# A Meta-Object Protocol for Parameterized Types

Andrew C. Myers
andru@lcs.mit.edu
MIT Laboratory for Computer Science*

## Abstract

An important feature of extensible object-oriented systems is a meta-object structure that describes a language's types and other system facilities. Parametric polymorphism is also coming to be recognized as an essential programming abstraction mechanism, which complements but is separate from subtype polymorphism and inheritance. This paper examines the meta-object protocol of Theta, a statically-typed object-oriented language, particularly focusing on how parameterized types are represented within this structure.

## 1 Introduction

Thor [6] is a persistent object system that provides access to client applications written in a variety of programming languages. The semantics of Thor objects are defined in Theta [1], a safe, statically-typed object-oriented programming language. In addition to subtype polymorphism and inheritance, Theta provides parametric polymorphism, an abstraction mechanism that is coming to be recognized as an essential language feature.

Thor also provides a meta-object protocol that explicitly manifests Theta types as objects. These *type objects* can be queried from the client or from within Theta code to obtain information about the type system, or to extend it. The type-object protocol makes Thor easily extensible, which is especially important in a system with long-lived persistent objects. The type-object protocol also supports important client applications: programming environments, compilers, automated interface translation between Theta and client languages, and other applications that dynamically add new types to the system. The ability to describe parameterized types is particularly important for interfacing to client languages with some support for parameterization, such as C++.

This paper examines Theta's type-object protocol, especially focusing on how parameterized types are represented in the protocol.

## 2 Basic Type Protocol

Types are represented in Thor by type objects: meta-objects that the compiler uses for type-checking and code generation. There are several different kinds of types in Theta. For example, it has primitive types like `int` and `char`, object types (which explicitly declare their subtype relationships), and procedure types (which implicitly participate in subtype relationships). Each of these types is a subtype of the universal supertype `any`.

All types are represented by type objects of type `Type`, though different kinds of types correspond to different implementations of `Type`. A type is described by a collection of *slots*, which for most types correspond to methods. The slot-based approach is similar to that in CLOS [2, 4], though as shown later, Theta also provides encapsulation. In Theta, the specification of `Type` appears as follows:

*545 Technology Square, Cambridge, MA 02139, USA. Phone: (617) 354-1575. This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

```
Type = type
  name() returns(string)
  unparse() returns(string)
  subtype(t: Type) returns(bool)
  slots() yields(slot)
end Type

slot = type
  name() returns(string)
  slot_type() returns(Type)
  mutable() returns(bool)
end slot
```

The name and unparse methods produce textual descriptions of the type, for debugging and browsing purposes. The subtype method is use to determine whether two types are in a subtype relationship. The slots method iterates over the operations supported by the type. For ordinary object types, these slots are methods; for a record type, these slots correspond to fields. A slot contains a name and a type, and also indicates whether it can be assigned to. Method slots cannot be assigned.

## 2.1 Object Types

There are several kinds of types in Theta, each implementing Type differently: object types, procedure types, record types. This paper will focus on object types.

For example, the type real_array is an abstract object type (Actually, array[real] would normally be used instead). Because real_array is abstract, it only defines an external method interface.

```
real_array = type
  length() returns(int)
  fetch(i: int) returns(real) signals(bounds)
  store(i: int, x: real) signals(bounds)
  elements() yields(real)
end real_array
```

An abstract object type is a member of the type ObjType. Because object types are a kind of type, the type ObjType is a subtype of Type. For object types, the slots are interpreted as object methods (which are not mutable) and the usual subtyping rules apply: an object may have multiple supertypes, and the supertype relation is transitive.

For example, the type object real_array, which is an instance of ObjType, will report that it has four immutable slots named length, fetch, store, and elements, all with procedure types. The type of the length slot is proc() returns(int). The type real_array does not appear in the method's signature, because the receiver object is implicitly curried as a part of method selection.

To facilitate browsing, ObjType extends Type with some methods, as shown in the following specification. The syntax "< Type" indicates that ObjType is a subtype of Type.

```
ObjType = type < Type
  methods() yields(slot)
  supertypes() yields(ObjType)
  renamings() yields(ObjType, string, string)
end ObjType
```

The methods iterator yields the methods directly declared in this object type, as distinguished from those methods which come from supertypes (slots will yield these methods as well). The supertypes method iterates over the explicitly declared supertypes of this object type; renamings reports what methods from the supertypes have been renamed.

This section has described how abstract object types interact with the type-object protocol; other kinds of types, such as record types, provide their own interpretation of the slots; instances of Record have slots that are mutable and correspond to fields in the record object.

## 2.2 Classes

Theta distinguishes between abstract object types, which have no associated implementation, and classes, which implement an abstract type. Code is written in terms of abstract object types, and classes are only used as types within their own implementations. Ordinary Thor objects are instances of classes, which define their physical representation and their method code. The representation is a set of instance variables that are only visible within the class implementation itself.

Classes are described by the type Class, which is a subtype of ObjType. This subtype relationship exists because a class, like an abstract object type, has a external method interface. It also has instance variables and private methods, but their encapsulation is guaranteed by exposing the instance variables and private methods as slots only while the class is being compiled.

For browsing purposes, the class can report its instance variables and method implementations directly, as well as an optional superclass from which it inherits part of its implementation.

```
Class = type < ObjType
  fields() yields(slot)
  privateMethods() yields(slot)
  superclass() returns(Class)
  methodImpl(m: slot) returns(any)
end Class
```

A class is understood to be an implementation of zero or more object types. These implemented types are yielded by the supertypes iterator.

Note that in Theta, object types do not describe creator routines. Objects are created by stand-alone procedures. This approach differs from Smalltalk-80, where object creation methods are added to the class object itself [3]. Extending class objects requires a new metaclass for every class in the system. Theta avoids this complexity without sacrificing the ability to guarantee that objects are only produced through the approved creators.

# 3 Parameterized Types

Statically-typed object-oriented languages often lack parametric polymorphism, with the result that they do not handle collection types gracefully. Here is an example of a parameterized set type in Theta, the semantics of which cannot be captured solely through subtype polymorphism:

```
set = type[T]
        where T has equal(T) returns(bool)
        < collection[T]
  % a mutable set
  insert(x: T) signals(duplicate)
  contains(x: T) returns(bool)
  equal(s: set[T]) returns(bool)
end set
```

The parameterized type set is not a type; it is a type generator that, for appropriate types T, generates the corresponding set types [5]. This process is called *instantiation*, and the individual types, such as set[int], are called *instantiations*.

A parameterized object type, such as set, is a member of the type PType. Because parameterized types are not types, PType is not a subtype of ObjType, even though it conforms to ObjType's signature:

```
PType = type
  name() returns(string)
  unparse() returns(string)
  methods() yields(slot)
  supertypes() yields(ObjType)
  instantiate(sequence[Type]) returns(Instn)
      signals(wrong_number_params,
              bad_parameter(Param))
  parameters() yields(Param)
end PType
```

The methods and supertypes methods act as a template for instantiation. The object types yielded by supertypes may be instantiations themselves, in which some parameters are taken from the parameterized type; for example, the supertype of set[T] is collection[T]. These supertypes are meaningless until the parameterized type is instantiated. Similarly, the slots yielded by methods may name types that depend on the parameters.

## 3.1 Type Parameters

A key to understanding PType is the pseudotype Param, which represents a type parameter and its associated where clauses. An example of a Param is the type T in the declaration

```
set[T] where T has equal[T] returns(bool)
```

No object is ever an instance of a Param; a Param is merely a useful placeholder for some other type. However, as far as the compiler is concerned, Param *is* a type. For example, in the following code, an operation is performed using a Param. The invocation of equal is achieved by a process different from ordinary method dispatch [7, 8].

```
  x,y: T
  if x.equal(y) then ... end
```

A type parameter has operations corresponding to its where clauses, and these operations are yielded by the slots iterator if invoked on the corresponding Param object.

A Param is defined by some parameterized type, and only has meaning in the context of that parameterized type. In addition to the usual type operations, a Param also can report the parameterized type that defined it:

```
Param = type < Type
  ptype() returns(PType)
end Param
```

Representing type parameters as types is advantageous when implementing the compiler, and also simplifies the type-object protocol for parameterized types.

### 3.2 Instantiation

As the specification of PType indicates, instantiation produces an Instn: a type like set[int], or set[set[int]]. In addition to being an abstract object type, an Instn also knows how it was instantiated:

```
Instn = type < ObjType
  ptype() returns(PType)
  pargs() returns(Type)
end Instn
```

### 3.3 Parameterized Classes

A parameterized class implements a parameterized type, providing a representation and method code. However, a parameterized class does not function as a class until instantiated. A parameterized class definition looks like the following, which implements set[T]:

```
set_class = class[T]
      where T has equal(T) returns(bool)
      implements set[T]
  rep: array[T]
  ...
  contains(x: T) returns(bool)
    for e: T in rep.elements() do % iterator call
      if x.equal(e) then return(true) end
    end
    return(false)
  end contains
end set_class
```

A parameterized class is represented by a member of the type PClass:

```
PClass = type < PType
  fields() yields(slot)
  privateMethods() yields(slot)
  superclass() returns(Class)
  methodImpl(m: slot) returns(any)
  instantiate(sequence[Type])
        returns(ClassInstn)
        signals(wrong_number_params,
                 bad_parameter(Param))
end PType
```

In addition to methods that correspond to their class equivalents (fields, superclass, methodImpl), PClass also specializes instantiate to return a ClassInstn: a parameterized class instantiation.

```
ClassInstn = type < Instn, Class
  pclass() returns(PClass)
end ClassInstn
```

A class instantiation acts both as an ordinary type instantiation and as a class. It is in implementing ClassInstn that most of the complexity of parameterization appears, since class instantiations must actually instantiate code. In Thor, code instantiation does not require duplicating code for each instantiation [8].

## 4 Creating New Types

As described in Section 3, new types can be added to the system by calls to the instantiate methods of parameterized type and class objects. New record types are created similarly, though this is not described here.

Creation of new parameterized type and classes, and of non-instantiated object types and classes, is performed by calls to the Compiler meta-object, which also is not described here.

Currently, the set of type kinds is fixed, and includes the object types described here, as well as record, oneof, and procedure types. Since Type is an ordinary abstract object type, it would seem possible for it to be reimplemented, adding a new kind of type to the system. Currently, this is not possible, because the compiler would not understand how to select and store the slots of this new type, and because there is no mechanism for creating a new object labeled with a user-defined type.

An interesting topic for future research is the addition of methods to the type object that specify how slots should be manipulated. Such a user-defined type kind would not be as efficient as the built-in Theta type kinds (which are about as fast as persistent C++ implementations [7]), but could provide different slot access semantics.

## 5 Conclusion

Thor type objects engage in a meta-object protocol that describes the behavior of Theta, a statically-typed object-oriented language with parametric polymorphism. The type-object protocol allows client applications written in a variety of languages to manipulate Theta types as first-class objects, and to add new types to the system.

The ability to manipulate type objects is important for a variety of important applications, including programming environments, compilers, automated interface and implementation translation between Theta and client languages, and other applications that dynamically extend the set of available data types.

## References


[1] Mark Day, Robert Gruber, Barbara Liskov, and Andrew C. Myers. Theta reference manual, 1994. To appear as memo of the Programming Methodology Group, Laboratory for Computer Science, MIT.

[2] L. DeMichiel and R. Gabriel. The Common Lisp Object System: An overview. In *Proc. of European Conference on Object-Oriented Programming*, 1987.

[3] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[4] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[5] B. Liskov and et al. *CLU Reference Manual.* Springer-Verlag, 1984.

[6] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management.* Morgan Kaufmann, San Mateo, California, 1993.

[7] Andrew C. Myers. Fast object operations in a persistent programming system. Technical Report MIT/LCS/TR-599, MIT Laboratory for Computer Science, Cambridge, MA, January 1994. Master's thesis.

[8] Andrew C. Myers and Barbara Liskov. Efficient implementation of parameterized types despite subtyping, July 1994. Thor Design Note #9. Also at: ftp::/pion.lcs.mit.edu/pub/thor/ptypes.ps.