Managed Data: Modular Strategies for Data Abstraction

Alex Loh
University of Texas at Austin
alexloh@cs.utexas.edu

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI)
storm@cwi.nl

William R. Cook
University of Texas at Austin
wcook@cs.utexas.edu

Abstract

Managed Data is a two-level approach to data abstraction in which programmers first define *data description and manipulation mechanisms*, and then use these mechanisms to define *specific kinds of data*. Managed Data allows programmers to take control of many important aspects of data, including persistence, access/change control, reactivity, logging, bidirectional relationships, resource management, invariants and validation. These features are implemented once as reusable strategies that can apply to many different data types. Managed Data is a general concept that can be implemented in several ways, including reflection, metaclasses, and macros. In this paper we argue for the importance of Managed Data and present a novel implementation of Managed Data based on *interpretation* of *data models*. We show how to inherit and compose interpreters to implement the features described above. Our approach allows Managed Data to be used in object-oriented languages that support reflection over field access (overriding the "dot" operator) or dynamic method creation. We also show how self-describing data models are useful for bootstrapping, allowing Managed Data to be used in the definition of Data Managers themselves. As a case study, we used Managed Data in a web development framework from the Ensō project to reuse database management and access control mechanisms across different data definitions.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data management

General Terms Data management, Aspect-oriented programming, Model-based development

Keywords Schema, Interpretation, Composition

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.

1. Introduction

Mechanisms for organizing and managing data are a fundamental aspect of any programming model. Most programing models provide built-in mechanisms for organizing data. Well-known approaches include *data structure definitions* (as in Pascal, C [16], Haskell [12], ML [20]), *object/class models* (as in Java [1], Smalltalk [9], Ruby [30]), and *predefined data structures* (as in Lisp [25], Matlab [13]). Languages may also support abstract data types (as in ML, Modula-2 [34], Ada [33]), or a combination of multiple approaches (e.g JavaScript [7], Scala [21]). A key characteristic of all these approaches is that the fundamental mechanisms for structuring and manipulating data are predefined. Predefined data structuring mechanisms allow programmers to create specific kinds of data, but they do not allow fundamental changes to the underlying data structuring and management mechanisms themselves.

Predefined data structuring mechanisms are insufficient to cleanly implement many important and common requirements for data management, including persistence, caching, serialization, transactions, change logging, access control, automated traversals, multi-object invariants, and bi-directional relationships. The difficulty with all these requirements is that they are pervasive features of the underlying data management mechanism, not properties of individual data types. It is possible to define such features individually for each particular kind of data in a program, but this invariably leads to large amounts of repeated code. To implement these kinds of crosscutting concerns, developers often resort to preprocessors [14], code generators [26], byte-code transformation [2], or modified runtimes or compilers [23]. The resulting systems are typically ad-hoc, fragile, poorly integrated, and difficult to maintain.

This paper presents *Managed Data*, an approach to data abstraction that gives programmers control over data structuring mechanisms. Managed Data has three essential components: (1) *schemas* that specify the desired structure and properties of data, (2) *data managers* that enable creation and manipulation of instances of data that conform to the data specification, and (3) *integration* with a programming

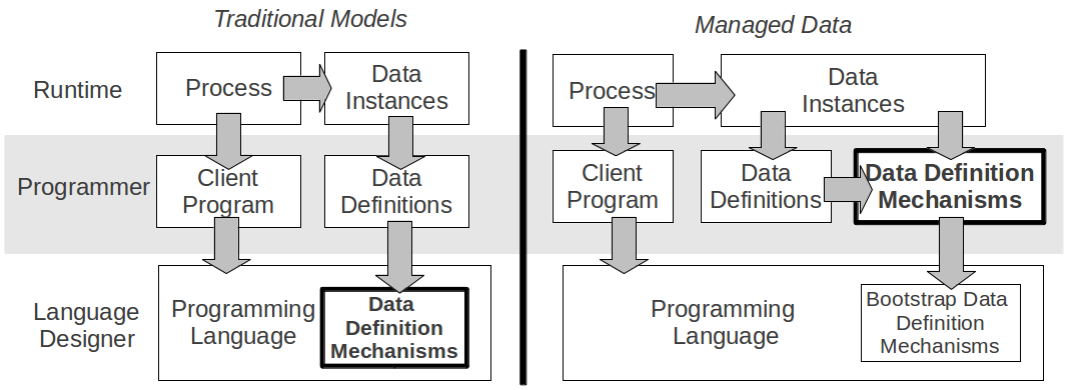

Figure 1. Traditional Data Mechanisms versus Managed Data

language, so that managed data instances are used in the same way as ordinary objects. Managed Data has a strong emphasis on modularity, allowing schemas and data managers to be modularly defined and reused. Additionally, schemas may also themselves be defined using Managed Data via a *bootstrapping* process, extending the benefits of programmable data structuring to their own implementation.

Figure 1 illustrates the difference between traditional built-in data structuring mechanisms and Managed Data. In the traditional approach, the programming language includes a process and data sublanguages, which are both predefined. With Managed Data, the data structuring mechanisms are defined by the programmer by interpretation of data definitions. Since a data definition model is also data, it requires a meta-definition mechanism. This infinite regress is terminated by a boot-strap data definition that is used to build the Managed Data system itself.

One way to understand Managed Data is as a *design pattern* that allows the programmer to define the behavior of data manipulation operations traditionally considered built-in primitives: initialization, field access, type tests, casting, and pointer equality.

This pattern can be implemented in many different ways in different languages, or in different programming styles, including object-oriented or functional. Some programming systems support a degree of control over the data structuring mechanisms. Meta-classes in Smalltalk define how classes are instantiated and compiled [9]. The reflective features of Ruby, Python and Smalltalk can trap and handle undefined methods and properties, allowing creation of dynamic proxies or virtual objects [9, 30, 32]. Attributes and bytecode manipulation can specify and implement pervasive data management behaviors in Java [2] Scheme macros are often used to create data structuring mechanisms [15]. For example, the `defstruct` macro defines mutable structures with a functional interface. The Adaptive Object Model Architecture [35] provides an architecture for this approach, but does not discuss how it is bootstrapped or integrated with existing languages. In general, static languages are less able to support Managed Data directly, so they require the use of external code generators. Dynamic languages often provide reflective hooks that can be used to implement Managed Data.

Our implementation of Managed Data uses the reflective capabilities of Ruby. Sections 2 and 3 implement Managed Data with dynamic proxies, which declare properties and methods on the fly using Ruby's `method_missing` mechanism. A second, more static implementation, introduced in Section 4, uses `define_method` to define instance methods as closures at run-time. Section 5 demonstrates the use of Managed Data in EnsōWeb, a web development framework. Managed Data is used to configure pervasive data management concerns such as persistence, security and logging in a data-independent way, without introducing boilerplate into the specific type definitions. Finally, Section 6 compares and classifies related work.

2. Example of Managed Data

The definition and use of *records*, or labeled products, provides a good initial example of Managed Data. Records are a built-in feature of many languages, including Pascal and ML. Managed Data can be used to implement similar functionality, although without static type checking. On the other hand, Managed Data supports dynamic checking of both types and other invariants. To implement records using managed data, it is necessary to define a schema language that describes record structures, define data managers that implement the appropriate record behavior, and also specify hooks into the programming language so that records can be created and used. The following sections first introduce a simple schema language, then discuss use of records, and finally implement a data manager.

2.1 Simple Record Schemas

Record schemas describe the structure of records, which are mappings from field names to a value of an appropriate type

for each field. A record schema specifies a class of records that have a given set of field names and types. In this section schemas are defined using Ruby hashes. More complex schema languages, including stand-alone languages, are introduced in Section 4. A schema that describes simple two-dimensional points is defined as a Ruby hash as follows:

```
Point = { x: Integer, y: Integer }
```

`Point` defines a *hash* in Ruby 1.9 syntax. The hash is an object that represents a mapping from values to values. In this case the keys of the hash are the symbols `x` and `y`. Both these symbols are mapped to the class `Integer`. Classes are values in Ruby, as in Smalltalk. Although this definition appears to be a *type*, it is actually just a value. One interpretation of this value as a kind of specification, or dynamically checked type, but it is important to keep in mind that `Point` is just a hash value.

`Point` describes records with `x` and `y` fields whose values are integers. `Point` is a simple example of a *schema*. It is easy to describe many different kinds of records using this simple notation. For example, information about persons can be described by the following record schema:

```
Person = { name: String,
           birthday: Date,
           erdos: Integer }
```

Schemas are not complete specifications without a corresponding data manager. In this case a record schema is a definition of a data type but does not state whether the records are immutable or mutable, whether they are stored in a database, or transformed in other ways. Schemas can be interpreted in many different ways to create different kinds of records.

2.2 Using Managed Data

Before showing how to implement managed data, it is important to consider how managed data is used. This study will generate the intuitions and requirements needed to guide the implementation.

The goal is to create objects that conform to the specification given by a basic record schema, and which can be easily used within a programming language. At the same time, the objects may have additional behavior, such as logging and access control, as defined by the data manager. Assume we have a data manager, `BasicRecord`, which enables creating and updating basic records, given a schema. Here is an example of how this data manager might be used:

```
p = BasicRecord.new Point
p.x = 3
p.y = -10
print p.x + p.y
```

In this example, the `BasicRecord` manager is instantiated and given the `Point` schema as an input. The result is a new point object, that is, an object that conforms to the point schema. The object `p` has fields `x` and `y` which can be accessed and assigned.

Attempting to violate the schema results in an error. Some examples are given below:

```
print p.z # unknown field z
p.x = "top"  # x must have type Integer
p.z = 3   # assigning unknown field z
```

The `BasicRecord` manager interprets the `Point` schema to manage points. The code illustrates that `BasicRecord` supports mutable fields, and that it checks types and validity of field names. These simple checks are typical of how data is managed in most object-oriented programming languages. Later sections consider data managers that implement a variety of features, including immutability, persistence, invariants, etc.

The fields of the managed data object are dereferenced using the "dot" operator, like in most object-oriented languages. For languages such as Java, C#, JavaScript, and PHP, this requires the class the object belongs to statically declare those fields or methods. However, since the schema that contains those fields is only known dynamically, the data manager must be able to determine the fields and methods of the managed data object dynamically.

`BasicRecord` *interprets* a record schema to create dynamic objects that act according to its specification. Exactly *how* the data manager is implemented has been left unspecified. It could work by code generation, reflection, byte-code manipulation, or other techniques. The next section presents a implementation based on dynamic method handling. Section 4.3 illustrates an alternative implementation based on dynamic method creation from closures. Both of these implementation techniques avoid any form of explicit code generation.

2.3 Implementing a Data Manager

A reflective data manager for records is defined in Figure 2. The class `BasicRecord` is defined as a subclass of `BasicObject`, a minimal base class that only defines primitive equality and some reflective methods.

The schema is passed to the constructor of the basic record, as shown in the previous section. The `initialize` method stores the schema in a member variable and creates an empty hash `{}` to store the field `values` for this record. Finally, it initializes the fields to default values appropriate for the field's type. `BasicRecord` does not allow fields to be undefined.

`BasicRecord` includes two generic getter and setter methods, `_get` and `_set`, which access and update a field by name. The `_get` method looks up the field name in the schema and returns an error if the field is not defined. If the field does exist, then the current value of the field is returned from the `@values` hash.

The `_set` method takes the field name and the new value as inputs. The `_set` method also checks that the field is

defined, and that the new value is of the appropriate type. If the field exists and the value has the right type, then the `_set` method updates the `@values` hash to store the new value.

The `_get` and `_set` method provide the necessary functionality to create and use records, but calling them explicitly is cumbersome. What is needed is for methods to be called implicitly when fields are accessed or updated.

2.4 Managed Data as Ruby Objects

To allow basic records to be used as if they were ordinary Ruby objects, `BasicRecord` uses the reflective capabilities of Ruby.

When an unknown method or property is accessed, or an unknown field is assigned, rather than raising an error immediately Ruby invokes `method_missing` on the object. The arguments to `method_missing` are the name of the undefined method and the arguments of the original call. For example, a call to a missing method `obj.m(3)` is converted into a call to `obj.method_missing(:m, 3)` where `:m` is a *symbol* representing the name of the method. Access `obj.field` to an undefined field is converted into a call to `obj.method_missing(:field)`. As a special case, field assignment `obj.field = val` is converted into a call of the form `o.field=(val)`, which then follows the normal rules for methods. The default `method_missing` method in `BasicObject` raises an error. But if a class overrides `method_missing` then it can perform any action in response to an unknown method, including returning normally.

Since `BasicRecord` does not define any ordinary methods `method_missing` is always called whenever a field access or assignment is attempted. `BasicRecord` defines `method_missing` to dispatch to `_get` or `_set`. The second formal argument `*args` of `method_missing` captures all remaining actual arguments as an array. The `method_missing` method first determines if the call is a field assignment by checking if the method name ends with `=`. If so, it calls `_set` to handle the field assignment, passing the field name (with `=` removed) and the new value. If not, it checks that there are no additional arguments and then calls `_get`.

One drawback of using `method_missing` is that it does not handle method name clashes elegantly. Specifically, if a field has the same name as an existing method, such as `_set` and `_get`, or one of the methods defined by Ruby's `BasicObject` class (e.g. `equal?`) then `method_missing` will call that method instead. To avoid such problems, programmers must avoid using field names that begin with an underscore or correspond to a method defined in `BasicObject`.

3. Alternative Data Managers

Managed Data drops the idea of a single predefined data manager and schema description language, and allows programmers to create or extend their own. Data managers naturally manage many schemas, but there can also be multiple data managers for a given schema to, for example, im-

```
class BasicRecord < BasicObject
  def initialize(schema)
    @schema = schema
    @values = {}
    # assign default values to all fields
    schema.each do |name, type|
      @values[name] = type.default_value
    end
  end

  # internal methods for getting and setting fields
  def _get(name)
    if @schema[name].nil?
      ::Kernel.raise "unknown field #{name}"
    end
    @values[name]
  end

  def _set(name, value)
    type = @schema[name]
    if type.nil?
      ::Kernel.raise "setting unknown field #{name}"
    end
    if not value.is_a?(type)
      ::Kernel.raise "#{name} must have type #{type}"
    end
    @values[name] = value
  end

  # all properties and methods are handled here
  def method_missing(name, *args)
    if name =~ /(.*)=/ # setters end with a '='
      name = $1.to_sym # $1 is name without trailing '='
      _set(name, *args)
    else # getter
      if args.length != 0
        ::Kernel.raise "getter must not have arguments"
      end
      _get(name)
    end
  end
end
```

Figure 2. A data manager for simple records

```
class LockableRecord < BasicRecord
 def _lock
  @locked = true
 end

 def _set(name, value)
  if @locked
   ::Kernel.raise "Changing {name} of locked object"
  end
  super
 end
end
```

Figure 3. A lockable data manager

```
class InitRecord < LockableRecord
 def initialize(schema, init)
  super(schema)
  # assign default values to all fields
  init.each do |name, value|
   _set(name, value)
  end
  _lock()
 end
end
```

Figure 4. A data manager with field initialization

plement an in-memory versus a database-based strategy for storing the data. Data managers may be composed to form a stack of managers that has their combined behavior.

This section presents a few alternative data managers that enhance the basic data structuring mechanism provided by object-oriented programming languages.

3.1 Immutability

The data manager in Figure 3 introduces a locking mechanism to protect a record from changes. This is useful for certain types of optimizations or to implement constant values. `LockableRecord` inherits from the default data manager and overrides its `_set` method to check if the record is locked before invoking the original `_set` via **`super`**. In this implementation locking is irrevocable, but it would be easy to include an `_unlock` option.

3.2 Instance Initialization

Constant objects are initialized at the beginning of their lifespan and immutable henceforth. The data manager in Figure 4 extends `LockableRecord` with the option to initialize fields during construction. The constructor parameter `init` is a map from field names to initial values and can be used as follows:

```
p = InitRecord.new Point, x: 3, y: 5
```

```
class ObserverRecord < BasicRecord
 def initialize(schema)
  super
  @_observers = ::Set.new
 end

 # add an observer to this record
 # &block is a lambda expression
 def _observe(&block)
  @_observers.add(block)
 end

 def _set(name, value)
  super
  @_observers.each do |obs|
   obs.call(self, name, value)
  end
 end
end

class DataflowRecord < ObserverRecord
 def _get(name, dependent=nil)
  if not dependent.nil?
   _observe do |obj, field, value|
    if field == name
     # inform dependent
     dependent.set_dirty
    end
   end
  end
  super(name)
 end
end
```

Figure 5. A data manager for the Observer Pattern

We extended `InitRecord` from `LockableRecord` because we wanted to build constant objects, but in general immutability and initialization are orthogonal concepts that can apply independently.

3.3 Observers

Figure 5 presents a data manager that supports the OBSERVER PATTERN [8]. The following code snippet logs changes to the record by printing out a message whenever a point is changed.

```
p = ObserverRecord.new Point
p._observe do |obj, field, value|
 print "updating #{field} to #{value}\n"
end
p.x = 1
p.y = 6
p.x = p.x + p.y
```

```
class Schema          | class Field      | class String
  classes: Class*     |   name: String   | class Bool
class Class           |   type: Class    |
  name: String        |   many: Bool     |
  fields: Field*      |                  |
```

Figure 6. A minimalist self-describing schema

Output:

```
updating x to 1
updating y to 6
updating x to 7
```

An observer is a useful pattern especially for event-driven tasks such as enforcing inverses and dataflow programming. `DataflowRecord` is a record that allows callers who access a particular field to register themselves as dependents. Dependents will be told to re-compute their cached values when the value in that field changes.

Compared to `LockableRecord` and `InitRecord`, `ObserverRecord` and `DataflowRecord` demonstrate another kind of dependency where one data manager rely on the services provided by another. These examples expose the need for a general strategy for combining data managers such that dependencies between data managers are respected and independent data managers can be selected modularly.

4. Self-Describing Schemas

A self-describing schema is a schema that can be used to define schemas (including itself). Self-describing schemas are important because they allow schemas to be managed data. The concept of self-description is well known. The Meta-Object Facility (MOF) meta-metamodel [22] is self-describing. It is possible to write a BNF grammar for BNF grammars. Rather than starting from an existing meta-model, we will first develop what we believe to be a minimal self-describing schema, and then present a more complete and useful version based on this foundation.

4.1 A Minimalist Schema Schema

In the previous sections, the schema was a simple mapping from field names to primitive types, which can describe the structure of simple records. This simple schema format cannot be used to describe itself, because a simple schema is not a record. To model the structure of a schema, we need to be able to describe a record type as a collection of fields, each of which having a name and a type. This immediately requires a schema to have two concepts, namely "type" and "field". A third concept arises, a "schema", as a collection of types. Finally, we must recognize that some fields are single values, for example the name or type of a field, while other fields are many-valued, including the fields of a type and the types in a schema. These concepts are represented in the minimalist self-describing schema shown in Figure 6.

```
class Schema
  types! Type*

class Type
  name# str
  schema: Schema / types

class Primitive < Type

class Class < Type
  supers: Class*
  subclasses: Class* / supers
  defined_fields! Field*
  fields: Field*
    = supers.map() {|s|s.fields} + defined_fields

class Field
  name# str
  owner: Class / defined_fields
  type: Type
  optional: bool
  many: bool
  key: bool
  inverse: Field? / inverse
  computed! Expr?
  traversal: bool

primitive string
primitive bool
```

Figure 7. An Ensō Schema Schema

In this notation, a *class* is introduced by the keyword **class** followed by the class name and then a list of *field definitions*. Each field definition has the form *name*:*type* giving the name and type of the field. A *type* is a class name optionally followed by *, which indicates that the field is many-valued, i.e. a collection of values of the given type.

In the example, the class `Schema` has a single field, `classes`, which is a collection of `Class` values. A `Class` has a `name` field of type `String` and a collection of `fields`. A `Field` has a `name`, a `type`, and a boolean flag indicating whether the field is single or many-valued. To be complete, it is necessary to define `String` and `Bool` as classes.

This explanation of the content of the schema also demonstrates why it is self-describing, because every concept used in the explanation is included in the definition. One small point is that the type of a field is a `Class` in the schema, but the type is written as a name in the figure. During parsing or interpretation of the textual presentation of the schema, the named must be looked up to find the corresponding `Class`.

While this minimalist schema could be used directly, we believe that it is more useful to work with a slightly more complex self-describing schema. There are two major prob-

lems with this schema: (1) it does not distinguish between primitive classes (e.g. `String`) and structured classes (e.g. `Field`), and (2) it does not support *inverse* relationships. Primitives could be distinguished by adding a boolean flag to the class `Class`, but we find it more natural to introduce a structural distinction between the concept of a `Class` and a `Primitive`. These are two specific cases of the general notation of a `Type`. Representing this concept in the schema requires introducing a type hierarchy, or inheritance, into the schema.

4.2 The Ensō Schema Schema

Figure 7 defines a condensed version of the Schema schema used in Ensō. It is similar to the minimalist schema, but Ensō introduces several new concepts:

1. A class definition can include a list of superclasses, written `<` *classes*. In this schema, `Primitive` and `Class` are subclasses of `Type`, and `Type` is used in where `Class` was used in the minimalist schema. Multiple inheritance is allowed but no two ancestors may define a field with the same name.
2. A field type can include an *inverse* field, written *type* `/` *field*. The inverse field must be a field in the class given by the type. The schema introduces three inverses: the `schema` for a class is the schema that it belongs to, the `owner` of a field is the class it belongs to, and the `inverse` of a field is the field that it is an inverse of.
3. A field can be *computed*, written `=` *expression*. A computed field cannot be assigned. A single computed field is used to implement inheritance! The `fields` of a class are computed as the union of the fields of all its superclasses, combined with the fields that are defined on the subclass. The `defined_fields` field contains only the fields directly defined in a class.
4. Many-valued collections are marked with a $*$. Orderedness in many-valued fields is implicitly defined by whether the type of the field is keyed. By default, collections containing keyed objects are unordered hash tables while unkeyed objects are ordered indexed arrays. A single-valued field may be *optional*, indicated by `?`. Inverses and computed expressions are optional while defined fields in classes are unordered many-valued collections.
5. A field can be marked as a key with #, which forces its value to be unique within collections of the field's class. The `name` fields in `Class` and `Field` are both marked as keys, so the schema cannot have duplicate class names, and a class cannot have duplicate fields.
6. A field can be marked as a *traversal* with `!` after its name. Traversal fields delineate a distinguished minimum spanning tree called a *spine*. Spines provide a standardized way to view the object graph as a tree, avoiding inconveniences such as returning different result for different implementations of a depth-first search. Additionally, the spine is used to a derive unique, canonical address for each object in the graph by tracing its path from the root. In practise, traversal fields often loosely correspond to *composition*, or 'is a part of', relationships, but they do not necessarily have to be.

Additional properties are added to `Field` and `Class` to represent these new capabilities. There are many possible self-describing schemas, and Ensō does not stipulate that one must be used over another.

4.3 The Ensō Data Manager

Figure 8 defines the data manager used in Ensō, called a *factory*, and Figure 9 shows the managed object it creates.

`Factory` has only one method, `_make`, which it uses to build a `ManagedObject`. One convenience method is defined for each type in the schema to create managed objects of that type. Managed data object created by the factory generally indistinguishable from ordinary Ruby objects. If desired, the factory can completely replace class definitions without methods. Note also that even though `Factory` refers to `ManagedObject` by name here, in the actual implementation it uses the PROTOTYPE PATTERN [8], so the factory can add data managers to its private copy of `ManagedObject` without polluting the shared copy.

`ManagedObject`'s constructor takes a type and a set of initial values. Fields are set to an initial value if available or else the default value. Collection fields are set to empty lists. Note that the snippets shown here are stripped down versions of our actual implementation, in particular we omitted code listings for `ManyField` and `ManyIndexedField`, which handle collections and can themselves be overridden by other data managers. Just like in earlier examples, `ManagedObject` uses two methods, `_get` and `_set`, to manipulate the underlying data. `ManagedObject`'s `_get` method checks if its field is a computed field and evaluates the computed expression in the context of its current object if so. The `_set` method perform a simple check on types and implements the update notification system described in 3.3. Update notification is used to maintain inverses.

The implementation of this data manager improves on the earlier examples in two ways. Firstly, instead of using `method_missing`, `ManagedObject` defines methods for field access and assignment directly in its constructor. This avoids the problems with name clashes between field access methods and Ruby `Object` methods. Secondly, `Factory` and `ManagedObject` both define their methods within *modules*. Modules in Ruby implement mixin inheritance, a feature that is also present in languages like Smalltalk, Python and Scala. Mixins allow any arbirary combination of data managers to be selected, forming a 'stack' of data managers with their combined behavior. Normal inheritance does not work as the inheritance chain needs to be predefined. In the earlier examples, we could not define a data manager with

```
class Factory
  module FactoryBase
    def initialize(schema)
      @schema = schema
      # create object methods
      schema.classes.each do |c|
        _create_methods(c.name)
        define_singleton_method(c.name) do |*inits|
          _make(c.name, *inits)
        end
      end
    end

    # create a new object of type 'name'
    def _make(name, *inits)
      ManagedObject.new(@schema.classes[name], *inits)
    end
  end
  include FactoryBase
end
```

Figure 8. Data manager for any schema

both `ObserverRecord` and `LockableRecord` since they both must inherit `BasicRecord`. Languages that do not support mixins, such as Java, can use the DECORATOR PATTERN [8] if all data managers share the same interface. Refering to our earlier examples from Section 3, `InitRecord` and `LockableRecord` can be re-written as decorators on the basic record, but that will not allow the `_observe` method in `ObserverRecord` to be overridden. Alternatively, data managers can also be implemented using some form of metaprogramming or by explicitly passing around a `this` reference, depending on how much boilerplate is tolerated.

4.3.1 Bootstrapping

The Schema schema is itself Managed Data, and a bootstrapper is used to load the first Schema schema into memory.

Figure 10 summarizes the relationships between the different levels of schemas and data managers. At the lowest level, data objects such as points are described by the Point schema. This schema is managed by a data manager capable of initialization, allowing points to be created with starting values. The Point schema is in turn described by the Schema schema. The Schema schema is self-describing, following the spirit of modularity, we bootstrap the Schema schema from the minimal bootstrap schema that has only classes and fields. This minimal bootstrap schema is necessarily self-describing as it must manage itself, and it possesses a simplistic data manager that only allow updating. It is also hardcoded. Bootstrapping from a minimal schema allows us to customize even the Schema schema in very fundamental ways.

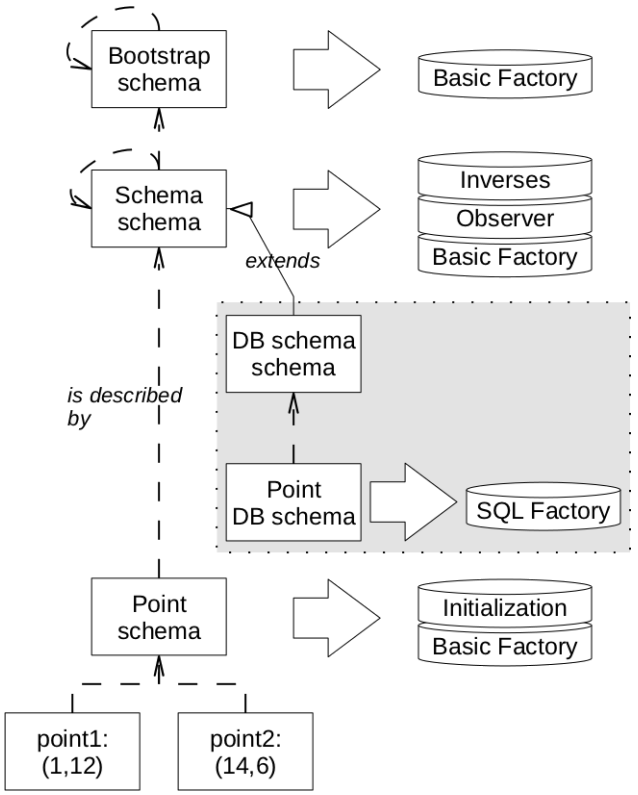

Figure 10. Bootstrapping in Ensō

This is not the only path the schema can take, however. Some data managers may require additional information from their schema. For instance, a data manager with support for relational databases will require the schema to provide a mapping from classes and fields to table and column names, as well as additional information on indices and keys. In the diagram, Database schema extends Schema schema so that DB Point schema can provide it with the relevant fields. Note that even though it is different from Point schema, DB Point schema is still able to describe Point objects, so it is possible to migrate them from one data manager to another.

5. Case Study: EnsōWeb

We have used Managed Data to build EnsōWeb, a web development framework. EnsōWeb loosely follows the Model-View-Presenter architecture and comprises a number of DSLs for expressing data models, web interfaces, and business logic such as security policies. In this section we are primarily concerned with how data models are managed. This part of EnsōWeb is analogous to ActiveRecord in Ruby on Rails [30] or Java's Hibernate [2]. The data model manager can take on a few different roles:

- At its core, the data manager is expected to perform standard data modeling functions such as enforcing inverses and cardinality constraints. This basic manager is

```
class ManagedObject
  module MObjectBase
    attr_reader :schema_class
    def initialize(schema_class, *initializers)
      @schema_class = schema_class; @values = {}
      # initializes object with values where available
      schema_class.fields.each do |field|
        init = initializers.shift # shift pops the leftmost element of an array
        if !field.many
          @values[field.name] = (init!=nil ? init : field.type.default_value)
        else
          # create the appropriate collection
          if (key = ClassKey(field.type))
            @values[field.name] = ManyIndexedField.new(key.name, self, field)
          else
            @values[field.name] = ManyField.new(self, field)
          end
          if !init.nil?
            init.each {|x| @values[field.name] << x} #initialize values
          end
        end

        # create convenience accessor methods for this field
        define_singleton_method(field.name) { _get(field.name) }
        define_singleton_method(field.name+"=") {|new| _set(field.name, new) }
      end
    end

    def _get(name)
      field = @schema_class.fields[name];
      raise "Accessing non-existant field '#{name}'" unless field
      return field.computed ? _eval(field.computed, ObjectEnv.new(self)) : @values[name]
    end

    def _set(name, new)
      field = @schema_class.fields[name]
      raise "Assign to invalid field '#{name}'" unless field
      raise "Can't assign field '#{name}'" if field.computed || field.many
      check_type(field.type, new) # check_type not shown in this snippet
      if @values[name] != new
        _notify(name, new) # notify observers that this field has been changed
        @values[name] = new
      end
    end
  end
  include MObjectBase
end
```

Figure 9. Data manager for a schema class

very similar to the Ensō data manager presented in Section 4.2.

- The data manager may optionally be required to implement low-level security. In turn, security failures may need to be logged.
- The data manager needs to support different backends for persistence depending on the requirements of the application. Possible options include in-memory, SQL, and XML databases.
- Versioning may be needed for backup and recovery.

For the most parts, the data manager in EnsōWeb provides similar services to Hibernate and Active Records, the chief difference being that Managed Data, as used in EnsōWeb, allows data management to be specifically tailored for an application (e.g. adding security constraints where none existed before), while ActiveRecord and Hibernate both only allow configuration within a fixed framework (e.g. defining a different set of validation rules). Managed Data also takes a modular approach to data management, allowing different predefined managers to be selected and composed.

Note that *all* of EnsoWeb is built around Managed Data, so models for web pages, logging, access control all have their own data managers performing generic tasks like versioning, merging, and validation, although they are not shown here.

5.1 Security

The optional security module extends the basic data manager and administers *policy-based access control* [5]. There are three parts to this module. The security specification language allows the administrator to define permissions based on the state of the system, the current logged-in user or his role, the operation requested (i.e. create, read, update, or delete) and object to be operated on. Internally, its security policy is a set of user-defined rules on a predicate. The data manager then interprets this security language to allow or deny any operation on the data model, denied reads return an empty record while denied writes are silently dropped. Finally, the managed data object can then be used in the client code without additional boilerplate and with access control transparently enforced underneath, as shown below.

```
Policy.auth:
deny read(s:Student) if s.grade == 'F'
allow update(s:Student{grade}) if s.section == 1

SecTest.rb:
# Data:
#  {name: 'Alice', grade: 'A', section: 1}
#  {name: 'Bob', grade: 'B', section: 2}
#  {name: 'Cathy', grade: 'F', section: 1}

for s in students
  print s.name
end
# Output: Alice, Bob

students['Alice'].grade = 'A+'
print students['Alice'].grade
# Output: 'A+'

students['Bob'].grade = 'A+'
print students['Bob'].grade
# Output: 'B'
```

`Policy.auth` is written in the security language defined using Ensō, while `SecTest.rb` is a snippet from the code to access secured Managed Data objects. The policy file states that students with an 'F' grade cannot be read and the grades of students can be modified only if they are in section 1. In the example, `Cathy` is hidden when iterating over the list of students since she has a failing grade. Likewise, attempts to modify `Bob`'s grade is ignored because of his section.

```
module SecureObject
  def _set_user(user); @user=user; end
  def _check(op, obj, field=nil)
    # returns false if operation is not allowed
    # implementation omitted for brevity
  end
  def _get(name)
    if _check("Read", self, name)
      result = super
      _check("Read", result) ? result : nil
    else
      nil
    end
  end
  def _set(name, new)
    super if _check("Update", self, name)
  end
end
module LoggedSecureObject
  include SecureObject
  def _check(op, obj, field=nil)
    auth = super
    # write error to log file
    log.write "Security fail: #{op} #{obj}" if !auth
    auth
  end
end

class ManagedObject
  include SecureObject
  include LoggedSecureObject
end
```

Figure 11. Security extension for the data manager

Figure 11 shows how to implement such a secured data manager. Like in previous examples, `_get` and `_set` are overridden, this time to perform security checks before reading and writing. `SecureObject` introduces a new method, `_check`, which is in turn overridden by `LoggedSecureObject` to record any attempts to access unauthorized data.

Low-level security at the data model is the second line of defense to augment security at the user interface level, which is still needed to control the behavior of on-screen widgets and to provide meaningful error messages. The capability to filter out unpermitted records from the result of reads, either to return a shortened list of records or a null object, and to block write attempts to unpermitted records, is built into the data manager.

5.2 Persistence Layer

There are several alternatives for persisting data depending on the needs of the application: the relational databases (RDBMS) used for this example, in-memory, and XML-based options are the most common choices. Even among databases, the need for configurable data management exists [27]. Domain-specific SQL dialects for streams and sensor networks, different indexing schemes like B-trees and R-trees, OLAP cubes, query language features like stored procedures, recursive queries, views, and biases between read and write operations, are some of the many decisions database designers need to make. Managed Data facilitates switching between these choices by making persistence a part of the data structuring mechanism and largely independent of the client code.

Figure 12 presents a data manager that persists the managed data object with a RDBMS backend. Instead of using a hash table to store values like before, `DBFactory` creates a new database for this schema when it is initialized. In this database, classes are mapped to tables and fields to columns. Many-to-many relationships have to be transformed to use a junction table. Incidentally, many-to-one relationships without an inverse also use a junction table because the target table does not have a column to serve as a back pointer. Another thing this data manager needs to do is add a `key` field to every class, because unlike in-memory objects that are identified by their memory reference, database tuples can only be uniquely identified by their primary keys.

This RDBMS data manager is an example of coupling data managers with schema extensions. Because EnsōWeb is running on top of a relational SQL database, the schema of the data model needs some way of associating classes to tables and fields to columns and specifying a name for the schema and root table. Figure 13 the additions to the schema used by the database manager. This schema extension is merged into the schema schema via *overriding union* to produce the DB schema schema, which allows schemas to specify table and column names. Note that only the schema that describes the managed data is changed, the managed data objects themselves are the same regardless of which scheme is used.

```
module DBFactory
 def initialize(schema, params={})
  # create a new database
  SQLexec("create database #{schema.name};")
  SQLexec("use #{schema.name};")
  @schema = schema
  # create object tables
  schema.classes.each do |c|
   str = "create table #{c._table_name} ("
   str += "DBKey int,"
   c.fields.each do |f|
    if !f.many
     if f.type.Primitive? # single−valued prim
      str += "#{f.name} #{convert(f.type)},"
     else # single−valued reference
      str += "#{f.name} int,"
     end
    else
     if f.inverse.nil? # 1−to−many (no inv)
      # create a junction table
      junction(f._field_name, c._table_name,
            f.type._table_name)
     elsif !f.inverse.many? #1−to−many
      # do nothing, resolved by inverse
     else #many−to−many
      # create a junction table
      junction(f._field_name, c._table_name,
            f.inverse.ownder._table_name)
     end
    end
   end
   str += ")"
   SQLexec(str)
  end
 end
end
class Factory
 include DBFactory
end
```

Figure 12. Data manager for an RDBMS backend

5.3 A Family of Data Managers

By selecting the relevant data managers, the programmer can dynamically construct managed data objects with the desired set of behavior. Ideally, the different data managers should be orthogonal and not interact with each other. This is true in many cases such as the security and versioning data managers. However, in practice data managers have dependencies on other data managers whose service they rely on. There might also be unanticipated interactions between

```
class Schema
  root: str?
  root_class: Class = classes[root]

class Class
  table: str?

class Field
  column: str?

primitive str
primitive bool
```

Figure 13. DB schema extension for specifying table and field names

data managers. For instance, when using a secured manager backed by a relational database, the generated `key` field must be made readable by all users who have access to the object, as this is the only way to identify records in the database. This has to be resolved by applying a patch that adds security permissions for the new field.

6. Related Work

One of our goals in this work is to provide a name and a better understanding for programming practices that have been used for many years, in many different forms. These existing approaches to managed data can be classified along several dimensions.

- The first, and most important, is *how the schema is defined*. The schema is the metadata that describes the structure and behavior of the data being managed. The schema can be defined as *external data*, *program data*, or *class attributes*. External data refers to any data outside the program, whether an XML file or a special schema file. Internal program data is dynamic data that is instantiated within the program code, for example a Lisp S-expression. A third way is to define not define the schema explicitly, but to derive them by using reflection on the program itself. The program might also be annotated with class-based metadata to define additional attributes not normally part of the language. Alternatively, the schema could also be defined by a combination of these approaches.
- The second dimension is whether the data manager is implemented by *code generation* or *interpretation*. Code generators can either generate program source code or byte-code instructions. Interpretation is characterized by a lack of explicit manipulation of code syntax. Thus creating closures does not count as code generation.
- The final dimension is whether on not the client language, which manipulates the managed data, is *statically typed*. In some cases the typing is a *hybrid* of static checking with dynamically generated type information.

The following table summarizes the related work relative to these criteria:

System:	Schema	Implementation	Typing
Ensō	External	Interpreted	Dynamic
AOM	External	Interpreted	Dynamic
MOP	Internal	Generated code	Dynamic
Macros	Internal	Generated code	Dynamic
Type Providers	*	*	Hybrid
RDBMS	External	Interpreted	Dynamic
Ruby on Rails	Reflection	Generated code	Dynamic
MorphJ	Reflection	Generated code	Static
IDispatch	*	*	Hybrid
EMF	External	Generate code	Static

The Adaptive Object-Model (AOM) Architectural Style [35] is closely related to Managed Data. This architectural style is a "reflective architecture" or "meta-architecture" because the actual code of the system does not define the behavior and properties of the domain objects and business rules that are manipulated by the system. Instead these domain objects and rules are defined by explicit metadata, which is interpreted by the system to generate the desired behavior. There is generally no static type checking of the resulting system. In some ways the Adaptive Object-Model style is more general than Managed Data, because it is described at a very high level as a pattern language and it also covers business rules and user interfaces, in addition to data management. On the other hand, the Adaptive Object-Model does not discuss issues of integration with programming languages, the representation of data schemas, or of bootstrapping, which are central to Managed Data. The Adaptive Object-Model is also presented as a technique for implementing business systems, not as a general programming or data abstraction technique.

The languages in the Lisp family, including Scheme [15], have a long tradition of supporting user-defined data abstraction mechanisms. The `defstruct` macro is a widely used and has many variations and options, including mutability, serialization, pattern matching, and initialization [25]. A variant, the `define-type` macro, supports immutable sums-of-products [19]. Despite these prominent examples, it is not clear how widespread the practice of Manage Data is in Lisp-based languages. The `defstruct` macro is generally presented as a standard feature of the language or as part of the standard library, rather than an example of a general concept that users might practice themselves.

Object-oriented extensions of Lisp are often implemented within Lisp itself using macros or other encodings [4, 24, 29]. This approach was developed in a pure form in *The Art of the Metaobject Protocol* (MOP) [18]. The principles underlying MOP are the same as those underlying Managed

Data: that the programmer should be able to control the interpretation of structure and and behavior in a program. In the case of MOP, this focus is on behavior of objects and classes, while in Managed Data the focus is more narrowly defined as data management. The MOP approach is general enough to include Managed Data as a special case.

F# 3.0 [28] has recently introduced *type providers* as a new mechanism for accessing and manipulating external data. Type providers are a form of Managed Data, because the type provider defines the structure and behavior of data values that appear as native data types in F#, but are in fact virtual values that can be drawn from any source. Type providers are a very interesting example of Managed Data, because they provide semi-static access to dynamic data from a strongly-typed functional language. The access is semi-static because the types provided by the provider can change between the time the program is compiled and when it is executed.

Relational Database Managed Systems (RDBMS) can be viewed as providing a form of Managed Data, where the SQL Data Definition Language (DDL) defines a schema. The RDBMS interprets the schema to create tables, which can then be accessed in SQL queries. Managed Data imports this approach from databases into the core of a programming language. One of the key issues with RDBMS has been the problem of integration with existing programming languages.

6.1 Class-based Metadata

One common approach to managed data is to extend an existing object-oriented language with attributes, which are interpreted to add additional behaviors to the class instances. In statically typed languages, the attributes are usually processed by a compiler or code generator, while in dynamic languages the attributes can be interpreted dynamically.

Ruby on Rails [31] implements a form of Managed Data. The schema information comes from metadata that is attached to class definitions. Ruby allows the information in a class definition to be extended easily, allowing many kinds of metadata to be included. These metadata attributes are interpreted at runtime. The Ruby on Rails engine can be viewed as a kind of data manager.

Hibernate [2] implements a restricted form of Managed Data for Java, but uses byte-code manipulation and code generation rather than dynamic interpretation. Hibernate is best understood as a specific data manager that supports binding to a relational database, rather than a general system for Managed Data.

MorphJ [11] is a system for compile-time transformation of class definitions. A source class provides metadata for a user-defined generation of a new class definition. One of the advantages of MorphJ is that the transformations are statically typed. However, using a class as metadata is more restrictive than other systems which allow arbitrary schema definitions.

6.2 Proxies

Many languages support a form of *dynamic proxy* that can be used to implement Managed Data. Proxies have long existed in dynamic languages. The Data Managers in Ruby defined in this paper use a form of dynamic proxy. A similar implementation is possible in Smalltalk. More recently, JavaScript 1.8.5 has proposed a Proxy API which provides similar functionality. Proxies are also possible in statically-typed languages, including Java [6] and C#, although they generally cannot implement the full range of Managed Data features as described in this paper. Rather than wrapping an existing object, a proxy can also be used to implement a new object dynamically. The main problem with this approach is that the interfaces for the managed objects must be predefined.

The `IDispatch` interface in Microsoft COM is a powerful tool for implementing Managed Data. It provides two main operations, `GetTypeInfo` and `Invoke`. The first operation returns a (possibly dynamically generated) description of the operations that are possible on the object. The `Invoke` operation allows operations to be invoked dynamically by passing an operation identifier and a list of parameters. Visual Basic, VBScript and JScript all provide special support for invoking COM objects that implement the `IDispatch` interface. The syntax `o.m(args)` is automatically converted to a call to `Invoke`.

6.3 Data Modeling

There is a long history of developing high-level data modeling languages and notations externally. While the concept of Managed Data is independent of the particular data description language being used, it is worth noting that the Ensō Schema schema is closely related to a large body of existing work. Examples include the Semantic Data Model [10], Entity-Relationship modeling [3], and Eclipse Modeling Framework's (EMF) [26] ECore model.

EMF [26] is a toolchain for model-driven engineering (MDE) in Java for Eclipse. MDE is a programming paradigm based on definable data descriptions specialized for a specific application domain. Their data models are managed, with native support for inverse and cardinality constraints. EMF uses the data description, called an ECore model, to generate code for commonly used scaffoldings, such as persistence and logging.

Model-driven engineering share our goal of reusing data management services across different data descriptions, however, in most implementations, including EMF, data management is configurable only as part of the tool, whereas in Managed Data data managers can be configured programmatically. The other key difference is that the EMF's meta-metamodel, while self-describing, is fixed and cannot ever be changed. In comparison, the Schema scheam in Ensō is programmer-definable and attributes can be added depending on usage, as demonstrated in section 5.2.

6.4 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [17] enables compiler support for injection of code at quantified join points. While AOP is not directly related to the idea of Managed Data, it is commonly used to modularize pervasive data management mechanisms such as logging and security, similar to how we define data managers. Like AOP, our approach allow functionality to be weaved in without the need for explicitly prepared variability hooks. However, our approach is coarser in granularity, operating only at method boundaries. Also, because we are dealing exclusively with data managers, we know beforehand the set of possible join points, and thus quantification become unnecessary. Nevertheless, the Ensō interpreter framework does support universally quantified modifications like aspects in the general context.

7. Conclusion

Managed Data is a powerful approach to data abstraction that gives programmers control over the fundamental mechanisms for creation and manipulation of data. A *schema* provides a description of the structure and behavior of desired data. The schema is interpreted by a *data manager* that implements the necessary strategies for managing data. Managed Data gives programmers more degrees of freedom while separating concerns. Programmers can change specific schemas, or the schema language, or the data managers themselves, depending on what level of functionality is needed.

While the idea of Managed Data has appeared in various forms in the past, it has not been identified and studied as a fundamental programming concept. We have analyzed previous approaches to Managed Data, and proposed a new implementation based on interpretation of external schema languages. Our approach to Managed Data is the foundation of the Ensō system. As a case study, we demonstrated how Managed Data can be used in the context of a web development framework to reuse database management and access control services across different data definitions while hiding their implementation from the client code. Both our implementation of Managed Data and the example web framework EnsōWeb are available from `http://enso-lang.org`.

In the future, we intend to explore static typing for Managed Data. Another promising direction is improving data manager performance through partial evaluation, especially since bootstrapping introduces a significant slowdown based on our current implementation.

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Professional, 4th edition, 2005.

[2] Christian Bauer and Gavin King. *Java persistence with Hibernate*. Manning Publications Co., second edition, 2006.

[3] Peter Pin-Shan Chen. The entity-relationship model - towards a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, March 1976.

[4] Pierre Cointe. Une extension de VLISP vers les objets. *Science of Computer Programming*, 4:291–322, 1984.

[5] William R. Cook. Policy-based authorization. (Unpublished manuscript), 2003.

[6] Oracle Corporation. Proxy (Java 2 Platform SE v1.4.2). http://docs.oracle.com/javase/1.3/docs/guide/reflection/proxy.html.

[7] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., 3rd edition, 1998.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[10] Michael Hammer and Dennis McLeod. The semantic data model: a modelling mechanism for data base applications. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 26–36. ACM Press, 1978.

[11] Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 4609, pages 399–424. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-73589-2-19.

[12] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language Haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992.

[13] Brian R. Hunt, Ronald L. Lipsman, and Jonathan M. Rosenberg. *A guide to MATLAB: for beginners and experienced users*. Cambridge University Press, 2001.

[14] Bjorn Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, first edition, 2005.

[15] Richard Kelsey, William Clinger, and Jonathan Rees. Revised 5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), 1998.

[16] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1988.

[17] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer Berlin / Heidelberg, 1997.

[18] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[19] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. April 2007.

[20] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, 1997.

[21] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-Step Guide*. Artima Incorporation, 2nd edition, 2011.

[22] OMG. *Meta Object Facility (MOF) Specification*. Object Management Group, 2000.

[23] Alexander Reelsen. *Play Framework Cookbook*. Packt Publishing, 2011.

[24] Guy Steele. *Common Lisp: The Language*. Digital Press, 1990.

[25] Guy L. Steele, Jr. *Common LISP: the language (2nd ed.)*. Digital Press, 1990.

[26] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, second edition, 2008.

[27] Michael Stonebraker and Ugur Cetintemel. "One Size Fits All": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE)*, pages 2–11. IEEE Computer Society, 2005.

[28] Don Syme, Adam Granicz, and Antonio Cisternino. *Expert F# (Expert's Voice in .Net)*.

[29] Éric Tanter. *Object-Oriented Programming Languages: Application and Interpretation*. 2010.

[30] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide*. Addison-Wesley Professional, second edition, 2008.

[31] Dave Thomas, David Hansson, Leon Breedt, Mike Clark, James Duncan Davidson, Justin Gehtland, and Andreas Schwarz. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2006.

[32] Guido van Rossum. *The Python Language Reference Manual*. Network Theory Ltd, second edition, 2006.

[33] David A. Watt, Brian A. Wichmann, and William Findlay. *Ada language and methodology*. Prentice Hall International (UK) Ltd., 1987.

[34] Niklaus Wirth. *Programming in MODULA-2 (3rd corrected ed.)*. Springer-Verlag New York, Inc., 1985.

[35] Joseph W. Yoder and Ralph E. Johnson. The adaptive object-model architectural style. In *Proceedings of the IFIP Conference on Software Architecture: System Design, Development and Maintenance*, pages 3–27. Kluwer, B.V., 2002.