Behavioral Subtyping and Behavioral Enrichment of Multimethods

Neelam Soundarajan and Stephen Fridella
Computer and Information Science
The Ohio State University
Columbus, OH 43210
e-mail: {neelam,fridella}@cis.ohio-state.edu

Abstract

Multimethods have attracted a lot of attention in the literature over the last several years. Much of the focus has been on such questions as, how to choose the 'correct method' to dispatch to if no single method is most specialized for the given set of arguments; or, alternately, how to impose sufficient conditions that will prevent the problem from occurring in the first place.

The focus of this paper is different. We are interested in extending the notion of behavioral subtyping to the context of multimethods. We show that behavioral subtyping can indeed be naturally extended to deal with multiple dispatch; as an immediate bonus, we show that the problem of ambiguous methods disappears. We also investigate how, when reasoning about code that invokes multimethods, we can appeal to the enriched behavior provided by the specialized methods.

1 Introduction

Multimethods, that is methods for which dispatching is done on the basis of the dynamic class of several (or all) of the method's arguments rather than just that of the distinguished first argument as is the case with single-dispatch methods, promise to be a powerful mechanism in OO design. Not surprisingly, multimethods have attracted a lot of attention in the literature [1, 2, 3, 4, 10] over the last several years. Much of this work has focussed on typing issues or on such questions as how to choose the 'correct method' to dispatch to if no single method is most specialized for the given set of arguments. *Behavioral* questions, i.e., how to reason about the behavior of an OO program that exploits multimethods, seem to have received relatively little attention. Our goal in this paper is to address some of the behavioral questions that arise with multimethods and to develop techniques that may be used to reason about the behaviors of OO programs that utilize these methods.

One of the fundamental ideas in dealing with the behavior of OO programs is the notion of *behavioral subtyping* [6, 7]. In effect, a derived class[1] D is a behavioral subtype of its base class B if the behaviors that the operations of D exhibit are *consistent* with their specifications in B. More precisely, the requirement is expressed as appropriate relations between the pre- and post-conditions in the base class and the derived class of the respective operations, but the net result is that any reasoning we may have performed, using the base class specifications, concerning the behavior of a piece of client code C continues to be correct if we replace some (or all) instances of B that this code manipulates by instances of D. The key role that behavioral subtyping plays in building reliable OO systems has been well documented in the literature, see for example [8]; but all of this work assumes that method dispatch is on the basis of the distinguished first argument, that is, only single-dispatch methods are considered.

The first question we consider in this paper is that of defining a notion of behavioral subtyping that would be appropriate for multimethods. One problem we immediately encounter is that multimethods, unlike single-dispatch methods, are not members of any par-

[1] For simplicity, and following the practice of many standard OO languages such as *Eiffel* and *Java*, we will not make a distinction between 'class' and 'type', as well as between 'inheritance' and 'subtyping', in this paper. There are certainly OO languages for which such distinctions would be essential, but the issues involved do not have anything directly to do with multimethods which is the focus of this paper; further, it would be straightforward to recast our work in a setting in which such distinctions are indeed made.

ticular class, so it is not clear on which methods we should impose the behavioral requirements corresponding to behavioral subtyping. But this is really a syntactic problem, not a behavioral one; a simple parsing tool could easily identify the related methods. Indeed, any implementation of multimethods must include such tools since this information is essential when dispatching the methods. Ignoring these syntactic issues, it turns out, as we will see in the next section, that there is indeed a reasonable and natural way to define behavioral subtyping for multimethods; and the standard notion of behavioral subtyping for single-dispatch methods turns out to be a special case of our definition. As an added bonus, given our definition of behavioral subtyping for multimethods, the problem of 'ambiguous methods', that is the situation where no single method is most specialized for the given set of arguments, essentially disappears. That is, in this situation, *any* method may be chosen without causing (behavioral) difficulties.

But behavioral subtyping by itself is not enough to reason completely about the behavior of OO programs that use multimethods. Indeed, even for the single-dispatch case, behavioral subtyping is inadequate since it only takes account of the behavior provided by the base class methods, and ensures that the behavior provided by the derived class methods are consistent with those of the base class. The power of polymorphic methods, whether single or multiple-dispatch, arises because the OO designer is able to redefine these methods to provide behaviors tailored to specific derived classes corresponding to the objects to which the methods are applied, rather than using the common behavior provided by the base definitions. A piece of client code which may have been designed when only the base classes had been defined, can nevertheless be applied to objects that are instances of the derived classes and the (single or multiple) dispatch mechanism ensures that the newly redefined methods, appropriate to the specific derived classes corresponding to these objects, are applied. The reasoning problem then becomes, how do we account for the *richer behavior* that these redefined methods exhibit when understanding the behavior of the client code? Behavioral subtyping assures us that any reasoning we have performed about the client code, based on the specification of the base methods, continues to be valid, but it does not provide us any means to arrive at this richer behavior.

One possible solution would be to simply reanalyze the client code and during this reanalysis, appeal to the the richer behavior provided by the new definitions of the methods corresponding to the appropriate derived classes. But this would be completely opposed to the spirit of reuse that OO is meant to encourage. What we need instead is a way to enrich our analysis/specification of the client code so that when new subclasses are introduced and corresponding new definitions that exhibit appropriate richer behaviors are provided for the (multi)methods, we can 'plug-in' these richer behaviors into the specification of the client code, to arrive at a specification of the richer behavior that the client code will exhibit when applied to objects that are instances of these subclasses. In Section 3 we will see how to enrich our specification of the client code in this manner. The key is to include in this specification, information about the sequence of calls that this code makes to methods that might be enriched; specifically, information must be included about the identity of the methods invoked, the objects that are passed as arguments to these methods, and what is done with the results that the methods return. All of this information, as we will see, can be captured in the specification in terms of a *trace* (or sequence) of method calls. As we will see, our *Behavior Enrichment Rule* will allow us to plug-into such a specification, information about the richer behavior of the new definitions of the multimethods, to arrive at the richer behavior of the client code.

We will make some simplifying assumptions in our work; specifically, we will assume that there are no aliasing issues; thus if X and Y are two program variables, they may not refer to the same object. We will also assume that no object may internally contain a reference to another object; this assumption is, in fact, related to the first because it just ensures that there are no 'behind the scenes' ways of accessing a given object, that is, of creating aliasing. Although in practice real systems often do not satisfy these conditions, the problems that aliasing introduces are not directly related to behavioral issues concerning multimethods which is our focus.

The main contributions of this paper may be summarized as follows:

- Presents the case for developing reasoning techniques for dealing with the behavior of OO systems that exploit multimethods.

- Develops such reasoning techniques for dealing with both behavioral subtyping as well as the richer behavior that the code utilizing the multimethods will exhibit if applied to objects that are instances of subclasses instead of the base classes.

- Applies the reasoning technique to a simple example to show how this richer behavior may be arrived at without reanalysis of the code.

- Shows that the problem of 'ambiguous methods' can be ignored if the methods satisfy the requirements of behavioral subtyping as defined.

The rest of the paper is organized as follows: In the next section we define the notion of behavioral subtyping for multimethods. In Section 3 we present our trace based reasoning technique to specify the behavior of code that invokes multimethods and show how, from this specification, the richer behavior of this code when applied to derived class objects may be derived. In Section 4 we apply the reasoning technique of Section 3 to a simple example; of course, the multimethod in this example satisfies the behavioral subtyping requirement of Section 2. In Section 5 we consider the by-now classic multimethod example: the *equal* function defined on *points* and *color points*. Our analysis shows that this apparently simple example exhibits a rather surprising twist when we consider behavioral issues. In the final section we summarize our work, contrast it with other work on multimethods, and identify some problems for future research.

2 Behavioral Subtyping

Most standard OO languages do not include multiple dispatch, hence do not provide any syntax to express multimethods. Several language notations have been designed to code multimethods; in our discussion, we will use the *tuple class* notation introduced by Leavens and Millstein [5] as a convenient way to express multimethods; but note that our approach will be applicable to the other notations as well. If m() is a multimethod to be dispatched on, say, two arguments of (base) types B1 and B2, and has a third argument (on which no dispatching is done) of type C3, this would be written as:

```
tuple class( x:B1, y:B2 ) {
    method m( z:C3 ) { ... }
}
```

In order to keep the discussion simple, we will assume that a tuple class will define exactly one method. This is not a serious restriction since we can have more than one tuple class corresponding to a given set of classes B1, ..., Cn. In our discussion, we will refer to the above tuple class as ⟨m, (B1, B2)⟩; occasionally when the method in question is clear, we will abbreviate this to (B1, B2).

If D1 and D2 are derived classes (or subclasses – we use the terms interchangeably) of B1 and B2 respectively, and we wish to define a specialized form of m() for D1, D2, we would do that as follows:

```
tuple class( x:D1, y:D2 ) {
    method m( z: C3 ) { ... }
}
```

The fact that this definition of m() is a specialization of the previous one can be deduced by the system (and the designer) given that D1, D2 are subclasses of B1, B2. A call to m(), may be written as:

(X, Y).m(Z)

Let us assume that the static type of X is B1, of Y is B2 and of Z is C3. If the objects that X, Y refer to are of type D1, D2, the specialized definition of m() provided in the tuple class (D1, D2) would be dispatched to. If the object that X refers to is of type B1 or if the object that Y refers to is of type B2 (or both are true), the original definition of m() provided in the tuple class (B1, B2) would be dispatched to. The actual type of the object that Z refers to, does not enter the picture since no dispatching is done on that argument.

Note that a tuple class ⟨m, (B1, B2)⟩ is quite different from a standard class. Perhaps the most important difference is that it is not possible to declare variables to be of a tuple class type. So the term 'tuple *class*' is perhaps a bit confusing; nevertheless, the notation for method call, (X,Y).m(Z) is very intuitive; it makes it clear that the dispatching is done on (and only on) the types of objects that X, Y refer to. Hence we will use this notation in our examples and in our discussion of multimethods. Further, given two tuple classes, ⟨m, (B1, ..., Cn)⟩ and ⟨m, (D1, ..., Dn)⟩ that both define the multimethod m(), we will say that the latter is a *tuple subclass* of the former if for each i=1,...,n, it is true that Di is a subclass of (or is the same as) the class Ci.

Let us return to our two tuple classes defined earlier. If we wanted to, we could also have defined tuple classes corresponding to either, or both, of ⟨m, (B1, D2)⟩ and ⟨m, (D1, B2)⟩, with each of these providing its own definition of m(). If both of these were defined, and the tuple class corresponding to ⟨m, (D1, D2)⟩ was *not* defined, then the call (X,Y).m(Z) would be considered *ambiguous* if X and Y referred to objects of type D1, D2 respectively. This is because both the definition in the tuple class ⟨m, (D1, B2)⟩ as well as the one in ⟨m, (B1, D2)⟩ is applicable in this case, but neither is 'more specific' than the other; a multimethod definition in a given tuple class is more specific than its definition in another tuple class if the former tuple class is a tuple subclass of the latter. In general, a multimethod call will be dispatched to the *most* specific definition of the method that is applicable given the dynamic types of the objects involved in the call. In our example since neither of (B1,D2), (D1,B2), is a tuple subclass of the other, and since both definitions are applicable given that the dynamic types of the arguments are D1, D2, we do not have a unique most-specific method to dispatch to. In most systems, such a situation is considered erroneous and will result in a message along the lines of "*message-ambiguous error*".

Let us now turn to behavioral questions. For specifying the behavior of a (definition of a) method, we will use the standard pre- and post-condition approach. When specifying the behavior of class methods, invariants are also often very useful, but in the case of multimethods, invariants pose a problem. Consider the following: Suppose m() and m'() are two binary methods; suppose the variables X, Y refer to two objects A and B, and we have the following:

(X,Y).m(); ...; (X,Y).m'();

It would be convenient when reasoning about the effect of each call to be able to assume, immediately prior to the call, that the arguments satisfy some invariant relation. Normally we would expect that such an assumption could indeed be made provided we check that these two methods, as well as any other binary methods that receive A and B as arguments, ensure that the relation holds when they finish, if it was satisfied when they started. But there is a problem; the object A (or B) might be used in isolation for other purposes –for example a method of the class that A is an instance of may be applied to it– between the calls to m() and m'(). In that case, even if the invariant relation that we have in mind involving A and B was satisfied when the call to m() finishes, it will not necessarily be satisfied when m'() is called. The point is that A and B are not in any sense *encapsulated* into a single unit; rather, either of them may, at any time, be used by itself (or possibly paired with some other object, to be passed to yet other multi-methods), so relations like invariants involving these objects are not generally available for use in tuple classes, and we will not appeal to such invariants in our formalism. Note however that invariants on *single objects* are entirely appropriate. Thus in the discussion above, if the operations of the class that A is an instance of behave in such a way that when any of these operations finishes it ensures that the state of the object satisfies some condition I (assuming that the object satisfied the same condition at the start of the operation), then the fact that we might pass A as an argument to the multimethods m() or m'() makes no difference. The multimethods do not have any privileged access to the internals of A; like the rest of the code, they too have to invoke the operations provided by the class that A is an instance of, in order to manipulate it any way. Thus invariants for individual classes are available as usual.

Our pre- and post-conditions will be written in terms of the *concrete* states of the objects, rather than their abstract states. This is not because abstraction is not important when dealing with multimethods but rather to keep the presentation simple. Indeed, multimethods do not introduce any additional complications as far as abstraction is concerned, but taking an abstract view would introduce an extra step in the formalism, so we prefer to take a concrete view.

Suppose we have a tuple class ⟨m, (B1, B2)⟩. A specification of this tuple class will then consist of a pre-condition, $\mathsf{m}_{(b1,b2)}.pre$, and a post-condition, $\mathsf{m}_{(b1,b2)}.post$. Both of these will be assertions over the member variables of both B1 and B2, as well as any other parameters of m(); since no dispatching is done on these additional parameters, we will assume that these are 'simple' types of parameters (such as integers etc.), and that these parameters are passed by value-result. Our post-condition may also refer to the values at the time of the call; so in the post-condition we will use $\#x$ to refer to the value of x at that time whereas x will refer to the value when the method finishes. As usual, this specification asserts that if the values, at the time of the call to m(), of the member variables of the objects on which m() is applied and the other parameter values passed to m(), satisfy $\mathsf{m}_{(b1,b2)}.pre$, then the values of the member variables of the objects and the parameter values at the time that the operation finishes execution are guaranteed to satisfy the assertion $\mathsf{m}_{(b1,b2)}.post$. Suppose ⟨m, (D1, D2)⟩ is a tuple subclass of ⟨m, (B1, B2)⟩. The tuple class ⟨m, (D1, D2)⟩ will have its own specification consisting of the assertions $\mathsf{m}_{(d1,d2)}.pre$ and $\mathsf{m}_{(d1,d2)}.post$ as the pre- and post-conditions respectively.

Consider now a call to m() of the form (X, Y).m(Z) in a piece of client code C, in which X has been declared to be of type B1, Y of type B2, and Z of type C3. Then, in reasoning about the behavior of C, we would appeal to the specification $\mathsf{m}_{(b1,b2)}.pre$, $\mathsf{m}_{(b1,b2)}.post$; suppose, on the basis of this reasoning, that we have established some property $\mathcal{P}$ for C. Suppose now that the actual objects that X, Y refer to are of type D1, D2; then the above call would be dispatched to the definition of m() provided in the tuple class (D1, D2). How can we be sure that $\mathcal{P}$ will still hold in this situation? The answer will be provided by behavioral subtyping which we will define next and impose as a requirement on tuple base classes and corresponding derived classes (such as ⟨m, (B1, B2)⟩ and ⟨m, (D1, D2)⟩ respectively).

In defining behavioral subtyping, we will not only have to refer to specifications of the base and derived tuple class, but also the class invariants of the various classes; we will denote the invariants by I_{b1}, I_{b2} for B1, B2 and I_{d1}, I_{d2} for D1, D2 respectively.

Definition: Suppose that ⟨m, (D1, D2)⟩ is a tuple subclass of ⟨m, (B1, B2)⟩, and that D1, D2 are behavioral subtypes of B1, B2 respectively. Then ⟨m, (D1, D2)⟩ is a *behavioral subtype* ⟨m, (B1, B2)⟩ if the following conditions are satisfied[2]:

$$\begin{aligned} &\mathsf{m}_{(b1,b2)}.pre \wedge I_{d1} \wedge I_{d2} \Rightarrow \mathsf{m}_{(d1,d2)}.pre \\ &\mathsf{m}_{(d1,d2)}.post \Rightarrow \mathsf{m}_{(b1,b2)}.post \end{aligned} \qquad (1)$$

The first implication requires that the only additional requirements that the pre-condition of the multimethod defined in the tuple subclass may impose, compared to the pre-condition of the same method in the tuple base class, correspond to requiring that the objects passed to the method satisfy the invariants of the corresponding derived classes (I_{d1}, I_{d2} respectively). Similarly, the second implication requires that the multimethod defined in the tuple subclass must be such that it necessarily satisfies the post-condition corresponding to the tuple base class. The above definition corresponds to *binary* methods, but the generalization to dispatching on any number of arguments is straightforward. Note also for the case when the dispatch is on a single argument, that this definition reduces, as may be expected, to the standard definition of behavioral subtyping. Given the requirements of (1), it is straightforward to see that the code C considered in the last paragraph will indeed continue to exhibit the property $\mathcal{P}$ that has been established for it by appealing to base tuple class specification even though the actual call will be dispatched to the derived tuple class definition of m().

Before we conclude this section, let us briefly return to the problem of the ambiguous multimethod. We had D1, D2 as subclasses of B1, B2 respectively; we had defined the tuple classes, ⟨m, (B1, B2)⟩, ⟨m, (B1, D2)⟩, and ⟨m, (D1, B2)⟩; and we considered call of the form:

```
(X,Y).m();
```

where the static types of X, Y were B1, B2, and their dynamic types were D1, D2 respectively.

As we noted before, this would typically be flagged as an error, but let us consider it from the point of view of the behavior it exhibits. If, when reasoning about this segment of code, we are unaware of the dynamic types of X, Y, we will only be able to appeal to the specification of m() from the tuple base class ⟨m, (B1, B2)⟩. Suppose we have come to some conclusion about the behavior of this piece of code on the basis of this specification; suppose also that the actual call is dispatched to the m() defined in ⟨m, (B1, D2)⟩. It is easy to see, given the behavioral subtyping requirement that this definition of m() must satisfy with respect to the one defined in ⟨m, (B1, B2)⟩, that our conclusion will still be valid: Since the object that X refers to is of dynamic type D1, it satisfies the invariant I_{d1}; hence, D1 being a behavioral subtype of B1, it must also satisfy I_{b1}. And the object that Y refers to is of dynamic type D2, so it satisfies I_{d2}. Hence, from the first requirement of behavioral subtyping, we can be sure that at the time of the call to the m() defined in ⟨m, (B1, D2)⟩, that the required pre-condition, $\mathsf{m}_{(b1,d2)}.pre$ will be satisfied. Hence, we may assume that the corresponding post-condition, $\mathsf{m}_{(b1,d2)}.post$ will be satisfied when the multimethod finishes execution. Hence, by the second requirement of behavioral subtyping, $\mathsf{m}_{(b1,b2)}.post$ will be satisfied at this point. Our original conclusion must have been arrived at on the basis of the assumption that exactly this assertion will be satisfied at this point, so that conclusion remains valid.

A similar argument applies if the method that is dispatched to is the one defined in ⟨m, (D1, B2)⟩. The same argument can also be made if there is a method defined in ⟨m, (D1, D2)⟩, and it is this method that is dispatched to at this point (again assuming the requirements of behavioral subtyping are satisfied). Thus the problem of ambiguous methods is not really a problem as far as the behavior we have established, on the basis of the base tuple class specification of the method, is concerned. This is, of course, precisely the idea of behavioral subtyping – ensure that the behavior of the definition of the method in the derived class is consistent with that in the base class so that any conclusion drawn on the basis of the specification of the base class definition continues to remain valid.

But as noted earlier, the reason for the new definitions of m() in tuple classes such as ⟨m, (D1, B2)⟩ is to provide for *richer* behavior based on the fact that the first argument will be an instance not just of B1, but of D1. We need to be able to arrive at this richer behavior, not just assert that the original behavior con-

[2]There is a possibility of name clashes between, for example, the member variables of the classes B1 and B2; in practice, it is important to take care of such clashes, but we will ignore such possibilities in our discussion.

tinues to hold. This is the problem we will address in the next section. In the case of ambiguous situations, the formalism of the next section will let us conclude that the richer behavior that the code will exhibit will be a *disjunction* of those corresponding to the various methods that we might dispatch to.

3 Enriching Behavior

Our goal in this section is to see how to specify the behavior of a piece of client code C that includes calls to multimethods, and how to arrive at its enriched behavior once we know the specific subclasses of the objects that C manipulates, by appealing to the enriched behavior of the multimethods defined in appropriate tuple subclasses. A key requirement that our formalism must satisfy is that it does not require us to reanalyze the client code to take account of the behavior of the multimethods as defined in the tuple subclasses. In simple examples this can actually be more trouble than it is worth since reanalysis might be relatively simple. But if the code is complex, or if the source code is not available (because of proprietary or other reasons), a formalism such as the one we develop in this section is essential.

Let σ denote a list of all the objects manipulated by our code C; we assume that the individual objects are numbered 1 through n, so they may be referred to as $\sigma(1), \ldots, \sigma(n)$. The key feature of our formalism that will enable us to achieve the requirements stated in the last paragraph is the sequence or *trace*, which we will denote by τ, of all dynamically dispatchable method calls that the code makes. Note that both calls to single-dispatch methods as well as multimethods will be recorded on τ since both types of methods are capable of exhibiting enriched behavior when we consider subclasses. But it is not sufficient to simply record the *names* of the invoked methods. Rather, an element of τ, corresponding to a call to a method m will look as follows:

$$\begin{aligned}(m, (\sigma_{a_1}, \ldots, \sigma_{a_k}), (v_1, \ldots, v_j),\\ (\sigma'_{a_1}, \ldots, \sigma'_{a_k}), (r_1, \ldots, r_j))\end{aligned} \tag{2}$$

The first component m here is the name of the method being called. The second component is the state, immediately prior to the call, of the objects that are passed as arguments in this call to m. The third component specifies the values of any additional arguments to m. The fourth component is the state, immediately after return from the call, of the objects that were passed as arguments to m. The fifth component specifies the return values of (any) additional arguments to m. Suppose, for example, that m is a binary method and in the call that this element represents, the objects $\sigma(2), \sigma(3)$ are passed to m as arguments, that their states at the time of the call are $s2, s3$, and that there is one more argument u, whose value is the (integer) 5 at the time of the call; and that on return from the call, the states of $\sigma(2), \sigma(3)$ are $s2', s3'$, and the value of u is 10. Then the element representing this method call (and return) would look like:

$$(m, (s2, s3), (5), (s2', s3'), (10)) \tag{3}$$

The trace τ is a *sequence* of such elements, each representing a single method call and return.

The specification of C will be as usual in the form of a pre- and post-condition each being an assertion involving the objects manipulated by C (and any other simple values that C may use). In addition, these assertions will also involve τ. The pre-condition will assert that τ is ε, the empty sequence since at its start, C has not called any method.

The post-condition will provide information about the final state of the objects manipulated by C as well as the content of τ when C finishes execution; this may include information about the number of elements in τ i.e., the number of calls that C made to dispatchable methods during execution; relations between the values that appear in the different elements of τ; or, most commonly, relations between the elements of τ and the final states of the objects manipulated by C.

A couple of important points should be noted: First, the post-condition will not, in general, exhaustively list each possible τ and each element thereof. Rather, based on the specific application, the designer must decide how much detail to include in the specification; this is no different than in standard specifications of program behavior but here it is even more important since including unnecessary detail can make the specifications unmanageable. We will return to this point later in the paper. Second, our assertions will be such that when specifying the state of an object, for example as in the trace element in (3) above, we will not specify that a particular object, say the one numbered 2, has a specific state with specific values for each member variable. This is because, for all we know, this object might be enriched in a derived class, and correspondingly the methods will also be enriched. So what our assertions will say is only that the portion of this object's state that corresponds to the particular member variables has particular values. This will allow our specifications of C to be enriched (using the enrichment rule below) as appropriate for these new derived classes.

The initial analysis of C will assume that the objects $\sigma(1), \ldots, \sigma(n)$ to be of their static types, that is the types of the corresponding variables as declared in C. We will assume that these types are $B_1, \ldots, B_n$; so during the initial analysis, the calls that appear in C to the various (single-dispatch and multi)methods will be taken to be to the corresponding definitions of these methods. Once we have additional information about the dynamic types of these objects, which we will denote by $D_1, \ldots, D_n$ (each D_i must of course be a subclass of the corresponding B_i), we will be able to arrive at the enriched behavior of C based on our previous analysis of C, by appealing to the behaviors of the definitions of the methods corresponding to $D_1, \ldots, D_n$. The essence of these two steps is captured in the two rules we present below. The first rule corresponds to the initial analysis of C, in particular ensuring that the method call being considered is appropriately recorded in τ; note that only method calls have an effect on τ; other statements that may appear in C are handled using entirely standard axioms and rules, so we omit them. Our second rule corresponds to the enrichment step.

In order to simplify the presentation of the method call rule, we assume the following: There are three objects, $\sigma(1), \sigma(2), \sigma(3)$ in C. The method being called is a binary method named m; m expects one additional parameter. The objects passed to m as arguments in this call are $\sigma(2), \sigma(3)$; the additional parameter is t. The specification of m from the tuple class ⟨m, (B2, B3)⟩ consists of the pre-condition $m_{(B2,B3)}.pre$ and the post-condition $m_{(B2,B3)}.post$. The formal parameter names of m are x, y, u (x, y being the object parameters and u the additional parameter); hence $m_{(B2,B3)}.pre$ is in terms of x, y, u, and $m_{(B2,B3)}.post$ is in terms of x, y, u as well as their values at the time of the call (these being identified by the # prefix as we noted in the last section).

R1. Method Call Rule

$$\frac{\begin{array}{l} p \Rightarrow m_{(B2,B3)}.pre^{x,\ \ y,\ \ u}_{\sigma(2),\sigma(3),t} \\ \exists(\sigma(2)', \sigma(3)', t'). \\ \quad [p^{\tau,\ \ \ \ \sigma(2),\ \sigma(3)}_{\mathsf{abl}(\tau)\,\sigma(2)',\sigma(3)'} \wedge \\ \quad m_{(B2,B3)}.post^{\#x,\ \#y,\ \#u,x,\ \ y,\ \ u}_{\sigma(2)',\sigma(3)',t',\ \sigma(2),\sigma(3),t} \wedge \\ \quad \mathsf{last}(\tau) = (m, (\sigma(2)', \sigma(3)'), (t'), (\sigma(2), \sigma(3)), (t))\,] \\ \qquad\qquad\qquad \Rightarrow q \end{array}}{\{\,p\,\}\ (\sigma(2), \sigma(3)).m(t)\ \{\,q\,\}} \tag{4}$$

The first antecedent of the rule ensures that at the time of the call, the pre-condition expected by the method is satisfied for the arguments that are passed to it; the super/sub-script notation is used in the standard manner to indicate substitutions. Thus on the right side of the implication, we are replacing all occurrences of x by $\sigma(2)$ etc.

The second antecedent ensures, given the specification of m, that the post-condition q we wish to assert is indeed appropriate. The function abl denotes 'all but last'; when applied to τ it returns a sequence that contains all elements of τ except the last one. Thus the first clause on the left side of the implication allows us to assume that the assertion p is satisfied following the call except that in place of τ we need to use $\mathsf{abl}(\tau)$ since a new element has been added to τ and in place of $\sigma(2), \sigma(3)$, we must use the states of these objects immediately before the call. The second clause on the left side allows us to use the information from the post-condition of m with the appropriate substitutions for the parameters (both at the time of the call and immediately after the return). The last clause gives us information on what the new element added to τ looks like, corresponding to this call and return. We are required to show that the assertion q contains nothing that cannot be deduced from these three clauses. We will not attempt to show the soundness of this rule formally, but note that it is a fairly standard method call rule, modified to account for the recording, in the trace, of the call and return.

What if the the objects $\sigma(2), \sigma(3)$ are in fact of dynamic type $D2, D3$ and the call we just considered is dispatched to the method defined in the tuple class ⟨m, (D2, D3)⟩? Will the post-condition q that we established by appealing to the above rule still be correct? Yes, indeed, and this follows from our behavioral subtyping requirement ((1) from Section 2). The first clause of that definition ensures that the pre-condition of this new m will be satisfied, and the second clause ensures that, on return, the original post-condition $m_{(B2,B3)}.post$ that we used in (applying) the above rule will be satisfied. Our goal of course is to also establish the richer behavior that results from dispatching to the method defined in (D2, D3) so let us turn to the rule that will allow us to do this.

We will first look informally at the key idea underlying the method to enrich the specification of C since although the basic idea is fairly straightforward, the formal details are a bit involved; the complexity in the formal details is primarily notational, not conceptual. Suppose we have obtained a specification of C based on the static types of the objects it manipulates, by using the rule R1 and other standard rules. The post-condition of C will be a relation that will relate the states of the various objects manipulated by C to the elements of the trace, τ, of calls to various methods that have been made in C. Suppose one of these calls is to the method m that we just considered, and that the dynamic types of the objects passed to m are $D2, D3$. The most common and useful type of enrichment is when the derived class introduces new components (i.e., new member variables) to the state of the object.[3] Thus, $\sigma(2)$, for example, has components that we were not aware of when we analyzed C, these being the member variables introduced in $D2$; similarly for $\sigma(3)$. An enriched post-condition of C should give us information about these extra components of $\sigma(2), \sigma(3)$ that the original post-condition of C is silent about. In principle, this is easily done. The only changes to the components of, for example, $\sigma(2)$ must come as a result of method calls in which $\sigma(2)$ is passed as an argument, and information about all the method calls, and which objects are passed as arguments, are recorded in τ. So if we knew the initial state of these additional components of $\sigma(2)$ at the start of C –and in general, this information will be given to us as a richer pre-condition for C– then we can go through the elements of the trace that involve $\sigma(2)$ and 'compute' the net effect on these additional components of $\sigma(2)$, given the enriched specifications of the methods in question. The rule below is essentially a formalization of this idea but before we look at the rule, one point is worth noting: If the original specification of C leaves out critical information about the elements of τ, as might happen if the the designer did not expect certain kinds of enrichment, we would be unable to arrive at a complete enriched specification of C from that specification and the rule below. In that case, we may well have to resort to re-verification. But this is not a problem that is unique to multimethods; single-dispatch methods will have the same problem if the specifier of the original code did not foresee some potential enrichments.

In order to simplify the presentation of the enrichment rule, we will consider the single multimethod $\mathsf{m}()$ (but it may be invoked multiple times in C). Let us assume that, as described before, we have established, on the basis of the static types $B1, \ldots, Bn$ of the objects that it manipulates, the specification $\langle pre, post \rangle$ for C. Suppose we now know that these objects are in fact of dynamic types $D1, \ldots, Dn$, each Di being, of course, a subclass of the corresponding Bi. Rule **R2** can then be applied to determine an enriched specification $\langle d.pre, d.post \rangle$ for C, corresponding to the enriched behavior that will be provided by the $\mathsf{m}()$ defined in the corresponding tuple subclass, and its corresponding enriched specification whose specification we will denote (using a slightly different notation than we used previously) as $\langle m_d.pre, m_d.post \rangle$.

R2. Enrichment Rule

$$\frac{\begin{array}{l} d.pre \Rightarrow pre \\ [\, pre^{\sigma(1), \;\ldots,\; \sigma(n)}_{\#\sigma(1)\downarrow B1, \;\ldots,\; \#\sigma(n)\downarrow Bn} \wedge \\ \;post^{\#\sigma(1), \;\ldots,\; \#\sigma(n), \;\sigma(1), \;\ldots,\; \sigma(n), \;\tau}_{\#\sigma(1)\downarrow B1, \;\ldots,\; \#\sigma(n)\downarrow Bn, \;\sigma(1)\downarrow B1, \;\ldots,\; \sigma(n)\downarrow Bn, \;\tau\downarrow\{B1,\ldots,Bn\}} \\ \;\wedge TBR(\tau, \#\sigma(1), \ldots, \#\sigma(n), \sigma(1), \ldots, \sigma(n)) \,] \\ \qquad\quad \Rightarrow \; d.post \end{array}}{\{d.pre\} \;\; C \;\; \{d.post\}}$$

Let us examine each of the two antecedents of the above rule in turn. The first antecedent states that the enriched pre-condition $d.pre$ must imply the original pre-condition pre. Without this condition, we would have to consider cases when C is invoked in a state that does not meet the original pre-condition, but in such a case we cannot be sure that *any* of our reasoning about C remains valid – including our reasoning about the structure of τ. This is clearly unacceptable. The second antecedent states that the enriched post-condition $d.post$ follows by implication from three assumptions: (1) the original pre-condition for C continues to hold on the non-enriched portions of the initial states of the parameters, (2) the original post-condition continues to hold on the non-enriched portions of the initial and final states of each of the objects of interest to C and the non-enriched elements of τ, and (3) the *Trace Behavior Relation* holds between the trace τ and the initial and final states of each object. What is "non-enriched portion" portion of a state? Consider that the original pre-condition for C will make assertions about its objects assuming that they have the types $B1, \ldots, Bn$. But when these parameters actually are objects of type $D1, \ldots, Dn$, each object will have additional state components which are not mentioned in pre. Thus pre only applies to the state components of each parameter which are defined in the classes $B1, \ldots, Bn$. To refer to state components of an object $\sigma(i)$ which were defined in a certain class B we use the notation $\sigma(i) \downarrow B$. Similarly, the notation $\tau! \downarrow \{B1, \ldots, Bn\}$ applies the appropriate projection to each state recorded in τ.

We are left to specify the *Trace Behavior Relation*. This is a relation that relates the elements of the method call trace of C to the initial and final states (for C) of the objects of C. Note that a particular object may not be part of the arguments to a partic-

[3] Another type of 'enrichment' is for the derived class to implement a method in a 'more deterministic' manner than specified in the base class; while examples of this kind are common in the behavioral subtyping literature, we will focus on the type of enrichment based on enriching the state since in practice this is by far the more useful kind.

ular multimethod. We will therefore define a function $nextuse(\tau, i, \sigma(k))$ which returns the position of the next entry in τ after or including position i, which corresponds to a multimethod invocation on a tuple of which $\sigma(k)$ is a part. $prevuse(\tau, i, \sigma(k))$ is defined similarly, but to return the most recent entry up to and including position i which uses $\sigma(k)$. If an object $\sigma(k)$ is a member of the tuple upon which the multimethod at position i in τ was invoked, then we can write $\sigma(k) \in \tau[i]$. Also, when referring to the element at position i in the trace τ (i.e. $\tau[i]$), $\tau[i].name$ is the name of the multimethod called, $\tau[i].\#\sigma(k)$ is the state of parameter $\sigma(k)$ recorded immediately before the call, $\tau[i].\sigma(k)$ is the state of this object recorded immediately following the call, and $\tau[i].\mathsf{result}$ is the result (for simplicity, we assume that there the method may return at most a single result and that the initial value of this is not important) of the call that is recorded in τ. Finally, $\tau[i].\#\bar{\sigma}$ is the list of initial states of arguments passed to the multimethod at position i of τ, and $\tau[i].\bar{\sigma}$ is the list of final states of those arguments. We can now define the relation as follows:

Trace Behavior Relation

$$TBR(\tau, \#\sigma(1), ..., \#\sigma(n), \sigma(1), ..., \sigma(n)) \equiv$$

$$(\#\sigma(1) {\downarrow} D1 \;=\; \tau[nextuse(\tau, 0, \sigma(1))].\#\sigma(1) {\downarrow} D1) \wedge$$

$$(\sigma(1) {\downarrow} D1 \;=\; \tau[prevuse(\tau, |\tau|, \sigma(1))].\sigma(1) {\downarrow} D1) \wedge$$

$$(\forall i, \sigma(1) \in \tau[i]).[\tau[i].\sigma(1) {\downarrow} D1 \;=\; \tau[nextuse(\tau, i+1, \sigma(1))].\#\sigma(1) {\downarrow} D1] \wedge$$

...

repeat above three clauses for each $\sigma(i)$ ... $\wedge$

...

$$(\forall i, 0 \leq i < |\tau|).[m_d.post^{\#\bar{\sigma},\ \bar{\sigma}}_{\tau[i].\#\bar{\sigma},\ \tau[i].\bar{\sigma}}\ ]$$

The definition consists of three clauses for each object $\sigma(i)$ of C plus one final general clause. The first clause for a object $\sigma(i)$ states that the enriched portion of this object does not change between the beginning of C and the first time the object is part of a tuple upon which a multimethod is invoked. This is because all the calls to enrichable methods that C makes are recorded on τ. Similarly, the second clause states that the enriched portion of $\sigma(i)$ does not change between the call to the final multimethod for which it is an argument, and the end of C. The third clause states that the enriched state of $\sigma(i)$ remains constant between a call to a multimethod which receives it as an argument, and the *next* call to a multimethod which uses it as an argument. These three clauses essentially ensure us that the only parts of C which affect the extended state of the object $\sigma(i)$ are the appropriate multimethod calls recorded in the trace. The final clause tells us how to determine the effect of these calls, by appealing to the enriched specification $\langle m_d.pre, m_d.post \rangle$ of $m()$ to determine how the final states recorded in position i of τ are related to the initial state recorded in the same trace entry.

```
class Plane {
  private member int stop_dist;
  public method int GetStopDist() { return stop_dist; }
  ...
}
class Runway {
  private member int length;
  public method int GetLength() { return length; }
  ...
}
```

Figure 1: Planes and Runways

4 Example: Planes and Runways

In this section we will illustrate our approach using a (necessarily) simple example. Imagine a large system for managing air traffic over a wide region. Two classes likely to found in such a system are Plane and Runway. These classes are likely to have a rich state modeling a large number of features of real planes and runways, but the two features that concern our example at the moment are the stopping distance required by the plane and the length of the runway. Figure 1 shows the structure that we envision encoded.

Now consider the question: when is it safe for a Plane p to land on a Runway r? Obviously the stopping distance of the plane should be less than the length of the runway. But with the huge variety of planes that a program like this must handle (passenger planes, private prop planes, cargo planes, etc.), each having their own indiosyncratic features, it is highly likely that there will be many different subclasses of Plane, and the same goes for runways. With this in mind it is quite easy to envision interaction between the specialized features of a subclass of Plane and the specialized features of a subclass of Runway that make it unsafe for some some of those planes to use some of those runways. For instance, JumboJet might be a subclass of Plane which has been enriched with information about the decibles of noise put out by its engines. Meanwhile, QuietRunway might model a runway with a noise restriction. How then can we write code that accounts for all of the different possible interactions that might affect whether a runway is safe for a plane to use?

The answer is, of course, that the Safe routine should be a multimethod, and the system should dis-

```
tuple class ( p:Plane, r:Runway ) {
  method boolean Safe() {
    if (p.GetStopDist() < r.GetLength()) { return true; }
    else { return false; }
  }
}

tuple class ( p:JumboJet, r:QuietRunway ) {
  method boolean Safe() {
    if ((p.GetStopDist() <= r.GetLength()) and
          (p.Noise() <= r.MaxNoise()) {
      return true; }
    else { return false; }
  }
}
```

Figure 2: Two versions of multimethod Safe

patch to the most specialized version of the method available based on *both* of its arguments. Figure 2 shows how two versions of Safe can be defined, one for the "most abstract" case of a Plane and a Runway, and one for the more specialized case of a JumboJet and a QuietRunway.

The advantages of this scheme should be clear: client code which is manipulating references to Planes and Runways does not need to be concerned with the particular subclasses that it is dealing with, it can simply put a plane and a runway into a tuple and call the Safe method, trusting the system dispatching mechanism to choose the appropriate version of Safe to invoke. But how does a client reason about such a call? We certainly need to provide specifcations for each version of Safe. Here is a first attempt (the notation $!\{x\}$ is used in a post-condition to indicate that the value of x does not change):

$$\mathsf{Safe(r,p)}_{R,P}.pre \equiv true$$

$$\mathsf{Safe(r,p)}_{R,P}.post \equiv$$
$$(\ (result = true) \Leftrightarrow$$
$$(\mathsf{r.length} \geq \mathsf{p.stop\_dist})\ ) \wedge !\{r, p\}$$

$$\mathsf{Safe(r,p)}_{Q,J}.pre \equiv true$$

$$\mathsf{Safe(r,p)}_{Q,J}.post \equiv$$
$$(\ (result = true) \Leftrightarrow$$
$$((\mathsf{r.length} \geq \mathsf{p.stop\_dist}) \wedge$$
$$(\mathsf{r.maxnoise} \geq \mathsf{p.noiselevel}))\ ) \wedge !\{r, p\}$$

Unfortunately, there is a problem with this approach, namely, the more specialized version of Safe is not a behavioral subtype of the more abstract version. Suppose Safe were called on a tuple with a JumboJet with a stopping distance of 1000 and a noise level of 200, and a QuietRunway with a length of 2000 but a maximum noise level of 180. The specification of Safe in the abstract case indicates that the method should return *true*, but the specification of the specialized version of Safe disagrees. More precisely, the post-condition of the specialized Safe is not stronger than the post-condition of the abstract version.[4]

But this is not surprising. The specification of the abstract version of Safe is overly specific. It does not take into account the fact that there are likely to be more specialized versions of Safe in the future that may reinterpret the notion of "safety". Of course, not every reinterpretation of "safety" should be allowed; its never "safe" to land a plane on a 1000 foot runway if the plane needs 1100 feet to stop! These, then, are the kinds of conditions that the specification of the abstract version of Safe should express: those properties that no more specialized version should be allowed to violate. Here is our second attempt:

$$\mathsf{Safe(r,p)}_{R,P}.pre \equiv true$$

$$\mathsf{Safe(r,p)}_{R,P}.post \equiv$$
$$(result = true) \Rightarrow (\mathsf{r.length} \geq \mathsf{p.stop\_dist})$$

This specification only requires that if *true* is returned, then the runway is long enough (and this will never be violated by more specific versions of Safe). It does not describe the specific conditions under which *true* is guaranteed to be returned, since it cannot anticipate what effect enrichments to Plane or Runway will have on this question. The specialized version of Safe is now a behavioral subtype of this version. Note that if we anticipated further enrichments of JumboJet and QuietRunway we would be wise to change the specification of that version of Safe to have a one-way implication as well.

Let us consider some client code which makes use of Safe. Suppose there exists a method in the class AirController which receives as a parameter a reference to a Plane and an array of references to Runway instances. The purpose of this method is to return to its caller the index in the array of a Runway on which it is safe to land the Plane. If no Runway in the array is safe, then the method returns -1. The code for this method appears in Figure 3.

First of all, we would like to be able to establish a specification for FindRunway that is valid in the case when we do not know what specific subclasses of Plane and Runway we are working with. We would also like this specification to remain valid regardless of the kind of subclasses that are passed to FindRunway and re-

[4] Interestingly, the same problem appears in a classic example of multimethods, as we will see in the next section.

```
method FindRunway ( p:Plane, rs:Runway[10] ) {
  for (int i = 0; i < 10; i++) {
    if ( (p, rs[i]).Safe() ) { return –1; }
  }
  return –1;
}
```

Figure 3: Client code: FindRunway

gardless of the kind of specialized versions of Safe that are introduced. Secondly, we would like to be able to use this general specification to arrive at the enriched behavior that is exhibited by FindRunway when it uses JumboJet and QuietRunway objects. Let us consider the general specification first.

$\mathsf{FindRunway(p,rs)}.pre \equiv true$

$\mathsf{FindRunway(p,rs)}.post \equiv$
$\quad !\{p, rs\} \wedge$
$\quad (\mathsf{result} \neq -1 \;\Rightarrow$
$\qquad \mathsf{rs[result].length} \geq \mathsf{p.stop\_dist}) \wedge$
$\quad (\mathsf{result} \neq -1 \;\Rightarrow\; |\tau| = \mathsf{result} + 1) \wedge$
$\quad (\mathsf{result} = -1 \;\Rightarrow\; |\tau| = 10) \wedge$
$\quad (\forall i, 0 \leq i < |\tau|).[\tau[i].name = \mathsf{Safe} \;\wedge$
$\qquad \tau[i].arg = (\mathsf{p},\ \mathsf{rs}[i]) \wedge$
$\qquad \tau[i].\mathsf{p} = \#\mathsf{p} \;\wedge\; \tau[i].\mathsf{rs}[i] = \#\mathsf{rs}[i]] \wedge$
$\quad (\mathsf{result} \neq -1 \;\Rightarrow$
$\qquad ((\forall i, 0 \leq i < \mathsf{result}).[\tau[i].res = false] \wedge$
$\qquad (\tau[\mathsf{result}].res = true))\ ) \wedge$
$\quad (\mathsf{result} = -1 \;\Rightarrow$
$\qquad (\forall i, 0 \leq i < |\tau|).[\tau[i].res = false])$

The first two clauses of the specfcation say all that we can say about the states of the runway objects, the plane, and the result of FindRunway in the general case. If the method returns a number not equal to –1, then we know that the runway at that index has sufficient length (otherwise Safe would have returned $false$). But this is a very weak result, and highlights the need for reasoning about the enriched behavior which this method will exhibit when making use of JumboJet and QuietRunway objects. In particular, with the specification above, we cannot come up with a sufficient pre-condition on the plane and list of runways that would guarantee that a safe runway can be found. Imagine an air traffic control program where we cannot prove to ourselves that planes will be allowed to land! In support of the enriched reasoning that we would like to perform, the remaining clauses of the specification give specific information about the way FindRunway interacts with the Safe multimethod. The third and fourth clauses show how the number of calls to Safe is related to the value returned by FindRunway. The fifth clause describes the contents of the ith trace element, in particular it says that the method called is Safe, the arguments are p and rs[i], and the values of those arguments are the same as their initial values. The sixth and seventh clauses say that –1 is returned by FindRunway only if *all* calls to Safe returned $false$, and if i is returned then that the first i calls to Safe returned $false$ and the last returned $true$. To establish the validity of this specification we would use rule **R1** to reason about the effect of the call to Safe in the code of FindRunway, and it would allow us to establish the appropriate assertions about the contents of τ. **R1** would allow us to appeal to the behavior of the version of Safe which is specialized for the declared types of p and rs, namely Plane and Runway. To make sure that out specification continues to hold in the presence of subclasses and more specialized versions of Safe we would simply have to check that each of the more specialized versions of Safe that we add to the system is a behavioral subtype of $\mathsf{Safe}_{R,P}$. We have already done this for $\mathsf{Safe}_{Q,J}$ so we can be sure that these subclasses will not cause FindRunway to behave in an unexpected manner, and if this was all we were interested in, no further work would need to be done.

As we stated above, however, our ultimate goal is to come up with the enriched behavior of FindRunway which is appropriate to JumboJets and QuietRunways. We can do this using **R2** and the specification above. FindRunway(p,rs).*post* tells us that –1 will be returned only if *all* calls to Safe fail. Also, an index i will be returned only if the call at position i to Safe (with p and rs[i] as parameters) succeeds. Finally **R2** tells us that we can appeal to the specification of the more specialized version Safe to reason about whether calls in the trace will succeed or fail. Thus we can precisely characterize the circumstances that will result in FindRunway failing to find a safe runway: if all runways are too short or have a noise maximum that is too quiet. On the other hand we can precisely characterize which runway will be chosen in the case that one is found: the first runway that is long enough and has a high enough noise maximum. This allows us to conclude with the following enriched specification:

$\mathsf{FindRunway(p,rs)}_{Q,J}.pre \equiv true$

$\mathsf{FindRunway(p,rs)}_{Q,J}.post \equiv$
$\quad !\{p, rs\} \wedge$
$\quad (\mathsf{result} = -1 \;\Rightarrow$
$\qquad (\forall i, 0 \leq i < 10).$

$[(\mathsf{rs}[i].\mathsf{length} < \mathsf{p.stop\_dist}) \vee$
$(\mathsf{rs}[i].\mathsf{maxnoise} < \mathsf{p.noiselevel})]) \wedge$
$(\mathsf{result} \neq -1 \;\; \Rightarrow$
$(\forall i, 0 \leq i < \mathsf{result}).$
$[(\mathsf{rs}[i].\mathsf{length} < \mathsf{p.stop\_dist}) \vee$
$(\mathsf{rs}[i].\mathsf{maxnoise} < \mathsf{p.noiselevel})] \wedge$
$(\mathsf{rs}[\mathsf{result}].\mathsf{length} \leq \mathsf{p.stop\_dist}) \wedge$
$(\mathsf{rs}[\mathsf{result}].\mathsf{maxnoise} < \mathsf{p.noiselevel}))$

Note that we were able to arrive at this specification using only the information provided in the trace specification, we did not have to look at the code of FindRunway a second time.

5 Behavioral Subtyping Revisited

A classic examples of multimethods is the *equal* operation defined on *points* and *color points*. The example consists of a tuple class ⟨equal, (Point, Point)⟩ and derived class ⟨equal, (ColorPoint, ColorPoint)⟩ defined as follows:

```
tuple class (p1: Point, p2: Point)
    { method equal() : bool
        {return ((p1.x()==p2.x()) &&
                (p1.y()==p2.y()));} }
tuple class (cp1: ColorPoint, cp2: ColorPoint)
    { method equal() : bool
        {return ((cp1.color()==cp2.color())&&
           (p1.x()==p2.x())&&(p1.y()==p2.y()));}
}
```

The equal() in the tuple subclass provides richer behavior than the one in the base tuple class. Given two points, the equal() function of the base tuple class only checks their x, y coordinates, whereas the one defined in the tuple subclass also checks their color. A piece of client code that manipulates two objects whose static types are Point, may invoke the equal() method. If the dynamic types of these objects are ColorPoint, multiple dispatch will ensure that the method defined in the tuple subclass is invoked with the intended effect.

But there is a problem. With the natural specifications for the equal() methods in the two tuple classes, behavioral subtyping is violated! Intuitively, the problem may be seen as follows: Suppose we have two ColorPoint objects that have the same x and y coordinates, but different color values. An analysis of the client code, on the basis of the static types (which is Point) of these two objects tells us that the value returned by a call to equal() will be true. But this conclusion will be incorrect operationally because the call will be dispatched to the method defined in the tuple subclass, and that method will return false because the color values of the two points do not agree.

There are two possible solutions to the problem. The first would be to say that the method equal() defined in the tuple base class is already completely defined, and specifies exactly what value it will return based on just the x, y components of the state. Any attempt at enriching the state with a component such as color and using information about this component to influence the value that equal() returns will contradict what we have already determined about the behavior of equal(), and hence such an attempt should not be made.

The other solution would be to weaken the specification of the base tuple class equal(). We could specify that it returns false if either the x or the y coordinates of the two points are unequal, but if both coordinates of the two points are equal, then we do not specify what equal() would return. In that case, the natural specification of the equal() defined in the tuple subclass, that it returns true if all components including the color are equal, and false otherwise, is an acceptable enrichment. Of course, the specification of the base tuple class may now be considered too weak but this seems to be a general problem rather than a peculiarity of this example; in general, for methods whose only purpose is to return a value, once the base class definition specifies exactly what value is to be returned, any attempt to enrich this method is likely to lead to problems.

6 Discussion

Multimethods promise to extend the power of the OO approach by making it possible to dispatch on the dynamic types of more than one argument. Several language notations have been designed to investigate the power of multimethods. Much work has also been done on such questions as ensuring type safety, investigating techniques for efficient implementation of multiple dispatch, etc. Our focus in this paper has been different. We have focussed on behavioral questions. Specifically, we have attempted to answer two fundamental questions: First, how do we extend the notion of behavioral subtyping to multimethods? Second, how do we arrive at the richer behavior that a piece of client code exhibits as a result of calls to multimethods it makes being dispatched to methods defined in tuple subclasses, depending on the dynamic types of the objects involved?

In answering the first question, one important issue we faced was that objects passed as arguments to a multimethod are not encapsulated into a unit, they may be used singly, or paired with other objects in calls to other methods; hence we could not introduce invariants that were relations between the different arguments of a multimethod. Nevertheless, we were able to provide a natural definition of behavioral subtyping for multimethods, and this definition reduced to the standard definition in the case of single dispatch methods. In Section 5 we looked at the classic equal() method on Points and ColorPoints; our analysis, based on our notion of behavioral subtyping, led us to conclude that if we just use the intuitive specifications for the equal() method, we may arrive at invalid conclusions about the client code.

The second question was more involved since we had to provide a way to arrive at the richer behavior corresponding to the tuple subclass. The key idea underlying our approach was to associate a *trace* τ with the client code and record in τ all calls the code makes to various multimethods and include information about τ in the specification of the client code. Once we have such a specification, we can use knowledge of the dynamic types of the objects that the client code uses and that it passes as arguments to the methods, to arrive at a richer specification by appealing to the specifications of the behaviors of the methods as defined in the tuple subclasses. The formalization of this enriching process was captured in our rule R2, while the (simpler) task of arriving at the original specification of the client code, in particular the recording of information about method calls in the trace was captured by the rule R1. In Section 4, we applied our technique to a simple, but non-trivial example to see how the enriched behavior of the client code may be arrived at without reanalysis of that code.

We conclude by considering some problems for future research. The first is a syntactic, but important, task. We need to develop notations and conventions involving traces and their elements in order to make it simple to specify commonly occuring patterns of calls to multimethods. Classic work on formal languages holds considerable promise here since our traces have a very uniform structure. The second problem is much more challenging: extend our work to allow for aliasing among objects. Standard work on reasoning about program behavior has generally, as we have done, ignored aliasing issues. But in OO, this is not a reasonable approach since aliasing seems to be used in fundamental ways in OO systems. We have developed an approach [9] that works in simple cases, but it does seem very detailed and tedious. More useful would be approaches that perhaps exploit specific aliasing patterns to make the reasoning task easier.

References

[1] K. Bruce, L. Cardelli, G. Castagna, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[2] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM TOPLAS*, 17:431–447, 1995.

[3] C. Chambers. Object-oriented multi-methods in cecil. In *Proc. of ECOOP '92*, volume 615, pages 33–56. Springer-Verlag, 1992.

[4] C. Chamers and G. Leavens. Type checking and modules for multi-methods. *ACM TOPLAS*, 17(6):805–843, 1995.

[5] G. Leavens and T. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of OOPSLA '98*, pages 374–387. ACM Press, 1998.

[6] B. Liskov and J. Wing. A new definition of the subtype relation. In *ECOOP*, 1993.

[7] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16:1811–1841, 1994.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[9] N. Soundarajan and S. Fridella. Reasoning about polymorphic behavior. In Ege, Singh, Meyer, Riehle, and Mitchell, editors, *Proceedings of TOOLS 26*, pages 346–358. IEEE Computer Society Press, 1998.

[10] G. Steele. *Coomon LISP: The language*. Digital Press, 1990.