Prototypes with Multiple Dispatch: An Expressive and Dynamic Object Model

Lee Salzman and Jonathan Aldrich

Carnegie Mellon University, Pittsburgh, PA 15217, USA,
lsalzman@alumni.cmu.edu, jonathan.aldrich@cs.cmu.edu

Abstract. Two object-oriented programming language paradigms—dynamic, prototype-based languages and multi-method languages—provide orthogonal benefits to software engineers. These two paradigms appear to be in conflict, however, preventing engineers from realizing the benefits of both technologies in one system. This paper introduces a novel object model, prototypes with multiple dispatch (PMD), which seamlessly unifies these two approaches. We give formal semantics for PMD, and discuss implementation and experience with PMD in the dynamically typed programming language Slate.

1 Overview

We begin the paper by describing a motivating example that shows the limitations of current, popular object-oriented languages for capturing how method behavior depends on the interaction between objects and their state. The example shows that multi-methods can cleanly capture how behavior depends on the interaction between objects, while dynamic, prototype-based languages can cleanly capture how behavior depends on object state. Unfortunately, unifying highly dynamic, prototype-based languages with multi-methods is hard, because traditional multi-methods assume a static class hierarchy that is not present in dynamic prototype-based languages.

In section 3 we describe Prototypes with Multiple Dispatch (PMD), an object model that combines the benefits of dynamic, prototype-based languages with multi-methods. PMD supports both paradigms by introducing a role concept that links a slot within an object to a dispatch position on a method, and defining a dynamic multi-method dispatch mechanism that traverses the graph of objects, methods, and roles to find the most specific method implementation for a given set of receiver objects.

Section 4 defines the PMD model more precisely using operational semantics. Section 5 demonstrates the expressiveness of PMD through the standard library of Slate, a dynamically-typed language that implements the PMD object model. Section 6 describes an efficient algorithm for implementing dispatch in Slate. Section 7 describes related work, and section 8 concludes.

2 Motivating Example

In this section, we use a simple running example to examine the benefits and limitations of two current trends in object-oriented programming: prototype-based languages and multi-method languages. Objects were originally invented for modeling and simulation purposes, and our example follows this tradition by modeling a simple ocean ecosystem.

```
class Animal {
   abstract method encounter (other : Animal);
   method swimAway (other : Animal) { ... }
}

class Fish inheriting Animal {
   method encounter (other : Animal) {
      if (other.isShark())
         if (other.isHealthy())
            swimAway();
   }
}

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

Fig. 1. A simple inheritance hierarchy modeling an ocean ecosystem. The encounter method illustrates behavior that depends both on an object's class (Shark or Fish) and its state (healthy or not). In conventional class-based languages, the behavior specification is complex, imperative, and hard to extend with additional classes.

Figure 1 presents the running example in a conventional class-based language like Java or C#. The inheritance hierarchy is made up of an abstract `Animal` superclass and two concrete subclasses, `Fish` and `Shark`. An animal's behavior is defined by the `encounter` method. Fish swim away from healthy sharks, but ignore other animals. If a shark is healthy, it will eat any fish it encounters and fight other sharks; if the shark is not healthy it will swim away from other animals. When a shark fights, it becomes unhealthy.

This example illustrates behavior that depends on both an object's class and its state, echoing many important real-world programming situations. For example, a fish's behavior depends on the type of animal that it encounters. A shark's behavior depends both on the type of animal it encounters and its current health.

In this example, object-oriented programming is beneficial in that it allows us to encapsulate a shark's behavior within the shark code and a fish's behavior within the fish's code. However, it also shows problems with current object-oriented languages. The specification of behavior is somewhat complex and hard to understand–even for this simple example–because the control structure within the `encounter` methods branches on many conditions. The program is also relatively hard to extend with new kinds of animals, because in addition to defining a new subclass of `Animal`, the programmer must add appropriate cases to the `encounter` methods in `Fish` and `Shark` to show how these animals behave when they encounter the new type of animal.

2.1 Multiple Dispatch

A language with multi-methods dispatches on the classes of all the argument objects to a method, rather than on just the class of the receiver. Multiple dispatch is useful for modeling functionality that depends on the type of multiple interacting objects.

Figure 2 shows the ocean ecosystem modeled using multi-methods. Instead of being written as part of each class, multi-methods are declared at the top level and explicitly include the first (or receiver) argument. Multi-methods dispatch on all argument positions, so that one of four `encounter` methods can be called, depending on whether the two animals are both sharks, both fish, or one of each in either order.

Typically, multiple dispatch is resolved by picking the most specific method that is applicable to all of the arguments, with a subtype relation among classes determining this specificity. For example, if a fish encounters a shark, at least two methods are applicable: the first method defined accepts a fish in the first position and any animal in the second position, but the second is more specific, accepting a fish in the first position but only sharks in the second position. In this case the second method would be invoked because it is more specific.

In cases where two methods are equally specific, languages differ. Languages like Cecil that use symmetric dispatch would signal a *message ambiguous* error [5], while languages like CLOS and Dylan would choose a method by giving the leftmost arguments greater priority whenever the specificities of two methods are compared [2, 8].

The example shows that multiple dispatch has a number of advantages over single dispatch. It is more declarative, concise, and easy to understand, because the control-flow branches within the `encounter` method have been replaced with declarative object-oriented dispatch. It is more extensible, because the system can be extended with new objects and new methods without changing existing

```
class Animal { }
   method swimAway (animal : Animal) { ... }

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

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

Fig. 2. Modeling the ocean ecosystem using multi-methods. Here, the `encounter` method dispatches on both the first and second arguments, simplifying the control structure within the methods and making the system more declarative and easier to extend.

objects and methods. These advantages are similar to the advantages that object-oriented programming brings relative to procedural programming.

However, there remain problems with the example, as expressed. It is still awkward to express stateful behavior; this is still represented by the control flow branches inside `encounter` methods. Furthermore, the code describing that unhealthy sharks swim away from all other animals is duplicated in two different `encounter` methods. This redundancy makes the program harder to understand, and creates the possibility that errors may be introduced if the duplicated code is evolved in inconsistent ways.

2.2 Prototype-Based Languages

Prototype-based languages, pioneered by the language Self [14], simplify the programming model of object-oriented languages by replacing classes with prototype objects. Instead of creating a class to represent a concept, the programmer creates an object that represents that concept. Whenever the program needs an instance of that concept, the prototype object is cloned to form a new object

that is identical in every way except its identity. Subsequent modifications to the clone diverge from the original and vice versa.

Prototype-based languages also emphasize the step-wise construction of objects over a static and complete description. Methods may be added to an individual object at any time, and in languages like Self, inheritance relationships may also be changed at any time. This emphasis on incremental construction occurs because objects are now self-sufficient entities that contain behavior as a genuine component of their state, rather than being instances of a class which merely describes their behavior for them.

```
object Animal;
object Fish;
object Shark;
object HealthyShark
object DyingShark

addDelegation (Fish, Animal);
addDelegation (Shark, Animal);
addDelegation (Shark, HealthyShark);

method Animal.swimAway () { ... }

method Fish.encounter(other) {
   if (other.isA(HealthyShark))
      swimAway();
}

method HealthyShark.swallow (other : Fish) { ... }
method HealthyShark.fight (other : Shark) {
   removeDelegation(HealthyShark);
   addDelegation(DyingShark);
}

method HealthyShark.encounter (other) {
   if (other.isFish())
      swallow (other)
   else if (other.isShark())
      fight (other)
}
method DyingShark.encounter (other) {
   swimAway();
}
```

Fig. 3. Modeling the ocean ecosystem using a prototype-based language. Here, the health of a shark is modeled by delegation to either the HealthyShark or the DyingShark. These abstractions represent behavior more cleanly and declaratively compared to the solutions described above.

Figure 3 shows how the ocean ecosystem can be expressed in a prototype-based language. The programmer first creates a prototype Animal object, then creates prototype Shark and Fish objects that delegate to the Animal.

The health of a Shark is represented by delegation to either a HealthyShark object or a DyingShark object. These objects encapsulate the behavior of the shark when it is healthy or dying, respectively. Sharks begin in the healthy

state, delegating to the HealthyShark object and thus inheriting its `encounter` method. When a HealthyShark fights, the current object's delegation is changed from HealthyShark to DyingShark, and from that point on the shark inherits the `encounter` method from DyingShark.

This example shows a strength of prototype-based languages: delegation can easily be used to represent the dynamic behavior of an object. The behavior can be changed dynamically when some event occurs simply by changing the object's delegation. Although we use dynamic inheritance as an illustration of the malleability provided by prototype-based languages, other features of these languages provide expressiveness benefits as well. For example, we could just as easily have redefined Shark's `encounter` method in the `fight` method to model changes in health.

Despite the advantages that prototypes bring, some problems remain. Like the original class-based code, the prototype-based implementation of the `encounter` methods branch explicitly on the type of the object being encountered. As discussed earlier, this makes the code more difficult to understand and harder to extend with new kinds of animals.

2.3 Discussion

The advantages of multiple dispatch and prototypes are clearly complementary. Multiple dispatch allows programmers to more declaratively describe behavior that depends on multiple interacting objects. Prototypes allow programmers to more cleanly describe stateful behavior, in addition to other benefits accrued by more malleable objects such as more accessible object representation, finer-grained method definition, and arbitrary object extension.

Because of these complementary advantages, it is natural to suggest combining the two models. Such a combination is difficult, however, because multiple dispatch depends on a predetermined hierarchy of classes, while prototypes generally allow a delegation hierarchy to change arbitrarily at any time.

Thus previous languages such as Cecil that combine these two models restrict delegation and method definition to be relatively fixed at a global scope that may be easily analyzed [5]. Unfortunately, restricting the manipulation of objects and methods, without compensating with additional mechanisms, also eliminates a key advantage of prototypes: the elevation of behavior to state. This fixed delegation hierarchy and method definition becomes reminiscent of classes which also, in general, emphasize this fixed inheritance and construction.

While other techniques for declaratively specifying the dependence of object behavior on state do exist, [6, 7, 11], they are more complex and restricted than dynamic inheritance and method update mechanisms in Self.

3 Prototypes with Multiple Dispatch

The contribution of this paper is describing how a dynamic, prototype-based object model in the style of Self can be reconciled with multiple dispatch. Our

```
object Animal;
object Fish;
object Shark;
object HealthyShark;
object DyingShark;

addDelegation (Fish, Animal);
addDelegation (Shark, Animal);
addDelegation (Shark, HealthyShark);

method swimAway (animal : Animal) { ... }

method encounter(animal : Fish, other : Animal) /* A */ { }
method encounter(animal : Fish, other : HealthyShark) /* B */ {
   swimAway();
}

method swallow (animal : Shark, other : Fish) { ... }
method fight (animal : HealthyShark, other : Shark) {
   removeDelegation(animal, HealthyShark);
   addDelegation(animal, DyingShark);
}

method encounter (animal : HealthyShark, other : Fish) /*C*/ {
   swallow (other)
}
method encounter (animal : HealthyShark, other : Shark) /*D*/ {
   fight (other)
}
method encounter (animal : DyingShark, other : Animal) /*E*/ {
   swimAway();
}
```

Fig. 4. Modeling the ocean ecosystem in Prototypes with Multiple Dispatch (PMD). PMD combines multiple dispatch with a dynamic, prototype-based object model, leading to a declarative treatment of both state and dispatch.

object model, Prototypes with Multiple Dispatch (PMD), combines the benefits of these two previous object models.

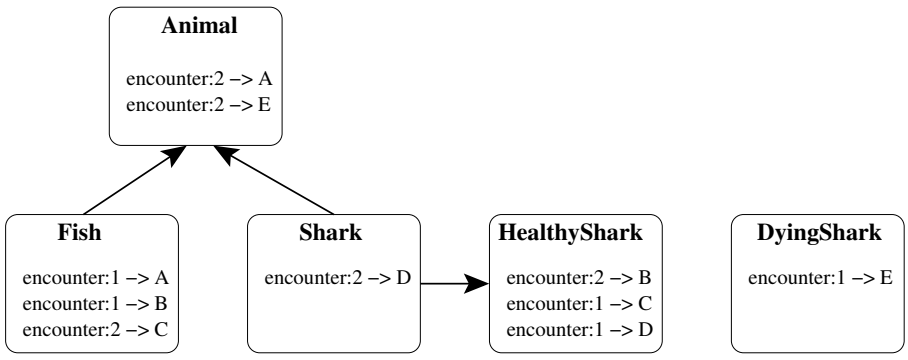

Fig. 5. A conceptual view of prototypes with multiple dispatch.

Figure 4 shows the programmer's view of PMD. The programmer creates an object structure that mirrors the prototype code given earlier. When defining methods, however, the programmer declares all arguments (including the receiver) explicitly, as in the multi-method code given earlier. Instead of giving the class that each argument dispatches on, a prototype object is given.

The code in Figure 4 combines the best of both prototypes and multiple dispatch. As in the prototype case, the behavioral dependence on the health of the shark is modeled as delegation to a HealthyShark or a DyingShark object. This delegation can be changed, for example, if the shark is injured in a fight. At the same time, behavioral dependence on multiple interacting objects is expressed through multiple method declarations, one for each relevant case. In a sense, the code is as clean and declarative as it could possibly be: no state variables or control-flow branches remain.

3.1 Dispatch Model

The key insight that makes PMD work is that multi-methods must be internalized into objects, rather than treated as external entities that dispatch across a fixed inheritance hierarchy. Retaining a largely extrinsic dispatch process, as in previous multi-method languages, inevitably restricts the capability of developers to manipulate the behavior of an object through dynamic inheritance or method update.

In Self, methods are internalized by storing them in slots of the receiver object. PMD cannot use this strategy, however, because a multi-method must operate on multiple objects; there is no distinguished receiver.

We solve this challenge by introducing the concept of the *role* played by a particular object in an interaction defined by a multi-method. Each multi-method defines a role for each of its argument positions. For example, in the last method of Figure 4, the `encounter` method's first role is played by a DyingShark object, while the second role is played by an Animal object.

Each object keeps track of which roles it plays for which multi-methods. Figure 5 shows the roles that different objects play in different `encounter` methods. Animal plays the second role in two different `encounter` method bodies: the ones marked A and E in the code above. Fish plays the first role in the first two methods (since their first parameter dispatches on Fish) and the second role in method C.

Dispatch occurs by searching the delegation hierarchy for inherited methods with the right name and appropriate roles for each of the arguments. For example, consider what happens when a fish encounters a shark that is healthy (i.e., still is delegating to the HealthyShark object). Fish can play the "encounterer" role (role #1) in both methods A and B. The shark can play the "encounteree" role (role #2) in methods A and E, inherited from Animal, method B, inherited from HealthyShark, and method D, defined in the Shark object itself. Only methods A and B will work for both roles. We choose the method to invoke by ordering the delegations for a given object, in case there are multiple such delegations.

A number of semantics are possible for determining the precedence of different applicable methods. The semantics we chose implements a total ordering of methods by considering (first) arguments farther to the left to take precedence over arguments to the right, (second) multiple delegations within a single object to be ordered according to the most recent time of definition, and (third) methods closer in the delegation hierarchy to the supplied method arguments to be more precise.

We chose a total order rather than a partial order, as in Cecil [5], to avoid the possibility of ambiguity in dispatch. A left-to-right ordering is standard, as is the criteria of closeness in the dispatch hierarchy. We chose to prioritize more recent delegations because this gives developers more flexibility to affect the behavior of objects by adding a new delegation. We prioritize first on the argument position, then on the ordering of delegations, then on the distance in the delegation hierarchy, because this gives us a convenient depth-first search algorithm to find the appropriate method (Section 6.1). This algorithm is both more efficient and easier for a programmer to understand than a breadth-first algorithm that would otherwise be required.[1]

4 Formal Model

This section provides a formal model of prototypes with multiple dispatch through a new object calculus. Our calculus borrows ideas from several previous object calculi, but the differences between PMD and previous object models are too great to use a straightforward extension of a previous calculus. For example, only one previous calculus that we know of supports imperative updates of methods [1]. However, this calculus, like most others [3], compiles away delegation by simply copying methods from the delegatee into the delegator. This strategy cannot possibly work in PMD because delegation can change after an object is created. Thus, the representation of objects in the calculus must maintain information about delegation to support this properly.

The most significant difference with all previous calculi is our modeling of multiple dispatch through roles on objects; this makes the common approach of modeling objects as records of methods inappropriate [9, 1, 3]. Although a few object calculi model multi-methods [12, 4], they all model multi-methods as external functions that dispatch over a fixed dispatch hierarchy, while PMD allows the developer to change the methods that are applicable to an object, as well as to modify the inheritance hierarchy at run time.

We instead sketch an untyped, imperative object calculus, PMD (Prototypes with Multiple Dispatch), that precisely describes the semantics of our proposed object model. The main contributions of the model are formalizing multi-method dispatch based on roles, and exposing the choices language designers have for

[1] If depth in the delegation hierarchy were considered first, for example, then simply adding an extra layer of delegation would affect dispatch, which seems extremely counterintuitive.

determining dispatching strategies. We hope that the calculus can be extended with a type system, but this remains challenging future work.

$l \in \mathit{locations}$	possible object identities in the store
$f ::= \lambda\overline{x}.e$	lambda expressions defining a method body
$e ::= x$	bindings
$\mid l$	locations that the store maps to objects
$\mid e_s(\overline{e})$	invokes method identified by selector e_s upon arguments $\overline{e}$
$\mid e_s(\overline{e}) \leftarrow f$	defining a method at selector e_s with body f, dispatching on $\overline{e}$
$\mid \mathit{clone}(e)$	copies an object
$\mid e \triangleright e_d$	updates e to delegate to e_d
$\mid e \not\triangleright$	removes the last delegation that was added to e
$v ::= l$	reduced values
$d ::= l$	delegations
$r ::= (l, i, f)$	roles contain a method selector, a method parameter index, and a method body
$O ::= (\langle\overline{d}\rangle, \{\overline{r}\})$	objects contain a list of delegations and set of roles
$S ::= l \mapsto O$	store mapping object identity to representation

Fig. 6. The syntax of PMD. The notation $\overline{z}$ denotes a syntactic sequence of z.

4.1 Syntax

Figure 6 explains syntax of PMD. This syntax provides lambda expressions for defining method bodies, object construction through ordered delegation and method definition, and roles that define the various connections between objects. As in Smalltalk [10], method selectors are themselves objects and can be computed.

As PMD is an imperative calculus, the model further assumes a store mapping a store location, used to represent object identity, to an object's representation. The object representation consists first of a sequence of locations denoting the objects the particular object delegates to and then a set of roles identifying the methods connected to the particular object.

The notation $S[l]$ will be used to denote object representation corresponding to the location l in the store S, and the notation $S[l \mapsto O]$ will be used to denote the store S adjusted to map the location l to the object representation O.

4.2 Example

Figure 7 presents the 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 lower-level calculus. The PMD semantics sufficiently captures the mechanisms that lead to the minimal factoring of the running example.

$$
\begin{array}{l}
Animal \stackrel{def}{=} clone(Root) \\
Fish \stackrel{def}{=} clone(Root) \\
Shark \stackrel{def}{=} clone(Root) \\
HealthyShark \stackrel{def}{=} clone(Root) \\
DyingShark \stackrel{def}{=} clone(Root) \\
Fish \triangleright Animal \\
Shark \triangleright Animal \\
Shark \triangleright HealthyShark \\
encounter(Fish, HealthyShark) \leftarrow \lambda xy.swimAway(x) \\
encounter(Fish, Animal) \leftarrow \lambda xy.x \\
fight(HealthyShark, Shark) \leftarrow \lambda xy.x \not\triangleright DyingShark \\
encounter(HealthyShark, Fish) \leftarrow \lambda xy.swallow(x, y) \\
encounter(HealthyShark, Shark) \leftarrow \lambda xy.fight(x, y)
\end{array}
$$

Fig. 7. The example scenario represented in the formal model.

The example assumes the existence of a distinguished empty object Root from which blank objects may be cloned as well as sufficient unique objects defined to cover all method selectors used in the example. Otherwise, the example remains a straight-forward translation of the earlier informal PMD example.

4.3 Dynamic Semantics

Figure 8 presents the core dynamic semantics of PMD. These reduction rules take the form $S \vdash e \hookrightarrow e', S'$, to be read as "with respect to a store S, the expression e reduces in one step to e', yielding a new store S'". The reduction rules define method invocation, method definition, object cloning, and delegation addition and removal. The congruence rules define the order of evaluation in the standard way.

The rule **R-Invoke** looks up the body of the most applicable method, with respect to a method selector and a sequence of method arguments, given by the *lookup* function (defined below in Figure 9). The method arguments are then substituted into the lambda expression/method body which is the result. Substitution occurs as in the lambda calculus.

The rule **R-Method** defines a new method body, to be invoked with a given selector, and dispatching on the given set of objects. A new role is added to each object v_i, stating that the object (or any object that delegates to it) can play the ith role in a dispatch on method selector v_s to method body f. The object representations are updated in the store to reflect this, yielding a new store. The first condition ensures this definition is unique to the particular method selector and arguments; there is no other method body defined upon this exact invocation. The expression reduces to the first argument here only to simplify presentation. We omit a rule for method removal for brevity's sake, which would be a straight-forward inversion of this particular rule.

Reduction Rules

$$\frac{lookup(S, v_s, \overline{v}) = \lambda \overline{x}.e}{S \vdash v_s(\overline{v}) \hookrightarrow [\overline{v}/\overline{x}]\, e, S'} \textbf{ R-Invoke}$$

$$\frac{\begin{array}{c} \nexists f' \left(\forall_{0 \le i \le n} \left(S_0\left[v_i\right] = \left(\langle \cdots \rangle, \{\cdots, (s, i, f')\}\right)\right)\right) \\ \forall_{0 \le i \le n} \left(\begin{array}{c} S_i\left[v_i\right] = \left(\langle \overline{d} \rangle, \{\overline{r}\}\right) \\ \bigwedge S_{i+1} = S_i\left[v_i \mapsto \left(\langle \overline{d} \rangle, \{\overline{r}, (v_s, i, f)\}\right)\right] \end{array} \right) \end{array}}{S_0 \vdash v_s(v_0 \cdots v_n) \leftarrow f \hookrightarrow v_0, S_{n+1}} \textbf{ R-Method}$$

$$\frac{S\left[v\right] = O \qquad l \notin dom(S) \qquad S' = S\left[l \mapsto O\right]}{S \vdash clone(v) \hookrightarrow l, S'} \textbf{ R-Clone}$$

$$\frac{S\left[v_o\right] = \left(\langle \overline{d} \rangle, \{\overline{r}\}\right) \qquad S' = S\left[v_o \mapsto \left(\langle \overline{d}, v_t \rangle, \{\overline{r}\}\right)\right]}{S \vdash v_o \triangleright v_t \hookrightarrow v_o, S'} \textbf{ R-AddDelegation}$$

$$\frac{\begin{array}{c} S\left[v\right] = \left(\langle d_0 \cdots d_n \rangle, \{\overline{r}\}\right) \qquad n \ge 0 \\ S' = S\left[v \mapsto \left(\langle d_0 \cdots d_{n-1} \rangle, \{\overline{r}\}\right)\right] \end{array}}{S \vdash v \not\triangleright \hookrightarrow d_n, S'} \textbf{ R-RemoveDelegation}$$

Congruence Rules

$$\frac{S \vdash e_s \hookrightarrow e_s', S'}{S \vdash e_s(\overline{e}) \hookrightarrow e_s'(\overline{e}), S'} \qquad \frac{S \vdash e_s \hookrightarrow e_s', S'}{S \vdash e_s(\overline{e}) \leftarrow f \hookrightarrow e_s'(\overline{e}) \leftarrow f, S'}$$

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

$$\frac{S \vdash e_i \hookrightarrow e_i', S'}{S \vdash v_s(v_0 \cdots v_{i-1}, e_i, e_{i+1} \cdots e_n) \leftarrow f \hookrightarrow v_s(v_0 \cdots v_{i-1}, e_i', e_{i+1} \cdots e_n) \leftarrow f, S'}$$

$$\frac{S \vdash e_o \hookrightarrow e_o', S'}{S \vdash e_o \triangleright e_t \hookrightarrow e_o' \triangleright e_t, S'}$$

$$\frac{S \vdash e_t \hookrightarrow e_t', S'}{S \vdash v \triangleright e_t \hookrightarrow v \triangleright e_t', S'} \qquad \frac{S \vdash e_o \hookrightarrow e_o', S'}{S \vdash e_o \not\triangleright \hookrightarrow e_o' \not\triangleright, S'}$$

Fig. 8. The dynamic semantics of PMD.

Note that method definition here affects intrinsic properties of the supplied argument objects' representations, rather than appealing to some extrinsic semantic device. This becomes significant in the rule **R-Clone**, which provides the ubiquitous copying operation found in prototype-based languages. To ensure that the copied object, which bears a new location and representation in the store, responds to all method invocations in similar fashion as the original object, the rule only needs to do duplicate the list of delegations and set of roles. This simple duplication of relevant dispatch information in turn simplifies the implementation of these semantics.

The rules **R-AddDelegation** and **R-RemoveDelegation** together manipulate the ordered list of delegations of an object in stack-like fashion. The rule **R-AddDelegation** adds a target object as a delegation to the top of the ordered list of delegations of the origin object. The rule **R-RemoveDelegation** removes the top of this ordered list and returns the removed delegation target. These two particular rules were merely chosen to simplify presentation, and other alternative rules allowing for arbitrary modification of the list are certainly possible.

4.4 Dispatch Semantics

$$\frac{f \in applic(S, v_s, \overline{v}) \quad \forall_{f' \in applic(S,v_s,\overline{v})} \left(f = f' \bigvee rank(S, f, v_s, \overline{v}) \prec rank(S, f', v_s, \overline{v})\right)}{lookup(S, v_s, \overline{v}) = f} \textbf{ R-Lookup}$$

$$applic(S, v_s, v_0 \cdots v_n) \stackrel{def}{=} \left\{ f \middle| \forall_{0 \le i \le n} \left(\begin{array}{c} delegates(S, v_i) = \langle d_0 \cdots d_m \rangle \bigwedge \\ \exists_{0 \le k \le m} \left(S\left[d_k\right] = (\langle \overline{d'} \rangle, \{\cdots, (v_s, i, f)\}) \right) \end{array} \right) \right\}$$

$$rank(S, f, v_s, v_0 \cdots v_n) \stackrel{def}{=} \prod_{0 \le i \le n} min_{0 \le k \le m} \left\{ k \middle| \begin{array}{c} delegates(S, v_i) = \langle d_0 \cdots d_m \rangle \\ \bigwedge S\left[d_k\right] = (\langle \overline{d'} \rangle, \{\cdots, (v_s, i, f)\}) \end{array} \right\}$$

Fig. 9. The dispatch semantics of PMD.

Figure 9 presents the dispatch semantics provided by the *lookup* function. The rule **R-Lookup** is a straight-forward transcription of the idea of multiple dispatch. It states that a method body should be dispatched if it is applicable - a member of the set of applicable methods - and it is the most specific of all such method bodies, or rather, is the least method body according to an operator that compares the applicable method bodies. The *rank* function and $\prec$ operator together implement this comparison operator.

We then define the *applic* set of methods as those methods for which every argument either contains a satisfactory role for the method, or delegates to an object with such a role. A role here is satisfactory if index of the method argument on which it is found matches that in the role, and the method selector matches that in the role as well. This definition relies on the *delegates* function, which returns an ordered list of all delegated-to objects transitively reachable by the delegation lists in objects, and including the original method argument itself. In the case of a statically-fixed delegation hierarchy, this rule exactly mirrors the applicability criteria in previous multi-method languages such as Cecil, Dylan and CLOS.

Note that because of the first condition of **R-Method**, only one method body can be directly defined on a tuple of objects at a particular selector. Thus, in the absence of delegation, dispatch is trivial since the applicable set of methods

contains at most a single method body. Ranking the applicable methods is thus a necessary consequence of delegation.

Finally, the *rank* function, conceptually, finds, among those methods in the *applic* set, the distance at which the roles corresponding to some method appeared. Given the ordered list of *delegates* for an argument described above, it determines the minimal position at which a delegated-to object contains a satisfactory role corresponding to the method. The $\prod$ operator, which also parameterizes the *rank* function, combines these minimal positions for each argument and produces a single rank value - conceptually, a n-dimensional coordinate in the ranking. The total ordering given by the *delegates* function facilitates the ordering that *rank* provides, without which these semantics would be trickier to define.

We assume here that a particular method body is unique to a single method definition. So, in the absence of the rule $R-Clone$, for a specific method selector and parameter index, there can only exist a single satisfactory role corresponding to that particular method body. However, since we do include $R - Clone$, and a role may thus be copied to another object, multiple satisfactory roles corresponding to the method body exist and the closest role among them in the *delegates* ordering is chosen.

We leave the *delegates* function, the $\prec$ operator, and the $\prod$ operator undefined. The reader may define these arbitrarily to suit their intended dispatch semantics. Slate's semantics for these operators are defined along with Slate's dispatch algorithm in Section 6.1.

5 Slate

Prototypes with Multiple Dispatch has been implemented in Slate [13], a dynamically typed programming language. Self [14], Cecil [5], and CLOS [2] directly inspired the design of Slate and the PMD model on which it is based. However, due to the retained flexibility of prototypes in PMD, Slate most strongly resembles Self and retains much of its language organization without greatly compromising its simple object model.

The following section provides a taste of the Slate language through small case studies describing how the implementation of the Slate standard library within Slate either benefits or suffers from various aspects of PMD in actual use.

5.1 Brief Overview

The syntax and system organization of Slate strongly resembles Self and so should be discernable to the reader if moderately familiar with Self or even Smalltalk [10]. We omit discussion at length of Slate's syntax, as it is presented more extensively elsewhere [13]. However, we still provide the following small case studies in Slate, as some are slightly difficult to improvise in a more common syntax.

For readers familiar with Smalltalk or Self who wish to examine the various examples more closely, we provide a brief overview of some of the more confusing aspects of the syntax.

Method definition simply affixes a block (the method body) to the normal message syntax, with the arguments also qualified with parameter bindings. These qualified message arguments are of the form "`parameterName @ roleArgument`", with "`roleArgument`" identifying the argument to method definition wherein to place a role, and "`parameterName`" being a variable bound when the method body is actually applied. If the message argument is merely of the form "`parameterName`", this behaves as if the role were placed on the distinguished object "`Root`" to which most other objects in Slate delegate. The presence of atleast one fully qualified parameter is what signals a method definition in the grammar as opposed to a method invocation.

Some important messages to be used in the subsequent examples include:

resend Resends the message that invoked the current method while ignoring any methods of greater or equal precedence in the dispatch order than the current method during dispatch.

prototype clone Returns a new copy of "`prototype`" that contains all the same slots, delegation slots, and roles.

object addSlot: name valued: initialValue Adds a new slot to "`object`" and defines the method "`name`" with which to access its value and the method "`name:`" with which to set it. "`name`" must evaluate to a symbol. The slot's value is initially "`initialValue`".

object addDelegate: name valued: initialValue This method behaves exactly like "`addSlot:valued:`", except only that the slot is treated as a delegation slot. The value of the delegation slot is treated as an object that "`object`" delegates to.

object traits Accessor message for the "`traits`" delegation slot, which holds an object sharing method roles for a whole family of "clone"-d objects.

block do Evaluates "`block`".

collection do: block Evaluates "`block`" with each element of collection "`collection`" supplied in turn as an argument.

cond ifTrue: trueBlock ifFalse: falseBlock Evaluates "`trueBlock`" if "`cond`" evaluates to "`True`", or instead "`falseBlock`" if it evaluates to "`False`".

5.2 Example: Instance-specific Dispatch

Instance-specific dispatch is an extensively used idiom in Slate. Yet, it is intrinsically supported by PMD, since it is purely based on prototypes, and requires no special handling in Slate.

When combined with multiple dispatch, it begins to strongly resemble pattern-matching while still within an object-oriented framework. For example, much of the boolean logic code in Slate is written in a strikingly declarative form using instance-specific dispatch:

```
_@True and: _@True [True].
_@(Boolean traits) and: _@(Boolean traits) [False].
_@False or: _@False [False].
_@(Boolean traits) or: _@(Boolean traits) [True].
```

The code dispatches directly on "`True`" and "`False`" to handle specific cases. It then defines methods on "`Boolean traits`" to handle the remaining default cases.

5.3 Example: Eliminating Double Dispatch

Smalltalk [10] and similar languages based on single dispatch typically rely on an idiom called "double dispatch" to work around the limitations this model imposes. Double dispatch bottles up dispatch code for a second or subsequent argument of a method either directly within the method dispatching on the distinguished receiver or is manually factored into a dispatch object written by the programmer.

This idiom frequently surfaces in such places as Smalltalk numerics system and makes it a chore to integrate new numeric entities into the system. All the double dispatch code, distributed among many diverse classes, must be updated to take the new entity into account where necessary.

Slate natively supports multiple dispatch and does not fall victim to these limitations. It is relatively simple to extend Slate's numerics system while keeping these extensions well-encapsulated and without needing global changes to other objects. For example, the following code illustrates how an epsilon object, a negligibly small yet non-zero value, may be easily integrated:

```
numerics addSlot: #PositiveEpsilon valued: Magnitude clone.

_@PositiveEpsilon isZero
[False].
_@PositiveEpsilon isPositive
[True].
x@(Magnitude traits) + _@PositiveEpsilon
[x].
```

It is also common in Smalltalk to find many methods such as "`asArray`" or "`asDictionary`" for converting a certain object to the type indicated by the message name. This is, in effect, the programmer manually performing the double dispatch.

With the aid of PMD, Slate easily supports a more expressive and uniform protocol for coercing objects of one type to another via the message "`as:`". The object to convert is supplied along with an instance (as opposed to a class) of some object type the programmer would like the original to coerce to. To define coercions, the programmer need only define a particular method for his new objects as in the following code:

```
x@(Root traits) as: y@(Root traits)
[(x isSameAs: y)
  ifTrue: [x]
  ifFalse: [x conversionNotFoundTo: y]
].
c@(Collection traits) as: d@(Collection traits)
[d newWithAll: c].
s@(Sequence traits) as: ec@(ExtensibleCollection traits)
[| newEC |
  newEC: (ec newSizeOf: s).
  newEC addAll: s.
  newEC
].
s@(Symbol traits) as: _@(String traits)
[s name].
```

5.4 Example: Supporting System Reorganization

Another benefit of using a prototype object system as the language core is that it easily supports reorganizing the language to support new features or remodel old ones.

For instance, Slate uses a depth-first search strategy for finding roles on delegated-to objects. Whichever roles are found first according to this order take precedent over ones found later. However, this simplistic scheme, while allowing an efficient dispatch algorithm and providing the illusion of single inheritance, easily becomes inappropriate in the presence of multiple inheritance.

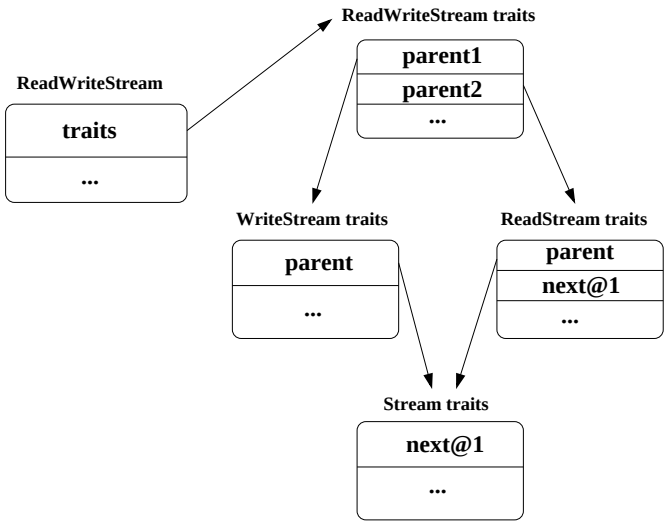

Fig. 10. Slate's original traits inheritance model. Multiple inheritance occasionally results in problems with sequencing methods.

Figure 10 illustrates the problem. A `ReadWriteStream` is derived from both a `WriteStream` and a `ReadStream`, and so its traits object delegates to both of their traits objects as well. `ReadStream`, in particular, might override a method "`next`" on the basic `Stream` prototype. However, should the dispatch algorithm visit `ReadWriteStream`'s traits, `WriteStream`'s traits, `ReadStream`'s traits, and `Stream traits` in that order, the "`next`" method on `Stream` will resurface and take precedence over the version on `ReadStream`. Ideally, the search should proceed topologically, so that `Stream`'s traits object is visited only after both `WriteStream`'s and `ReadStream`'s traits objects have been visited.

This behavior became severely confusing at times, yet merely throwing an error in this case forces the programmer to manually disambiguate it by defining a new method. Instead of adding a barrier to object reuse, a simple reorganization of the traits object system, still only replying upon native delegation and dispatch of PMD, allowed a much more satisfactory behavior.

Instead of having traits directly delegate to their parent traits, an extra layer of indirection was added in the form of a traits window. Objects now delegate to this

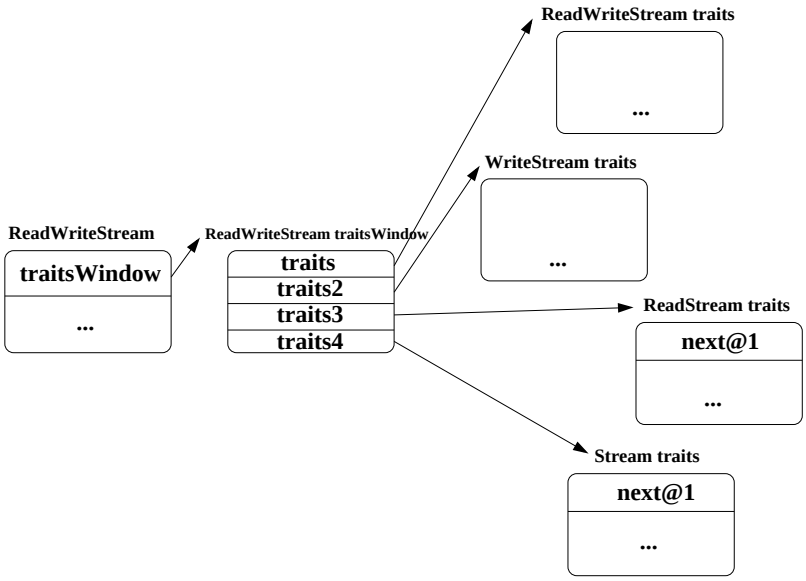

Fig. 11. Slate's new traits inheritance model allows the more desirable topological sequencing of methods.

window instead of directly to traits, and this window merely keeps a list of traits in the exact order they should be visited by the dispatch algorithm. This has the added benefit that even orderings that are expensive to compute and might otherwise defeat various optimizations Slate uses to enhance the performance of dispatch may be freely used and customized at a whim without any negative impacts. Figure 11 illustrates this new organization.

Yet, because Slate is based upon a prototype object system, this did not require any profound changes to the language's implementation to effect this new organization. Most of the changes were localized within the standard library itself, and mostly to utility methods used to construct new prototypes. Only a few lines of code in the interpreter itself that depended on the structure of certain primitively provided objects needed revision.

5.5 Example: Subjective Dispatch

In earlier work on the Self extension Us [15], it was noted that any language possessing multiple dispatch can easily implement subjective dispatch similar to that provided by Us. In this view of subjective dispatch, a subject is merely an extra implicit participant in the dispatch process, supplied in ways other than directly via a message invocation.

As PMD provides multiple dispatch, Slate supports subjective dispatch of this sort with some slight changes to its semantics. It need maintain a distinguished subject in its interpreter state, which is implicitly appended to the argument lists of message invocations and method definitions whenever either is evaluated within this subject. The interpreter also provides a primitive message "`changeSubject:`" to modify the current subject. The semantics of Slate otherwise remain unchanged.

Further, prototypes naturally support composition of subjects by delegation, allowing for a sort of dynamic scoping of methods by merely linking subjects together with dynamic extent. The message "`seenFrom:`" is easily implemented to this effect:

```
addSlot: #Subject valued: Cloneable clone.
Subject addDelegate: #label.
Subject addDelegate: #previousSubject.

m@(Method traits) seenFrom: label
[| newSubject |
  newSubject: Subject clone.
  newSubject label: label.
  newSubject previousSubject: (changeSubject: newSubject).
  m do.
  changeSubject: newSubject previousSubject
].
```

This subjective behavior can easily allow for the implementation of cross-cutting aspects of a program. The following code illustrates this through the implementation of an undoable transaction, which works by intercepting any modifications to objects via the message "`atSlotNamed:put`", logging the original value of the slot, then allowing the modification to proceed:

```
addSlot: #Transaction valued: Cloneable clone.
Transaction addSlot: #undo valued: ExtensibleArray newEmpty.
Transaction addSlot: #replay valued: ExtensibleArray newEmpty.

t@Transaction log: object setting: slot to: newValue
[| oldValue |
  oldValue: (object atSlotNamed: slot).
  t undo addLast: [object atSlotNamed: slot put: oldValue].
  t replay addLast: [object atSlotNamed: slot put: newValue].
].
t@Transaction undo
[t undo reverseDo: [| :action | action do]].
t@Transaction replay
[t replay do: [| :action | action do]].

[object atSlotNamed: slot@(Symbol traits) put: value
  [Transaction log: object setting: slot to: value.
    resend
  ].
] seenFrom: Transaction.
```

6 Dispatch in Slate

Many optimizations have been explored to enhance the performance of programs in Slate. This section details implementation strategies used in Slate and that may be applied to implementations of PMD for other languages.

6.1 Dispatch Algorithm

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 ranks of these methods and how to represent them. Various optimizations also expediently reduce the memory and processing requirements of the algorithm.

```
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 if not yet visited
      }
      position := position + 1
    }
  }
  return most specific method
}
```

Fig. 12. Pseudo-code for the basic dispatch algorithm used in Slate.

The programming language Slate serves as a canonical implementation of PMD and utilizes a dispatch algorithm 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 12 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 latter of which is used in Slate. The order in which delegations from a given object are pushed onto and popped from the ordering stack, analogous to the *delegates* function in the formalism, determines the ordering under multiple and non-trivial delegation and should be chosen as is applicable to the implementation. A simple vector of positions in a rank here provides the $\prod$ operator of the formalism. If one overlooks the necessary 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, 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 its 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.

6.2 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 overflows 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, if ever.

6.3 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.

6.4 Partial Dispatch

Because of Slate's sparse representation of roles, the dispatch algorithm may determine a method to be applicable, or rather, its 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, its 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 a special-case optimization of the PMD dispatch semantics. Further, almost all the dispatches in the Slate standard library 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.

6.5 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. The method caching optimization assumes that there are no changes to delegation relationships and no method addition or removal; if these changes are made, the caches must be invalided in the general case.

7 Related Work

Section 4 described related work in the area of formal object models. Three programming languages significantly influenced the development of PMD and the implementation in Slate: Self [14], CLOS [2], and Cecil [5].

Self attempted to provide a Smalltalk [10] better suited for interactive programming and direct manipulation of objects by dispensing with classes and providing self-representing objects. These objects simply contain slots - modifiable, named value-holders - which can serve as ordinary bindings, method definitions, or delegations. Further, objects are used to more flexibly represent traditionally fixed implementation facilities such as namespaces, shared behavior, and user interfaces. Slate, to date, borrows and benefits from much of this system organization while expanding upon the notion of method definition as in PMD.

CLOS [2] is an extension to Common Lisp that provides object-oriented programming through classes and generic functions. Generic functions are functions made up multiple method cases, which a multiple dispatch algorithm chooses among by examining the classes of all the arguments to a method call. A subtyping relation between the classes of parameters and arguments determines the applicable method bodies and their relative specificities. CLOS linearly (totally) orders both the class and method. The class hierarchy is sequenced into a precedence list to disambiguate any branches in

the hierarchy as a result of multiple inheritance. Leftmost parameters also take precedence over the rightmost, disambiguating cases where not all the parameter classes of one method are subtypes of the respective parameter classes of another. The formalism of PMD borrows the idea of a total ordering of inheritance and method arguments in its dispatch semantics to avoid appealing to subtyping, but dispenses with classes and the extrinsic notion of generic functions.

Dylan [8] is another dynamically-typed object-oriented language with multi-methods. Like CLOS, it gives precedence to the leftmost parameter of a function during object-oriented dispatch.

Cecil [5] is the first language known by the authors to integrate a prototype-inspired object model with multiple dispatch. Cecil dispenses with the slot-based dynamic inheritance of Self, opting instead to fix delegation between objects at the time an object is instantiated. Method definition is similarly limited to a global scope, restricting certain higher-order uses. Cecil provides multiple dispatch by a form of subtyping upon this relatively fixed delegation hierarchy. This multiple dispatch only provides a partial ordering among objects and method arguments. Dispatch ambiguities arise from the use of multiple delegation or incompete subsumption among the methods according to the subtyping relation. Such ambiguities raise an error when encountered, and recent work has focused on finding these ambiguities statically [12].

Instead of the slot-based dynamic inheritance of Self, however, Cecil provides predicate classes [6] wherein a fixed delegation relationship is established to a predicate class that is qualified by a predicate. When the predicate of a predicate class is satisfied for some object delegating to it, the object delegating to it will inherits its behavior. When this predicate is not satisfied, this behavior will not be inherited. Predicate dispatch is thus ideal for capturing behavior that depends on a formula over the state of the object, while the dynamic delegation mechanism in PMD captures behavior changes due on program events more cleanly. More recently, predicate classes have been generalized to a predicate dispatch [7, 11] mechanism which unifies object-oriented dispatch with pattern-matching in functional programming languages.

One other closely related system is Us, an extension of Self to support subject-oriented programming [15]. Subject-oriented programming allows a method to behave differently depending on the current *subject* in scope. Intuitively, subject-oriented programming can be modeled as an additional layer of dispatch, and multiple dispatch is a natural mechanism for implementing this concept, especially when combined with flexible objects with which to dynamically compose subjects, as the authors of Us noted. However, as Us extends a language only providing single-dispatch, the authors of Us instead chose to separate objects into pieces, all addressed by a single identity. Dynamically composable layer objects implicitly select which piece of the object represents it, effectively implementing a specialized form of multiple dispatch only for this extension. Since PMD provides multiple dispatch and dynamic inheritance, it naturally supports subjects with only a bit of syntactic sugar.

8 Conclusion

This paper introduced a new object model, Prototypes with Multiple Dispatch, that cleanly integrates prototype-based programming with multiple dispatch. The PMD model allows software engineers to more cleanly capture the dynamic interactions of multiple, stateful objects.

9 Acknowledgments

We thank Aaron Greenhouse, Jonathan Moody, and the anonymous reviewers for their feedback on earlier drafts of this material. This work was supported in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
2. D. G. Bobrow, L. G. DiMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System Specification. In *SIGPLAN Notices 23*, September 1988.
3. V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *European Conference on Object-Oriented Programming*, 1998.
4. G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. In *Lisp and Functional Programming*, 1992.
5. C. Chambers. Object-Oriented Multi-Methods in Cecil. In *European Conference on Object-Oriented Programming*, July 1992.
6. C. Chambers. Predicate Classes. In *European Conference on Object-Oriented Programming*, 1993.
7. M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate Dispatching: A Unified Theory of Dispatch. In *European Conference on Object-Oriented Programming*, 1998.
8. N. Feinberg, S. E. Keene, R. O. Mathews, and P. T. Withington. *Dylan Programming*. Addison-Wesley, Reading, Massachusetts, 1997.
9. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
10. A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, Reading, Massachusetts, 1989.
11. T. Millstein. Practical Predicate Dispatch. In *Object-Oriented Programming Systems, Languages, and Applications*, 2004.
12. T. Millstein and C. Chambers. Modular Statically Typed Multimethods. *Information and Computation*, 175(1):76–118, 2002.
13. B. Rice and L. Salzman. The Slate Programmer's Reference Manual. Available at http://slate.tunes.org/progman/, 2004.
14. D. Ungar and R. B. Smith. Self: The Power of Simplicity. In *Object-Oriented Programming Systems, Languages, and Applications*, pages 227–242. ACM Press, 1987.
15. D. Ungar and R. B. Smith. A Simple and Unifying Approach to Subjective Objects. *Theory and Practice of Object Systems*, 2(3):161–178, 1996.