Static meta-object protocols : towards efficient reflective object-oriented languages

CLARK, Tony <http://orcid.org/0000-0003-3167-0739>

Static Meta-Object Protocols: Towards Efficient Reflective Object-Oriented Languages

Tony Clark
Department of Computing Sheffield Hallam University Sheffield, South Yorks, UK
t.clark@shu.ac.uk

ABSTRACT

Reflection and extensibility in object-oriented programming languages can be supported by meta-object protocols (MOP) that define class-based interfaces over data representation and execution features. MOPs are typically *dynamic* in the sense that type-based dispatching is used to select between feature implementations at run time leading to a significant difference in execution speed compared to non-MOP-based languages. Defining a corresponding *static*-MOP would seem to be a solution whereby type-dispatching can occur at compile time. Such an approach requires the integration of a static type system with a MOP. This paper introduces a new reflective and extensible language called JMF written in Java that aims to generate efficient code through the use of a static-MOP. The contribution of this paper is to characterise a static-MOP and to show how it integrates with a type system for JMF.

CCS Concepts

•Software and its engineering → Reflective middleware; Extensible languages; Classes and objects;

Keywords

reflection; meta-object protocol; type-checking

1. INTRODUCTION

Dynamic features including reflection [14, 10, 13] are becoming increasingly important in programming languages in order that applications can adapt to changes in their environment [3], to support tools such as debuggers [17], and to support domain specific languages (DSLs). These requirements, addressed by *metaprogramming* [5], would otherwise require modification to the underlying language execution engine in order to support variation points and instrumentation leading to multiple language implementations and corresponding compatibility problems [16].

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Many languages provide reflective features to support *introspection* and *intercession* to facilitate adaptation of structure and behaviour. Such features are often made available via a *metaobject protocol* (MOP) [9] which is an interface that gives users the ability to incrementally modify the behaviour of a language. A MOP can be derived by analysing the key orthogonal execution mechanisms of a language.

There are several approaches to implementing a MOP that can be broadly categorised into: (a) providing access to the language interpreter via *continuation-passing*, for example [2]; (b) code-rewriting, for example [12, 7]; (c) type-based extension of language features via an interface. MOP-based reflective features are often suited to interpreted or dynamically typed languages, *e.g.*, [17], Smalltalk, CLOS, Python [1], Lua [8].

2. PROBLEM AND HYPOTHESIS

A MOP is a mechanism to allow the key execution features of an OO language to be extended. Since *inheritance* is an intrinsic feature of most OO languages, it makes sense to use inheritance *at the type level* to define a MOP. The particular features to be extended will differ from MOP to MOP; however, since they seem to be ubiquitous in OO languages, this paper addresses a MOP in terms of object creation, slot access and update, and message passing.

MOPs are typically used to support multiple object representation and to abstract away from the details of representation management. Consider a program that wants to mix two different types of class: those with small storage requirements and those with large sparsely populated storage requirements. In the first case we want to use an array and in the second case to use a hash-table. A language supporting MOPs might allow the following[1]:

```
class Point metaclass Array(Int) {
  names = ['x','y'];
}

class Employee metaclass Sparse {
  name:Str;
  ...
  salary:Float;
}
```

In each class definition the declaration of the meta-class determines how object creation, slot access and update, and message passing will occur for instances of the class. For example if `p` is an instance of `Point` then `storage(p)` is an array and `p.x:Int` is `storage(p)[0]` whereas if `e` is an instance of

[1]Since JMF and Java concrete syntax are similar, JMF keywords are underlined.

`Employee` then `storage(e)` is a hash-table and `e.name:Str` is `get(storage(e),"name")`.

Many MOP-based languages are dynamic in the sense that the language feature switching mechanism occurs at run-time. This is most often a result of the reflective nature of the languages resulting in types being run-time entities. Therefore in the general case it is not possible to statically determine the type of a program expression since its type may only exist at run-time. However, this is not always the case. In particular, a MOP-based language will have a default MOP whose types are statically known. Since many such languages are bootstrapped, being able to efficiently compile expressions that use the default MOP ought to result in a significant speed improvement.

The hypothesis of this work is that it is possible to extend a standard dynamic MOP in such a way as to allow type-based static analysis of a program and consequently generate efficient code for the language features governed by the MOP. To accommodate reflection, it will not always be possible to perform static analysis, in which case the default dynamic switching mechanism can be employed.

The hypothesis is to be evaluated in the context of a new language called JMF that is a MOP-based OO language written in Java to be compiled to the JVM. If the hypothesis holds then the code generated on the JVM will have a similar performance profile to equivalent Java code in the case that meta-types can be statically determined.

The first step is to define a dynamic MOP for JMF and to show how the MOP can be extended with static features. This paper describes the result of taking the first step as follows: section 3.4 describes core JMF, type dispatching and and how it is implemented in Java; section 4 shows how the JMF default dynamic MOP is implemented and section 5 shows how this is extended to produce dynamic MOPs for `Sparse` and `Array`; section 6 describes JMF type-checking and the associated default static MOP and section 7 extends this for `Sparse` and `Array`.

3. CORE JMF

JMF consists of a core collection of Java objects that serve the same function as the classes defined by the Java package `java.lang.reflect`. Each Java object must implement an interface called `Obj` as defined in section 3.1. This paper uses a subset of the JMF core relating to MOP definition and use; an overview of the core is given in section 3.4. A syntax for JMF programs is necessary in order to perform static analysis, this paper uses a subset both to provide examples and to define how the static MOP is defined. Section 3.2 provides an example of how a simple class is defined using the JMF syntax and subsequently extended using core classes via Java.

3.1 Everything is an Object

JMF is defined as a collection of Java objects that all implement an interface called `Obj`:

```
public interface Obj {
 public Obj get(String name);
 public Obj of();
 public Obj send(String name, Obj... args);
 public Obj set(String name, Obj value);
 public Obj slots();
}
```

The operation `of()` returns the class of an object in the same way that `getClass()` does in Java. The operations `get(n)` and `set(n,v)` are used to access and update named storage. The operation `slots()` returns a list containing slots (essentially name-value pairs) of the receiver. The `send(n,v,...,v)` operation sends a named message to the receiver.

3.2 A Simple Program

The following is an example of a JMF program that uses the default MOP:

```
class Point {
  x:Int;
  y:Int;
  inc():Point {
    x := x + 1;
    y := y + 1;
  }
}
p:Point = Point.new();
p.inc();
```

Dynamic MOP dispatch is implemented using the Java operations `send`, `get` and `set`. The JMF class definition for `Point` is translated to the equivalent Java program as follows:

```
Obj x = send(Attribute,"new", send(Str("x"), Int);                    1
Obj y = send(Attribute,"new", Str("y"), Int);                         2
Obj name = Str("Point");                                              3
Obj supers = list(Obj);                                               4
Obj atts = list(x,y);                                                 5
Obj Point = send(Class,"new",name,atts,supers);                       6
                                                                      7
send(Point,"addOperation",                                            8
  Op("inc", (self, so) ->                                             9
    set(                                                              10
      set(                                                            11
        self,                                                         12
        "x",                                                          13
        send(                                                         14
          get(self,"x"),                                              15
          "+",                                                        16
          Int(1))),                                                   17
      "y",                                                            18
      send(                                                           19
        get(self,"y"),                                                20
        "+",                                                          21
        Int(1))))));                                                  22
                                                                      23
Obj p = send(Point,"new");                                            24
send(p,"inc");                                                        25
```

The builtin operations `Str` and `Int` are used to map between Java values and JMF objects. The class `Point` is created by sending a `new` message to `Class` (since no explicit meta-class was supplied in the class definition for `Point`, the default meta-class is used) supplied with the name of the new class, its attributes and the super-classes. The builtin operation `list` takes any number of arguments and returns an JMF object representing list. By default, attributes are like Java fields and inheritance is the same as Java in the case where a class has a single super-class.

Operations that are added to classes (lines 8-22) work the same way as Java methods. Note that the operator `Op` is used to map between Java closures and JMF operations where the closure must have at least 2 arguments: the first is the receiver of the message that invokes the operation and the second works like **super** in Java.

3.3 Dynamic Dispatch

The Java code shown above uses dynamic dispatch to determine the implementation of language features in terms of Java methods `send`, `get` and `set`. These have been inserted by default when translating from JMF to Java since there has been no static analysis of the original JMF code. Dynamic

dispatch checks for `Class` in order to prevent *meta-recursion* [11]. The definitions of the methods are as follows:

```
public static Obj get(Obj o, String n) {
 if(o.of().of() == Class)
   return o.get(n);
 else return send(o.of(),"get",o,Str(n));
}

public static Obj set(Obj o, String n, Obj v) {
 if(o.of().of() == Class)
   return o.set(n,v);
 else return send(o.of(),"set",o,Str(n),v);
}

public static Obj send(Obj o, String n, Obj... vs) {
 if(o.of().of() == Class)
   return o.send(n,vs);
 else return send(o.of(),"send",o,Str(n),list(vs));
}
```

The definitions above show that the dynamic dispatch mechanism is fixed for objects whose meta-class is `Class`. Otherwise, an appropriate message is sent to the class of the object which must implement a suitable implementation, *i.e.* a MOP.

3.4 An Overview of the JMF Core

The JMF Core consists of a collection of bootstrapped classes that are shown in figure 1[2]. The syntax of the diagram is a variation on standard class diagrams: boxes are classes, edges with open arrows relate sub-classes to super-classes, full edges with arrows are labelled with the name of an attribute of the source class, dashed edges relate a class with its meta-class. Note that JMF classes define addition features including operations, constraints, constructors, and daemons, that are not shown.

The following conventions are used: a class with no super-class relation inherits from `Obj`; a class with no explicit meta-class is an instance of `Class`; attributes with simple types are shown within the class box; classes whose name has a white background are abstract; a class that is surrounded by a box is a meta-class; an attribute with a prefix `*` means that the corresponding class is a *container* for the elements of the attribute type; an attribute with a prefix `::` means that the owning class is a *name-space* with corresponding operations to index the values of the attribute by name; an attribute with a suffix `$` means that the owning class transitively *inherits* the values of the the attribute via a correspond operation; an attribute with a suffix means that the attribute is a specialisation of an attribute with the same name that is defined by a super-class; a package is shown using UML-style package notation; a type `[T]` represents a list of elements of type `T`.

For example, the class `Slot` is a sub-class of `NamedElement` and therefore inherits the attribute named `name` of type `Str`, and has an additional attribute named `value` of type `Obj`.

The class `Class` is an instance of itself, is a name-space for attributes, is a container of constructors, and inherits from the meta-class `Classifier`.

The meta-classes `InheritsOf`, `ContainerOf`, `NameSpaceOf`, `FunctionOf`, `ListOf`, and `TableOf` are all polymorphic in the sense that they are instantiated to produce classes by providing one or more types. For example `TableOf(String,Int)` is an instance of `TableOf` whose instances are tables mapping strings to integers.

A class that inherits from `ContainerOf(X)[contentName='x']` is shown on the diagram as the source of an edge labelled `*x` with target `X`. Similarly `NameSpaceOf(X)[contentName='x']` is shown using an edge-label `::x`. A class that inherits from `InheritsOf(X)[inheritedName='x']` is shown with an edge label `x$` and implicitly defines an operation named `allX` in the source class that constructs a list of all the inherited values of the attribute named `x`.

The package `Kernel` is an instance of the class `Package` and is its own meta-package. The entire diagram shows the contents of the package `Kernel`.

4. THE DEFAULT DYNAMIC MOP

A MOP allows the programmer to extend key aspects of the language. In the case of JMF we choose to have a dynamic MOP that supports **new**, `get`, `set`, and `send`. The MOP definition for `new` controls the representation for instances of a class. This is in contrast to Java where there is a single representation for objects generated by `java.lang.Class.newInstance()`. Having allowed a class to control the representation of its own instances, the MOP must provide `get` and `set` in order to support state access and update. Finally, the behaviour of a class of objects is controlled by the MOP definition of `send`.

The default JMF MOP defines the semantics of the language and must be defined in Java. The definition involves a Java class called `ConcreteObj` that implements the `Obj` interface:

```
class SlotTable extends Hashtable<String,Obj> {}

class ConcreteObj implements Obj {
  Obj of;
  SlotTable slots = new SlotTable();
  public ConcreteObj(Obj of) { this.of = of; }
  public Obj get(String n) { return slots.get(n); }
  public Obj set(String name,Obj value) {
    slots.put(name,value);
    return this;
  }
  public Obj send(String n,Obj... vs) {
    return sendCO(0,flatten(of),this,n,vs);
  }
  public Obj of() { return of; }
}
```

The default representation for objects uses a hash-table to represent the slots. The implementation uses the operation `sendCO` which is defined below.

The next step is to define an operation that creates new instances of classes that are governed by the MOP. A new instance is created by sending `new` to a JMF class, therefore the definition of `new` is owned by a meta-class. The basic meta-class is `Class` and the following code is part of the JMF bootstrap:

```
send(Class,"addOperation",Op("new",Class_new));

static Obj Class_new(Obj type,Obj so,Obj args) {
  Obj atts = send(type,"allAttributes");
  ConcreteObj o = new ConcreteObj(type);
  for(Obj att : iterate(atts)) {
    Obj value = get(get(att,"type"),"default");
    set(o,att.get("name").toString(),value);
  }
  return send(type,"initObj",o,args);
}
```

[2]The diagrams in this paper are created by a walker written in JMF that translates JMF packages to dot files that are processed by GraphViz.

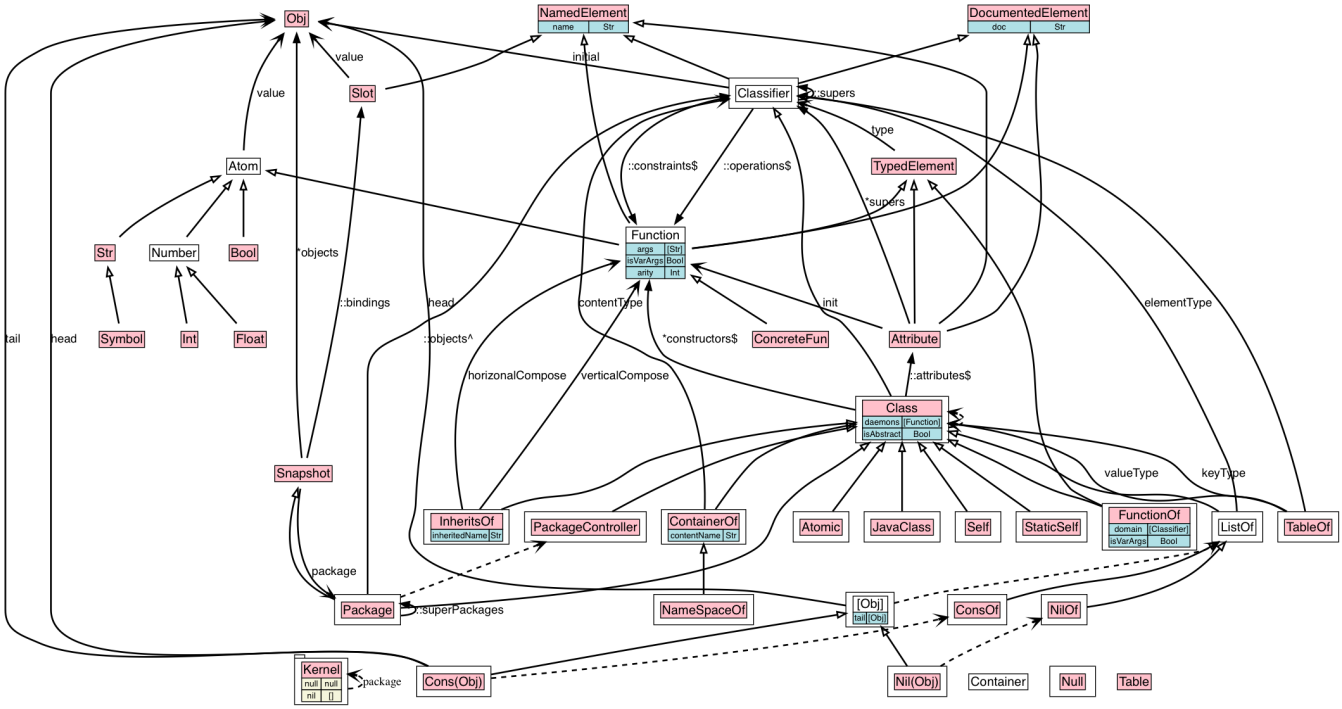

Figure 1: The JMF Kernel

The operation added to `Class` in line 1 is implemented by the Java static operation `Class_new`. All classes provide an operation called `allAttributes` that is the equivalent of `java.lang.Class.getFields()`. The Java method `iterate` translates a JMF list to a Java iterator and is used in line 6 to process each attribute. An attribute has a name and a type. A new slot is added to the concrete object containing the default value associated with the type (lines 7 and 8). Finally, the class is sent an `initObj` message that applies a constructor with the appropriate arity to `args` in the context of the new object `o`.

The default JMF MOP is completed by operations defined by `Class`:

```
Op("get",(c,o,n) -> o.get(n.toString()))
Op("set",(c,o,n,v) -> o.set(n.toString(),v))
Op("send",(c,o,m,vs) -> o.send(m.toString(),vs))
```

It remains to define the default mechanism for delivering a message. The operator `sendCO` is defined below. It is supplied with an array of classes `cs` together with an index that represents the current position in the array. Each JMF class is the bottom element of a lattice that has its top element as `Obj`. The operator `flatten` maps a class to an array by performing a depth first, left to right, lattice traversal. Such a traversal can encounter the same class multiple times, however the class will occur in the resulting array only once by ignoring all but the last occurrence. Sending a message `n` to an object `o` with arguments `vs` is handled by `sendCO(0,flatten(o.of()),o,n,vs)`:

```
1  Obj sendCO(int i,Obj[] cs,Obj o,String n,Obj vs) {
2    if(i == cs.length)
3      return send(o,"noOperationFound",n,vs);
4    else {
5      Obj op = lookupOp(cs[i],n,length(vs));
6      if(op != null) {
7        Obj so = new Obj() {
8          public Obj get(String n) { return o.get(n); }
9          public Obj of() { return Class; }
10         public Obj send(String n,Obj... args) {
11           return sendCO(i+1,cs,o,n,list(args));
12         }
13         public Obj set(String n,Obj v) {
14           return o.set(n,v);
15         }
16         public Obj slots() { return o.slots(); }
17       };
18       return send(op,"invoke",o,so,args);
19     } else return sendCO(i+1,cs,o,n,vs);
20   }
21 }
```

If the lattice is exhausted (line 2) then no operation is defined. Otherwise, the `i`th class is examined to see if it contains an operation with the name `n` and whose arguments match `vs`. If an operation is found then it can be applied to the arguments using the message `invoke`. The first two arguments to `invoke` are the target of the message and a *super object*. The super object can be used as the target of a message that will continue the current lattice traversal. Lines 7-17 create an object that performs the required behaviour of a super object.

5. IMPLEMENTING A DYNAMIC MOP

The default JMF MOP defined above uses a hash-table for storage that is pre-populated with values for all attributes defined by a MOP-compliant class. The two classes shown in section 2 are different: `Point` uses a fixed size array for the storage and `Employee` uses a hash-table that is not pre-populated. This section describes how these classes are supported by two different MOPs.

5.1 The Sparse MOP

The Sparse MOP uses a Java hash-table as object stor-

age and populates the table incrementally unlike the default MOP defined in section 4. Figure 2 shows the desired arrangement for a particular sparse-class instance where only the name slot has been given a value. The meta-class named `Sparse` is created by defining a class in JMF that inherits from `Class`:

```
class Sparse extends Class {}
```

The meta-class `Sparse` must override the `new` operation defined by `Class` so that new sparse-class instances implement the `Obj` interface appropriately. This is done in Java:

```
1  send(Sparse, "addOperation", Op("new",Sparse_new));
2
3  public static Obj Sparse_new(Obj c,Obj so,Obj args) {
4    SlotTable storage = new SlotTable();
5    return new Obj() {
6      public Obj get(String n) {
7        if(storage.containsKey(n))
8          return storage.get(n);
9        else {
10         Obj att = send(c,"getAttributeNamed",Str(n));
11         return get(a,"initial");
12       }
13     }
14     public Obj of() { return c; }
15     public Obj send(String n,Obj... vs) {
16       if(n.equals("clear") {
17         storage.clear();
18         return this;
19       }
20       return sendCO(0,flatten(of),this,n,vs);
21     }
22     public Obj set(String name, Obj value) {
23       storage.put(name,value);
24       return this;
25     }
26     public Obj slots() {
27       slots = theObjNil;
28       for (String n : storage.keySet()) {
29         Obj s=send(Slot,"new",Str(n),storage.get(n));
30         slots=send(slots, "cons", s);
31       }
32       return slots;
33     }
34   };
35 }
```

The object storage is created on line 4 as a standard `SlotTable` and the new instance is created on lines 5-34. Notice how the `get` operation checks whether the name exists in the table and returns the initial value for the appropriate attribute type if the slot has yet to be set.

The sparse MOP implements a pseudo-operation called `clear` that is detected on line 16 and which clears the storage. If the message is not `clear` then the default message passing MOP is used.

5.2 The Array MOP

The `Array` MOP uses a Java array as object storage. Each element of the array must be of the same JMF type. A class that conforms to the `Array` MOP declares the names of the slots that will be associated with successive elements in the array. Figure 3 shows the desired arrangement for a particular point instance: the class `Point` is an instance

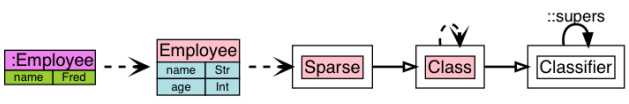

Figure 2: The Sparse MOP

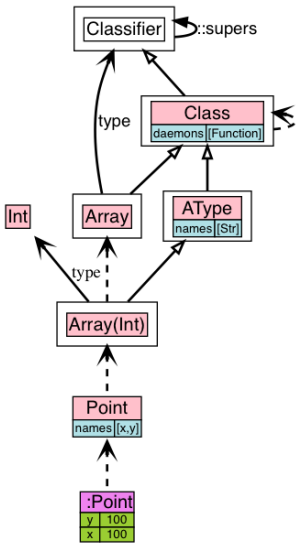

Figure 3: The Array MOP

of the meta-class `Array(Int)` which itself is an instance of the meta-class `Array`. The rest of this section provides the implementation in terms of JMF and Java code.

Each array class (such as `Point`) must specify a collection of names. The following class called `AType` is used as a *mixin*:

```
class AType extends Class { names:[Str]; }
```

The meta-class `Array` is used to create new array types by supplying the type of the elements. The constructor for `Array` is supplied with a classifier `t` that types the elements of the array. Note how the `supers` are set to `[AType]` thereby mixing in the `names` attribute:

```
class Array extends Class {
  type:Classifier;
  Array(t:Classifier) {
    name := 'Array(' + type + ')';
    attributes := [];
    type := t;
    supers := [AType];
  }
}
```

When an instance of an array type (such as `Point`) is created, the MOP must arrange for a Java array to be used for the object storage. This is achieved below by adding a Java-defined operation called `new` to `AType`:

```
send(AType, "addOperation", Op("new",AType_new));

public static Obj AType_new(Obj c,Obj so,Obj args) {
  Obj[] storage = new Obj[length(get(c,"names"))];
  Obj type = get(c.of(),"type");
  for (int i = 0; i < storage.length; i++)
    storage[i] = get(type, "initial");
  return new Obj() {
    private int index(String name) {
      int i = 0;
      for(Obj n : iterate(get(c,"names")))
        if(n.toString().equals(name)) return i;
      return -1;
    }
    public Obj get(String name) {
      return storage[index(name)];
    }
    public Obj of() { return c; }
    public Obj send(String n,Obj... vs) {
      return sendCO(0,flatten(of),this,n,vs);
```

```
    }
    public Obj set(String name, Obj value) {
      storage[index(name)] = value;
      return this;
    }
    public Obj slots() {
      slots = theObjNil;
      for (Obj n : iterate(get(c,"names"))) {
        Obj s = send(Slot,"new",n,storage[index(n)]);
        slots = send(slots, "cons", s);
      }
      return slots;
    }
  };
}
```

The storage for the new object is created in line 4 and initialised with values in lines 6-7. The new instance is created on lines 8-34 as a Java object that implements the `Obj` interface.

6. ADDING TYPES TO JMF

The previous section has defined two different dynamic MOPs using JMF. A dynamic MOP is a meta-class that redefines some of the operations that are provided by `Class`: **new**, `get`, `set`, or `send`. Since the MOP is dynamic, there is no scope for efficiencies that would be possible via static analysis of the JMF code. In order to achieve such efficiencies, the MOP must be statically available and be defined in terms of the syntax of JMF and its type system. This section analyses the key parts of the JMF and an associated type-checking relation that are used to define a static MOP.

6.1 JMF Expressions and Types

JMF is a Java-like language that contains constructs for class definitions, operations, commands and expressions. The key expressions that relate to a static MOP are defined below:

```
e ::= e.n                  slot access
  |   e.n := e             slot update
  |   e.n[n,...](e,...)    message passing
  |   ...                  more expressions
```

The syntax of slot access and update are straightforward, but the syntax of message passing includes an optional sequence of type names after the message name. This allows operations to be polymorphic. In particular it solves an issue with reflective languages whereby the type of an expression can depend upon its run-time value, for example if operation **new** defined by `Class` must return a value whose type depends upon the receiver, for example:

```
let c:Class = Point
in c.new().x
```

A question arises regarding the type of the expression `c.new()` since it depends upon the run-time value of `c`. A pragmatic solution is to require the programmer to supply the type when sending the message, therefore the type of the operation `new` defined by the class `Class` is: `Λ[t]()->t` and the type is explicitly supplied:

```
let c:Class = Point
in c.new[Point]().x
```

A JMF type `t` is a JMF class such as `Int`, `Class` and `Obj`. Three special types are used that are created by instantiating meta-classes:

`FunctionOf([t,...],t)` which denotes a function defined in terms of its argument types and its return type. This class is written `(t,...) -> t`

`ListOf(t)` which denotes lists whose elements are of type `t`. This class is written `[t]`.

`Λ[n,...]te` which denotes a type function whose body is the type expression `te`.

A type expression is defined as follows:

```
te ::= c                   a JMF class
   |   n                   reference to a named type
   |   [te]                a list type
   |   (te,...) -> te      a function type
   |   Λ[n,...]te          a type abstraction
```

Given a type environment ρ that maps names to types, a type expression `te` can be mapped to a type ρ(`te`).

6.2 The JMF Type-Check Relation

The JMF expression type-checker relates an expression `e` with a type `t` in the context of an environment ρ that maps names to types: $\rho \vdash$ `e : t`. The environment contains all of the core JMF classes and the user defined classes that are referenced by the expression. The type checker is defined by a collection of rules.

The rules are defined recursively on the structure of the expressions. The details of those rules that do not relate directly to a static MOP are not important, but consider a rule that types an if-expression:

$$\text{IF-1}\ \frac{\rho \vdash \texttt{e} : \texttt{Bool},\quad \rho \vdash \texttt{e1} : \texttt{t1},\quad \rho \vdash \texttt{e2} : \texttt{t2} \qquad \texttt{t1.inherits(t2)}}{\rho \vdash \textbf{if}\ \texttt{e}\ \textbf{then}\ \texttt{e1}\ \textbf{else}\ \texttt{e2} : \texttt{t2}}$$

The rule states that an if-expression has a type `t2` when the test expression `e` is a boolean and when the consequent and alternative expressions have types `t1` and `t2` respectively and when `t1` inherits from `t2`. Those rules that are relevant to the static MOP of JMF are defined below.

Slot access is defined by a rule named GET that uses an operation named `typeCheckGet` to calculate the type of the slot reference:

$$\text{GET}\ \frac{\rho \vdash \texttt{e} : \texttt{t} \qquad \texttt{t.typeCheckGet(n) = t'}}{\rho \vdash \texttt{e.n} : \texttt{t'}}$$

The default static JMF MOP for slot access is defined by `Class` as follows:

```
typeCheckGet(n:Str):Classifier =
 find a:Attribute in allAttributes() when a.name=n {
    a.type
 } else undef
```

Therefore the default type check behaviour for `e.n` selects the attribute `a` with the name `n` from those defined and inherited by `t.of()` where `t` is the type of `e`. The result is the type of the attribute `a.t`. If no attribute can be found with the appropriate name then the result is `undef` which denotes a type error.

Type checking slot update uses `typeCheckSet` and then ensures that the slot type and the value type conform:

$$\text{SET}\ \frac{\rho \vdash \texttt{e1} : \texttt{t1} \qquad \rho \vdash \texttt{e2} : \texttt{t2} \qquad \texttt{t1.typeCheckSet(n,t2) = t'} \qquad \texttt{t2.inherits(t')}}{\rho \vdash \texttt{e1.n := e2} : \texttt{t'}}$$

The default static JMF MOP for slot update is defined below:

```
typeCheckSet(n:Str,c:Classifier):Classifier =
  find a:Attribute in allAttributes()
  when a.name=n && c.inherits(a.type) {
    a.type
  } else undef
```

The final element of type checking is message passing which uses `typeCheckSend` to ensure that the argument types match. Note that we take into account the optional type arguments that can be passed to the type checker in case the type of the operation is polymorphic:

$$\text{SND}\ \frac{\begin{array}{c}\rho \vdash \texttt{e} : \texttt{t} \\ \rho \vdash \texttt{ei} : \texttt{ti} \\ \texttt{t.typeCheckSend(n,[}\rho\texttt{(n1),...,}\rho\texttt{(nk)],[t1,...,tn])=t'}\end{array}}{\rho \vdash \texttt{e.n[n1,...,nk](e1,...,en):t'}}$$

The corresponding definition of `typeCheckSend` in the class `Class` must find an operation with the appropriate name and where the types of the supplied arguments match those of the operation domain. In the following definition it is assumed that the type of the operation is of the form `Λ[n1,...nk].te`:

```
typeCheckSend(n:Str,ts,dom:[Classifier]):Classifier =
  find o:Operation in allOperations()
  when o.name = n && o.arity() = dom.length() {
    let t:Classifier = o.getType().apply(ts)
    in if ∀(t1,t2) ∈ dom × t.dom(). t1.inherits(t2)
       then t.range()
       else undef
  } else undef
```

7. IMPLEMENTING A STATIC MOP

The previous section has outlined JMF type-checking and defined the default static MOP. This section defines 3 static MOPs: section 7.1 extends the `Sparse` dynamic MOP; section 7.2 extends the `Array` dynamic MOP; section 7.3 defines a complete MOP as a JMF class.

7.1 Static Sparse MOP

The `Sparse` MOP does not need to redefine `typeCheckGet` or `typeCheckSet` since the default implementation can use the attributes. However, the pseudo-message `clear` must be caught since it will not exist as an operation that can be detected by the default message passing MOP:

```
typeCheckSend(n:Str,ts:[Classifier]):Classifier {
  if n = 'clear' && ts = []
  then self;
  else super.typeCheckSend(n,ts);
}
```

7.2 Static Array MOP

The `AType` meta-class is completed by defining the type checking operations. The definition of `typeCheckSend` does not need redefinition and is inherited from `Class`. Type checking slot access and update must be redefined since the attributes are implicitly defined via the `names` attribute:

```
typeCheckGet(n:Str):Classifier {
  if names.contains(n)
  then of().type;
  else undef;
}

typeCheckSet(n:Str,c:Classifier):Classifier {
  if names.contains(n) && c.inherits(of().type)
  then self;
  else undef;
}
```

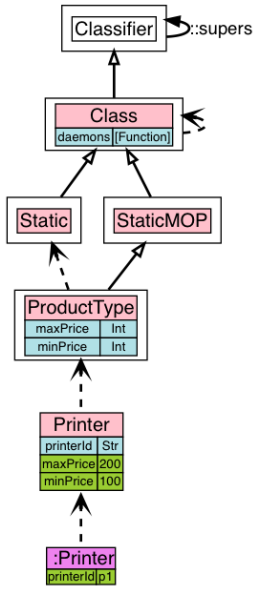

Figure 4: The Static MOP

7.3 A Self Contained JMF-Defined MOP

The previous section showed how to extend a dynamic MOP with static features in order to perform type checking. The examples of `Array` and `Sparse` were implemented using a mixture of Java and JMF code. The JMF language is intended to be self contained and this section shows how a full MOP that defines a new form of attribute that is equivalent to **static** in Java can be defined as a collection of JMF classes.

The motivation for this MOP is shown in figure 4 that shows a printer instance with id `p1`. All printers are priced in the same range so the min and max values are defined as slots on on the class `Printer` and are accessible as conventional slots via each instance such as `p1`.

In order to be a *static* class, `ProductType` is defined as an instance of the meta-class called `Static` which uses the following *mixin*:

```
class StaticMOP extends Class {

 typeCheckGet(n:Str):Classifier =
  find a:Attribute in of().attributes when a.name=n {
   a.type
  } else super.typeCheckGet(n);

 typeCheckSet(n:Str,type:Classifier):Classifier =
  find a:Attribute in of().attributes when a.name=n {
   if type = a.type
   then self
   else null
  } else super.typeCheckSet(n,type);

 get[T](target:Obj,n:Str):T =
  find a:Attribute in of().attributes when a.name=n {
   get(n)
  } else super.get(target,n);

 set(target:Obj,n:Str,value:Obj):Self =
  find a:Attribute in of().attributes when a.name=n {
   set(n,value)
  } else super.set(target,n,value);
}
```

The class `StaticMOP` defines a full MOP by allowing the meta-class slots to be included in type-checking, slot access and

slot update. The MOP is added as a mixin by the meta-class `Static`:

```
class Static extends Class {
  Static(name:Str,staticAtts:[Attribute]) {
    self.name := name;
    self.attributes := staticAtts;
    self.supers := [StaticMOP];
  }
}
```

Given the definitions above, the meta-class `ProductType` is defined as follows:

```
class ProductType metaclass Static {
  minPrice:Int;
  maxPrice:Int;
}
```

The class `Printer` is an example of a product type and each instance will have a different printer id but share the same max and min price.

```
class Printer metaclass ProductType {
  printerId:Str;
  minPrice = 100;
  maxPrice = 200;
  Printer(id:Str) { printerId := id; }
}

p1 := Printer('p1');
```

8. CONCLUSION

This paper has described the separation of a MOP into a *static*-MOP and a *dynamic*-MOP. The latter is a well-known approach to providing reflection and intercession in dynamic languages. The former is a new concept that is a pre-requisite to efficient compilation based on type information. The approach is similar to that described by [6] which introduces the idea of *exotypes* for abstracting away from data layouts in Lua, although exotypes are not fully integrated into the bootstrap as in JMF.

The code given in this paper is taken from an implementation of JMF that is based on the XMF language of the XModeler toolkit [4]. Unlike JMF, XMF is a compiled dynamic language that has no type-checker; the motivation for developing static MOPs in JMF is based on the limitations of XMF. JMF is under development; the current snapshot is at https://github.com/TonyClark/JMF where the bootstrap is defined in `value.Obj`.

Dynamic approaches to achieving efficiency in MOP-based languages include caching and *optimistic optimisation* [15]. These solutions can adapt when types are dynamically modified and this provides an interesting challenge for a static-MOP based approach. One option is to index code that is statically determined in a way that can be modified if the indexing types change: this is an area for further investigation.

This paper has demonstrated that static MOPs can be implemented in JMF. However, much work remains in order to achieve the overall aim: demonstrate that JMF can be compiled to the JVM and produce code that approximates the efficiency of Java. The static MOP will need to be extended with the ability to produce code in addition to type checking, and the JMF bootstrap must be completely amenable to static analysis.

9. REFERENCES

[1] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. Rpython: a step towards reconciling dynamically and statically typed oo languages. In *Proceedings of the 2007 symposium on Dynamic languages*, pages 53–64. ACM, 2007.

[2] K. Asai. Reflection in direct style. In *ACM SIGPLAN Notices*, volume 47, pages 97–106. ACM, 2011.

[3] G. Chari, D. Garbervetsky, S. Marr, and S. Ducasse. Towards fully reflective environments. In *Onward!*, 2015.

[4] T. Clark, P. Sammut, and J. S. Willans. Super-languages: Developing languages and applications with XMF (second edition). *CoRR*, abs/1506.03363, 2015.

[5] R. Damaševičius and V. Štuikys. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2), 2015.

[6] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. *ACM SIGPLAN Notices*, 49(6):77–88, 2014.

[7] R. Douence and M. Südholt. A generic reification technique for object-oriented reflective languages. *Higher-Order and Symbolic Computation*, 14(1):7–34, 2001.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes. The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1. ACM, 2007.

[9] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[10] P. Maes. Concepts and experiments in computational reflection. In *ACM Sigplan Notices*, volume 22, pages 147–155. ACM, 1987.

[11] N. Papoulias, M. Denker, S. Ducasse, and L. Fabresse. Reifying the reflectogram. In *30th ACM/SIGAPP Symposium On Applied Computing*, 2015.

[12] B. Redmond and V. Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *ECOOP 2002?Object-Oriented Programming*, pages 205–230. Springer, 2002.

[13] B. C. Smith. *Reflection and semantics in a procedural language*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1982.

[14] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 23–35. ACM, 1984.

[15] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[16] E. Tanter. Reflection and open implementations. Technical report, Citeseer, 2004.

[17] T. Verwaest, C. Bruni, D. Gurtner, A. Lienhard, and O. Niestrasz. Pinocchio: bringing reflection to life with first-class interpreters. *ACM Sigplan Notices*, 45(10):774–789, 2010.