Prototypes with Multiple Dispatch

Lee Salzman*

26th April 2004

Abstract

Modern object-oriented programming languages have a diverse and disjoint set of features that all prove beneficial to the task of designing and organizing maintainable programs. Three emerging families of object-oriented languages that are the focus of this paper - prototype-based languages, multi-method languages, and subject-oriented languages - each provide many orthogonal benefits to program design that prevent programs from becoming procedural and brittle. These language families, however, rely on differing interpretations of object-oriented programming such as the message passing model and classification. Language designers have, so far, failed to address this semantic schism with a single object model that not only unified these approaches but also preserves the original expressiveness of these approaches without undue complications or restrictions. This paper introduces a novel object model, prototypes with multiple dispatch (PMD), that unifies these concepts by dispensing with class-based and message-based conceptions of object-oriented programming and instead reinterprets it in a purely prototype-based model consisting of interacting objects and context. A formal semantics of PMD is presented, and finally, the paper discusses various implementation techniques in the context of the programming language Slate, based on PMD.

1 Introduction

Programming practice has evolved into a maze of unique programming languages, each with expedient yet differing or incompatible feature sets. This offers the programmer the dilemma of choosing the lesser of evils, or rather which features he can more easily live without, rather than the choice of which tool is best for the job. The programmer must often use awkward facilities to glue various programs written in different programming languages together, as one language often does not fulfill his needs. Unifying approaches to programming languages simplify the programmer's job, both reducing the number of tools he needs to work with and allowing him to consolidate his expertise with one system toward solving a problem.

*This paper is submitted both as independent study research and as an undergraduate thesis, advised by Jonathan Aldrich, to the Carnegie Mellon University Department of Philosophy's Logic and Computation program.

This problem prevails especially in the field of object-oriented programming languages, which model programs as a collection of small, anthropomorphic units of program functionality (objects). Because this model carried wide-spread intuitive appeal, many different families of object-oriented programming arose which all espouse different views of what exactly an object is. However, the underlying conceptual model of interacting object remains the same, so one desires that all these differing notions of objects be unified into one model of programming that does not require a choice of benefits.

This paper considers three recent, disparate innovations in object-oriented programming: prototype-based languages, multi-method languages, and subject-oriented languages. Prototype-based languages and multi-method languages provide naturally orthogonal feature sets, yet at their core depend on conflicting notions of objects that complicate their integration into a single language. Either model naturally supports some essential characteristics of subject-oriented languages, yet neither one alone sufficiently captures all of them. A paradigm providing both the features of prototype-based languages and multi-method languages would allow their orthogonal feature sets to address problems together, as well as provide a natural host for a subject-oriented language by capturing all of its essential properties.

This paper presents prototypes with multiple dispatch and its intuitive basis, otherwise called "PMD", a model integrating the features of prototype-based languages, multi-method languages, and subject-oriented languages into a single, consistent paradigm of interacting objects. The paper further provides a formal semantics for understanding PMD and its implications upon language structure. Lastly, the paper discusses the practical implementation of PMD in the programming language Slate [Rice and Salzman, 2004] and various expedient optimizations to the basic algorithm, including sparse representation of dispatch information, partially dispatching upon method arguments, and method caching.

2 Background

Object-oriented programming encompasses a wide diversity of programming languages that frame the problems of program design and reuse in a model similar to how people often describe and manage real-world phenomena, as a system of interacting objects, each with individual behaviors and motives. This conception of programming dates back to the language Simula [Dahl et al., 1970], which was literally intended for simulating real-world problems in a programming environment, a task which proved cumbersome in contemporary procedural languages. Simula gave birth to a progression of families of programming languages which are collectively known as object-oriented programming languages, and which either trace much of their semantics back to ideas presented

```
variable Fish : integer;
variable Shark : integer;
data Animal
{
    type : integer;
    healthy : boolean;
}
procedure swallow (animal : Animal, other : Animal);
procedure swimAway (animal : Animal);
procedure fight (animal : Animal, other : Animal)
{
    animal.healthy := False;
}
procedure encounter (animal : Animal, other : Animal)
{
    if (animal.type = Shark)
    {
        if (animal.healthy)
        {
            if (other.type = Fish)
              swallow (animal, other);
            else
            if (other.type = Shark)
              fight (animal, other);
        }
        else
          swimAway (animal);
    }
    else
    if (animal.type = Fish)
    {
        if (other.type = Shark)
        {
            if (other.healthy)
              swimAway (animal);
        }
    }
}
```

Figure 1: Procedural example

in Simula or at least address many of the issues Simula was intended to, rather, by framing programming problems as one might real-world phenomena.

Figure 1 presents an example of a procedural program modeling a small food chain consisting of sharks and fish. This running example shall be used both to exhibit differences in various object-oriented paradigms and motivate their introduction. In this example, unhealthy sharks will swim away from any other animals they encounter, while healthy sharks will eat any fish they find or fight any other sharks they encounter. Fish will simply swim away from any healthy sharks.

While the example is concise, it is surprisingly complex to understand despite its small size. Notice how the program must explicitly provide all the details of identifying the type of an animal and how this identification code is entangled with the more relevant details of the program, the behavior of the animals. Further, the program must explicitly choose an awkward representation of both animal type and health.

2.1 Class-based Programming Languages

The original family of object-oriented programming languages models objects by classifying them according to their intended purposes. These classes subsequently become explicit structural descriptions of how ideal objects, which are merely instances of some class, appear and how they should interact with other objects. Objects may then be instructed to perform some named task, and depending on the behavior of objects in its class, will perform the task in an appropriate manner, as determined by methods, polymorphism, and inheritance.

2.1.1 Methods

Contemporary languages of Simula structured programs as a collection of procedures, lists of instructions on how exactly to compute a result, that may in turn call upon other procedures to help compute this result. However, at best, this only allows a programmer to describe how the result of some real-world phenomena should be approximated. As a departure from this model, class-based programming such as in Simula appeals to the idea of methods, or rather, that different classes of objects have different methods of performing a task. Each class may then implement a method for performing some named task on an object in that class, possibly involving some other objects which serve as arguments to the method. The class thus plays a role in selecting which method should be used to perform the task, rather than as a catch-all procedure for which the user is entirely burdened with determining when and where to apply it. The provider of a class may instead abstract some of these details from the user, allowing for a simple reuse of expertise.

2.1.2 Polymorphism

Even methods, however, remain somewhat obtuse and retain shades of their procedural roots, for they still focus extensively on concrete details of how some method should be performed with respect to some known class of objects. The programmer needs to know the exact contexts in which he is using some method and what specifically he is using it with. In effect, he must know at all times, while writing a program, the exact implementation of any method he is using. In large programs, such global knowledge may be impossible, if not less than expedient, to come by.

Simula introduced the "virtual" method to ease this burden. The virtual method is a place-holder that allows the actual method implementation to vary based on the class of an object. Each class is responsible for defining its own version of the method, and the virtual method interfaces with the differing versions. Invoking the virtual method will invoke the respective method implementation for the class, without any knowledge of the exact class of the object it is invoked upon. Objects become interchangeable, a property known as polymorphism, where any object can substitute for another so long as they implement the same necessary interface of virtual methods. This frees the programmer to focus on the abstract behavior of objects, without worrying about specific details of how the object implements them. However, he still must decide exactly which methods must be virtual ahead of time. This creates the problem of fragile interfaces that mirrors the fragility of how normal procedures must decide all details of the data they are manipulating. Interfaces must be constantly updated to accommodate any extra necessary virtual methods of objects implementing the interface, until they become sufficiently general to support most implementations. For rapidly evolving or very large interfaces, this may require many tedious iterations of updating.

Subsequent languages such as Smalltalk [Goldberg and Robson, 1989], however, improved upon this idea by reconsidering what a method is in anthropomorphic terms. One instructs an object to perform some method by entering into a discourse with it, or rather, by sending it a message which it receives and to which it responds with some reply depending on its class. The details of what method the object is using to generate this reply are not exposed to the programmer, only that he is sending it a message. He is not only no longer burdened with knowing the class of objects he is using, but he also no longer need laboriously specify and maintain interface definitions. This level of polymorphism avoids the fragile interface problem while also freeing up the process of specifying abstract interfaces from notions of implementation so that it may be orthogonally provided in the most humanly expedient fashion possible.

2.1.3 Inheritance

Another central idea of class-based programming languages is that a class may optionally inherit all the methods provided by another class, or rather, that a class may be a subclass of some other class which serves as its superclass such that objects in the subclass may perform any methods which objects in the superclass may, as well as any restrictions or extensions of these methods particular to that subclass. This allows for a significant reuse of structure as a programmer can enable a class to implement an entire collection of methods without having to rewrite any of them. If any of these methods are inadequate for whatever purpose of the class, the programmer may then extend them to suit the new desired behavior or define entirely new methods not prescribed by the original superclass. When combined with polymorphism, inheritance provides great expressiveness in easily constructing new objects which may fit into a variety of contexts without having to rewrite or restructure large bodies of code.

Smalltalk, for its message-passing model of class-based programming, interprets subclassing as a type of referral between classes. When a message is sent to an object, its class is called upon to interpret the message for that object. Should the class not define any method responding to a message, or rather not understand it, the class will refer the message to its superclass to see if the superclass understands it. The process recurses until any of the class' superclasses understands the message and may respond to it, or until all superclasses have been exhaustively searched with no appropriate method found, in which case the message is in error.

2.1.4 Example

Figure 2 presents the running example in the class-based programming style. The first striking feature of this example is that it loses conciseness over the procedural version due to the necessary interface code. However, the example does show factoring of some concerns, as fish and sharks are responsible for deciding what happens when they encounter other animals, but only in the case the shark or fish is the encounterer. The programmer is forced to decide whether the encounterer or encounteree is the more dominant decider of animal behavior. The Shark class now encapsulates the details of its health and need not expose the particular implementation details. Further note that each class is still responsible for deciding determining the animal type of the encounteree in an ad-hoc and tangled manner much as in the procedural example.

2.2 Multi-methods

Despite its flexibility, this message-passing model of polymorphism, which assumes a one-way discourse with an object, is not well-suited for modeling situations where an object must have a method for interacting with a variety of

```
class Animal
{
    method swimAway ();
    virtual method encounter (other : Animal);
}
class Fish inheriting Animal
{
    method encounter (other : Animal)
    {
        if (classOf (other) = Shark)
        {
            if (other.isHealthy())
              swimAway ();
        }
    }
}
class Shark inheriting Animal
{
    variable healthy : boolean;
    method swallow (other : Animal);
    method isHealthy ()
    {
        return healthy;
    }
    method encounter (other : Animal)
    {
        if (isHealthy())
        {
            if (classOf (other) = Fish)
              swallow (other);
            else
            if (classOf (other) = Shark)
              fight (other);
        }
        else
          swimAway ();
    }
    method fight (other : Shark)
    {
        healthy := False;
    }
}
```

Figure 2: Class-based example

different classes of other objects. The programmer, when faced with this problem, ends up defining one monolithic method which is not only responsible for determining which class of objects its arguments belong to, but which is further entirely responsible for dispatching to appropriate code to handle objects of whichever class it manually determined them to be in. Essentially, the programmer is returned to an entirely procedural mode of programming and loses many of the benefits of object-oriented programming.

The message-passing model of polymorphism dispatches to a method for some respective message based upon a single object's class, referred to as single dispatch, such that the method is chosen from the most specific subclass to which the object belongs to and to which the method is defined. It is effectively taking the set of all methods for a given message defined at either the object's class or any of its subclasses, and finally choosing the least element of a linearly ordered set of methods as defined by the subclass relation on each method's receiver argument. A method is thus less than another method in this ordering if the class of its receiver argument is a subclass of the class the other methods receiver argument, and so the least method in this ordering must be the most-specific according to the subclass relation.

There exists a corresponding multiple object form of polymorphism, or multiple dispatch, used in systems such as CLOS [Steele, 1990], Cecil [Chambers, 92], and many other programming languages, where a method, referred to as a multi-method, is selected based upon the class of all arguments to the method, not just the argument distinguished as the receiver of a message. In this model, a class product is generated from the classes of all arguments to a method, and the collection of all such methods with the same name, a method family, is ordered according to the subproduct relation on their class product, rather than by the class of some receiver alone. Essentially, the subproduct relation is substituting for the subclass relation such that one multi-method is more specific than another if the classes assigned for all its arguments are subclasses of the classes assigned to the corresponding arguments in the other. This ordering, as used in Cecil, is partial in that if at one argument position the subclass relation holds and at another it does not, the multi-methods are no longer ordered. Alternatively, the class products of multi-methods may be lexicographically ordered, as in CLOS, so that the subclass relation for the classes of earlier argument positions is more significant than and overrides the subclass relation for later arguments, resolving any ordering ambiguities. Thus, all multi-methods are now linearly ordered provided the ordering contains no methods with duplicate argument classes.

In a multiple dispatch model, the task of dispatching to code based on all argument positions is thus offloaded from the programmer and back into the programming language, offering a stronger and more expressive form of polymorphism. Methods for dealing with specific combinations of argument classes

need no longer be in one monolithic method and may be independently defined. The resulting method family may even extend as necessary, without having to deal with large and brittle program code.

2.2.1 Example

Figure 3 presents the running example framed in terms of a language with multi-methods. Similarly to the class-based example, it achieves a factoring of concerns into classes. However, note that the multi-methods examples completely eliminates the ad-hoc protocol needed to determine the animal type of the encounteree and is more concise than the class-based example. Other animals could be trivially added to the example by simply defining new methods, rather than having to modify each class as with the solely class-based example. The example becomes satisfyingly simple, but it still retains the awkward protocol for deciding on and responding to the health of the shark. The new factoring has also exposed new redundancies, such as having to describe that an unhealthy shark should swim away from other animals in two different places.

2.3 Prototype-based Programming Languages

While class-based programming languages offer a more familiar simulation of objects than procedural programming languages, classes sometimes fail as units of program design in that they may either be too restrictive or not restrictive enough for representing objects. A class must specify the behavior of an object for its entire lifetime, in which case it is too restrictive for objects that evolve. Differing behavior over the lifetime of an object must be entangled in a single class or parameterized in an ad-hoc manner, and objects which must significantly evolve during their life in the program prove cumbersome. Further, one often deals with unique objects that don't easily subscribe to any classification or for which multiple instances of its chosen class may be contradictory. Similarly, one may wish to uniquely specify the behavior of an object over only a portion of its lifetime or behavior, in which case a class is not restrictive enough.

Classes do not easily express a host of these design issues that often surface in programs, and so necessitate a more general unit of object representation that subsumes the uses of classes as well as addressing these issues. The programming language Self [Ungar and Smith, 1991], a descendant of Smalltalk, however, is the progenitor of a family of programming languages, referred to as prototype-based programming languages, that reexamine the notion of objects and their representation in this context.

2.3.1 Prototypes

Prototype-based programming languages confront head-on the issues of how exactly an object is represented and whether notions traditionally taken for granted, such as classes, best represent object-oriented programming system.

```
class Animal;
method swimAway (animal : Animal);
class Fish inheriting Animal
{
}
method encounter (animal : Fish, other : Fish)
{
}
method encounter (animal : Fish, other: Shark)
{
    if (other.healthy)
      swimAway (animal);
}

class Shark inheriting Animal
{
    healthy : boolean;
}
method swallow (animal : Shark, other : Animal);
method encounter (animal : Shark, other : Fish)
{
    if (animal.healthy)
      swallow (animal, other);
    else
      swimAway (animal);
}
method encounter (animal : Shark, other : Shark)
{
    if (animal.healthy)
      fight (animal, other);
    else
      swimAway (animal);
}
method fight (animal : Shark, other : Shark)
{
    animal.healthy := False;
}
```

Figure 3: Multi-methods example

Classes are eliminated in favor of the idea of prototypical objects, objects that serve as examples from which like objects may be constructed, and which are self-representing, no longer depending upon classes to describe their behavior nor belonging to any particular classes, and so may be uniquely constructed. Objects directly contain methods and other information normally posited within a class, and so are free to vary in behavior on a per-object basis, rather than on a per-class basis. Methods may be added or remove from such objects at any times, and so objects are free to evolve and differentiate independent of the restraints of any classification. Similarly to how the notion of message passing divorces the specifying of method interfaces from method definition, prototype-based languages divorce the classifying of object functionality from the task of object construction, again freeing it to be provided for in more humanly expedient ways when possible.

2.3.2 Cloning

As objects in prototype-based programming languages are self-representing, they may not rely upon some external blueprint for construction such as a class. So, prototype-based languages instead provide for reuse of objects by cloning, where and only where it makes sense to do so, prototypical objects that embody all the necessary behavior of the object. The idea of cloning appeals to a simple biological metaphor of creation where, given some object, a clone of it may be constructed that bears all the methods and information the original object contained. This new clone is a distinct object and is no way constrained to remained at all like its originator, unlike in a class system where the classes of objects are usually fixed and an object may not change its behavior in any way not prescribed ahead of time by its class. Once cloned, the new object may thus be refined to whatever purpose the programmer desires, whether it is to create a new sort of prototypical object or to employ it as just another instance.

2.3.3 Delegation

Cloning, however, suffers from the fact that the newly cloned object is entirely separate from its originator, and so beneficial changes made to the originator won't propagate into the clone. Objects and their methods may need to change and evolve while maintaining their identity, thus creating a new problem of consistent and updated behavior for existing objects. To overcome this, Self introduces the idea of delegation where, in a manner analogous to the superclass referral process for messages in Smalltalk, an object may choose to delegate the responsibility of responding to undefined messages to some other object. Should an object have no method defined to that will respond to a message, it will refer the message to any objects it delegates to and utilize whatever method it finds they would have used to respond to the message.

Both the object and any objects it delegates to are free to change independently, and any changes to the objects delegated to will be dynamically inherited into

said object. So, if the objects delegated to redefine or define any new methods for some messages, the object delegating to them will automatically gain the ability to respond to those messages with the particular methods if it does not already define its own methods for the messages. Moreover, objects in Self are free to change the objects they delegate to at any time, and so may use delegation as a simple means for reconfiguring the behavior of their methods. Otherwise, one would be forced to explicitly parameterize predetermined methods based on the object's state and fall into a more procedural programming style. Delegation thus not only allows for dynamic reuse of methods, but also provides an expressive means of factoring object behavior for varying conditions into distinct units.

2.3.4 Example

Figure 4 presents the running example in a prototype-based language. The first striking feature of this example is that, similarly to the multi-methods example, it has become much more conciseness than the solely class-based example, and is possibly more concise than the multi-methods example. However, in terms of requiring the programmer to decide on the dominant decider of encounter behavior, it achieves a factoring no better than the class-based example. The ad-hoc protocol for deciding the animal type of the encounteree remains almost unchanged from the class-based example. The prototype-based example, however, achieves the new conciseness by modeling the health of a shark in a drastically different fashion than the preceding examples. Instead of using an ad-hoc protocol for health, it represents health as an intrinsic property of the Shark object itself. When a shark is healthy, it delegates to the HealthyShark behavior, and otherwise evolves to delegate to the DyingShark behavior once it is injured. The differing behavior for each state is now factored into two distinct objects, at least for the encounterer. This example, while also pleasingly simple, never the less retains some of the awkwardness of the class-based example.

2.4 Subject-oriented programming

While object-oriented programming allows objects and their expertise to be reused, such behavior is objective in that it must be fixed ahead of time with respect to and must provide for all anticipated uses. An object may parameterize its behavior on its state or employ delegation, but this requires that the object be explicitly notified any time it needs to behave differently, by changing state or delegations, using ad-hoc protocols for doing so. Users of the object must manage, at the granularity of individual objects and without limitation, how the object behaves, any changes to its behavior temporarily necessary, and restoring the object's original behavior after usage. This again leads to fragile, ill-factored code where behavioral details of an object are entangled in all the code using it. Further, if one wishes to restrict usage of an object by certain parties for security, these restrictions must be provided wholesale, as any explicit behavior parameterization scheme would quickly be exploited to access this forbidden

```
object Animal;
object Fish = clone (Animal);
object Shark = clone (Animal);
object HealthyShark;
object DyingShark;
addDelegation (Shark, HealthyShark);
method Animal.swimAway ();
method Fish.encounter (other)
{
    if (other.isA(HealthyShark))
      swimAway ();
}
method HealthyShark.swallow (other);
method HealthyShark.fight (other)
{
    removeDelegation (HealthyShark);
    addDelegation (DyingShark);
}
method HealthyShark.encounter (other)
{
    if (other.isA(Fish))
      swallow (other)
    else
    if (other.isA(Shark))
      fight (other)
}
method DyingShark.encounter (other)
{
    swimAway ()
}
```

Figure 4: **Prototype-based example**

behavior.

Subject-oriented programming, as espoused by the Self language extension Us[Ungar and Smith, 1996], allows for differing, subjective views of an object that are implicit properties, as opposed to explicitly managed, of what is using it. The different views restrict which methods are visible of all objects used within them. This not only allows methods to be hidden in some perspectives and visible in others, but also allows entirely different versions of a method to be provided depending on how the method is viewed. This provision for contrasting behavior of the same object's methods allows for a much stronger notion of security and multiple user orientation than mere behavior hiding would otherwise.

2.4.1 Subjects

Subjects provide the unit of perspective in subject-oriented programming, and appeal to the idea that depending on the subject at hand, one may expect an object to perform differently than usual of some objective description of its behavior. The language Us views subjects as implicit method arguments that, exactly as with multiple dispatch, are to be dispatched upon in addition to normal method arguments. These subjects are also represented by objects, and as with any other object in Self, may use cloning and delegation to both construct new subjects and to compose them with other subjects in the usual manner. By delegating from one subject to another, a subject inherits all of the peculiarities of the subject it is delegating to.

The programmer further manages subjects at the granularity of units of program design, rather than at the usage of specific objects. Individual objects no longer manage the bookkeeping details of providing different behavior depending on use. The current subject is a global property of the running program such that, so long as it is in use, it both globally and implicitly effects the behavior of all objects viewed within it and may be changed at will. Delegation, in combination, allows the composition of novel subjective behavior with the currently prevailing behavior, by creating new subjects that delegate to the current subject. Further, if the operation of changing a subject is itself a method, the current subject may restrict, where appropriate, what changes of subject are allowed.

3 Prototypes with Multiple Dispatch

3.1 Prototypes Gone Wrong

The entangling of data usage concerns with all the program code that uses the data motivated the introduction of Simula and object-oriented programming. This same problem arose in new contexts, despite this advance, motivating new families of object-oriented programming to cope with it. While ultimately suffering the same problem, these contexts do not easily yield to the same conceptual

machinery for eliminating it. Moreover, the implementations of the differing mechanisms often cross-cut each other, preventing language designers from implementing all of these orthogonal solutions in a single language or considering them in a single framework [Abadi and Cardelli, 1996]. This partitioning of problems and solutions forces a degree of schizophrenia upon the programmer. Despite the diversity of program design problems, he must choose a single tool that does not adequately address all of the problems and in many cases rules out the use of other solutions.

Multi-methods, for instance, depend upon the notion of a fixed inheritance hierarchy, usually but not always provided for by classes, for a globally applicable ordering from most specific to least specific of multi-methods. It generalizes the specificity of individual classes into a class product for the method parameters that may subsequently be ordered. Any modification or restructuring of the class hierarchy forces this order to be reevaluated, and so it must be constantly synchronized. Further, methods must be specified upon classes entirely for the sake of generating this ordering. Should one wish to only define a multi-method upon some unique object, a class must be created explicitly by the programmer or implicitly by the implementation so that this multi-method will sequence properly in the global ordering of multi-methods.

Prototype-based languages, however, take as their main hypothesis that the classes needed in languages with multi-methods do not accurately represent the process of object construction. They do not, however, rule out classes being developed as an expedient organization of pure objects and delegation, as they may express classes as normal objects that serve as repositories of methods and are delegated to by convention. Yet to require the usage of classes or some external typing system as a prerequisite for using multi-methods at all would prevent the two from being used upon the same objects and problems, although while still allowing both to be used in a mutually exclusive fashion within the same language. This only serves to reinforce the programmer's growing sense of schizophrenia.

Further, in a prototype-based language, each object is conceptually in its own class. A cloned object must support all the original methods of its source object and allow for revision into an entirely new prototype. The new object is not simply in a subclass of the class of the object it was cloned from, since in a true prototype-based languages, methods may be removed as well as added. The new object should not be forever constrained to provide all the methods and behavior of its original source. If a multi-method implementation is to properly co-exist with prototypes, it must define new methods for the cloned object's class based upon all methods existing for the source object's class so that they properly inherit and yet still support removal. Further, any changes in what an object delegates to generate an entirely new class hierarchy upon which to determine method specificity, again reinforcing that the new object's class is not

merely a subclass of the source object's class, but instead is a disjoint class of its own.

In the face of these implementation difficulties, languages such as Cecil that attempt to integrate prototype-orientation and multi-methods enforce restrictions upon the prototype-oriented model that result in a level of expressiveness little better than classes. Cecil does not allow delegation relations to be changed, and so they are fixed at the point an object is created. Further, objects may not remove methods from themselves once they are added. Prototypes thus become essentially classes, where the operation of instantiating a class has merely been conceptually washed over with the idea of cloning, but not to its fullest extent. While methods may still be added to classes at any time, this feature exists in many normal class-based languages incorporating multiple dispatch such as CLOS.

Subject-oriented programming benefits from the usage of either paradigm, but currently does not have an adequate host that may easily supply both. Multi-methods naturally represent the role subjects play in dispatching to individual multi-methods by treating the subject as just another argument to be dispatched upon. Prototype-based languages naturally represent distinct and evolving user contexts or subjects using cloning and delegation to compose new subjects. One noteworthy subject-oriented language, Us, chooses the prototype-based model to flexibly represent the construction of subjects via the normal prototype-oriented features of its host language Self. However, because Self is inherently a single dispatch language, Us must use an ad-hoc form of double-dispatch not provided for by Self which the authors admit is slow and costly. Were a language to provide both mechanisms, such compromises could be avoided.

3.2 Internalizing Multi-methods as Interactions

This apparent conflict between prototype-based languages and multi-methods arises because of their contrasting views of object representation. Prototype-based languages espouse an internalized approach, that objects must be self-representing and carry all the necessary information to describe their behavior. Multi-methods favor, but do not necessitate, an externalized approach, that external decisions must be made about a multi-method's applicably based upon classes of its arguments, where classes themselves are a notion external to objects regardless of their utility. To cleanly reconcile the two, the notion of multi-method applicability must be internalized into an object's representation, rather than retaining its external character.

Traditionally, a multi-method is applicable to some list of arguments if the class of each argument is a subclass of the class of each parameter in the respective positions. Further, one multi-method is more specific than another if the class of each of its parameter is a subclass of the class of the other multi-method's

parameters in the respective positions. So, one dispatches a multi-method by both finding the set of applicable methods, and then selecting the most specific of those methods to dispatch to, if one exists.

Instead of making an external decision about the class of an object, so that the subclass relation may be evaluated for method applicability, I propose that one directly query the object if it supports a given multi-method. I introduce the concept of a role, such that each multi-method represents an interaction, or consensus, between its arguments where each argument plays a specific role in the execution of that multi-method. Each multi-method defines roles upon all the objects it is applicable to such that the objects may only perform specific roles in the interaction, but not others.

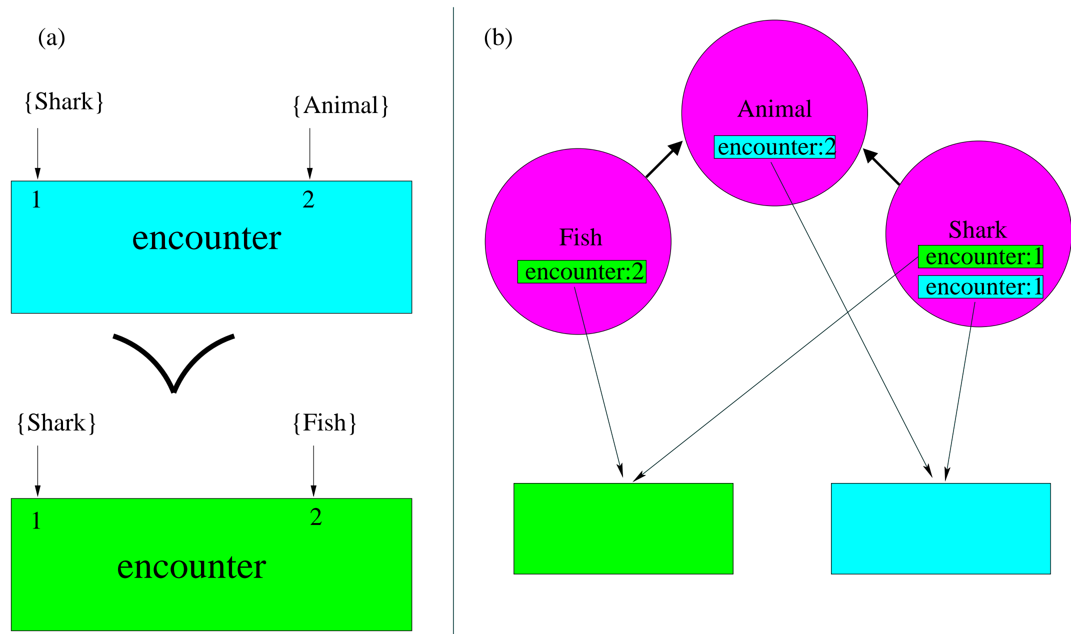

Figure 5: Comparison of (a) traditional multi-method organization and (b) PMD

The task of determining multi-method applicability is revised so that each argument's role is decided at the point of invocation, and one queries each argument to determine if it contains the specified role for the multi-method. Should the role not be found on the argument, its delegation relationships are traversed to determine if the delegated-to objects support the role for the argument. Further, the process of finding candidates for applicability may be folded into the process of role discovery, as roles also supply information about which multi-methods may be partially applicable, not just those which are entirely applicable. This

separates the decisions of multi-method applicability from any notion of class, and the role information may be freely and cheaply reproduced should an object need to be cloned.

Without external class information, there is no provision for deciding whether one multi-method is more specific than another. An expedient notion of specificity independent of classes must be introduced to deal with the delegation present in prototype-based languages. I propose a linear ordering of the objects an argument delegates to that may be easily evaluated at the time of dispatch, rather than at the time a multi-method is defined, so that no global ordering of multi-methods needs to be maintained.

The process of dispatch then shifts from subclass determination to role discovery. Each argument to a method invocation and the objects it delegates to are traversed, searching for any roles those objects may support while noting the position in the delegation ordering the roles were found. Multi-methods whose full set of roles are discovered on the arguments serve as applicable methods. The positions at which the roles were found for each argument are composed into a specificity ranking for a method so as determine which method is the most-specific. This neither depends upon any notion of class nor requires the delegation relationships among objects to be predetermined, yet still preserves the discovery of applicable and most-specific multi-methods. As objects internalize role information, the process further behaves well under cloning and removal of methods. Prototype-based languages may now coexist peacefully with multi-methods, without compromising the expressiveness of either approach.

Figure 5 illustrates this difference in organization. Under the traditional multi-method approach, the encounter methods, illustrated as rectangles, appeal directly to the Animal, Fish, and Shark classes for their parameters, numbered within the methods, and these classes ultimately determine the respective specificity of the two methods. The encounter method for Shark and Fish is more specific than the method for Shark and Animal, as Fish is a subclass of Animal. Under the new organization, the encounter methods no longer contain any information about dispatch (unless otherwise desired). The prototypes for Animal, Fish, and Shark, illustrated as circles with delegations as thick arrows, internalize the dispatch information as roles which reference the original methods, illustrated as rectangles within the objects containing a selector name and argument position and pointing to the methods they dispatch. This role information alone determines which method is dispatched, and no ordering is represented beyond the delegations between objects. For example, should an encounter be invoked upon the Shark prototype in the first argument position and the Fish prototype in the second argument position, dispatch will first traverse the delegation ordering of the Shark prototype and notice it supports two roles for an encounter in the first argument position, assigning both these roles an ordering position of 1, as Shark is the first object traversed. Dispatch recurses to the

Animal prototype, finding no roles for this argument position. Next, the second argument, the Fish prototype, is traversed and the single encounter role it contains for the second argument position is assigned an ordering position of 1. Dispatch again recurses to the Animal prototype, but this time finds a role for the second argument position and assigns it an ordering position of 2, as Animal is the second delegated-to object traversed in this argument position. The resulting specificity rank of the Shark and Animal encounter method is thus (1,2) and for the Shark and Fish encounter method is (1,1). Since the specificity rank (1,1) is less than (1, 2), the Shark and Fish encounter method is chosen as most-specific.

3.3 Conceptual Motivations

One may view object-oriented programming as the process of authoring a system of objects in the third-person. These objects correspond to agents within the system, and the programmer must describe their interactions and behavior in an almost vicarious fashion. The resulting program is a story about objects which, beyond merely being code that a computer ultimately executes, is meant to be read, revised, and even enjoyed. Just as there are various styles of third-person writing, there are various spheres of third-person design emphasized by object-oriented paradigms.

The message passing model of object-oriented programming utilized in single dispatch, prototype-based languages corresponds to the third-person limited perspective. It is conceptually simple and appealing on the grounds that the programmer describes the behavior of an object from a purely internal point of view. However, this model forces the programmer to explicitly detail all the mechanisms for identifying other objects that he is interacting with, and in a procedural manner. In effect, he must guess at the behavior of other objects than the one serving as his frame of reference in a roundabout fashion. In many cases, he must even take this model too far and begin describing to other objects how they should behave in a vicarious manner, destroying factoring of design. This view denies that any other objects besides a single frame of reference explicitly exist.

Multi-methods correspond to the third-person objective perspective. They rely upon external, objective descriptions of object behavior and how they interact with each other. This allows one to more easily describe how different objects interact with each other compared to the message passing model. However, this removes the ability of objects to internalize their behavior, and soon the programmer is overwhelmed with overly explicit and inflexible descriptions of the objects he must manipulate, almost as a pedant who must examine an encyclopedia after encountering any new or different object in his environment. This view simply ignores any objects that are not amenable to objective description.

The combination of prototypes and multiple dispatch corresponds to the third-person omniscient perspective. In this view, many objects each, each with their own internalized behavior and representation, that achieve some goal by consensus, or rather, all agreeing on something to do based on their internalized perspectives. The programmer benefits from both the internalized representation of objects and the ability to externally describe how these objects interact. He is neither burdened by overly explicitly descriptions of his objects, as he may define them individually and directly, nor limited to design his system from a single point of reference, as he may describe the terms of consensus among objects. This appeals to the simple intuitive notion of objects interacting with other objects, where each object is playing a specific role in an interaction, just as actors in a play. Instead of the objects possessing a vocabulary of messages they understand, the objects now possess various roles they may fulfill. Instead of objects requiring an external authority mandating their structure and behavior, they are free to evolve and differentiate. This also provides a natural model of subject-oriented programming if one considers the author or context of interaction itself merely another object contributing to the consensus.

Further, this interaction model appeals to paradigms outside of object-oriented programming, namely pattern-matching, and offers an alternative understanding of polymorphism. Given the prototype-object model, the type of the interaction (the method selector) may itself be considered an object that is interacting with other method arguments. Delegations further regulate the overall interaction by their presence or absence. The method selector, which itself may be considered an argument, and arguments together provide "horizontal" polymorphism reminiscent of simple pattern-matching upon tuples, while delegations provide "vertical" polymorphism that matches on nested structure in an unbounded (transitive) and order-insensitive (commutative) fashion. Dispatching thus becomes a form of pattern-matching on this overall structure of objects, instead of considering polymorphism as two distinct mechanisms. If one further extends role positions to not merely contain the argument positions at which the methods must be found, but instead be the actual paths in the object structure or some more flexible criteria for matching the paths, then one may encode arbitrary patterns both stronger and more general than either multiple dispatch or traditional pattern-matching that integrate with objects. However, such a model is beyond the scope of this paper.

3.4 Example

Figure 6 finally presents the running example in a language is both prototype-based and provides multi-methods. Notice that the PMD-inspired example removes all of the awkwardness of the solely class-based example while retaining all of the benefits of both the solely prototype-based example and the solely multi-methods example. The factoring is minimal, with no conditional behavior to the program whatsoever. The description is extremely concise and reads

```
object Animal;
object Fish = clone (Animal);
object Shark = clone (Animal);
object HealthyShark;
object DyingShark;
addDelegation (Shark, HealthyShark);
method swimAway (animal : Animal);
method encounter (animal : Fish, other : HealthyShark)
{
    swimAway (animal);
}
method encounter (animal : Fish, other : Animal)
{
}
method swallow (animal : Shark, other : Animal);
method fight (animal : HealthyShark, other : Shark)
{
    removeDelegation (animal, HealthyShark);
    addDelegation (animal, DyingShark);
}
method encounter (animal : DyingShark, other : Animal)
{
    swimAway (animal);
}
method encounter (animal : HealthyShark, other : Fish)
{
    swallow (animal, other);
}
method encounter (animal : HealthyShark, other : Shark)
{
    fight (animal, other)
}
```

Figure 6: PMD-based example

on a simple case-by-case basis, with the behavior of each case specified individually. It is further trivially extensible in ways all of the previous running examples lacked, both by adding new objects to represent new animals or animal states and by adding methods to handle these new objects. The PMD version of the running example is not only pleasingly simple, but there are few ways (if there are any) it can be simplified further.

4 A Formal Model of Prototypes with Multiple Dispatch

The following formal model of prototypes with multiple dispatch, henceforth abbreviated as PMD (Prototypes with Multiple Dispatch), captures the essence of the system without reference to superfluous details of its incarnation. While the model retains similarity to a practical programming language, notable features are omitted, in the style of the lambda calculus, such as object fields and assignment, non-local returns, and syntactical conveniences, in so far as they may be framed in terms of or as simple extensions of the model presented and are not intrinsic to the model or its presentation. Despite these omissions, the model allows for reasoning about most relevant details of PMD.

4.1 Syntax

$$l, s, d \in locations$$

$$\begin{array}{lcl}
e & ::= & \lambda\overline{x}.e \mid e(\overline{e}) \mid x \mid l \\
v & ::= & l \\
r & ::= & \texttt{<}s,\ i,\ l\texttt{>} \\
O & ::= & \texttt{<}\langle\overline{d}\rangle,\ \{\overline{r}\},\ e\texttt{>} \\
S & ::= & l \longmapsto O \\
C & ::= & l
\end{array}$$

Figure 7: Syntax

Figure 7 shows the simplified syntax of PMD. The metavariable x ranges over variable names, e ranges over expressions, i ranges over valid method parameter indexes, r ranges over roles, S ranges over stores, l ranges over locations in the store (s and d serving as aliases for readability), v ranges over values, and C ranges over store locations identifying subjective contexts. The notation $\overline{z}$ denotes a sequence of the object z.

PMD assumes only a small number of syntactic constructs representing methods, which are objects holding an expression to be evaluated under certain method

parameters; method invocations, which identify a method to be invoked as well as a set of expressions whose values are to be passed to the method parameters; and method parameter names, which are placeholders for the store locations serving as values, as passed by a method invocation. The syntax emphasize similarities with the lambda calculus where applicable.

As PMD relies on object identity, the model further assumes a store mapping a store location, used to represent object identity, to an object's store representation which consists first of a sequence of locations denoting the objects the particular object delegates to, a set of roles identifying the methods defined upon the particular object, and an expression which, ostensibly, appeals to the syntactic function of an object without enforcing any particular details of its implementation. The notation $S[l]$ will be used to denote object representation corresponding to the location l in the store S, and the notation $S[l \longmapsto O]$ will be used to denote the store S adjusted to map the location l to the object representation O. Little is assumed about the initial store (which need not be unique), denoted $\bullet$, other than that it contains at least a location for some object serving as the initial subjective context, and any store locations that may be referenced as literal objects.

4.2 Dynamic Semantics

$$\frac{S,C \vdash e_{\mathrm{S}} \hookrightarrow e'_s, S', C'}{S,C \vdash e_s(\overline{e}) \hookrightarrow e'_s(\overline{e}), S', C'} R - Selector$$

$$\frac{S,C \vdash e_i \hookrightarrow e'_i, S', C'}{S,C \vdash v_s(v_0 \cdots v_{i-1}, e_i, e_{i+1} \cdots e_n) \hookrightarrow v_s(v_0 \cdots v_{i-1}, e'_i, e_{i+1} \cdots e_n), S', C'} R - Argument$$

$$\frac{l \notin dom(S) \quad S' = S[l \longmapsto < \langle\rangle, \{\}, \lambda\overline{x}.e >]}{S,C \vdash \lambda\overline{x}.e \hookrightarrow l, S', C} R - Method$$

$$\frac{lookup(S, C, v_s, \overline{v}) = l \quad S[l] = < \langle\overline{d}\rangle, \{\overline{r}\}, \lambda\overline{x}.e >}{S,C \vdash v_s(\overline{v}) \hookrightarrow [\overline{v}/\overline{x}]e, S', C'} R - Invoke$$

Figure 8: Dynamics semantics

Figure 8 presents the core dynamic semantics of PMD as a set of reductions rules of the form $S, C \vdash e \hookrightarrow e', S', C'$, to be read as "with respect to a store S and subjective context C, the expression e one-step reduces to e', yielding a new store S' and a new subjective context C'". $\hookrightarrow^*$is to be read as the reflective, transitive closure of $\hookrightarrow$. As PMD aims to be a simple, uniform object model, these rules only provide a subset of the necessary behavior for a practical language, but

suffice to encompass the essential core of evaluation. Further auxiliary rules will be described in a later section, which are to be taken as an optional set of primitives fleshing out a suggested object model to accompany PMD.

The rule $R-Method$ adds a new location to the store with an object bearing the particular method representation as its primitive behavior. The result of the reduction is the location of this object.

The rule $R-Selector$ ensures that a method selector expression evaluates before any of the arguments to a method invocation. The rule $R-Argument$ ensures that all arguments to the method invocation evaluate in a left-to-right order, and after the evaluation of the method selector.

The rule $R-Invoke$ looks up the location of a particular method, with respect to a method selector and a sequence of method arguments, given by the *lookup* function. Given an object implementing the method representation corresponding to the location, the reduction results in the method's body expression being evaluated with the method arguments substituted for the method parameters. The *lookup* method, detailing the actual multiple dispatch upon the method arguments, will be defined in a subsequent section.

4.3 Dispatch Semantics

$$\frac{compose(C,\overline{v}) = \langle\overline{v'}\rangle \quad l \in applicable(S,s,\overline{v'}) \quad \forall_{l' \in applicable(S,s,\overline{v})} \left(l = l' \vee rank(S,l,s,\overline{v'}) \prec rank(S,l',s,\overline{v'})\right)}{lookup(S,C,s,\overline{v})=l} R-Least$$

$$\frac{\forall_{0\leq i\leq n}\left(order(S,v_i)=\langle d_0,\cdots,d_m\rangle \wedge \exists_{0\leq\alpha\leq m}\left(S[d_\alpha]=<\left\langle\overline{d'}\right\rangle,\{\overline{r}\},e> \wedge <s,i,l> \in \{\overline{r}\}\right)\right)}{l \in applicable(S,s,v_0,\cdots,v_n)} R-Applicable$$

$$\frac{l \in applicable(S,s,v_0,\cdots,v_n)}{rank(S,l,s,v_0,\cdots,v_n)=\prod_{0\leq i\leq n} min\left\{0\leq k\leq m \mid order(S,v_i)=\langle d_0,\cdots,d_m\rangle \wedge S[d_k]=<\left\langle\overline{d'}\right\rangle,\{\overline{r}\},e> \wedge <s,i,l> \in \{\overline{r}\}\right\}} R-Rank$$

Figure 9: Dispatch semantics

The dispatch semantics presented in Figure 9, given an ordering of method ranks and a delegation ordering, find the least method applicable to all the method arguments within a particular subjective context. These semantics are intended to be intuitively similar to those of multiple dispatch via subclassing while providing for all the expressiveness of PMD. The functions *compose*, order,

$\prec$, and $\prod$ serve to parameterize these dispatch semantics and may be defined as desired so long as they adhere to the necessary semantics as described in the rules in which they are used.

The rule $R - Lookup$ describes *lookup* function. *compose* is assumed to be a function that, given a subjective context and a sequence of method arguments, will yield a new sequence of methods arguments considering the context. The simplest interpretation of *compose* is that it adds the context to the beginning of the sequence of method arguments. Given the new method arguments, the least member of the set of applicable methods is found such that the rank of this method is less than the rank of all other applicable methods besides itself. The asymmetric, transitive relation $\prec$ provides a total ordering of the ranks of applicable methods. If such a least applicable method exists, it is the result of the *lookup* function.

The rule $R - Applicable$ describes the *applicable* function, which yields the set of applicable methods with respect to some method selector and sequence of method arguments. *order* is assumed to be a function that, given a particular method argument, will return an ordered list of all particular objects in the linear ordering imposed by the delegation relation bounded by that particular method argument. Conceptually, *order* yields all objects reachable from the method argument (including the method argument itself) by traversal of the delegation links. A method is then applicable if for every method argument, there exists some object among those in the ordering bounded by that particular method argument which contains a role bearing that method and matching the method selector and the method argument's position.

Finally, the rule $R - Rank$ describes the *rank* function. This rule again relies on the existence of the function *order* that provides a linear order of objects on which to determine the rank. $\prod$ is assumed to be an operator that composes the indexes in the ordering for each argument position into an n-dimensional rank. So, with respect to a particular method selector and sequence of method arguments, the rank of a method is then the composition (as by $\prod$) of the minimal indexes in the ordering of an object that contains a role bearing that method and matching the method selector and method argument's position. In light of this rule, the preceding rule, $R - Applicable$, merely determines if a rank actually exists for a given method.

4.3.1 Suggested Dispatch Parameters

Figure 10 provides suggestions for the parameters *compose*, *order*, $\prod$, and $\prec$ of the dispatch semantics. The intended effect of these suggested parameters is to configure PMD for interleaving subjective contexts, a depth-first ordering of the delegation relation, and left-to-right lexicographic ordering of method arguments.

$$
\begin{array}{l}
compose(C, \overline{l}) = \langle \overline{l}, C \rangle \\
dfs(S, \langle l_0, \cdots, l_n \rangle, \langle \overline{l'} \rangle) = \\
\begin{cases}
\langle \overline{l'} \rangle & if\ n \ngtr 0 \\
dfs(S, \langle l_0, \cdots, l_{n-1}, \overline{d} \rangle, \langle \overline{l'}, l_n \rangle) & if\ l_n \notin \{\overline{l'}\} \wedge S[l_n] = < \langle \overline{d} \rangle, \{\overline{r}\}, e > \\
dfs(S, \langle l_0, \cdots, l_{n-1} \rangle, \langle \overline{l'} \rangle) & otherwise
\end{cases} \\
bfs(S, \langle l_0, \cdots, l_n \rangle, \langle \overline{l'} \rangle) = \\
\begin{cases}
\langle \overline{l'} \rangle & if\ n \ngtr 0 \\
bfs(S, \langle \overline{d}, l_0, \cdots, l_{n-1} \rangle, \langle \overline{l'}, l_n \rangle) & if\ l_n \notin \{\overline{l'}\} \wedge S[l_n] = < \langle \overline{d} \rangle, \{\overline{r}\}, e > \\
bfs(S, \langle l_0, \cdots, l_{n-1} \rangle, \langle \overline{l'} \rangle) & otherwise
\end{cases} \\
order(S, l) = dfs(S, \langle l \rangle, \langle \rangle) \\
\prod_{0 \leq i \leq n} k_i = \langle k_0, \cdots, k_n \rangle \\
\langle p_0, \cdots, p_n \rangle \prec \langle k_0, \cdots, k_n \rangle = \exists_{0 \leq i \leq n} (p_i < k_i \wedge \forall_{0 \leq j < i} p_j = k_j)
\end{array}
$$

Figure 10: Dispatch parameters

The provided *compose* places the subjective context in the least significant method argument position. Should the original method arguments not be sufficient for ordering the ranks, the subjective context is then consulted as a tie-breaker. This behavior is unobtrusive in that methods defined in a more specific context will integrate with existing methods defined in inherited contexts without any special care. An interesting alternative model, however, is to place the context in the most-significant argument position so that the context is consulted before any other method arguments to determine method rank. This alternative model provides a layering of contexts, wherein methods will not integrate with methods defined in inherited contexts, but will instead entirely override them.

The provided *order* constructs the ordering of objects by a depth-first search of the delegation links using the function dfs. A stack is maintained for objects that have yet to be searched, as well as the currently constructed portion of the ordering. At each step, the currently visited object is added to the order, and the ordered list of delegation links is added to the top of the stack, such that the last delegation link is traversed first on the next step. Objects that have already been visited are simply skipped so that they only appear in the ordering once in the earliest position they were found. This depth-first ordering provides a simple conceptual, a layering of inherited method definitions wherein later added delegation links always sequence their methods before those in earlier delegation links, as well as mapping to an efficient implementation. An alternative model, breadth-first search, is illustrated by the function bfs. The only significant difference between this model and the depth-first ordering is that delegation links are added to the bottom of the stack, effectively transforming it into a queue, such that the breadth of the delegation links are effectively scanned first before recursing lower. However, this model carries with it significant

conceptual overhead in that inherited methods will integrate in non-trivial ways in the presence of multiple delegation and force the programmer to carry a greater understanding of all involved delegation hierarchies when using objects. If PMD is limited to only single delegation, either of these models are trivially equivalent.

The provided $\prod$ and $\prec$ implement a left-to-right lexicographic ordering of method arguments, wherein $\prod$ merely composes the ordering indexes into an n-dimensional rank vector. $\prec$ then orders these rank vectors in the obvious way. This particular implementation maps efficiently to bit vectors and normal integer comparisons, especially if appropriate limits are placed on the maximal number of method arguments and size of the ordering. A trivial alternative to this is a right-to-left lexicographic ordering, while a far more interesting alternative is a partial ordering. For a partial ordering, $\prec$ is modified so that at least one position in a rank vector must be less than the respective position in another, but all other positions must be less than or equal, rather than only all preceding positions. This model is far stricter than a lexicographic ordering, and may prevent a certain amount of errors caused by unintended combinations of inherited methods. However, using a partial order complicates composition of subjective context, such that it will no longer work as illustrated unless the $\prec$ operator is extended beyond a simple partial ordering to support it such that contexts are still lexicographically ordered. If adopting a pure partial order, and one method is more specific than another except with respect to context for which the inverse might be true, then these methods are no longer ordered and dispatching them is in error despite any preferences for context.

4.4 Auxiliary Semantics

Figure 11 presents auxiliary semantics which ostensibly describe the primitive behavior of operations desirable for an expressive language based on the core PMD semantics, but do not, however, prescribe the exact behavior of these operations, nor are these an exhaustive nor uniquely distinguished set of such operations. It is taken for granted that the arguments to the method invocations have already been reduced to simplify presentation. It is assumed that *clone*, *addDelegation*, *removeDelegation*, *addMethod*, *removeMethod*, and *changeSubject* name method selectors present in the initial store $\bullet$, and that $\bullet$ also contains a method object for each particular operation and named by the respective method selectors. Further, all objects in $\bullet$ must have roles defined on them or delegate to an object such that they all serve as applicable arguments to these operations, and there is a smallest such object in the store containing these roles, the store location of which is known.

The *clone* operation, given the location of an object already in the store, will produce a new location that maps to an equivalent object representation. This is intended to be the primary mode of instantiating new objects.

$$\frac{\begin{array}{c} S[v] = O \quad l \notin dom(S) \\ S' = S[l \longmapsto O] \end{array}}{S, C \vdash clone(v) \hookrightarrow l, S', C} A - Clone$$

$$\frac{\begin{array}{c} S[v] = < \langle \overline{d} \rangle, \{\overline{r}\}, e > \\ S' = S\left[v \longmapsto < \langle \overline{d}, v' \rangle, \{\overline{r}\}, e >\right] \end{array}}{S, C \vdash addDelegation(v, v') \hookrightarrow v, S', C} A - AddDelegation$$

$$\frac{\begin{array}{c} S[v] = < \langle d_0, \cdots, d_n \rangle, \{\overline{r}\}, e > \quad n > 0 \\ S' = S[v \longmapsto < \langle d_0, \cdots, d_{n-1} \rangle, \{\overline{r}\}, e >] \end{array}}{S, C \vdash removeDelegation(v) \hookrightarrow d_n, S', C} A - RemoveDelegation$$

$$\frac{\begin{array}{c} compose(C, \overline{v}) = \langle v'_0, \cdots, v'_n \rangle \\ \forall_{0 \leq i \leq n} \left(S_i[v'_i] = < \langle \overline{d} \rangle, \{\overline{r}\}, e > \wedge S_{i+1} = S_i\left[v'_i \longmapsto < \langle \overline{d} \rangle, \{\overline{r}, < v_s, i, v_m >\}, e >\right]\right) \end{array}}{S_0, C \vdash addMethod(v_m, v_s, \overline{v}) \hookrightarrow v_m, S_{n+1}, C} A - AddMethod$$

$$\frac{\begin{array}{c} compose(C, \overline{v}) = \langle v'_0, \cdots, v'_n \rangle \\ \forall_{0 \leq i \leq n} \left(S_i[v'_i] = < \langle \overline{d} \rangle, \{\overline{r}\}, e > \wedge S_{i+1} = S_i\left[v'_i \longmapsto < \langle \overline{d} \rangle, \{\overline{r}\} - \{< v_s, i, v_m >\}, e >\right]\right) \end{array}}{S_0, C \vdash removeMethod(v_m, v_s, \overline{v}) \hookrightarrow v_m, S_{n+1}, C} A - RemoveMethod$$

$$\frac{}{S, C \vdash changeSubject(v) \hookrightarrow C, S, v} A - ChangeSubject$$

Figure 11: Auxiliary semantics

The *addDelegation* operation adds a new delegation to the tuple of delegations for a particular object. The new delegation is assumed to be the last element in the resulting tuple of delegations. The *removeDelegation* operation simply undoes the effect of an *addDelegation* operation, and taken together they may be used to implement positional modification of the tuple of delegations for a particular objects.

The *addMethod* operation will define roles upon the sequence of supplied objects such that the objects or any subsequent clones will respond to a method lookup for the given method selector with the given method object. The *removeMethod* operation, given those same objects, will remove the roles from the objects.

Finally, *changeSubject* is used to replace the current subjective context with a new object, which will subsequently effect all method lookups thereafter.

4.4.1 Example

$$
\begin{array}{l}
addDelegation(Root, Root) \\
addMethod(\lambda mv.do(addMethod(m, do, m, Root), m, v), apply, Root, Root) \\
addMethod(\lambda xy.y, seq, Root, Root) \\
addMethod(\lambda ogsv. \\
\qquad\qquad apply(\lambda fm. \\
\qquad\qquad\qquad addMethod(\lambda ov. \\
\qquad\qquad\qquad\qquad\qquad seq(removeMethod(m, g, o), addMethod(\lambda o.v, g, o)), \\
\qquad\qquad\qquad\qquad s, o, Root) \\
\qquad\qquad\quad addMethod(\lambda o.v, g, o)), \\
\qquad addSlot, Root, Root, Root, Root) \\
addSlot(Lobby, Boolean, Boolean :, clone(Root)) \\
addDelegation(Boolean(Lobby), Boolean(Lobby)) \\
addSlot(Lobby, True, True :, clone(Boolean(Lobby))) \\
addSlot(Lobby, False, False :, clone(Boolean(Lobby))) \\
addMethod(\lambda btf.apply(t, b), ifThenElse, True(Lobby), Root, Root) \\
addMethod(\lambda btf.apply(f, b), ifThenElse, False(Lobby), Root, Root) \\
addMethod(\lambda xy.False(Lobby), = \\
, Boolean(Lobby), Boolean(Lobby)) \\
addMethod(\lambda xy.True(Lobby), =, True(Lobby), True(Lobby)) \\
addMethod(\lambda xy.True(Lobby), =, False(Lobby), False(Lobby))
\end{array}
$$

Figure 12: Example

As a pathological but expedient example of PMD in action, Figure 12 illustrates the implementation of comparable boolean objects with a conditional control flow structure based on the auxiliary semantics presented above. The example

supposes the existence of an object *Root* which has defined on it all the methods described in the auxiliary semantics and to which all objects delegate to, an object *Lobby* which is a clone of *Root* and serves as a namespace, and distinct named objects serving as selectors sufficient to cover all uses of selectors in this example.

Firstly, it declares some essential utilities for later use in the example that illustrate some of the expressive power of PMD. It ensures *Root* delegates to itself so that all objects cloned from it will also properly delegate to it and support any new methods defined on it. Next, it illustrates a higher-order use of $addMethod$ by defining a method *apply*, which defines the method *do* on the supplied method m to invoke itself (on itself and its arguments), to allow for the binding of temporary names and simple application of closures. The method *seq* is defined, which merely returns the value of its last argument, so as to allow a sequence of expressions to appear in a method body. Finally, it defines $addSlot$ which implements assignable slots on objects as an accessor method with selector g that returns the current value of the slot, and a mutator method with selector s which, when supplied with a new value, will remove the old accessor from the object and install a new accessor to return the new value.

The next portion of the example illustrates the creation of useful boolean objects using the new utilities. Firstly, a slot with accessor $Boolean$ and mutator $Boolean:$ is created in the *Lobby* and initialized to hold a fresh clone of *Root*. The object, accessed by invoking the $Boolean$ accessor method on the *Lobby*, is set to delegate to itself so that new clones will delegate to it, as in the first part of example. Next, two new boolean objects, $True$ and $False$, are similarly defined as clones of $Boolean$. A method $ifThenElse$, serving as a useful control structure, is defined on both so that it will select one of two closures to apply depending on which boolean object it was invoked upon. $True$ applies the first supplied closure, while $False$ applies the second supplied closure. For example, $ifThenElse(True(Lobby), \lambda fb.Root, \lambda fb.Lobby)$ if invoked would evaluate to $Root$.

The final portion of the example demonstrates the use of inherited methods. A method $=$ is defined on $Boolean$ so that if no other methods apply, it will by default return $False$. Specialized versions of $=$ are defined on each of the two boolean objects such that if both of the arguments are the same, they will return $True$. For example, invoking $=(True(Lobby), False(Lobby))$ would use the version defined for $Boolean$ and so return $False$, whereas invoking $=(False(Lobby), False(Lobby))$ would use the version defined for $False$ and so return $True$.

Figure 13 presents the final running example in the PMD calculus. It still retains all of the conciseness and descriptiveness as the original PMD-inspired example and differs little from it, despite being framed in terms of the more

$$
\begin{array}{l}
addSlot(Lobby, Animal, Animal:, clone(Root)) \\
addSlot(Lobby, Fish, Fish:, clone(Animal(Lobby))) \\
addSlot(Lobby, Shark, Shark:, clone(Animal(Lobby))) \\
addSlot(Lobby, HealthyShark, HealthyShark:, clone(Root)) \\
addSlot(Lobby, DyingShark, DyingShark:, clone(Root)) \\
addDelegation(Shark(Lobby), HealthyShark(Lobby)) \\
addMethod(\lambda xy.swimAway(x), encounter, Fish(Lobby), HealthyShark(Lobby)) \\
addMethod(\lambda xy.x, encounter, Fish(Lobby), Animal(Lobby)) \\
addMethod(\lambda xy. \\
\qquad\qquad seq(removeDelegation(x), addDelegation(x, DyingShark(Lobby))), \\
\qquad fight, HealthyShark(Lobby), Shark(Lobby)) \\
addMethod(\lambda xy.swallow(x, y), encounter, HealthyShark(Lobby), Fish(Lobby)) \\
addMethod(\lambda xy.fight(x, y), encounter, HealthyShark(Lobby), Shark(Lobby))
\end{array}
$$

Figure 13: Formal PMD example

crude calculus. The PMD semantics sufficiently captures the mechanisms that lead to the minimal factoring of the running example.

5 Implementation

5.1 Dispatch in Slate

The formalization presented in the previous section leaves open a number of practical considerations about how to implement the core dispatch algorithm of PMD. These issues include determining the proper order of delegations, the candidate set of methods that may be applicable, and finally, the actual rank of this set of methods and how to represent it. Various optimizations also expediently reduce the memory and processing requirements of the algorithm.

The programming language Slate [Rice and Salzman, 2004] serves as a canonical implementation of PMD and utilizes a dispatch algorithm for geared toward a lexicographic ordering of methods and a number of optimizations, including efficient encoding of rank vectors, sparse representation of roles, partial dispatch, and method caching. Slate's dispatch algorithm shall guide and motivate subsequent implementation discussion.

Figure 14 outlines in pseudo-code a basic version of the dispatch algorithm. The comparison operator $\prec$is as in the formalism and may be chosen to implement either a partial or lexicographic ordering as desired. The order in which delegations from a given object are pushed onto and popped from the ordering stack determines the ordering under multiple and non-trivial delegation and should be chosen as is applicable to the implementation. If one overlooks the necessary

```
dispatch(selector, args, n)
{
  for each index below n
  {
    position := 0
    push args[index] on ordering stack
    while ordering stack is not empty
    {
      arg := pop ordering stack
      for each role on arg with selector and index
      {
        rank[role's method][index] := position
        if rank[role's method] is fully specified
        {
          if no most specific method
             or rank[role's method] ≺ rank[most specific method]
          {
            most specific method := role's method
          }
        }
      }
      for each delegation on arg
      {
        push delegation on ordering stack
      }
      position := position + 1
    }
  }
  return most specific method
}
```

Figure 14: Basic dispatch algorithm

bookkeeping for rank vectors, this algorithm strikingly resembles the message lookup algorithm utilized by Self.

The process for constructing a depth-first ordering of delegations is straightforward. One maintains a stack of visited but not yet ordered objects from which elements of the ordering are drawn. If the host language allows cyclic delegation links, one also need maintain a set of objects already visited, easily represented by marking the objects directly, to avoid traversing the same delegation twice. If one further assumes object structure is represented by maps, as in Self [Chambers et al., 1991], or classes, this visited set may be stored on a per-map or per-class basis without loss. The stack is then processed by popping objects off the top, assigning them the next position in the ordering, and then pushing all their delegations onto the stack unless they were already visited.

Role information is stored directly on the objects themselves (or their map or class) and each role identifies a potentially applicable method, or rather, a method that is supported by at least one of the arguments to the method invocation. One may conveniently collect all the candidate methods and their ranks while determining the delegation ordering, merely traversing an object's roles, for the given argument position and method selector, as it is popped off the ordering stack. An auxiliary table, which may be cheaply distributed among the methods themselves, stores the currently determined rank vector of the method, augmenting the method invocation argument's respective component of the rank vector with the current position in the delegation ordering. When a method's rank becomes fully determined, the method is noted as the most specific method (found so far) if it's rank is less than the previously found most specific method, or if it is the first such method found. Once the delegation stack has been fully processed for each method invocation argument, the resulting most specific method, if one exists, is a method whose rank is both minimal and fully specified at all argument positions.

5.2 Rank Vectors

One may represent rank vectors themselves efficiently as machine words, with a fixed number of bits assigned to each component up to some fixed number of components. If one assumes method arguments have lexicographical ordering, then simple integer comparisons suffice to compare ranks, where more significant components are placed in more significant bits of the integer represented in the machine word. However, if one assigns each component of the rank number a fixed number of representation bits and if the rank vectors themselves are fixed size, the maximum length of a delegation ordering that may be reflected in each component is also effectively fixed as well as the maximum number of method parameters. One need only provide a fall-back algorithm using arbitrary precision rank vectors in case the ordering stack is overflown or if an excessive number of arguments are present at a method invocation. Anecdotally, the majority of

methods contain small numbers of parameters and inheritance hierarchies (and similarly delegation hierarchies) are small, so this fall-back algorithm is rarely necessary. In 40,000 lines of code in the Slate standard library, only one pathological case was found where an ordering capacity of 16 or greater was necessary. An ordering capacity of 32 sufficed to handle all usable objects, without exception, and obviated the need for any fall-back algorithm entirely.

This dispatch procedure possesses a worst case algorithmic complexity of $O(n*e*r*v(e)*h(m)*c(n))$, where n is the number of arguments to a method invocation, e is the cumulative number of delegations in the store, m is the number of methods in the store, r is the number of roles in the store, $v(e)$ is the cost of inserting a delegation into the visited set and checking membership, $h(m)$ is the cost of mapping a method to its rank vector, and $c(n)$ is the cost of comparing rank vectors. If one maintains the visited set and rank mapping directly on the objects and methods, and represents the rank vector as a machine word, then $v(e)$, $h(m)$, and $c(n)$ become effectively constant. The practical complexity of dispatch, in the best case, thus becomes $O(n*e*r)$. e is usually quite small and constant as the algorithm only need traverse those delegation links reachable from each argument and, again anecdotally, delegation hierarchies are usually shallow. r is also small in practice, as the algorithm only need traverse those roles actually defined upon an object.

5.3 Sparse Representation of Roles

In Slate, the delegation hierarchy is rooted at one specific object so that certain methods may be defined upon all objects. However, since this object always assumes the bottom position in the delegation ordering, any roles defined upon it will always be found and always be the least specific such roles with respect to other roles with the same method selector and argument position. These roles do not aid in disambiguating the specificity of a given method since they occupy the bottom of the ordering and, in effect, contribute no value to the rank vector.

The majority of methods in the Slate standard library dispatch on the root object in most arguments positions, so representing these roles needlessly uses memory and adds traversal overhead to the dispatch algorithm. In the interests of reducing the amount of role information stored, one need not represent these roles if one identifies, for each method, the minimum set of roles that need be found for a rank vector to be fully specified and so allows the size of this set of roles to be less than the number of actual method parameters. This set of roles does not contain any roles specified on the root object. A method is now applicable when this minimum set of roles is found during dispatch, rather than a set of roles corresponding to all method parameters. In the interests of reducing duplication of information, Slate stores information about the size of this minimum set of roles on the method object linked by these roles.

5.4 Partial Dispatch

Because of Slate's sparse representation of roles, the dispatch algorithm may determine a method to be applicable, or rather, it's minimal set of roles may be found, before it has finished traversing the delegation orderings of all argument positions. The basic algorithm, however, requires that the entire delegation ordering of all arguments be scanned to fully disambiguate a method's specificity and ensure it is the most specific. The majority of methods in the Slate standard library not only dispatch on fewer non-root objects than the number of method parameters, but only dispatch on a single non-root object, and are, in effect, only singly polymorphic. Scanning the entire delegation orderings for all objects under such conditions is wasteful and needless if an applicable method is unambiguously known to be the most-specific method and yet dispatch still continues.

The key to remedying this situation is to take advantage of Slate's lexicographic ordering of method arguments and also note that a role not only helps identify an applicable method, but a role also indicates that some method is possibly applicable in the absence of information about which other roles have been found for this method. If no roles corresponding to a method are found, then the method is not applicable. If at least at least one role corresponding to a method is found, then this method may become applicable later in the dispatch and effect the result should its determined rank vector precede the rank vectors of other applicable methods.

Dispatch in Slate traverses method arguments from the lexicographically most significant argument to the least significant argument. So, for any role found, it's contribution to the rank vector will necessarily decrease with each successive argument position traversed. If some method is known to be the most specific applicable method found so far, and a role for a contending method is found whose contribution to its respective rank vector would still leave it less specific than the most specific method, then no subsequent roles found for the contending method will change the method result as they contribute lexicographically less significant values. Thus, one only need maintain the partial rank vector, representing the contention for most specific method, corresponding to the lexicographically most significant roles found up to the current point of traversal. If any applicable method's rank vector precedes this partial rank vector, then it is unambiguously the most specific method, since there are no other more specific methods that may later become applicable.

For example, if one method singly dispatches on the Shark prototype, and another similarly named method dispatches on the Animal prototype in a lexicographically less significant or equally significant argument position, then dispatch will determine the Shark prototype's method to be applicable as soon as the Shark prototype is traversed and before traversing the Animal prototype. If no other roles were found at lexicographically more significant positions, or

on preceding objects in the delegation ordering for the lexicographically equal argument position, then there is no possible contention for the resulting most specific method, and the Shark prototype's method must be the most specific.

Intriguingly, this optimization reduces the cost of dispatch to the amount of polymorphism represented in the entire set of candidate methods. So, if all methods only dispatch on their first argument, the dispatch algorithm effectively degenerates to a traditional single dispatch algorithm and need never examine more than the first argument or traverse farther down the delegation hierarchy than where the first candidate method is found. The algorithm then only incurs the cost of maintaining the rank information above the cost of single dispatching. Single dispatching becomes nothing more than a trivial and more general optimization of the PMD dispatch semantics. Further, almost all the dispatches in the Slate standard library (approximately 90%) were found to terminate early due to this optimization, rather than requiring a full traversal. This number closely corresponds to the fraction of methods dispatching on fewer non-root objects than their number of arguments, which supports this intuition.

5.5 Method Caching

Various global and inline method caching schemes may be extended to fit the dispatching algorithm and provide an essentially constant time fast-path for method invocation under PMD. Given partial dispatching and if for each method selector one identifies the global polymorphism of the set of methods it identifies (the set of argument positions any roles have been specified in), one only need store the significant arguments positions, as given by the global polymorphism, as the keys of the cache entries. However, cache entries must still have a capacity to store up to the maximally allowable amount of polymorphism for caching. In the degenerate case of global polymorphism of only the first argument, this extended caching scheme degenerates to an ordinary single dispatch caching scheme.

In the Slate standard library, a simple global cache indexed only by method selector and validated against the maps of the method arguments as by this scheme achieves a cache hit rate of approximately 80% and yields significant speed-ups above and beyond the contributions of partial dispatch, by avoiding any delegation or role traversal altogether.

6 Conclusions

PMD provides a coherent unifying approach to three disparate paradigms: prototype-based languages, multi-method languages, and subject-oriented languages. It offers insights into these disparate mechanisms and why they address orthogonal aspects of object polymorphism. Its implementation in Slate further draws new explicit parallels with existing techniques for implementing single

dispatch and why single dispatch algorithms are degenerate cases of a more general multiple dispatch algorithms.

PMD offers programmers new ways to reason about and construct their programs that leverage the extreme polymorphism afforded by the paradigm. The programmer need no longer choose between differing conceptions of object-oriented programming. He may utilize the unified concept to its fullest extent without the distraction of integrating conflicting approaches. He is no longer limited by the lesser of evils.

References

[Abadi and Cardelli, 1996] Abadi, Martin, and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Chambers, 92] Chambers, Craig. Object-Oriented Multi-Methods in Cecil. *ECOOP '92 Conference Proceedings*, Utrecht, the Netherlands, July, 1992.

[Chambers et al., 1991] Chambers, Craig, Elgin Lee, and David Ungar. An Efficient Implementation of SELF, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Lisp and Symbolic Computation*, 4(3), June, 1991.

[Dahl et al., 1970] Dahl, O. J., B. Myhrhaug, and K. Nygaard. *The SIMULA 67 common base language*. Technical report, Norwegian Computing Center, Oslo, Norway, 1970.

[Goldberg and Robson, 1989] Goldberg, Adele, and David Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Rice and Salzman, 2004] Rice, Brian, and Lee Salzman. *The Slate Programming Language*. April, 2004. http://slate.tunes.org/.

[Steele, 1990] Steele, Guy L. Jr. *Common Lisp: The Language, Second Edition*. Digital Press, 1990.

[Ungar and Smith, 1991] Ungar, David, and Randall B. Smith. SELF: The Power of Simplicity. *Lisp and Symbolic Computation*, 4(3), June, 1991.

[Ungar and Smith, 1996] Ungar, David, and Randall B. Smith. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2(3):161-178, 1996.