming some form of behavioral subtyping.) 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.

The two aliasing choices induce different reasoning principles, which we discuss below.

3.3 Reasoning Principles Induced by the Aliasing Choices

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 these types [2, 31, 34, 35, 41, 38, 48, 47].

The following kinds of supertype abstraction are both sound and practical with respect to each of the aliasing choices.

Reasoning Choice 1: Restrict the behavior of additional methods in subtypes, as in 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 method specifications [31, p. 1812]. The reason for this prohibition is that allowing the use of method specifications is unsound without considering all possible cross-type aliases, and, as we demonstrate below, considering all such aliases is impractical.

Reasoning Choice 2: Prohibit multiple viewpoints on objects, using aliasing choice 2, and allow clients to reason using method specifications taken from the specifications of the static types of expressions, as well as the invariants and history constraints of these types.

These reasoning choices determine two sound and practical notions of behavioral subtyping. For example, with reasoning choice 1, all additional methods of behavioral subtypes must satisfy the history constraint (and invariant) of their supertypes. This leads to a less restrictive form of Liskov and Wing's history constraint definition of behavioral subtyping [33, 31], which we call "minimal strong behavioral subtyping." In *minimal strong behavioral subtyping*, a subtype's methods must obey the invariant and history constraint of its supertypes, but Liskov and Wing's "methods rule" is not imposed. When the methods rule is imposed as well, we call their definition *strong behavioral subtyping*. Liskov and Wing's other definition of strong behavioral subtyping, based on explaining

how additional methods could be programmed using the supertype's methods [32, 31], also uses reasoning choice 1, except that the history constraints available for use with this definition are not specified directly.

The problem with reasoning choice 1 and with strong behavioral subtyping is that clients cannot use method specifications without considering all possible cross-type aliases. Indeed, the complications are worse than that, because one must consider not only cross-type aliases among objects, but for each such cross-type alias, one must consider all possible dependencies among model fields. To see this, consider the observation function in Fig. 5. This observation uses the types `PairFI`, from Fig. 1, and `IncSecond` from Fig. 6.

```
public class Observe2 {
    public static boolean obsFunc2(PairFI p, IncSecond s) {
        if (p == null || s == null) {
            return true;
        } else {
            int second = p.getSecond();
            s.incSnd();
            return second == p.getSecond();
        }
    }
}
```

Fig. 5. An observation function, `obsFunc2`.

```
public interface IncSecond {
    //@ model instance int snd;

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

Fig. 6. The interface `IncSecond`.

How is one to reason about the code in Fig. 5 using the method specifications of the types `PairFI` and `IncSecond`? One has to consider all possible aliases

among the arguments, and for each such aliasing pattern, one has to consider all possible aliases among model fields used in the specifications. In this example one must consider whether the model instance field `s.snd` can be aliased to `p.first` or `p.second`. In JML, such aliasing would be declared with a `depends` declaration [20, 27, 25, 26, 37]. When a specification such as that in Fig. 6 allows `incSnd` to assign to a model field such as `snd`, it means that a correct implementation may assign to any location on which `snd` depends, i.e., on its *dependees*. There are thus three possibilities. If there is no aliasing, that is, if `s.snd` does not depend on `p.first` or `p.second`, then the code in Fig. 5 will return `true`, since the frame axiom in Fig. 6 says that only the `snd` field and its dependees can be assigned to by the call `s.incSnd()`. Using the definition of strong behavioral subtyping, one can rule out the possibility that `s.snd` depends on `p.first`, since in that case the specification of `incSnd` would violate the history constraint of `PairFI`. The other possibility is that `s.snd` depends on `p.second`. In this case, the specification of `incSnd` in Fig. 6 says that `p.second` can be assigned (as a dependee) and thus that `p.second` may be changed by the call (whether or not this happens would depend on the exact way that `s.snd` is represented by `p.second`). Thus, the result of `obsFunc2` might not be true in this case.

This last possibility occurs if, for example, one adds the subtype `TripleFI`, which is a strong behavioral subtype of both `TripleFI` and `IncSecond`. In this case, when one invokes the observation function as follows,

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

an alias is created within `obsFunc2`, between `p` and `s`, and the observation returns false. This shows that such dependencies are not just a theoretical idea, but can cause observable effects.

In summary, when reasoning using method specifications and strong behavioral subtyping, one must consider each possible aliasing pattern and each possible dependency among the fields of the potentially aliased objects. While this is modular, it is surely complex. In particular, the predicates describing states in general will contain one case for each such possible aliasing pattern and each of the possible dependencies within those aliasing patterns. For client code containing many objects and many model fields, this is impractical. When applied informally, such reasoning is likely to be error prone. It is certainly simpler, more practical, and less error prone to follow Liskov and Wing and only allow reasoning using the safety properties guaranteed by invariants and history constraints when using strong behavioral subtyping [31, p. 1812].

Another way to look at this example is that it shows the advantage of prohibiting cross-type aliasing, as in reasoning choice 2. When this is done, i.e., when each object has at most one viewpoint, then the number of aliasing patterns that one needs to consider are reduced. Furthermore, because there are no cases of cross-type aliasing, one does not have to consider possible dependencies among model fields. If two objects of the same type are aliased, their viewpoint's type declares all the relevant dependencies, and so no additional combinations of possible dependencies arise.

```
public class TripleFI extends PairFI 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 TripleFI(int fst, int snd, int thd) {
        super(fst, snd);
        third = thd;
    }

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

    /*@ also
      @   public normal_behavior
      @     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++;
    }
}
```

Fig. 7. Java code for `TripleFI`.

Reasoning choice 2 has advantages for reasoning, since clients can use method specifications, and are not restricted to reasoning about safety properties using just the invariant and history constraints of types, as with strong behavioral subtyping. The definition of behavioral subtyping induced by reasoning choice 2 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 the kind of subtyping induced by reasoning choice 2 *weak behavioral subtyping* [10, 12]. With reasoning choice 2, every strong behavioral subtype is a weak behavioral subtype, but not vice versa.

4 Weak Behavioral Subtyping

In this section we explain weak behavioral subtyping in more detail and give some examples.

4.1 Definition of Weak Behavioral Subtyping

A formal definition of weak behavioral subtyping that permits reasoning choice 2 and that permits all the types of mutation discussed in Section 3.2 is given in Appendix A.

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 `weakly`, as in the first line of Fig. 4; 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 [33, 31], the difference in the way history constraints are applied to the subtype's additional methods explains the different effects they have on permitted subtype relationships.

Another way of interpreting the difference in the application 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 this way, when subtype methods are specified, because of the postcondition rule, the supertype's history constraint should be satisfied by the common methods. In strong behavioral subtyping, the history constraint is thought of as part of the postcondition of each non-static public method of each subtype. However, for weak behavioral subtypes, the history constraint need not be satisfied by the subtype's additional methods. Nevertheless, violation of the supertype's history constraints by

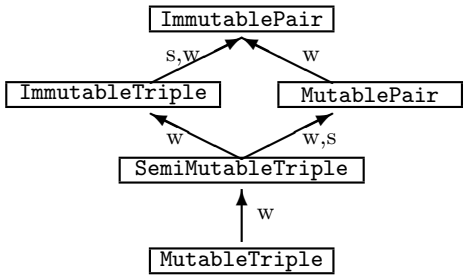

Fig. 8. Behavioral subtype relationships between tuple types. Arrows labeled with "w" are weak behavioral subtypes, those labeled with "s" are strong behavioral subtypes.

the subtype's additional methods will not be observable through the supertype's viewpoint, since the viewpoint restriction in aliasing choice 2 disallows CS-AM mutations.

4.2 Examples of Weak Behavioral Subtyping

As described above, `Triple` in Fig. 4 is a weak behavioral subtype of the type `PairFI` (from Fig. 1). 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, with the viewpoint restriction, every strong behavioral subtype is also a weak behavioral subtype. For example, consider Fig. 8, which shows several subtyping relationships. The subtype `ImmutableTriple` extends `ImmutablePair` with an additional third component and an observer for the third component. Hence, `ImmutableTriple` is both a strong and a weak behavioral subtype of the type `ImmutablePair`.

In Fig. 8 the types `MutablePair` and `ImmutablePair` share common state, but a `MutablePair` object has additional methods that can mutate its state. The history constraint on the supertype `ImmutablePair` says that its state is immutable, but with weak behavioral subtyping, this history constraint is only applicable to the common methods. Thus a `MutablePair` can be a weak behavioral subtype of `ImmutablePair`, although it could not be a strong behavioral subtype. Similarly, `SemiMutableTriple`, in which the third element is not mutable, is a weak behavioral subtype of both `ImmutableTriple` and `MutablePair`, but it is not a strong behavioral subtype of `ImmutableTriple`. Finally, `MutableTriple` is a weak behavioral subtype of `SemiMutableTriple`.

Fig. 9 is another example of a subtype hierarchy that would not be permitted by strong behavioral subtyping. Figures 10, 11, 12. and 13 give the formal specifications of these types in JML. The first four JML annotations in Fig. 10

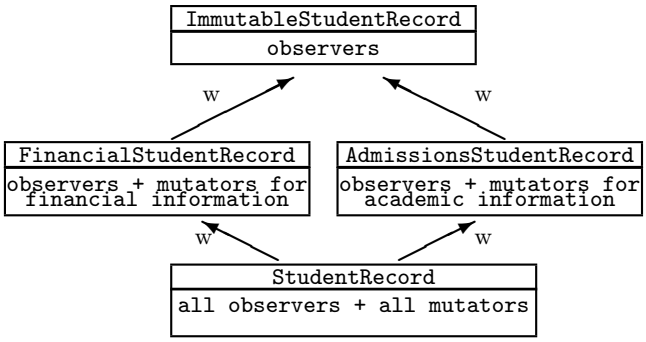

Fig. 9. Student records with weak behavioral subtype relations that provide multiple views for virtual supertypes.

declare the model instance fields of the type `ImmutableStudentRecord`. The history constraint of `ImmutableStudentRecord` states that none of its fields is mutable. Its subtypes `AdmissionsStudentRecord`, `FinancialStudentRecord`, and `StudentRecord` each provide a different view of `ImmutableStudentRecord` with their additional methods. For example, the additional methods of the type `AdmissionsStudentRecord` in Fig. 11, `changeAddress`, `setHighSchoolGPA`, and `admit`, mutate the fields `address`, `highSchoolGPA`, and `admitted` respectively, provide a way an admissions office can observe and mutate these model fields. Note that the additional methods in the subtype `AdmissionsStudentRecord` do not preserve the history constraint of the type `ImmutableStudentRecord`. This is a weak behavioral subtype relation that is not a strong behavioral subtype relation. Similarly, `FinancialStudentRecord` is a weak behavioral subtype of `ImmutableStudentRecord`, and `StudentRecord` is a weak behavioral subtype of both types `AdmissionsStudentRecord` and `FinancialStudentRecord`.

Another example is the `const` modifier, which, as in C++, takes the methods that have side-effects on objects out of a type's interface. So T is a weak behavioral subtype `const` T, but T is not a strong behavioral subtype of `const` T. Similarly, in flexible alias protection [40], the *arg* mode type modifier is such that T is a weak (but not strong) behavioral subtype of *arg* T.

5 Aliasing

In this section, we describe the alias restrictions of aliasing choice 2(b), which corresponds to weak behavioral subtyping. We then 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 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.

```
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();
}
```

Fig. 10. 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();
}
```

Fig. 11. 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 amt >= 0.0;
      @   assignable acctBalance;
      @   ensures acctBalance == \old(acctBalance - amt);
      @*/
    public void debit(double amt);

    /*@ public normal_behavior
      @   requires addr != null;
      @   assignable address;
      @   ensures address.equals(addr);
      @*/
    public void changeAddress(String addr);
}
```

Fig. 12. A JML specification for the Java interface `FinancialStudentRecord`.

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

Fig. 13. A JML specification for the Java interface `StudentRecord`.

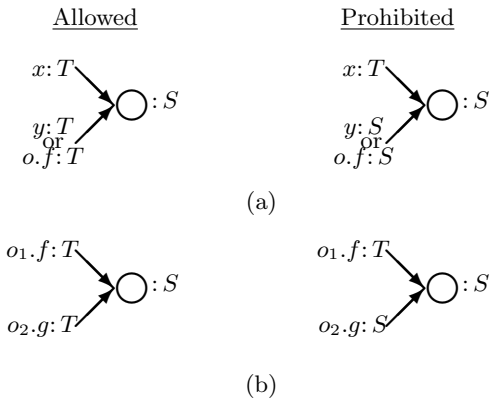

Fig. 14. A comparison of the kinds of aliases that are allowed and that are prohibited for weak behavioral subtyping. In the figure, x and y are variables that are distinct from "`this`", and the types S and T are distinct (and not necessarily related).

5.1 Alias Restrictions

Aliasing in OO programs can be either aliasing between variables or aliasing between fields of objects.[5] Fig. 14 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. Fig. 14 part (b) illustrates aliasing between fields of two objects $o_1.f : T$ and $o_2.g : T$. 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`").

5.2 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 new 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

[5] For purposes of this discussion, we think of static fields in classes as variables and array elements as fields of objects.

to the set of static types of fields and variables, other than "`this`", which may reference the object. The notation $E\colon T :: r$ means that E has static type T and viewpoint set r.

In a language like Java or C++, some expressions are primitive values, not objects. For an expression of such a primitive value type, such as `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 a field or variable reference, other than "`this`," is the singleton set containing its static type.

The pseudo-variable "`this`"[6] is assigned by dynamic dispatch, and thus may refer to 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, `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 `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 $\mathit{SubtypesOf}(S)$, which we specify as distinct from all other types. The important point to note for the discussion below is that the viewpoint set of the expression `this` is thus a set with at least 2 elements, the static type of `this`, say S, and $\mathit{SubtypesOf}(S)$.

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 $E\colon S :: r$, $S \leq_{\mathrm{W}} T$, 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 `this` is a set with more than one element, the rule prevents one from assigning `this` to a variable or field.

To illustrate this rule, consider the Java code in Fig. 15. On line 2, the expression `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).

[6] The name `this` is used in Java, and is `*this` in C++, and `self` in Smalltalk. In languages, like Smalltalk, where `super` can be be used as a synonym for `self` in some contexts, the remarks we make about `this` also apply to `super`.

```
PairFI p1, p2, p3;                  // 1
Triple t = new Triple(10, 20, 30);  // 2
p1 = t;                             // 3
p2 = new Triple(10, 20, 30);        // 4
p3 = p2;                            // 5
```

Fig. 15. Example of aliasing. Line 3 is illegal as explained in the text.

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_i to a formal of static type T_i, the viewpoint set of E_i must be a subset of $\{T\}$. Again, since `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 `may alias` clause, which declares an upper bound on the viewpoint set of the method's result.[7] 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 "`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 `this` as a result is also prohibited, since its 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 viewpoint set of the method it overrides. This allows the type checker to conservatively approximate the viewpoint set of a 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 `may alias` clause of an overriding method to be a subtype of a type mentioned in the `may alias` clause of the supertype. This restriction is necessary, because the purpose of the viewpoint set is to prevent cross-type aliasing, 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 thrown in exceptions. However, to simplify the type system, instead of adding `may alias` declarations for

[7] In a language like C++, function declarations also need a `may-alias` clause. Constructors also need such a clause, since they may cause aliases by assigning `this` to variables or fields.

exception results, the type system just requires that the viewpoint sets for all exception results be 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 $(S)E\colon S :: r$ if for some $S \leq_{\mathrm{W}} T$, $E\colon T :: r$. 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 Fig. 15 by changing it to the following.

```
p1 = 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 `:` $E_2)\colon T :: r$ if for some b, r_1, and r_2, E_0:`boolean`$\colon b$, $E_1\colon T :: r_1$, $E_2\colon T :: r_2$, 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 linked data structures and `this` are permitted as arguments. One approach might be to use restrictions that are more semantic, such as those proposed by Leino and Stata [28]. 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. Other work, such as that by Müller and Poetzsch-Heffter [41, 38, 37], Noble, *et al.* [40], and Vitek and Bokowski [49] aims 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 more flexible rules that are still sufficient for weak behavioral subtyping.

Another avenue for future work on this type system is ways of combining it with strong behavioral subtyping, which does not require aliasing restrictions. Such a change would require the language to syntactically distinguish between strong and weak subtypes. This may be another avenue to flexibility in practice.

6 Related Work

Liskov and Wing [31] were the first to point out the key problem of aliasing (and also concurrency) for modular reasoning in the presence of subtyping. They noted that aliasing allows the additional methods of the subtype to cause observable state changes in a supertype object. Their notion of strong behavioral subtyping is discussed throughout the present paper. Strong behavioral subtyping is more restrictive than weak behavioral subtyping, but offers sound modular reasoning about safety properties with unrestricted aliasing (our reasoning choice 1). However, since every strong behavioral subtype is also a weak behavioral subtype, if one can prove that an object satisfies the viewpoint restriction, then one can reason about it using its viewpoint's method specifications, thanks to the "methods rule" of strong behavioral subtyping. (For a more comprehensive discussion of other work on behavioral subtyping, see [21].)

Lewerentz and his colleagues [29] use refinement calculus to define simulations on programs that are observations on types. They do not consider aliasing or interference. Mikhajlova and her coauthors [36] 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 [44].

Abadi and Leino [1] extend the work of Cardelli's [7] 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 and does not provide any help to make reasoning about cross-type aliasing practical.

Recently, Huisman [16] and von Oheimb [50] have given sound (and in the case of von Oheimb, relatively-complete) verification logics for Java. However, these do not allow one to verify code in a way that is practical with respect to patterns of potential cross-type aliases, as we do. Furthermore, Oheimb's work does concern itself with modularity or specification-only variables such as JML's model fields.

The work of Müller and Poetzsch-Heffter [37, 38, 41] 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) [39] and invariants. They control aliasing through a "universe type system." They use supertype abstraction in reasoning about code using method specifications, 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 allows 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 help one reason about potential aliases among different types.

As noted earlier, weak behavioral subtyping is related to flexible alias protection's *arg* mode types [9]. The type *arg* T is like T but does not contain any methods that access the mutable state of T. Thus, T is a weak behavioral subtype of *arg* T. However, weak behavioral subtyping is a more flexible relation

than the relation between T and *arg* T, since it is defined in terms behavioral specifications; in particular, there can be several proper weak behavioral subtypes of *arg* T that are proper supertypes of T, these need not prohibit access to all of the mutable state of T.

The restrictions our type system imposes on the `this` reference are similar to the restrictions imposed on "anonymous methods" in Vitek and Bokowski's work on confining types within Java packages [49]. Both type systems do not allow a method to store the current instance, `this`, in a field, to pass it as an argument to another method, or to return it from a method. However, our type system is more restrictive because we impose these requirements on all methods, whereas the Vitek and Bokowski type system only imposes these restrictions on methods that are declared to be anonymous. On the other hand, anonymous methods are also prohibited from using object identity comparisons on `this`, which is not necessary in our work.

7 Conclusions

The main contributions of this paper are its discussion of issues related to mutation, aliasing, subtyping, and modular reasoning. The paper justifies a series of choices on these issues that lead to a more flexible notion of behavioral subtyping—weak behavioral subtyping. In contrast to strong behavioral subtyping [31], weak behavioral subtyping permits more subtype relations; for example, it permits types with mutable objects to be subtypes of types with immutable objects. One new aspect of our definition of weak behavioral subtyping in this paper is that we have specialized the definition for single dispatch languages (such as Java). We have also explained how weak behavioral subtyping interacts with formal specifications in the context of JML.

For soundness, weak behavioral subtyping requires that cross-type or multiple viewpoint aliasing be prohibited. We have demonstrated one way to enforce this viewpoint restriction statically; the details of this type system are new with this work.

Another important contribution of this paper is a clarification of why reasoning with method specifications needs the viewpoint restriction to be practical. The idea is that the viewpoint restriction not only limits the number of aliasing patterns that one must consider, but that it also eliminates the need to consider possible aliases between model fields of different types. This simplifies reasoning and makes it practical to use method specifications in reasoning. These advantages apply even when one reasons informally; moreover, when one reasons informally it is especially important to have a technique that does not require the consideration of many cases that are easy to forget.

Finally, we observed that, if one has the viewpoint restriction to enable reasoning with method specifications, then one can use weak behavioral subtyping and gain the benefits of its weaker restrictions on what types can be behavioral subtypes.

Future work includes implementing the viewpoint restriction in some practical language, so that we can experiment to see how difficult it is to write code with it. We expect that more sophisticated type systems, for example using alias burying [4] flexible alias protection [9], confined types [49], or universes [37, 38, 41] will be helpful in making the type system that enforces the viewpoint restriction more practical.

Acknowledgments

The work of Leavens was supported in part by a grant from Electronics and Telecommunications Research Institute (ETRI) of South Korea, and by grants CCR-0097907 and CCR-0113181 from the US National Science Foundation.

This work was done, in part, while Leavens was a visiting professor at the University of Iowa. Thanks to Art Fleck, Ken Slonneger, and Cesare Tinelli at the University of Iowa, and to Medhat Assaad, John Boyland, Yoonsik Cheon, Curtis Clifton, Peter Müller, Clyde Ruby, and the OOPSLA 2001 referees for comments on earlier drafts of this paper.

References

1. M. Abadi and R. Leino. A logic of object-oriented programs. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 682–696. Springer-Verlag, New York, NY, 1997.
2. P. America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.
3. K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.
4. J. Boyland. Alias burying: Unique variables without destructive reads. *Software—Practice and Experience*, 31(6):533–553, May 2001.
5. M. Broy and S. Jähnichen, editors. *KORSO: Methods, Languages and Tools for the Construction of Correct Software*, volume 1009 of *Lecture Notes in Computer Science*, New York, NY, 1995. Springer-Verlag.
6. K. B. Bruce and P. Wegner. An algebraic model of subtype and inheritance. In F. Bançilhon and P. Buneman, editors, *Advances in Database Programming Languages*, pages 75–96. Addison-Wesley, Reading, Mass., Aug. 1990.
7. L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, New York, NY, 1991.
8. Y. Chen and B. H. C. Cheng. A semantic foundation for specification matching. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 91–109. Cambridge University Press, New York, NY, 2000.

9. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, Oct. 1998.
10. K. K. Dhara. Behavioral subtyping in object-oriented languages. Technical Report TR97-09, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames IA 50011-1040, May 1997. The author's Ph.D. dissertation.
11. K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. http://www.elsevier.nl/locate/entcs/volume1.html.
12. K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.
14. J. V. Guttag, J. J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.
15. C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.
16. M. Huisman. *Reasoning about Java Programs in higher order logic with PVS and Isabelle*. Ipa dissertation series, 2001-03, University of Nijmegen, Holland, Feb. 2001.
17. D. H. H. Ingalls. A simple technique for handling multiple polymorphism. *ACM SIGPLAN Notices*, 21(11):347–349, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.
18. K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, NY, 1994.
19. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.
20. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, Aug. 2001. See `www.cs.iastate.edu/~leavens/JML.html`.
21. G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.
22. G. T. Leavens and D. Pigozzi. A complete algebraic characterization of behavioral subtyping. *Acta Informatica*, 36:617–663, 2000.
23. G. T. Leavens and W. E. Weihl. Reasoning about object-oriented programs that use subtypes (extended abstract). In N. Meyrowitz, editor, *OOPSLA ECOOP '90 Proceedings*, volume 25(10) of *ACM SIGPLAN Notices*, pages 212–223. ACM, Oct. 1990.
24. G. T. Leavens and W. E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, Nov. 1995.

25. K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.
26. K. R. M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, Oct. 1998.
27. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Technical Report 160, Compaq Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, 2000.
28. K. R. M. Leino and R. Stata. Virginity: A contribution to the specification of object-oriented software. *Information Processing Letters*, 70(2):99–105, Apr 1999.
29. C. Lewerentz, T. Lindner, A. Rüping, and E. Sekerinski. On object-oriented design and verification. In Broy and Jähnichen [5], pages 92–111.
30. B. Liskov and J. Guttag. *Program Development in Java*. The MIT Press, Cambridge, Mass., 2001.
31. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.
32. B. Liskov and J. M. Wing. A new definition of the subtype relation. In O. M. Nierstrasz, editor, *ECOOP '93 — Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany*, volume 707 of *Lecture Notes in Computer Science*, pages 118–141. Springer-Verlag, New York, NY, July 1993.
33. B. Liskov and J. M. Wing. Specifications and their use in defining subtypes. *ACM SIGPLAN Notices*, 28(10):16–28, Oct. 1993. *OOPSLA '93 Proceedings*, Andreas Paepcke (editor).
34. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, 1988.
35. B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.
36. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Stengthened Foundations of Formal Metohds*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101, NY, 1997. Springer-Verlag.
37. P. Müller. *Modular Specification and Verification of Object-Oriented programs*. PhD thesis, FernUniversität Hagen, Germany, Mar. 2001.
38. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 7, pages 137–159. Cambridge University Press, 2000.
39. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in jml. Technical Report 01-03, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, Apr. 2001. To appear in the Formal Techniques for Java Programs 2001 workshop at ECOOP 2001. Also available from archives.cs.iastate.edu.
40. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In E. Jul, editor, *ECOOP '98 – Object-Oriented Programming, 12th European Conference, Brussels, Belgium*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer-Verlag, July 1998.
41. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symosium un Programming (ESOP '99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

42. A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03c, Iowa State University, Department of Computer Science, Aug. 2001.
43. C. Ruby and G. T. Leavens. Safely creating correct subclasses without seeing superclass code. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10) of *ACM SIGPLAN Notices*, pages 208–228, Oct. 2000.
44. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. *ACM SIGPLAN Notices*, 21(11):38–45, Nov. 1986. OOPSLA '86 Conference Proceedings, Norman Meyrowitz (editor), September 1986, Portland, Oregon.
45. S. Stepney, R. Barden, and D. Cooper, editors. *Object Orientation in Z*. Workshops in Computing. Springer-Verlag, Cambridge CB2 1LQ, UK, 1992.
46. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Co., Reading, Mass., 1986. Corrected reprinting, 1987.
47. M. Utting. *An Object-Oriented Refinement Calculus with Modular Reasoning*. PhD thesis, University of New South Wales, Kensington, Australia, 1992. Draft of February 1992 obtained from the Author.
48. M. Utting and K. Robinson. Modular reasoning in an object-oriented refinement calculus. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction, Second International Conference, Oxford, U.K., June/July*, volume 669 of *Lecture Notes in Computer Science*, pages 344–367. Springer-Verlag, New York, NY, 1992.
49. J. Vitek and B. Bokowski. Confined types in Java. *Software — Practice and Experience*, 31(6):507–532, 2001.
50. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.
51. A. Wills. Capsules and types in Fresco: Program validation in Smalltalk. In P. America, editor, *ECOOP '91: European Conference on Object Oriented Programming*, volume 512 of *Lecture Notes in Computer Science*, pages 59–76. Springer-Verlag, New York, NY, 1991.
52. A. Wills. Specification in Fresco. In Stepney et al. [45], chapter 11, pages 127–135.
53. A. Wills. Refinement in Fresco. In Lano and Houghton [18], chapter 9, pages 184–201.
54. J. M. Wing. Writing Larch interface language specifications. *ACM Trans. Prog. Lang. Syst.*, 9(1):1–24, Jan. 1987.

A Formal Definition of Weak Behavioral Subtyping

To formally define weak behavioral subtyping, we use the following notation. We use $\leq_{:W}$ to refer to a weak behavioral subtype relation, which is a binary relation on types. Type symbols are represented by S and T, where S is by convention the subtype, and type vectors by $\vec{U}$ and $\vec{V}$. An invariant of a type T is denoted 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 `this` and additional parameters $\vec{x}$. The notation $assignable_T^m$ denotes the set of locations specified as assignable by method m of type T; this includes dependees of locations explicitly specified as assignable [25, 26]. Similarly, $normpost_T^m$ denotes the the normal postcondition of method m in type T; this is the postcondition that applies when no exceptions can be thrown. Finally, the notation $expost_T^m(\texttt{this}, \vec{x})$ means the postcondition of method m in T that applies when exceptions can be thrown; such a postcondition can refer to the exception result object but the exact notation is unimportant. In relating pre- and postconditions, we use $\texttt{\textbackslash old}(e)$ for the value of e in the pre-state. Substituting z for y in predicate $p(y)$ is written as $p(z)$.

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]. However, that definition does not treat exceptional postconditions separately, and does not treat the assignable clause. The definition below can also be extended to multiple dispatching languages [10]. This definition uses ideas from [2, 31, 12, 10].

Definition 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 (non-receiver) 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_S 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 in the theory of the S's specification.*[8]

- *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.*

$$pre_T^m(\texttt{this}, \vec{y}) \Rightarrow pre_S^m(\texttt{this}, \vec{y})$$

  - *Frame axiom rule.*

$$assignable_S^m \subseteq assignable_T^m$$

  - *Normal postcondition rule.*

$$\begin{aligned}(\texttt{\backslash old}(pre_S^m(\texttt{this}, \vec{y})) \Rightarrow normpost_S^m(\texttt{this}, \vec{y}))\\ \Rightarrow (\texttt{\backslash old}(pre_T^m(\texttt{this}, \vec{y})) \Rightarrow normpost_T^m(\texttt{this}, \vec{y}))\end{aligned}$$

  - *Exceptional postcondition rule.*

$$\begin{aligned}(\texttt{\backslash old}(pre_S^m(\texttt{this}, \vec{y})) \Rightarrow expost_S^m(\texttt{this}, \vec{y}))\\ \Rightarrow (\texttt{\backslash old}(pre_T^m(\texttt{this}), \vec{y}) \Rightarrow expost_T^m(\texttt{this}, \vec{y}))\end{aligned}$$

The postcondition rules given above [12] are less restrictive than those used by Liskov and Wing [31]. A condition that is logically equivalent to our normal postcondition rule (see appendix B for a proof) is the following [50], which we display in an unusual manner that illustrates how the condition can be used for reasoning at the level of the supertype's specification:

$$\begin{array}{ccc}\texttt{\backslash old}(\mathrm{pre}_T^m(\texttt{this}, \vec{y})) & & \mathrm{normpost}_T^m(\texttt{this}, \vec{y})\ ) \\ \Downarrow & & \Uparrow \\ (\ (\texttt{\backslash old}(pre_S^m(\texttt{this}, \vec{y})) & \Rightarrow & \mathrm{normpost}_S^m(\texttt{this}, \vec{y}))\end{array} \tag{1}$$

Chen and Cheng proved [8] that requiring both the precondition rule and the normal postcondition rule above is equivalent to requiring both the precondition rule and the following (also found in [30]):

$$\begin{aligned}&(\texttt{\backslash old}(pre_T^m(\texttt{this}, \vec{y})) \wedge \mathrm{normpost}_S^m(\texttt{this}, \vec{y}))\\ &\Rightarrow \mathrm{normpost}_T^m(\texttt{this}, \vec{y}).\end{aligned} \tag{2}$$

Chen and Cheng also proved that these equivalent conditions are the least restrictive sound conditions for reuse of methods.

[8] In the theory of a type's specification, one is allowed to assume the type's invariant for any object of the type in any visible state. Thus, for example, when proving the constraint rule, since $\texttt{this} : S$, one can assume $I_S(\texttt{this})$, which effectively means that it suffices to prove $(I_S(\texttt{this}) \wedge \texttt{\backslash old}(I_S(\texttt{this})) \wedge C_S(\texttt{this})) \Rightarrow C_T(\texttt{this})$.

B Equivalent Rules for Postconditions

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

Lemma 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.

$$\begin{array}{ll} & (S_{pre} \Rightarrow S_{post}) \Rightarrow (T_{pre} \Rightarrow T_{post}) \\ = & \quad \langle \text{by } P \Rightarrow (Q \Rightarrow R) \equiv (P \wedge Q) \Rightarrow R \rangle \\ & ((S_{pre} \Rightarrow S_{post}) \wedge T_{pre}) \Rightarrow T_{post} \\ = & \quad \langle \text{by symmetry of conjunction} \rangle \\ & (T_{pre} \wedge (S_{pre} \Rightarrow S_{post})) \Rightarrow T_{post} \\ = & \quad \langle \text{by } (P \wedge Q) \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R) \rangle \\ & T_{pre} \Rightarrow ((S_{pre} \Rightarrow S_{post}) \Rightarrow T_{post}) \end{array}$$