JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

TNF

Technisch-Naturwissenschaftliche
Fakultät

An efficient Meta-Object Protocol for Scheme

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
DI Peter Feigl

Angefertigt am:
Institut für Systemsoftware

Beurteilung:
a. Univ.-Prof. Dr. Günther Blaschek (Betreuung)
a. Univ.-Prof. Dr. Johannes Sametinger

Linz, November, 2011

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe verfaßt, andere als die angegebenen Quellen nicht benützt und die den benützten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, November 2011 DI Peter Feigl

An efficient Meta-Object Protocol for Scheme

DI Peter Feigl

November 2011

Abstract

Meta-object protocols offer introspective capabilities to object-oriented languages and provide the programmer with a way to interfere with the design decisions the language designer proposed. They open up the implementation, leaving the programmer with choice concerning the exact semantics of object representation, inheritance, generic function invocation etc.

This work presents a meta-object protocol for the programming language Scheme that offers similar power to existing meta-object protocols in other languages, yet is designed to allow efficient implementation.

A number of sample applications was implemented and is presented with details on how the meta-object protocol allowed their implementation. They are intended as an indication that the meta-object protocol is powerful enough for actual applications.

The final chapters summarise the design, the implementation and the applications. The proposed meta-object protocol is compared to other existing approaches, and its applicability to other languages is investigated.

Kurzfassung

Metaobjekt-Protokolle bieten introspektive Fähigkeiten für objektorientierte Programmiersprachen und erlauben dem Programmierer, in verschiedener Weise die Sprachentwurfsentscheidungen des Programmiersprachenentwicklers zu beeinflussen. Sie öffnen die Implementierung für den Programmierer und erlauben Einfluß auf die Speicherrepräsentation von Objekten, Vererbung, Aufruf von generischen Funktionen, etc.

Diese Arbeit präsentiert ein Metaobjekt-Protokoll für die Programmiersprache Scheme, das den anderen existierenden Metaobjekt-Protokollen entsprechende Mächtigkeit besitzt, aber dafür ausgelegt ist, dass eine effiziente Implementierung möglich ist.

Einige verschiedene Beispiele wurden ausimplementiert und werden vorgestellt, um zu zeigen, wie das Metaobjekt-Protokoll deren Implementierung erlaubt. Sie dienen dazu zu zeigen, dass das Metaobjekt-Protokoll mächtig genug für Anwendungen der realen Welt ist.

Die letzten Kapitel fassen das Design, die Implementierung und die Anwendungen zusammen. Das vorgeschlagene Metaobjekt-Protokoll wird mit anderen existierenden Herangehensweisen verglichen und dessen Anwendbarkeit auf andere Sprachen wird untersucht.

Acknowledgements

To Prof. Blaschek, who supported me in choosing a topic I found interesting and patiently encouraged me in my work; to my parents, my sister and my brother, who always believed in me; to all my colleagues and friends, who showed interest in the topic and discussed various points with me; to the people from several channels on `irc.freenode.net`, who answered questions about exotic programming languages. To Angelika, for the love and support I receive daily. To the inventors of Scheme, for creating the language I came to love.

Contents

List of Figures

List of Tables

1. Introduction

Object-oriented systems have been around a long time. Since the 1960s, they have been analysed and improved. Many paradigms have been offered to the research community, yet the mainstream object-oriented languages only differ in small details from their ancestors or from each other.

One reason for this unwillingness to change seems to be the dichotomy of power versus efficiency. Language designers must consider the trade-off between adding additional features to the language, yet they must also ensure that implementations of the language retain the ability to compile to efficient code. The designer cannot know all fields of application for the programming language, and thus some of these choices are almost always wrong in some application the language designer never thought of.

As an example, consider the traditional memory allocation of objects: Upon allocation of an object storage is allocated for all fields, whether they contain useful values or not. If a programmer were to design an object with a large number of fields, of which only few ever hold a value (a "sparse" object), a considerable amount of memory is wasted. On the other hand, languages that allocate fields by need in a hashtable-like manner often cannot guarantee efficient access to these fields. It would be beneficial for the programmer, if she could choose the preferred implementation for her classes.

Kiczales and Des Rivieres [15] have suggested that the use of meta-object protocols can open up object-oriented languages, allowing the programmer (as opposed to the language designer) to decide certain trade-offs. A meta-object protocol allows the programmer to intervene in the semantics of object-oriented programs, whether for reasons of succinctness, power of expression or efficiency.

The original meta-object protocol proposed in [15] is a very dynamic protocol, it mandates costly generic function calls in many places, only few of the available commercial compilers implement the necessary complicated optimisations for efficiently implementing the full meta-object protocol.

The EuLisp object system [4] was meant to design a meta-object protocol that alleviates these problems, however it never reached a proper degree of maturity.

This work presents a meta-object protocol that aims to fulfil the goals of maintaining the power of the MOP presented in [15], yet provide a protocol that is simpler to implement efficiently. The author has developed – on the basis of a common object-system, designed by the author but based on and compatible with most of the major Lisp object systems – a number of protocols allowing the user of the meta-object protocol to reflect upon object-oriented programs and interfere in their operation.

The language Scheme [17] has been chosen to implement a prototype because it offers a simple syntax, powerful semantics, and is amenable to syntactic extensions (which considerable simplify the use of the object system and the meta-object protocol for the programmer).

1.1. Overview

Chapter 2 introduces the elements of the language Scheme that are necessary to understand the code examples throughout the later chapters. A short introduction is given to basic Scheme datatypes, the control structures, and expression syntax. As Scheme consistently employs prefix syntax, very little knowledge of special syntax is necessary. This chapter also explains syntactic extensions (so-called "macros") in Scheme, which are a pervasive and fundamental way of expanding the language to match the problem domain.

Chapter 3 presents related work, especially the original meta-object protocol from [15] and related concepts in other programming languages. The meta-object protocol or similar construct of CLOS, TELOS, Smalltalk, OpenC++, Dylan and Ruby are presented.

Chapter 4 describes the proposed meta-object protocol (and the object system it is based on). This chapter (and appendix A) is a specification for implementors of the meta-object protocol. First the object system is described in detail, then the different protocols are shown. The design decisions and alternatives are shown and explained.

Chapter 5 describes the prototypical implementation, which was used to validate that the proposed meta-object system is feasible.

Chapter 6 shows how to use the meta-object protocol to achieve various effects:

- Changing the storage of class member fields from a vector-like model to a hashtable-like model
- Automatic tracing and logging of function entry and exit
- Class-allocated (static) class member fields
- Multiple dispatch (multimethods) based on predicates, not only class membership
- Automatic object persistence

These applications are meant to demonstrate the versatility and efficiency of the proposed meta-object protocol.

There are two explicit goals for the meta-object protocol proposed in this work:

- Provide a powerful conceptual framework for reflecting upon and interceding in object-oriented programs
- Offer a framework that does not preclude efficient implementation

Chapter 7 recapitulates on the features and compares them to the existing approaches mentioned in chapter 3; it also re-examines the proposed meta-object protocol with respect to the two goals mentioned above.

Appendix A lists all the relevant generic functions in alphabetic order and explains their use. It is meant as a reference for implementors.

2. Scheme

Scheme[17] is a programming language in the Lisp family. Its main defining aspects in that family are that it is lexically scoped, supports full continuations, mandates proper tail-call-optimisation and aims to provide minimal core of orthogonal features. Scheme has been under constant development since its beginnings in 1975, as of 2011 work is under way on the 7th Revision of the Scheme Standard. As the core language is small, there are many implementations, often targeted at very specific problems or new work in programming language research (for example [12]).

Scheme has been chosen as the implementation language for this work because it offers a simple way of adding objects to the core language and due to its uniform syntax it allows for syntactic extensions (macros), which greatly facilitate providing syntactic sugar for complex operations.

2.1. Syntax

Basic Scheme syntax is very simple, every expression is syntactically a function call. An opening parenthesis is followed by the operator, then all the operands separated by whitespace, then a closing parenthesis:

```
(operator operand1 operand2 operand3)

(+ 1 2)
  => 3
(string-append "foo" "bar")
  => "foobar"
```

Scheme passes parameters by value, i.e. all expressions in operand positions (and the expression in the operator position) are evaluated before the actual function call is made. Only very few so-called Special Forms violate this rule.

2.1.1. Datatypes

Scheme supports a number of primitive datatypes, most Scheme systems also support a large number of additional datatypes that are not mandated (or regulated) by the standard ([17]). The basic datatypes are:

- boolean
- symbol
- character
- number
- string
- vector
- pair (which extends to list)

Booleans

Boolean values in Scheme are denoted by `#t` for true and `#f` for false. In addition, any non-false value is considered as true by `if`.

Characters

Characters in Scheme are denoted by the two characters `#\` followed by the actual character. For non-graphic characters, the name of the character is used instead of the actual character:

```
#\a
#\!
#\newline
#\space
```

Numbers

Scheme supports integers, floating point numbers (including scientific notation), fractions and complex numbers:

```
2147483648
3.141592653589793
6.0221415e23
333/106
3+2i
```

Strings

Scheme strings are enclosed by `"`, which can be escaped by a backslash if it occurs inside strings itself (i.e. the string `"\""` denotes the string consisting of a single double-quote character). Most Scheme systems support a number of additional escape sequences for newline, tabulator, etc.

```
  "a␣simple␣string"
  "a␣string␣containing
␣␣␣an␣implicit␣and␣an␣explicit\nnewline"
```

Symbols

Any identifier (sequence of characters delimited by whitespace or parentheses and excluding a few special characters like single or double quote) in Scheme is a symbol. Symbols are used to denote bindings. The Scheme implementation ensures that two symbols that look the same are actually the same symbol. All the following are legal symbols:

```
add
+
list->string
%defmacro
```

A number of conventions for identifiers exist:

- Symbols ending in an exclamation mark (e.g. `set!`) mutate their argument
- Symbols ending in a question mark are predicates that return true or false (e.g. `even?`, `odd?`)
- Symbols enclosed in asterisks are global variables (e.g. `*print-base*`)
- Symbols enclosed in angle brackets are class names (e.g. `<integer>`, `<object>`)
- Symbols beginning with a percent sign are system-dependent or internal (e.g. `%fixnum:add` or `%write-char`)

None of these conventions is enforced in any way by the Scheme system.

Vectors

Vectors in Scheme are a fixed-size datastructure that can be indexed by successive non-negative integers (the first element has index 0) . A literal vector is written as the characters `#(` followed by the elements of the vector separated by whitespace, and ends with a matching closing parenthesis `)`:

```
(define my-vector #(1 2 3))
(vector-ref my-vector 0)
  => 1
(vector-length my-vector)
  => 3
```

Pairs

Pairs in Scheme consist of a first and second element (called "car" and "cdr" for historic reasons). The function that creates a pair is called `cons`. A pair is displayed as an opening parenthesis, followed by the first element, then a dot (`.`), then the second element, and ended by a closing parenthesis:

```
(cons 1 2)
  => (1 . 2)
(define my-pair (cons 1 2))
(car my-pair)
  => 1
(cdr my-pair)
  => 2
```

Lists

Lists are nested pairs, where each cdr of a pair is another pair, until – at the end of the list – the cdr is a special value denoting the empty list, called "null" (and written as an empty list `'()`):

```
(cons 1 (cons 2 (cons 3 '())))
  => (1 2 3)
(list 1 2 3)
  => (1 2 3)
(define my-list (list 1 2 3))
(car my-list)
  => 1
(cdr my-list)
  => (2 3)
(car (cdr my-list))
  => 2
```

All normal Scheme code consists of lists. Function calls, definitions, special syntactic forms, arithmetic operators, everything has the same, uniform syntax in Scheme.

2.1.2. Special Forms

Normally, all parameters to a function are evaluated before the actual function call is made ("call-by-value"). There are only five special forms (where the evaluation order and time of the operands is not as usual) in Scheme:

- `if`
- `quote`
- `set!`
- `lambda`
- `quasiquote`

In addition to these, any macro that expands to these forms may also appear to follow special evaluation rules (e.g. `let` or `and`).

`if`

`if` is the basic conditional expression. It takes two or three parameters, the first denotes the test, the second the consequence (if the test succeeds), the optional third the alternative (if the test fails):

```
(if (> number 0) 'positive 'negative)
```

The expression above returns the symbol `positive` if `number` is greater than zero, otherwise `negative`. If the alternative (the third argument) is not supplied, an unspecified value (or no value) is returned.

The consequence is only evaluated if the test succeeds, the alternative is only evaluated if the test fails.

`quote`

`quote` signals that an expression is a literal, not an expression to be evaluated. Only symbols and lists need to be quoted, as all other datatypes are self-evaluating:

```
(define foo 3)
foo
  => 3
'foo
  => foo
(+ 1 2)
  => 3
'(+ 1 2)
  => (+ 1 2)
```

Quote never evaluates its argument.

`set!`

`set!` is the basic assignment operator in Scheme. It is used to assign values to lexical and global bindings:

```
(define x 3)
x
  => 3
(set! x 5)
x
  => 5
```

`set!` never evaluates its first argument (that argument denotes the binding that is to be modified) and it always evaluates the second argument.

`lambda`

`lambda` creates anonymous functions. A lambda-expression consists of a list of parameter names (the names of the bindings that are in effect in the body) and a function body:

```
(lambda (a b)
  (+ a b))
```

When this lambda-expression is applied to parameters, the values of the parameters are bound to the names given in the parameter list, and the body is executed with these bindings in place:

```
((lambda (a b) (+ a b))
  1 2)
  => 3
```

`lambda` never evaluates any of its parameters.

`quasiquote`

`quasiquote` is used to create complex list structures that are partly dynamic. All the content of the quasiquote-expression is quoted, except for the parts that are unquoted by the `unquote` operator:

```
(quasiquote (+ 1 2 => (unquote (+ 1 2))))
  => (+ 1 2 => 3)
```

There is also a splicing operator, that splices a list into place inside a quasiquoted expression:

```
(quasiquote (+ 1 2 (unquote (list 3 4 5))))
  => (+ 1 2 (3 4 5))
(quasiquote (+ 1 2 (unquote-splicing (list 3 4 5))))
  => (+ 1 2 3 4 5)
```

As a convenient shorthand, Scheme systems support ` instead of `quasiquote`, `,` instead of `unquote` and `,@` instead of `unquote-splicing`:

```
`(+ 1 2 ,(+ 1 2) ,@(list 4 5))
  => (+ 1 2 3 4 5)
```

2.1.3. Syntactic Sugar

Scheme provides predefined syntactic expressions for defining functions, binding variables, looping, etc. Most of these require no special support from the underlying implementation, and could be (and often are) implemented by the mechanism described in 2.2.

`define`

`define` is used to define global bindings. An extended syntax is available to simplify function definitions.

```
(define x 5)
x
  => 5
(define add (lambda (a b) (+ a b)))
(add 1 2)
  => 3
(define (add a b) (+ a b))
(add 1 2)
  => 3
```

`begin`

`begin` denotes serial execution. All expressions inside `begin` are executed one after another, then the result of the last expression is returned:

```
(begin (- 1 2) 'done)
  => done
```

`let`

`let` is used to bind lexical variables. The first parameter must be a list of pairs of names and values for the bindings, which are in effect in the body of the let-expression:

```
(let ((a 3)
      (b 5))
  (+ a b))
  => 8
```

These bindings can be mutated by `set!`.

`and and or`

`and` and `or` are Scheme's version of the short-circuiting boolean operators:

```
(and #t #f (/ 1 0))
  => #f
(or #f #t (/ 1 0))
  => #t
(or #f #f (/ 1 0))
  => division by zero signalled by /
```

Scheme supports many more predefined syntactic expressions, and any programmer can transparently extend this list by defining custom Syntactic Extensions.

2.2. Syntactic Extensions

Due to the uniform syntax of Scheme expressions, it is simple to create different kinds of macro-systems that allow automatic transformations of source code. A macro is code that is given code as input, and generates code as output. Because all code in Scheme has the same list structure, this approach does not suffer from some of the difficulties and drawbacks that for example C or C++ macros face. It is not possible to generate unbalanced parentheses or illegal syntax with Scheme macros.

Macros are expanded before the compiler gets to see the code, thus they can be used not only for syntactic sugar, but also for all kinds of optimisations [16, 14]. The macro-writer can create macros that expand into different code depending on the specifics of the actual macro use, similar to user-specified custom compiler optimisations.

Macro systems are divided into Hygienic Macro Systems and Non-Hygienic Macro Systems. "Hygienic" means that certain precautions are taken to limit the power of the macro system and avoid the most common problems (for example multiple evaluation or identifier injection).

2.2.1. Hygienic macros

The Scheme Standard R5RS ([17]) defines hygienic macros based on syntax rules. This form of syntactic extension allows the definition of special syntax keywords that transform code before it is compiled. Rules consist of a pattern, and an expansion (or template). Each use of the rule is matched against the pattern, and the template is expanded accordingly.

```
(define-syntax push!
  (syntax-rules ()
    ((push! el lst)
     (set! lst (cons el lst)))))
```

In the example above, `(push! el lst)` is the pattern (the syntactic keyword `push!` followed by two parameters, `el` and `lst`). Whenever the macro-expander sees an expression that matches this pattern, it replaces this expression by the expanded form, in this case `(set! lst (cons el lst))`.

This example is a macro that prepends a value to a list (i.e. it assigns the old value of a variable with the element prepended to the variable). The macro `push!` cannot be written as a function, as it needs to mutate the original variable, and all normal functions in Scheme follow call-by-value semantics.

Pattern language

The pattern language of `syntax-rules` is rather simple, the following section gives a brief overview, for a detailed explanation see [17]. The pattern can be an arbitrary list structure, the identifier `...` is treated specially in the pattern (and expansion). An identifier followed by `...` will match zero or more expressions in the expression that the pattern is matched against. In the expansion, `...` is used similarly. Each identifier that does not appear in the pattern, but only in the expansion is assumed to denote the binding that is in effect at the time of the definition of the macro.

```
(define-syntax unless
  (syntax-rules ()
    ((unless condition body0 body1+ ...)
     (if (not condition) (begin body0 body1+ ...)))))
```

The example above defines a macro that expands to code which executes the body expressions if the condition is not true (i.e. unless the condition is true). At least a condition and one body expression must be specified (though more than one body expression is allowed), otherwise the macro does not match (and an error is raised).

```
(define-syntax unzip
  (syntax-rules ()
    ((unzip (a b) ...)
```

```
      (list (list 'a ...) (list 'b ...))))))
```

The example above defines a macro that "unzips" all lists it is given, by building a list of all the first expressions and another list of all the second expressions in the parameters.

```
(unzip (at austria) (de germany) (ie ireland) (in india))
  => ((at de ie in) (austria germany ireland india))
```

Literal keywords in syntax-rules

The parameter right after `syntax-rules` (in all examples above an empty list `()`) may contain a list of "keywords" that must appear exactly as given in the pattern.

```
(define-syntax for
  (syntax-rules (in)
    ((for element in list body0 body1+ ...)
     (for-each (lambda (element)
                 body0 body1+ ...)
       list))))

(for i in (list 1 2 3 4 5 6 7 8 9 10)
  (display (* i i))
  (display "␣"))
  => 1 4 9 16 25 36 49 64 81 100
```

The example above defines a simple for-loop. Like this example, all of the traditional looping constructs can be built in Scheme with syntactic extensions; it is for this reason that R5RS only includes a single iterative looping construct (`do`).

Limitations of and problems with `syntax-rules`

There are limitations to the macros that can be written with `syntax-rules`. The most severe of these limitations is probably the fact that no new identifiers (i.e. identifiers that do not appear in the pattern) can be introduced in the expansion. As an example, anaphoric if [14], a macro that behaves like the normal `if` special form, but automatically binds the identifier `it` to the result of evaluating the condition, cannot be written hygienically (because it introduces the new binding `it` that is not mentioned in the parameters of the macro).

A general problem with macros can be inadvertent multiple evaluation of parameters. If any of the parameters is repeated multiple times in the expansion, the expression will be evaluated multiple times. If this expression includes side effects, this behaviour can be observed. This is not generally an error, as it can be correct behaviour (as in looping constructs, where multiple evaluation is what is actually sought).

Sometimes an approach on a lower level than the one that `syntax-rules` provides is needed. Traditionally, Lisp systems have supported a very basic form of macro definitions: a macro is a function that is passed an expression (as a list structure), and that returns another expression, which is used instead of the one that was passed. This is a very powerful approach, as it allows arbitrary code transformations, but it can lead to subtle problems due to inadvertent capturing of identifiers. This happens if a macro introduces an implicit binding that shadows a name which the user wanted to permeate the macro. Ways of dealing with the complexity that this approach raises are Syntactic Closures [2] or Explicit Renaming [7].

3. Related Work

Several decades of research have gone into Meta-Object Protocols (MOPs) in the Lisp family. Many different approaches have been explored. In other programming language families, research and results have been more limited. For this reason, most of the related work in this chapter is related to the Lisp family.

Probably the best-known example of a Meta-Object Protocol is the CLOS Meta-Object Protocol (described in [15]). Related to this is the TELOS MOP [4, 22]. The language Dylan (which is heavily based on Lisp) also includes a MOP with several distinctive features [8]. Smalltalk includes a reflective MOP [11].

3.1. Definition of Terms

The term *Meta-Object Protocol* consists of three parts: Meta, Object, and Protocol. We will now define what these terms mean in this work:

Meta

> μετά
> with Accus. II. of Place, after, next after, **behind**
>
> *Greek-English Lexicon, Liddell and Scott*

> meta-
> 1. a prefix appearing in loanwords from Greek, with the meanings "after", "along with", "**beyond**", "among", "**behind**".
>
> *dictionary.com*

Terms that start with meta- often describe concepts that are in some way behind others, or that others are based on. In this sense, *meta-class* is a class that is behind a normal class. Meta-Objects are objects that describe how objects are created and used. They are part of the meta-language that describes the actually implemented language (Scheme with our meta-object protocol).

Object

> object

> From Medieval Latin *objectum*: something thrown down or presented (to the mind).
>
> *dictionary.com*

An object in this work is a region of memory that is assigned meaning depending on the class it belongs to. An object has an identity, a class, and a number of slots (fields) which can contain other objects.

Protocol

> πρωτόκολλον
> a leaf or tag attached to a rolled papyrus manuscript and containing notes as to the contents
>
> *dictionary.com*

A Protocol describes the correct interplay and co-operation of different parts of a system.

3.1.1. Classes

The object systems we look at are Class-based. This means that each object is an instance of a class, which defines its behaviour and structure. A class defines the following for each of its objects:

- How an object is created
- What the structure of an object looks like
- How slots in an object are accessed
- Which slots an object has
- Which superclasses an object has
- Which generic functions are applicable to an object

Classes are typically defined by specifying a list of superclasses (if multiple inheritance is supported, otherwise only a single superclass is allowed) and a list of fields, commonly called "Slots" in Lisp object systems.

```
(define-class <point> (<object>)
  (x (reader point-x) (writer set-point-x!))
  (y (reader point-y) (writer set-point-y!)))

(define my-point (make <point>))
```

```
(set-point-x! my-point 3)

(point-x my-point)
  => 3
```

This code sequence defines a new class `<point>` that inherits from another class `<object>` and has two slots, one called `x` which has a slot reader (an existing generic function of one parameter) called `point-x` and a writer (an existing generic function of two parameters) called `set-point-x!`; similarly for `y`. As seen in this example, new instances of a class can be created by calling the generic function `make`.

3.1.2. Generic Functions and Methods

Most of the MOPs in this chapter support Multiple Dispatch by providing multi-methods [3] (i.e. methods that dispatch based on the dynamic type of all parameters, not only a special "self" or "this" parameter). Methods are de-coupled from classes, they belong to a Generic Function. The generic function itself is like an interface or abstract definition, its methods provide the actual implementations for different types.

As an example consider a method `draw` which has two parameters: the shape that is to be drawn, and the medium that this shape should be drawn onto. In a traditional single-dispatch object system, the method `draw` has to be either a method in the class `Shape` or in the class `Medium`. If there are many subclasses of `Shape` and many subclasses of `Media`, there is no easy way to define special methods for those combinations (this is especially notable if only a few of these many possible methods are actually needed). One traditional solution to this is "double dispatch":

```
abstract class Medium {
  public abstract void draw(Shape s);
}

class Paper extends Medium {
  public void draw(Shape s) {
    s.drawOnPaper(this);
  }
}

class Screen extends Medium {
  public void draw(Shape s) {
    s.drawOnScreen(this);
  }
}
```

```
abstract class Shape {
  public abstract void drawOnPaper(Paper p);
  public abstract void drawOnScreen(Screen s);
}

class Circle {
  public void drawOnPaper(Paper p) { ... }
  public void drawOnScreen(Screen s) { ... }
}

class Square {
  public void drawOnPaper(Paper p) { ... }
  public void drawOnScreen(Screen s) { ... }
}
```

If extensibility is not an issue, a `switch` statement dispatching on the type can also be used. Both of these solutions are unsatisfactory.

A generic function with multiple dispatch could be defined as follows:

```
(define-generic-function (draw medium shape))

(define-method (draw (p <paper>) (c <circle>)) ...)

(define-method (draw (p <screen>) (c <circle>)) ...)

(define-method (draw (p <paper>) (s <square>)) ...)

(define-method (draw (p <screen>) (s <square>)) ...)
```

This defines all four necessary methods for the generic function `draw`. It is extensible by defining further methods for additional classes. Only the necessary methods must be defined, there is no need to define all possible combinations.

Dynamic dispatch of the methods of a generic function takes into account the dynamic type of each parameter. First, all methods that are not applicable to the combination of parameters are eliminated. Then, the resulting methods are sorted according to specificity (the method with the "closest" parameter type in the type hierarchy is most specific). Many ways of dealing with the ambiguities resulting from several methods being most specific for different parameters exist; the most common is to define that the method that is most specific for the left-most parameter is the most specific method overall.

```
;  (define-class <real> ...)
```

```
;  (define-class <integer> (<real>) ...)

(define-generic-function (add a b))

(define-method (add (a <integer>) (b <real>))
  'integer-first)

(define-method (add (a <real>) (b <integer>))
  'real-first)

(add 1 2)
  => integer-first

(add 1 1.2)
  => integer-first

(add 1.2 1)
  => real-first
```

In this example, `<integer>` is a subclass of `<real>`. Any generic function that is applied to an integer will choose methods that are specialised on `<integer>` before methods that are specialised on `<real>` (because `<integer>` is "closer" in the inheritance hierarchy).

Comparable to the call of the implementation of a method in a base class ("super-call"), each method has an extra parameter (customarily called `next-method`), which is bound to the next-most-specific method. This can be called during execution of the method. Thus it is simple to extend an existing method in a subclass.

Figure 3.1 shows how these types of objects are interconnected. Objects are instances of classes, classes are objects themselves. Generic functions are special objects that can be called as functions. Each generic function contains any number of methods. The figure shows the relationships for an object of type `<point>` and for the generic function `add` as shown in the example above.

3.2. Meta-Object Protocols

The Meta-Object Protocol describes which parts of the Scheme system are concerned with creating objects, defining classes, calling generic functions, etc. It is an informal specification of the interfaces and classes that the implementor of extensions can count on.

Historically, these decisions have been made by the language designer, with no way to change them when using the language. The rules for object creation, dynamic dispatch, multiple inheritance conflict resolution etc. were decided on by the language creator, and

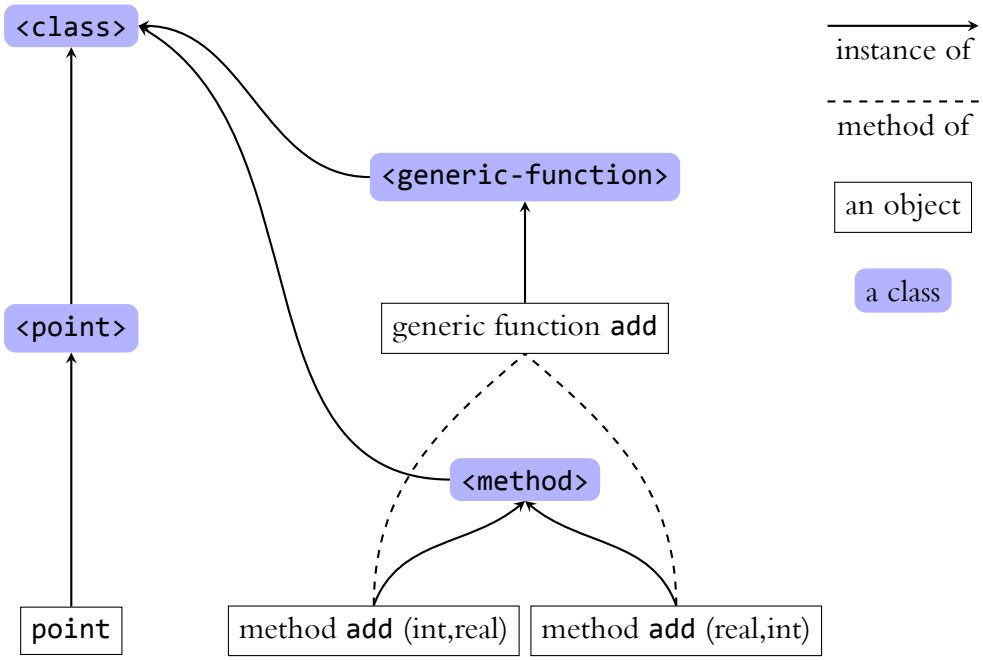

Figure 3.1.: Basic object relations

the user of the language was unable to adapt the object system to his or her own needs.

Meta-Object Protocols have the goal of opening up the implementation, so that the object systems we design do not occupy a single point in the space of possible systems, but actually describe a region of adjacent possible object systems. If the default spot chosen by the designer does not fit the user, she can adapt the object system (by modifying it via the meta-object protocol). Figure 3.2 shows a chart with several programming languages, where the meta-object protocol that is described in this work is shown as a region, not a single point.

Here are a few examples of the operations and adaptations that meta-object protocols allow

- Find out which slots a class has
- Find out which superclasses and subclasses a class has
- Find out which generic functions are specialised on a class
- Change the structure of objects to deal with objects that have a large number of slots of which only a few are ever used (“sparse” objects)
- Intercede in function calls to log the parameters and the return values
- Intercede in slot getters and setters to persist slot values in a database

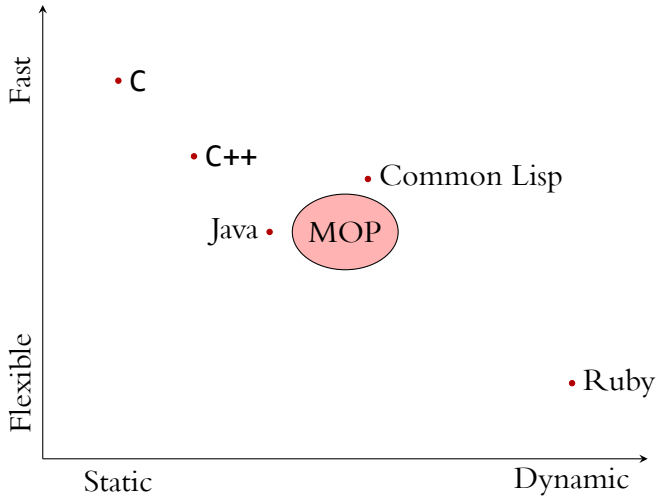

Figure 3.2.: Programming Language Design Space

- Change the way dynamic dispatch works by adding the possibility to specialise methods not only on classes but also on objects

These applications can be roughly divided into two fields of operation (according to [15]):

- *Introspective*: This part of the meta-object protocol allows to inspect the state of the program, the connections between objects, the applicability of functions, etc. Introspective protocols provide ways of acquiring information about the program. The first three items above are covered by introspective protocols.

- *Intercessory*: Intercessory protocols allow the programmer to change the way the default language works. The semantics of the object system can be changed and adapted at need (within certain bounds). The last four items above can be achieved by using intercessory protocols.

3.3. CLOS

The Common Lisp Object System (CLOS) [26] was developed to unify and expand the existing Lisp object systems (Flavors, New Flavors, Loops, CommonLoops, etc.). It consists of classes, generic functions, methods and many macros for defining and using these. Several compatible implementations of CLOS exist and all major Lisp systems support CLOS.

The fundamental and seminal work on meta-object protocols has been done in CLOS [15]. The protocol we describe here is the most wide-spread, detailed in [15].

The CLOS MOP provides a basic meta-object class for each program element. These classes are:

- `class`
- `slot-descriptor`
- `generic-function`
- `method`
- `method-combination`.

The following section details the information these classes contain. Figure 3.3 shows a diagram of the conceptual slots of the relevant classes (class `method-combination` is not shown as the meta-object protocol proposed later does not contain method-combinations in the same way as CLOS).

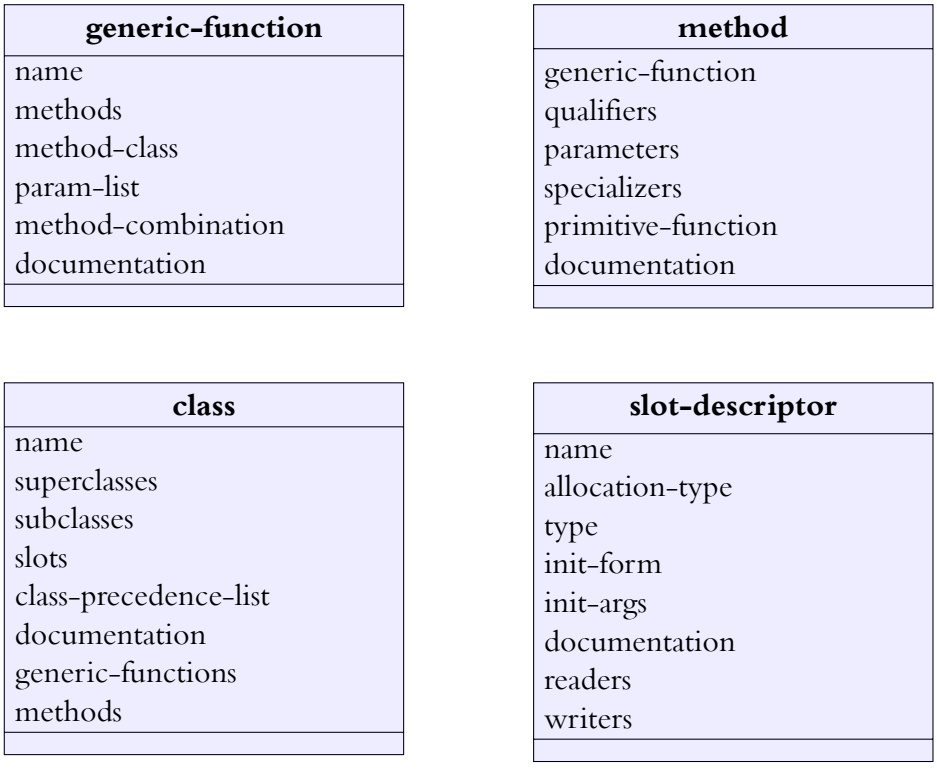

Figure 3.3.: CLOS basic meta-object classes

`class`

Each class meta-object defines the behaviour and structure of objects that are instances of this class. For reflection, the class meta-object may offer:

- The name of the class.
- All direct superclasses, direct subclasses, and the class precedence list (all direct and indirect superclasses).
- The direct slots and the effective slots (all direct and indirect slots that are actually accessible).
- Documentation for the class.
- All generic functions which have at least one method specialising on this class.
- All methods which specialise on this class.

`slot-descriptor`

A slot descriptor meta-object contains details about a single slot of a class. A class meta-object contains one slot descriptor for each direct and inherited slot. A slot descriptor meta-object may contain:

- The name of the slot.
- The allocation type of the slot (per instance, per class, ...).
- The type of the slot.
- The initialisation form (to automatically calculate a default value for the slot).
- Initialisation arguments (to automatically set the slot value based on parameters passed to the `make-instance` function).
- Documentation for the slot.
- The generic function names of reader and writer methods used for accessing the slot value.

`generic-function`

Generic function meta-objects contain information about a generic function (e.g. method dispatch), but not about the actual methods. This information includes:

- The name of the generic function.
- All methods of this generic function.
- The default class for the generic function's methods' meta-objects (i.e. the class which all methods of this generic function have).
- The list of parameters (to ensure that all methods have compatible parameter lists).
- The method combination of the generic function (this is a Common-Lisp specialty, explained in the section below).
- Documentation for the generic function.

`method`

Method meta-objects contain information about a single method:

- The generic function this method is a part of (either exactly one or none, no method can belong to multiple generic functions).
- The qualifiers for this method (relating to the method-combination).
- The list of parameters.
- The list of types this method is specialised on.
- The primitive function (an actual executable function, not a generic function).
- Documentation for the method.

`method-combination`

The CLOS MOP supports "method combinations", which are a means of applying not just a single method of a generic function, but multiple.

As an example, the predefined *standard* method combination allows the definition of multiple *before*, *after* and *around* methods (these keywords are the qualifiers in the `method` meta-objects), which are automatically executed before, after, or instead of the primary method.

The predefined *list* method combination automatically applies all applicable methods in order of specificity and returns a list of all the results. Other available predefined method

combinations are + (add all results), *and* (only apply later methods if all earlier methods returned a true value), *or* (only apply later methods if all earlier methods returned a false value), and a few more [26].

The structure of method combination meta-objects is left undefined by the CLOS MOP.

3.3.1. CLOS Protocols

A number of protocols are specified on different layers, for achieving different goals. The lowermost is the instance structure protocol, which influences the way class instances are built. The object initialisation protocol builds on this lower layer, providing ways to customise the initialisation of objects of any class. Upon these two the class finalisation protocol, which governs the creation of new classes, and the generic function invocation protocol, which deals with calling generic functions and selecting the correct methods to execute, are built. On top of all the others, a layer of syntactic extensions provides more convenient access to the functionality provided by lower levels:

- the instance structure protocol
- the object initialisation protocol
- the class finalisation protocol
- the generic function invocation protocol
- the syntactic extension layer

These protocols specify in detail how the specific generic functions and classes are related, when they are called, and – if additional methods are added – which restrictions these methods are subjected to.

The CLOS MOP differentiates between *functional* and *procedural* protocols: A generic function in a Functional Protocol is called for its result, which may be cached. The programmer cannot rely on a generic function of a functional protocol being called every time the value is needed. In contrast to this, a Procedural Protocol specifies that the implementor (and any programmer extending the generic methods) must ensure that the generic function is called each and every time as specified.

As an example, the list of superclasses of a class is calculated according to a functional protocol only once by the generic function `compute-class-precedence-list`. If it is requested multiple times by the user, `compute-class-precedence-list` may not be called again (depending on the implementation).

But in a procedural protocol, like the protocol for calling generic functions, the related generic functions must be called each and every time. For example, the generic function

`apply-method` must be called each time a method of a generic function is applied to arguments.

According to the specification the user of the Meta-Object Protocol can decide which generic functions to extend or override in which way.

3.3.2. Example protocol: Generic function invocation

As a representative of the protocols mentioned above, we will describe in more detail the generic function invocation protocol in CLOS MOP. As shown in figure 3.4,

```
apply-generic-function gf args                                  procedural
  compute-applicable-methods-using-classes gf classes           functional
    method-more-specific-p gf method1 method2 classes           functional
  apply-methods gf args methods                                 procedural
    apply-method method args next-methods                       procedural
      extra-function-bindings method args next-methods          functional
```

Figure 3.4.: Generic function invocation protocol in CLOS MOP

this protocol consists of nested calls to 6 different generic functions. The entry point is `apply-generic-function`, which must be called each time any generic function is called.

This generic function in turn calls `compute-applicable-methods-using-classes`, which computes all methods that are applicable to the list of classes that is given as a parameter. The result of this computation may be cached, so if the same list of methods and the same list of classes is used later on, it is possible (and allowed by the protocol) that the generic function `compute-applicable-methods-using-classes` is not called again.

The generic function `method-more-specific-p` (`-p` in Common Lisp has the same semantic denotations as `?` in Scheme) is called according to a functional protocol by `compute-applicable-methods-using-classes` in order to sort all applicable methods by specificity. Again, the results of calling it for two specific methods and a list of classes may be cached by the system.

Once the actually applicable methods of the generic function are calculated by the generic function `compute-applicable-methods-using-classes` and sorted (according to `method-more-specific-p`), the list of applicable methods is passed to the generic function `apply-methods`. This generic function decides on the exact order in which these methods are called. For the "standard" method combination, the order is all before methods, then the primary method, then all after methods.

For actually applying a method to the given arguments, `apply-methods` calls the generic function `apply-method`. This function in turn calls `extra-function-bindings`

in order to introduce any extra bindings that should be in effect during execution of the function.

3.4. TELOS

TELOS (or ΤΕΛΟΣ) is the object system of the EuLisp [21] dialect of Lisp . EuLisp was intended as a rival of Common Lisp, and was influenced by Common Lisp [26], InterLISP [27], Scheme [17] and T [23]. The basic objects of the meta-object protocol are the same as the ones in CLOS.

The Meta-Object Protocol of TELOS was explicitly meant to find a balance between efficiency, simplicity and extensibility [4]. TELOS consists of several layers, the lowest of which provides the building blocks for all the ones above it. For example this lowest level, called level-0, only implements single inheritance, but provides the mechanisms for multiple inheritance in higher levels.

As an example of the TELOS meta-object protocols, we will look at the slot access protocol. The TELOS slot access protocol works at class definition time, all slot accessors are set up when the class is defined.

For readers, the protocol is as follows:

```
(compute-slot-reader class
                     slot-descriptor
                     slot-descriptors)
(ensure-slot-reader class
                    slot-descriptor
                    slot-descriptors reader)
  (compute-primitive-reader-using-slot-descriptor
    slot-descriptor
    class
    slot-descriptors)
```

The function `compute-slot-reader` typically returns a generic function for the reader (calculated based on the slot descriptor) without adding any methods to it. To make sure that this generic function contains the correct method, `ensure-slot-reader` must be called on it. This method checks whether an appropriate reader is already defined, otherwise calls `compute-primitive-reader-using-slot-descriptor` to compute one.

A new slot descriptor type `<predicate-slot-descriptor>`, which checks a predicate before allowing write access, would be implemented as follows:

```
(defmethod compute-primitive-writer-using-slot-descriptor
  ((slot <predicate-slot-descriptor>)
   (class <class>)
   slotds)
```

```
(let ((prev-writer (call-next-method))
      (predicate (slot-descriptor-predicate slot)))
  (lambda (object new-value)
    (assert (predicate object))
    (prev-writer object new-value))))
```

This defines a new method that returns a primitive writer function on the generic function `compute-primitive-writer-using-slot-descriptor` which checks the predicate before actually writing the value (by calling the next method) (see [4] for details).

Unfortunately, TELOS was never completed, the class initialisation protocol and the generic function invocation protocol were never fully specified according to the original design goals. Development stagnated around the year 1993, when funding ran out.

3.5. Smalltalk

Smalltalk [13, 19] is one of the oldest object-oriented languages. It is dynamically typed (like Scheme) and fully object-oriented: everything is an object, there are no primitive types (like in Java) that are different from other objects. Smalltalk has a rather minimalistic syntax, only six reserved "keywords", and a large collection of standard library classes.

Smalltalk includes reflective facilities via meta-classes by default. Each class that is defined in Smalltalk automatically includes the definition of a meta-class. This meta-class has only one instance, the class object for the class that was originally defined. All meta-classes are instances of the class `Meta-class`, itself an instance of class `Class` (this resolves the metacircularity). Figure 3.5 shows the relationship between these objects. Meta-classes in Smalltalk are – rather unimaginatively – named `<classname> class` (e.g. the meta-class of class `Point` is named `Point class`). The following transcript shows the relationships also shown in figure 3.5. The square denotes a simple instance, the circles are class objects (either of classes or meta-classes). Arrows symbolise sub-classing, they point from the sub- to the super-class. Sending the message `class` to an object returns the class that this object is an instance of.

```
st> Point class
Point class
st> Point class class
Metaclass
st> Point class allInstances
(Point )
st> Point class class allInstances
(Class class Object class ObjectMemory class
 Message class ...)
```

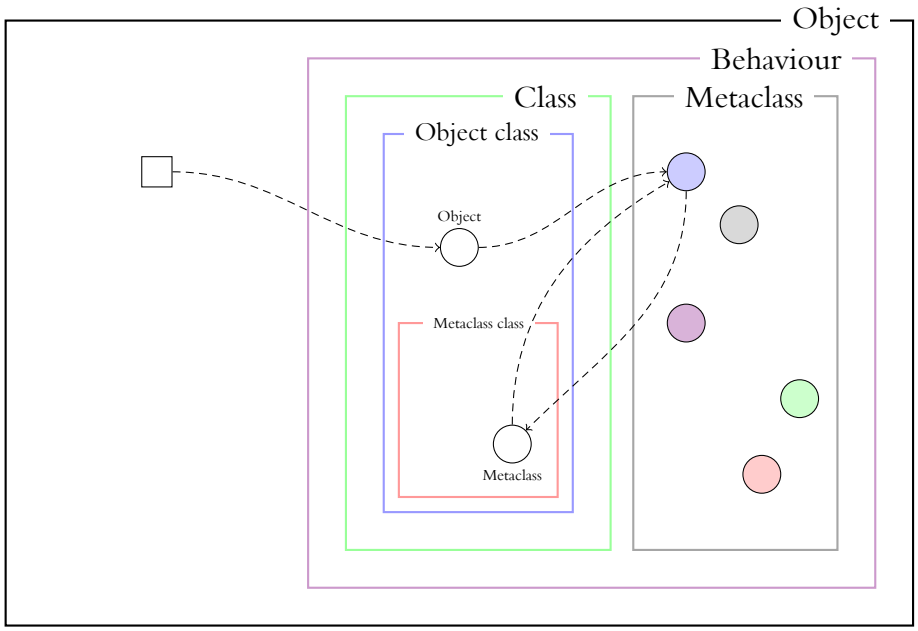

Figure 3.5.: Smalltalk Metaclasses (based on [19])

The class object `Point` is an instance of class `Point class`. That class, `Point class` is itself an instance of `Metaclass`. There is only one instance of `Point class`, namely the class `Point`. The class `Metaclass` has exactly one instance for each class defined in the entire Smalltalk system (in GNU Smalltalk `Point class class allInstances size` returns 246).

One application of these meta-classes is implementing class methods. A class method on a Smalltalk class is really an instance method on the meta-class.

Introspective facilities in Smalltalk reside mostly in the class `Behavior`, which is the common superclass of all class-like objects (mainly instances of `Class` and `Metaclass`). It contains – among many others – the following useful messages:

- `allSubclasses`: returns a list of all subclasses of this class
- `allSuperclasses`: returns a list of all superclasses of this class
- `instVarNames`: returns a list of all instance variable names
- `methodDictionary`: returns a dictionary of all messages this object directly responds to
- `inheritsFrom:`: returns true if this class inherits from the class passed as an argument
- `allInstances`: returns all direct and indirect instances of this class

Smalltalk lacks full intercessory protocols. Experiments in adding this ([11]) mostly involve changing the compiler to provide additional methods that are called during class creation etc.

Certain cases of intercession can be emulated by creating proxy objects that implement the message `doesNotUnderstand:`. This message is a special message that the Smalltalk VM generates when an object is sent a message that it normally doesn't implement. The single parameter is the reified message. This message can be inspected, and acted upon. This mechanism is for example used to implement proxies for distributed objects.

3.6. OpenC++

OpenC++ [6] is a compile-time Meta-Object Protocol for C++. It was developed with efficiency in mind, resulting in a pure compile-time protocol. The extended C++ code is compiled by an OpenC++ pre-compiler, then the resulting C++ code can be compiled by any normal C++ compiler. Any user of the MOP can generate custom code for the specified entry points into the MOP (e.g. getters and setters, function calls or variable declarations).

Meta-objects in OpenC++ only affect the compilation behaviour. There is (by default) no runtime representation of meta-objects. The OpenC++ compiler reads all relevant files into memory as an abstract syntax tree, builds an internal representation of the meta-objects, then applies the meta-objects at all necessary points in the code (e.g. getters and setters, function calls, memory allocation), which may result in a modified abstract syntax tree, and finally writes the resulting tree to C++ files for the normal compiler to work on.

The following behaviour in a C++ program can be influenced by meta-objects in OpenC++:

- member function calls
- data member readers
- data member writers
- variable declarations (of a class type)
- memory allocation (of a class type)
- class declaration

The meta-objects contain functions to deal with each of the constructs mentioned above. As an example, a meta-object that adds logging on data member readers is shown:

```
metaclass Point : ReaderLoggingClass;
class Point { public: int x, y; }

class ReaderLoggingClass : public Class {
  public:
  Expression
    CompileReadDataMember(Environment env,
                          String member_name,
                          String variable_name) {
    return
      MakeParseTree("cout␣<<␣\"Read␣%s.%s\"␣<<␣endl;%e",
                    variable_name,
                    member_name,
                    Class::CompileReadDataMember(...));
  }
}
```

In this example, the code sequence `cout << "Read␣...."<< endl;` is inserted right before the actual data member reader (which is compiled via a call to the superclass). The first line declares that `ReaderLoggingClass` is the meta-class of `Point`, telling the compiler to use the custom functions in that class to compile the class and all instances of `Point`.

3.7. Dylan

The dynamic language Dylan [24] was developed by Apple as an application-development language. It was designed as a purely object-oriented language from the ground up, different from Java or C++, which contain non-object primitives. The influence of Lisp (and especially CLOS) on Dylan is undeniable, in fact early Dylan interpreters and compilers were implemented in Lisp. The syntax was originally Lisp-like, but was changed to an ALGOL-like syntax in hope of wider acceptance.

The Dylan object system is based on the same primitives as CLOS, namely classes, generic functions and methods. For the sake of efficiency, a number of additions were made, the most important of which are *sealed* classes, which cannot be extended outside their original module (thus allowing efficient compilation).

The following shows how to define a generic function and two methods on it in Dylan:

```
define generic add (first :: <object>, second :: <object>)
 => (result :: <object>);

define method add (first :: <number>, second :: <number>)
```

```
 => (result :: <number>)
  first + second;
end method add;

define method add (first :: <string>, second :: <string>)
 => result :: <string>
  concatenate(first, second);
end method add;
```

The generic function `add` now has two methods, one that is applicable to two `<number>`s, and one that is applicable to two `<string>`s.

3.8. Ruby

The dynamic language Ruby [10] was developed primarily by Yukihiro Matsumoto. The most influential languages on its design were Perl, Smalltalk, Eiffel and Lisp. It is an exceedingly dynamic language, allowing modification of class structure and behaviour at runtime.

Due to its dynamic nature, the same effects that other languages achieve with meta-object protocols can be reached via reflection in Ruby. The main problem of this approach is its inefficiency, as the system cannot know which parts of a program are static, and which can change at any time.

As an example, we show how to override the `new` method in the built-in class `Class`, which is called when instantiating new objects:

```
class Class
  alias oldNew new
  def new(*args)
    print "Creating␣new␣", self.name, "\n"
    oldNew(*args)
  end
end
```

This code segment re-opens the class `Class` for defining and overwriting methods, it defines `oldNew` as an alias to the original `new` method. Then a the method `new` is overwritten with a new method, which first prints the string shown and then calls the old method. This method is now applicable for *all* classes, pre-existing and new. As an example, calling the method `new` on the class `String`, we get the following result:

```
String.new
  => Creating new String
  => ""
```

Ruby allows this sort of interference and redefinition in any place in a program, thus rendering optimisation very complex, because even the behaviour of built-in classes and methods can be modified easily.

3.9. Comparison

The preceding sections show that none of the investigated systems provide powerful and simultaneously efficient abstractions that achieve the goals mentioned in chapter 1. Figure 3.6 shows at which criteria the sample systems excel, and where there is still room for improvement. The criteria are:

- **Slot accessors:** Whether the language allows defining custom slot accessor code.
- **Object layout:** Whether the language allows changing object layout.
- **Static optimisation:** Whether the language provides the meta-object protocol in a way that still allows static optimisation.
- **Runtime component:** Whether the meta-object protocol is also available (in parts) at runtime.

Language	Slot accessors	Object layout	Static optimisation	Runtime component
CLOS	●	●	○	●
TELOS	●	●	●	●
Smalltalk	○	○	●	●
OpenC++	●	●	●	○
Dylan	○	○	●	●
Ruby	●	●	○	●

Figure 3.6.: Comparison of Languages

This comparison shows that TELOS meets all the criteria, but unfortunately work on TELOS was discontinued around 1993. The following chapter shows how to conceptually finish the work that TELOS began (though the choice of implementation language is Scheme instead of a custom Lisp).

4. Object System and Protocols

This chapter describes the object system this work is based on, and gives a detailed description of the proposed meta-object protocol.

For the meta-object protocol, a suitable object system has been built, mainly based on the CLOS and Dylan object systems. The basic building blocks and their interconnections are described in the next section. The following three sections describe in detail the three meta-object protocols for object instantiation, class creation and generic function invocation.

The meta-object protocol is designed to allow for efficient implementation, minimising the number of generic functions that are called at runtime, yet allowing for extensive customisation.

4.1. Components of the Object System

The object system that is used in this work consists of the following parts:

- Objects (or instances)
- Slots
- Classes
- Generic functions
- Methods

This section gives a detailed discussion of the exact semantics these parts have, and is an extension of section 3.1. The object system described here is based on CLOS, TELOS and Dylan, and is very similar to that of Smalltalk, Ruby and OpenC++, except for the use of multiple dispatch and generic functions, instead of class functions and single dispatch. There is nothing special about the object system, it is used as a medium of showing the applicability of the proposed meta-object system.

4.1.1. Objects

Objects are the basic entities of the object system. Each object is a *direct* instance of exactly one class (*the* class of the object). Each object has an identity, the programmer can distinguish two non-identical objects with the same layout and contents. An object is an abstraction that provides ways to access its class (`class-of`), compare its identity (`eq?`), and that can contain data in so-called "slots" (or fields). This data cannot be accessed directly (except via implementation-dependent functions), but the reader and writer functions for the slots can be found via reflection. This enables the user of the object system to redefine and optimise the actual memory layout of specific objects. Figure 4.1 shows the abstract layout.

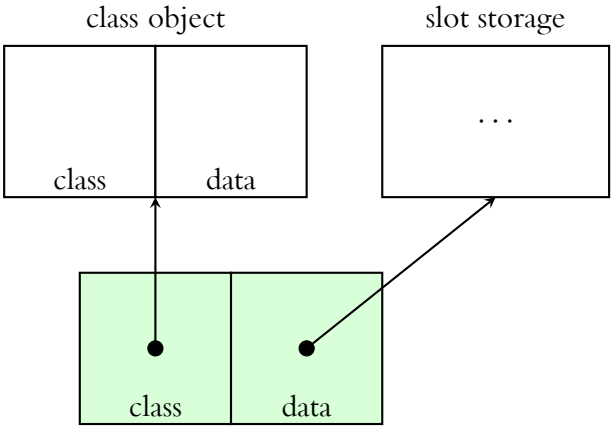

Figure 4.1.: Object layout

An object's class defines its layout (and more, as is described in later sections).

4.1.2. Slots

Slots are abstractions for fields or data members of an object. Each slot has a name, and optionally associated reader and writer functions. The normal way to access the slot value is via these functions. Slots are an abstraction; by default, a slot generates an actual data field that holds a value, but the meta-object protocol allows extensions where a slot value is e.g. calculated upon read, read from a database, transported over a network or distributed into other slots fields upon writing.

Each class has direct slots (the slots that are defined directly for this class) and indirect slots (all slots that are purely inherited, with no changes). The combination of these are the effective slots of the class.

4.1.3. Classes

Classes describe the layout and behaviour of objects. Classes are first-class objects, so they exist as objects at runtime, and can be used like any other object. Class objects are used for reflection (they contain objects that describe the instances of the class) and for intercession in the meta-object protocol (they contain information and functions that are used for e.g. constructing objects).

The default class objects (the extensible core) contain the following information:

- The name of the class
- The class this class is an instance of (the meta-class)
- The direct superclasses
- The direct slots
- The direct subclasses

The meta-class describes the layout of the class object. This leads to a conceptionally infinite Meta-Recursion (as the meta-class in turn needs a meta-meta-class that describes its layout, ad infinitum), that is resolved by the class `<class>`, which is its own meta-class (see figure 4.2). Because the class `<class>` describes the default behaviour and layout, which is clearly defined, no infinite recursion arises. Upon need, new meta-classes can be introduced, leading to new types of classes (for example classes that log all accesses to their slots or classes that automatically persistent their slot values). By default, all class objects are instances of `<class>`, the default meta-class object.

The direct superclasses specify this class's place in the inheritance hierarchy. By default, `<class>` only supports a single superclass (multiple inheritance can be implemented via the meta-object protocol as an extension).

The direct slots describe the slots that all instances of this class contain. This list only contains the slots that are new or redefined in this class (compared to its superclasses). The list of direct subclasses is automatically kept up-to-date whenever new subclasses are defined.

All these slots can be read via generic functions, forming a part of the reflective protocol. For example, accessing the slot descriptors and extracting the names and reader functions allows building a generic object inspection function that can print the slot names and slot values of any object.

Classes do not have specially associated functions (like in C++ or Java), but instead all generic functions that specialise any parameter on a class can be asked. This is conceptually similar to all methods of this class in C++ or Java.

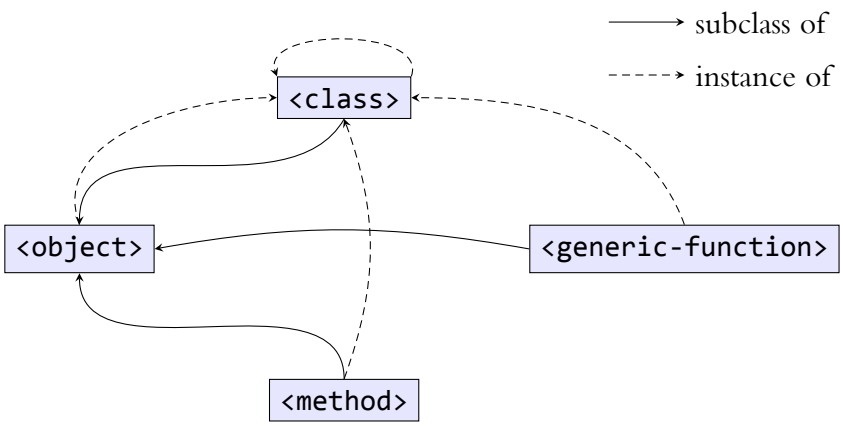

Figure 4.2.: Meta-Class Connections

4.1.4. Generic functions

Generic functions provide multi-methods (multiple dispatch). A generic function can have multiple methods, each of which is specialised on a certain list of parameter types. When the generic function is applied to a list of arguments, the best-fitting method is selected, and called.

Generic functions have the following slots:

- The name of the generic function
- The list of registered methods
- The common signature of all methods of this generic function
- A discriminating function, that selects and orders the methods upon function application

The signature describes the valid parameter lists, to ensure that all methods are compatible (for example, they must have the same number of required parameters). The discriminating function does the work of deciding which methods of the generic function actually apply, and in which order (see `next-method` in section 4.1.5) they will be called.

The default discriminating function works as follows: method specialisers can only be classes (i.e. the method applies if the parameter is an instance of the given class). All methods of a generic function where at least one specialiser does not match a parameter are discarded, the rest is sorted by specificity, where more specific means: From first to last parameter, if a method specialises on a hierarchically lower class, it is more specific. This algorithm is arbitrary (the sum of actual distances for all parameters could be used, or an entirely different approach), but has been used in many other Lisp object systems (among others [26],[21],[24]), and is simple to understand.

4.1.5. Methods

Methods encapsulate the actual code, that is executed when a generic function is called. Each method specialises on a certain set of parameters. It is guaranteed (by the discriminating function of the generic function that the method belongs to) that each method is only ever called with matching parameters (i.e. a type mismatch cannot happen when calling a method).

A method contains the following slots:

- The generic function this method is registered with (if any)
- The specialisers
- The method lambda (the actual function code)

By default, the specialisers can only be classes, i.e. dynamic dispatch compares the class of a parameter to the specialiser. This can be changed via the meta-object protocol (e.g. dispatch could be based on predicates or on singletons, see 6.4).

The method lambda is a primitive Scheme function, that does the actual calculations of the method. This function takes the same parameters as the method, but additionally has a parameter prepended that is called `next-method`. This parameter is used to implement inheritance-based super-calls. The discriminating function of a generic function orders all applicable methods. The first one is called, and the parameter `next-method` is set to the next-most specific method. Thus, the most specific method can call the next-most specific method, and so on. The passing of the correct next method to each invocation is handled behind the scenes. This is similar in spirit to a super-call in Java or C++. A more specific method can call a less specific method to extend upon its work.

4.1.6. Syntactic extensions for defining classes, generic functions and methods

To simplify the use of the object system, a number of syntactic extensions ("macros") is defined here, which allows easy definition of classes, generic functions and methods. It is important to note that these macros are not strictly necessary, they are only syntactic sugar around more verbose forms, however they make understanding what's going on much simpler. These are based on and similar to the corresponding macros in related Lisp object systems.

`define-generic-function`

To define a new generic function, a programmer would write code like the following:

```
(define-generic-function (binary-add a b))
```

This macro expands to the following definition:

```
(define binary-add (make <generic-function>
                         'binary-add
                         '(a b)
                         '()))
```

Here the first parameter to `make` is the class we wish to instantiate (in this example that is the class `<generic-function>`), the second argument is the name of the newly instantiated generic function (in this case the symbol `'binary-add`), the third parameter is the list of parameters that the newly created generic function will accept (in this case two parameters, name `a` and `b`).

The full form of the macro is:

```
(define-generic-function (<function-name>
                          <parameters> ...)
  <generic-function-option> ...)
```

The last parameter is a list of generic function options. The only option that is recognized by default is `instance-of`, which declares the class of the generic function (if it is different than `<generic-function>`). These options can be extended by the user of the meta-object protocol, new options can be added or old ones can be treated differently. For example, an option could be which of several competing dispatch mechanisms to use.

define-method

To add a new method to an existing generic function, the **`define`**`-method` macro is provided:

```
(define-method (binary-add (a <number>)
                           (b <number>))
               (+ a b))
```

This expands to the code that instantiates a new method and adds it to the generic function. If a method with the exact same specifiers already exists, it is overwritten.

The full form of the macro is:

```
(define-method (<generic-function-name>
                (<parameter-name> <parameter-type>)
                ...)
  <body-expression> ...)
```

define-class

The macro **`define`**`-class` supports the definition of new classes. An example of a use of **`define`**`-class` is:

```
(define-class <point-2d> (<object>)
  ((x
    (reader point-x)
    (writer set-point-x!))
   (y
    (reader point-y)
    (writer set-point-y!)))
  (predicate point-2d?))
```

This code segment defines a new class `<point-2d>`, which is a subclass of `<object>`. It has two direct slots (`x` and `y`), plus any further slots it inherits from its base class (in this case none, as the class `<object>` defines no slots). Both slots have an associated reader and writer generic function, appropriate methods for reading and writing `x` and `y` are automatically added; the generic functions must be defined beforehand. The `predicate` class option specifies that predicate methods should automatically be generated for the specified generic function (returning true for all general instances of the defined class, and false else).

The `predicate` class option is one of two predefined class options. The other one is `instance-of`, which is used to specify the meta-class of the newly defined class. Normally, class objects are instances of the class `<class>`, but if another class is specified, that one is used. As with generic functions, class options can be extended and added by the user of the MOP.

The full form of the macro is:

```
(define-class <class-name> (<superclass> ...)
  ((<slot-name> <slot-option> ...) ...)
  <class-option> ...)
```

4.2. Design considerations and concepts

This section explores the criteria that were shown at the end of chapter 3, motivating the design decisions that were taken when creating the proposed meta-object protocol and showing different choices.

There are many criteria for evaluating object systems, the ones chosen here are those deemed most important by the author (and corroborated in [15] and [4]) for creating an object system and meta-object protocol that are applicable to a wide range of problems and nonetheless retain enough power and expressivity to enable efficient and effective implementations.

4.2.1. Customising slot accessors

A very common operation in all object systems is access to "fields" or "members" (what we call "slots") of an object. In all investigated systems this is a major building block for abstractions. Several systems intermingle slot access with visibility and data hiding concerns (often making the situation even more complex due to the connections between visibility and inheritance).

A definite design goal is the ability to change the semantics of slot access. Consider a new "type" of objects, where all slot accesses are logged (in order to reconstruct later when slot values were accessed or modified). In order to implement this new class of objects in pure Smalltalk or Dylan, one would have to write the corresponding functionality by hand in *each* slot accessor for each class that uses it. There is no way to specify that – instead of the usual "normal" accessor – an extended version should be used. In Ruby, this is easily done, however the costs are prohibitive in cases where efficiency is needed.

CLOS allows interceding in slot access in arbitrary ways, which precludes efficient implementation. As there is one generic function (`slot-value`) in CLOS which governs the access to *any* slots of any class – with potentially hundreds of methods – static optimisation is very hard.

TELOS and OpenC++ both provide this functionality, by overriding the way slot accessors are created in a way that only relates to the class (and potentially a specific generic function).

4.2.2. Customising object layout

Connected to the previous criterion is the ability to change the actual memory layout of objects. If a specific slot value is always calculated, read from a database or transferred via a network, there is often no need to reserve space for it in the object instance. When creating such an object, memory should only be reserved for the slots for which this is necessary, as denoted by implicit rules or by the programmer. As for the previous criterion, neither pure Smalltalk nor Dylan provide means of achieving this. Ruby again makes it easy, at great runtime cost.

OpenC++ and CLOS both provide full customisability of object layout. OpenC++ does so by allowing the programmer to define the transformations of new object definitions (so that an object could be "unrolled" by allocating all fields as local variables instead of allocating it on the heap, for example). CLOS (and TELOS) provide generic functions that are called on each class instantiation, thus allowing arbitrary object layout in connection with custom slot accessors. However, the need to always call a generic function (with full multiple dispatch) is a potential problem if many objects are allocated.

4.2.3. Enabling static optimisation

Even though computers have become faster and faster, there are still many applications for which runtime matters. The meta-object protocol itself should not *preclude* efficient implementation (even though the default implementation may not implement all possible optimisations).

Dylan, OpenC++, Smalltalk and TELOS all fulfil this criterion, two due to the simplicity of their MOP (Dylan, Smalltalk), the other two (OpenC++ and TELOS) due to design decisions. OpenC++ was designed from the start to be a compile-time MOP, thus allowing as much optimisation as possible at compile-time. TELOS too was based on CLOS, but with the additional goal of making optimisation easier than it is in CLOS.

The main criterion for enabling static optimisation is providing the phases before runtime with as much information as possible on function calls. CLOS uses generic functions in many key places, which are hard for the compiler to analyse completely. By reducing the amount of generic functions that must be called in a program, runtime costs are diminished.

4.2.4. Meta-information accessible at runtime

The above three points dealt with intercessory abilities of the meta-object protocol. In addition to interfering in the execution of object-oriented programs, one very important characteristic of meta-object protocols is the ability to reflect upon programs. All the investigated languages except OpenC++ offer full reflective capabilities, e.g. inspecting the available classes, the slots of objects, the inheritance hierarchy, etc.

OpenC++ allows the programmer to add this information where needed by implementing the proper mechanisms herself, but provides no additional support for reflection.

4.3. Object Instantiation Protocol

The object instantiation protocol governs the creation of all objects, i.e. "normal" objects and meta-objects. The following steps are taken when instantiating a class:

1. The generic function `make` is called to instantiate an object of a class
2. Storage is allocated for the object (including storage for all necessary slots)
3. The generic function `initialize` is called on the uninitialised object, in order to correctly initialise the object according to any parameters passed to `make`

Note that there is only one generic function call (the call to `initialize`) after the initial call to `make`. This is by design, object instantiation should be as efficient as possible. A detailed description of the steps follows:

```
(make <class> <parameters>)
  ;; call slot storage allocator
  ;; allocate object
  (initialize <object> <parameters>)
```

Figure 4.3.: Object Instantiation Protocol

`make`

The generic function `make` is the common object instantiation function. Objects of all classes are created by calling this generic function. The first parameter must be an object of type `<class>` (defining the class that is instantiated). Any further parameters are passed along to `initialize`.

The default behaviour of the generic function `make` is to allocate slot storage and return a new object with the given class and the newly allocated slot storage. Overriding `make` is useful in cases where this method should not actually allocate a new object (but return an existing one).

Allocating storage

Storage is allocated in a two-step process:

1. Storage for slots is allocated
2. The object is allocated

Conceptually, an object consists of a pointer to the class and a slot storage (the actual object format is an implementation detail and cannot be changed). The user of the meta-object protocol can change the slot storage. This is done by adding a method to the generic function `calculate-slot-storage-allocator`. This method takes one parameter (the class to calculate the slot storage allocator for) and returns a function of zero parameters, that allocates the actual slot storage. The default behaviour is to return a function that allocates a vector of the same size as the number of effective slots.

Overriding the slot storage can be used to implement different slot allocations (like class-allocated slots, or hash-table-based slots). The generic function for calculating the slot storage allocator, `calculate-slot-storage-allocator`, is only called once, when the class object is instantiated, so it can only be used in the proper way in meta-classes.

`initialize`

After the object and slot storage is allocated, `make` calls `initialize` with the newly allocated object as the first parameter, and any parameters left from the original call to

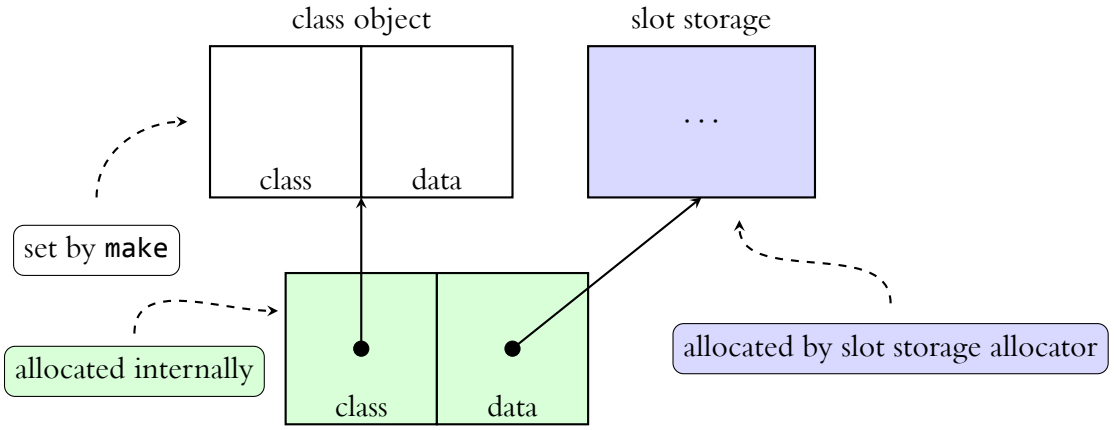

Figure 4.4.: Object instantiation protocol

`make`. The methods of `initialize` must finish any initialisation that is needed beyond pure allocation. The default behaviour is to do nothing for normal objects. For subclasses of `<class>`, the default behaviour is to correctly initialise the new class object.

New methods on `initialize` can be used to e.g. support automatic initialisation of slots via parameters to `make`, count the number of objects created, run custom initialisation code, etc. The generic function `initialize` is the equivalent of a constructor in Java or C++.

4.3.1. Design Decisions and Rationale

Creating new objects is a very common operation in all systems. If intercession is wanted, it should be as cheap as possible in terms of runtime overhead. This means that there should *not* be a generic function call on *each* object instantiation. For this reason, the proposed object instantiation protocol does not define a generic function to be called on each instantiation, but instead defines a generic function that is called upon *class* creation, which should return an allocator function. This is normally a standard Scheme function, which just returns a new object, guaranteeing no additional overhead beyond a normal Scheme function call. If needed, the user of the meta-object protocol can use a generic function here.

The object instantiation protocol controls the creation of new objects. It deals with allocating storage and initialising the object. These two steps can both be influenced, the first statically via a functional protocol – `calculate-slot-storage-allocator` is called once, when the class object is created; it returns an allocator function, which is then called to allocate actual objects – the latter via a procedural protocol – the generic function `initialize` is called for each object that is initialised. The main instantiation function (`make`) also follows a procedural protocol.

The decision to keep `make` and `initialize` procedural (as opposed to making them both functional) is based on the fact that, for a functional protocol to work, a new meta-class needs to be introduced, because functional protocols may be executed only once. The functional protocol then applies to all instances of instances of this meta-class. This is a noticeable design overhead for the common operation of customising object initialisation, in these cases the cost of a generic function call seems justified.

Customising slot storage allocation is a much less common operation, in this case introducing a new meta-class seems a sound design decision, as this constitutes a new class of objects.

In total, two generic function calls are necessary for each object instantiation. If multiple objects of the same class are instantiated in a row, caching in the generic function dispatch can amortise the costs.

4.4. Class Creation Protocol

The class creation protocol specifies all necessary detail for creating new classes. By making use of the class creation protocol, the user of the MOP can influence the way classes (and in turn instances of these classes) are generated.

The class creation protocol is the most complex of the protocols, it governs the following:

- The generic function `make` is called to instantiate an object of a class type
- The normal object instantiation protocol is followed, but the generic function `initialize` additionally does the following steps:
- Calculate and correctly order the superclasses
- Calculate a slot storage allocator function to use with instances of this class
- Calculate the effective slots
- Apply class options to the newly created class object
- Ensure that valid accessors for the effective slots are generated

Each of these steps is to be implemented by a generic function, and can be influenced by the user of the MOP by adding additional methods to them. A detailed explanation of each step is given in the next section. The entire class creation protocol is a functional protocol, all generic functions mentioned here are called for their results when the class is generated; they are not called again after class instantiation is finalized.

```
(initialize <class> <parameters>)
  (calculate-and-order-superclasses <class>
                                    <superclasses>)
  (calculate-slot-storage-allocator <class>)
  (apply-class-options <class> <options>)
  (ensure-slots-accessors <class> <slots>)
    ;; details in the Slot Accessor Protocol
```

Figure 4.5.: Class creation protocol

Calculation and ordering of superclasses

When creating a new class, the programmer passes a list of superclasses. This list may be empty, or may contain one or more classes. Based on this list, the generic function `calculate-and-order-superclasses` calculates a list of effective superclasses, and orders them according to their precedence. The default method keeps the list as it is given; if the list is empty, the single class `<object>` is inserted as the base class. This generic function is called once when the class object is created, and should return a list of superclasses.

The user of the MOP may wish to override this method to achieve specific effects in combination with multiple inheritance or e.g. to enforce of mixin-semantics.

Calculation of the slot storage allocator function

When creating the class object, a custom slot storage allocator is calculated by calling the generic function `calculate-slot-storage-allocator`. This allocator is called whenever a new instance of the class is allocated, it should return the appropriate slot storage for this instance.

As mentioned in 4.3, this is useful for classes that have special storage allocation, where the default (a vector with one field per slot) is not sufficient.

Calculation of effective slots

The effective slots of a class are a combination of the direct slots defined explicitly for this class, and all slots that are inherited from superclasses. The slot descriptors for these slots must be combined (and maybe merged, if a superclass contains a slot with the same name as the subclass). The generic function `effective-slot-descriptors` is called to calculate this list. The default method for instances of `<class>` just concatenates the direct slots of the newly generated class and the effective slots of all superclasses.

Overriding this method allows special effects in slot inheritance, the user of the MOP can decide which slot options to inherit, and how to merge them. This method also

influences calculation of slots if multiple inheritance is used and more than one base class has a given slot.

Application of class options

The options that are given when defining the class are processed by this function. Each option consists of a key (a symbol) and a list of arbitrary values. If a programmer wants to support a new class option, she should add another method to the generic function `apply-class-options` that tests for this key in the list of options, and takes appropriate action. Normally, the next method should then be called (to process class options that are handled higher up in the inheritance hierarchy).

Ensuring of accessors for effective slots

When a class is generated, each effective slot must be checked for the existence of proper accessor methods (if specified). For each effective slot in a class, if there is a reader or writer generic function specified, a method is added to that generic function that correctly accesses the slot in question. If the access method has not changed compared to the superclass, no new method is necessary. For example, by default for the single–inheritance case, all new slots are added at the end of the slots list, so the position of a slot never moves. There is no support for removing existing slots in sub–classes (as this would violate the interface contracts). The Accessors of the superclass can be used on all subclasses.

```
(ensure-slots-accessors <class> <slots>)
  (ensure-slot-accessors <class> <slot> <slots>)
    (ensure-slot-reader <class> <slot> <slots> <reader>)
    (ensure-slot-writer <class> <slot> <slots> <reader>)
```

Figure 4.6.: Slot Accessor Protocol

The generic function `ensure-slots-accessors` is called, which in turn calls the generic function `ensure-slot-accessors` for each slot. The methods of this generic function must ensure that a proper accessor method is registered with the generic reader or writer function. It does this by calling in turn `ensure-slot-reader` and (if necessary) `ensure-slot-writer`. These methods register the actual reader and writer method respectively with the generic reader and writer functions.

Overriding the appropriate methods of these generic functions allows the programmer to intercede when a slot is accessed for reading or writing. This enables efficient yet simple implementation of e.g. tracing accesses, controlling accesses, providing persistent slots, etc.

4.4.1. Design Decisions and Rationale

All investigated systems offer numerous ways to intercede in class definition and creation. The common pattern in all of them is the ability to customise which code is generated for slot accessors. The user must be able to generate custom slot accessors, that are not significantly slower than hand-written accessors would be.

As the creation of class objects is usually not a frequent operation, many generic functions are involved in order to provide a maximum of customisability. Care has been taken to eliminate generic functions from the actual productive code in *instances* of the classes (e.g. no generic functions are involved in slot access beyond the actual accessor function call). The protocol allows the programmer to adapt all the main characteristics of classes – inheritance, slots, slot accesses, and general class options – to his or her needs.

4.5. Generic Function Invocation Protocol

Probably the most central point of the entire object system is generic function invocation. Dynamic dispatch is the main means of extending existing objects, thus needs to be as efficient as possible. The generic function invocation protocol specifies many entrance points where customisation can occur:

- When a generic function is applied to any arguments, a primitive function is invoked, that directly calls the generic function's discriminating function with the arguments.
- The discriminating function by default calls `apply-generic-function`.
- `apply-generic-function` first filters and sorts the applicable-methods by calling `sort-applicable-methods`, then calls the first applicable method with the given parameters, and a function that – if called – invokes the next-most applicable method of the remaining applicable methods.
- `sort-applicable-methods` first calls the generic function `applicable-methods` to get a list of methods that are applicable to the given arguments, then sorts them according to the generic function `method-more-specific?`.
- The generic function `method-more-specific?` in turn invokes the generic function `specializer-more-specific?` to find out which of two methods is more specific.

All of these generic functions can be overridden or extended by the user of the MOP for new meta-classes. As generic function invocation plays a central rôle in the object system, the user cannot override the default behaviour (which is optimised for speed) for the existing meta-classes (but it can be arbitrarily extended for new meta-classes).

```
(apply-generic-function <generic-function> <arguments>)
  (sort-applicable-methods <generic-function>
                           <arguments>)
    (applicable-methods <generic-function> <arguments>)
    (method-more-specific? <method-1> <method-2>)
      (specializer-more-specific? <method-1>
                                  <method-2>
                                  <specializer-1>
                                  <specializer-2>)
;; apply the methods in order of specificity
```

Figure 4.7.: Generic Function Invocation Protocol

A programmer using the MOP can influence the generic dispatch on multiple levels, from the most general – overriding the discriminating function, which decides on which methods to apply in which order – to the most specific, e.g. influencing the exact ordering of methods depending on specialisers.

Figure 4.8 to 4.10 show the three most important steps in executing a call to a generic function:

- Select only the applicable methods from the set of all methods of this generic function (figure 4.8).

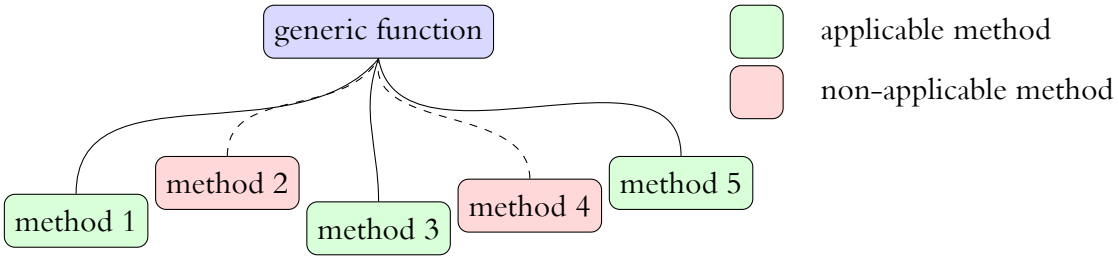

Figure 4.8.: Generic Function Invocation – Filter methods

- Sort all applicable methods (by the generic function `method-more-specific?`), to find the correct order for applying the applicable methods (figure 4.9).

- Call the most specific method with the parameters, including the next-most-specific method (which, if it is called in turn, will be passed the next-most-specific method, etc.) (figure 4.10).

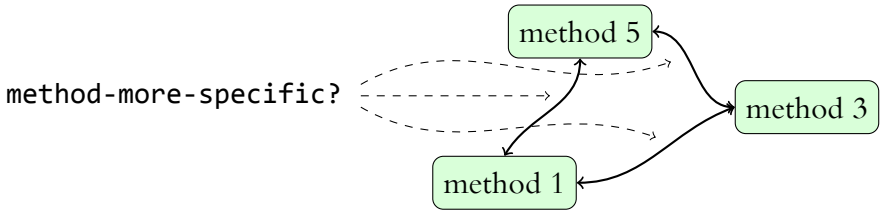

Figure 4.9.: Generic Function Invocation – Sort methods

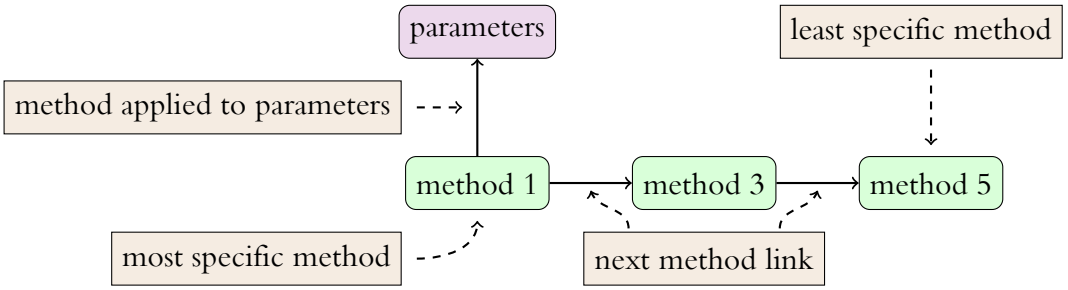

Figure 4.10.: Generic Function Invocation – Apply methods

4.5.1. Design Decisions and Rationale

The most complex part of the entire proposed meta-object protocol is generic function invocation. This is potentially the most important bottleneck in terms of runtime speed, as all other parts of the system depend on it. Special care has been taken to ensure that the user can tweak and optimise in many places.

It seems like the generic function invocation protocol contains many generic functions in turn, thus rendering it slow. However, the restriction that it cannot be overridden for the default classes (`<generic-function>`, etc.) means that these cases can be implemented efficiently. The possibility to override the discriminating function gives the user a handle to implement other efficient dispatchers as well.

The most efficient dispatcher is a (non-generic) function that just checks whether the correct number and type of arguments has been provided. This implementation incurs no overhead beyond the checking (as basic Scheme offers no typing of functions, this must be implemented anyway). More complex implementations could cache the most recently used method (thus speeding up dispatches to that exact method), or apply advanced heuristics when eliminating non-applicable methods.

5. Implementation

The design described in the above sections was implemented in MIT/GNU Scheme [20] by the author. This implementation is described in the following section.

The implementation consists of several layers (from bottom to top):

- The basic object layer
- The implementation of generic function invocation
- The implementation of the various meta-object protocols
- The syntactic extensions

Only the lowermost two layers (basic objects and generic function invocation) are implementation-specific, the others are portable across Scheme implementations.

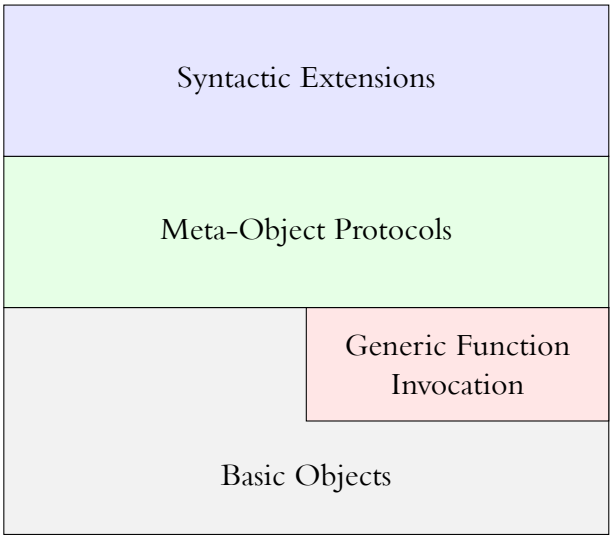

Figure 5.1.: Implementation layers

5.1. The basic object layer

Objects are implemented as a record type, consisting of two fields:

- The class of the object (i.e. a pointer to the class)
- The slot data of the object (i.e. a pointer to the slot data)

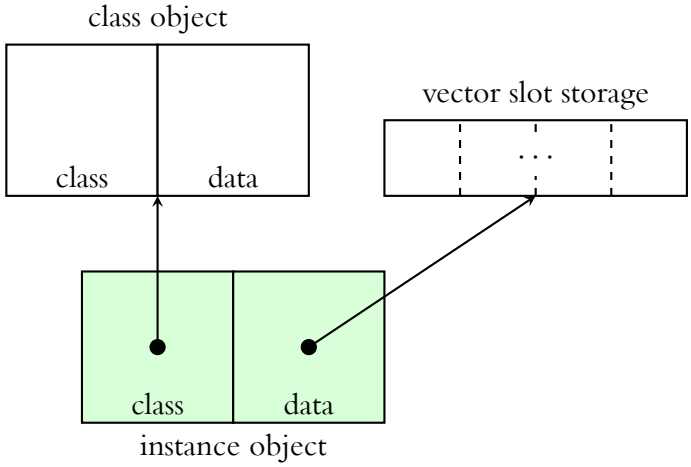

Figure 5.2.: Basic Object Layout

In addition to objects, several primitive data types (numbers, lists, symbols, etc.) are also supported transparently. This representation of objects was chosen because it allows the object identity to stay the same even when the slot data changes (this is needed to support operations like changing the class of an object). Figure 5.2 shows that the instance object only contains two pointers, one to a class object, one to the slot storage (which may be an object in the strict sense of the object system, but could also be a simple hashmap or vector – as it is in the default implementation).

This layer defines accessors for the class and data of objects, which are only used in the implementation of the meta-object protocols, they should not be used by the user of the meta-object protocol.

The object system defines the classes and slots shown in figure 5.3.

5.2. Generic Function Invocation

Generic function invocation is a special case, because a generic function is a new type of object that can be applied to parameters, but must be distinct from normal functions. MIT/GNU Scheme offers the concept of *apply hooks* here, which allows arbitrary data to be linked to functions. Thus the additional data needed for generic functions is stored in a generic function object which is bound to a simple trampoline function (which just accesses the generic function object and calls the discriminating function). This layer implements transparent conversion between generic functions and their data objects.

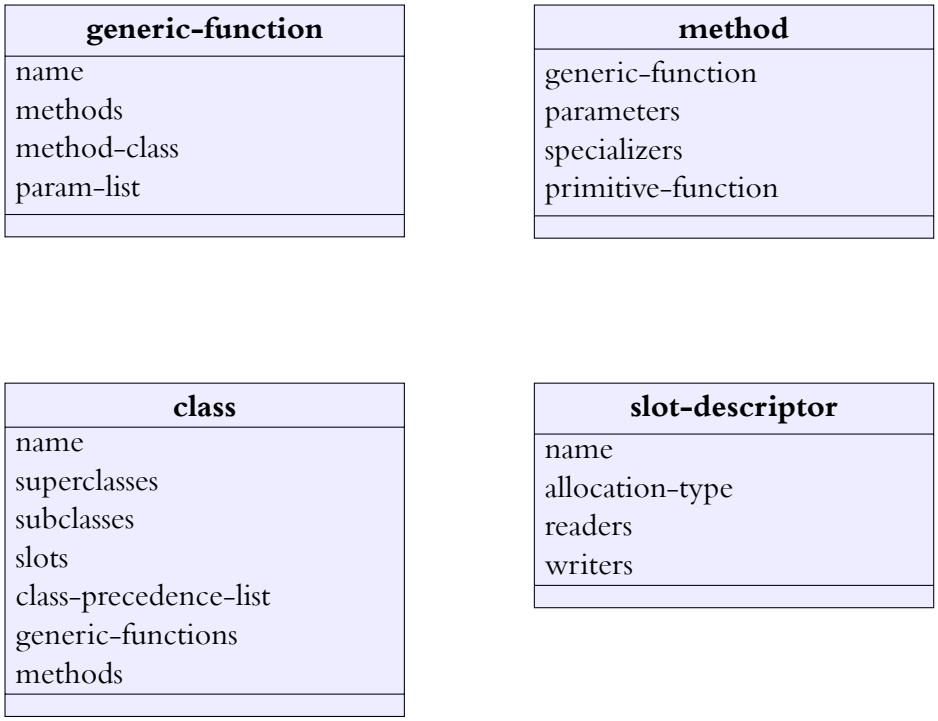

Figure 5.3.: Object system classes

5.3. The meta-object protocols

This is the main part of the implementation. All the generic functions and classes described in the sections above are implemented here. Mostly this is a direct application of the specification, the implementation-specific parts are:

- The default mechanism for instance slots (the object data) are fixed-size vectors, with one field per slot
- The function lambda of a method always receives a first parameter `next-method` in addition to any other parameters
- The generic dispatch functions for Direct Instances of various classes have been hardcoded (for efficiency and to eliminate infinite recursion). The specification forbids modification of the default methods for these classes, so this is not a problem.

The problems related to bootstrapping the system have been solved – as is usual – by defining the necessary structures by hand (i.e. without the use of convenient syntactic extensions) before using them.

5.4. Syntactic Extensions

The topmost layer of the implementation contains the following macros:

- `define-class`
- `define-generic-function`
- `define-method`

These are only for convenience, they do not offer any additional functionality, yet render the abstractions more convenient to use.

As an example, the macro `define-generic-function` is implemented as follows:

```
(define-syntax define-generic-function
    (rsc-macro-transformer
      (lambda (form environment)
        (define (make-argument argument)
          `(list ',(car argument) ,@(cdr argument)))
        (define (find-metaclass options)
          (let ((metaclass (assq 'instance-of options)))
            (if metaclass
              (cadr metaclass)
              '<generic-function>)))
        (define (make-option option)
          (make-argument option))
        (let* ((name (caadr form))
               (params (cdadr form))
               (options (cddr form))
               (metaclass (find-metaclass options)))
          `(,(close-syntax 'define environment)
            ,name
            (,(close-syntax 'make environment)
              ,metaclass
              ',name
              '(,@params)
              (list ,@(map make-option options)))))))))
```

It is defined as a macro transformer that transforms the input code to a definition using `define` and `make`. Its main task is to find the appropriate meta-class and to correctly transform generic function options.

5.5. Recapitulation of the additions to plain Scheme

This section shows in detail which concepts are provided by the Scheme language, and which are added to that by the proposed object system and meta-object protocol.

Basic Scheme ([17] extended by MIT/GNU Scheme) includes the following elements relevant to our implementation:

- Record type definitions (includes defining new predicates, constructors and accessors for a record type, but does not support inheritance)
- Function definitions (untyped and non-overloaded first-order functions)
- Basic datatypes (numbers, strings, etc. and operations on them)

In our implementation, all objects are instances of one basic record type (called `object` internally). Class types do not map onto distinct record types, instead each object includes a pointer to its class.

Generic functions are untyped functions that accept any number and type of parameters. When called, they check whether the correct number of parameters was supplied, and look for a method to call. Each method is again an untyped method that accepts a certain number of parameters. In addition, each method contains information on which type of parameters it is applicable to. This implementation (one generic function contains multiple methods) adds overloading and typing to generic functions (which doesn't exist in plain Scheme).

The entire class hierarchy and operations on it hinge on generic functions. All reflective operations and intercessory meta-object protocols build on generic functions.

There are no changes on the compiler level, the entire meta-object protocol and object system is an addition to plain Scheme, there was no need to modify the existing compiler or runtime (showing the extent to which Scheme can be customised). This also means that any programmer can use both plain Scheme and our extended meta-object protocol in the same program without performance hits on non-meta-object protocol parts. If the extensions are not used, they do not cost anything in terms of runtime.

6. Applications and Results

This chapter describes sample applications that were developed using the meta-object protocol specified in chapter 4. They range from simple to more complex, mostly the applications do not depend on external libraries (with the exception of section 6.5).

The prototypical applications are:

- Classes that have a hashtable-like slot storage (i.e. they only allocate space for slots that actually contain values)
- Generic functions that print a trace of their execution (the arguments when the function is entered, the results when it exits)
- Classes that have "common" slots, which share a value for all instances (also known as "static" fields in other languages)
- Generic functions that can dispatch not only based on which class they belong to, but also based on general predicates. As an extension, methods can even be specialised on single objects.
- Classes that automatically store their slots in a database upon each change. This can be used as a persistent storage, that is automatically initialised from the database upon program startup.

The section on each application first describes the wanted effects, then proceeds to describe how these can be achieved by means of the meta-object protocol.

6.1. Hashtable-Storage Classes

Traditionally, the slots of objects are stored in a vector-like fashion. Internally, all objects of a class are the same size, regardless of how many slots actually contain values. If an object has a great number of slots, yet only uses a few of them, much space is wasted. On the other hand, the vector representation is efficient, as all accesses are in constant time.

We would like a way of changing the slot storage model from vector-like to a slot hash table, where only the slots that actually have values take up space.

Most programming languages do not let the programmer decide which of these two models is better suited for her problems, but instead incorporate one or the other in their

design. By making use of the object-instantiation protocol (see 4.3), we can create a new meta-class, which changes the slot storage model for all its class instances.

This is achieved in two steps:

- Instead of allocating a vector for slot storage, allocate a hash table.
- Instead of defining readers and writers that access a vector, define readers and writers that access a hash table.

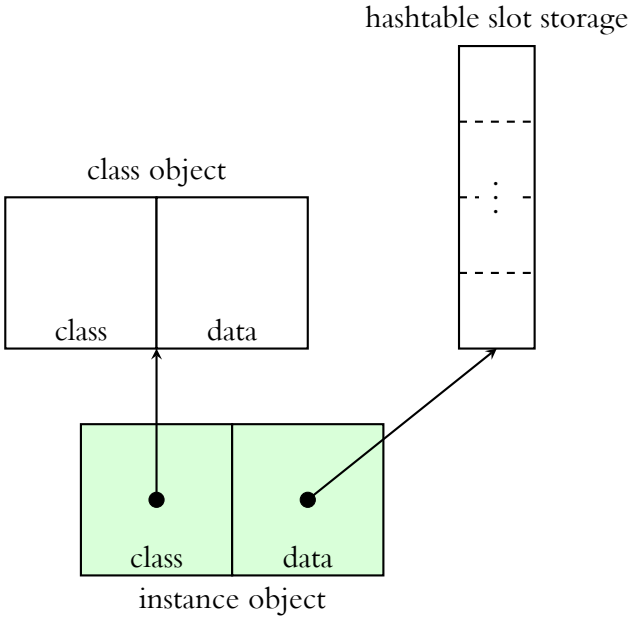

Figure 6.1.: Hashtable-storage Instance Layout

6.1.1. Implementation

First, a new class is defined, that serves as the meta-class for all classes that have hash table slots:

```
(define-class <hashing-class> (<class>)
  ())
```

Then – to change allocation from a vector to a hash table – the slot storage allocator function is overridden, according to 4.3:

```
(define-method (calculate-slot-storage-allocator
                (class <hashing-class>))
  (lambda () (make-strong-eq-hash-table)))
```

Note that this allocator does not use the actual slots of the class for deciding what to allocate (as opposed to the vector allocator, which must inspect the class to find out the size of the vector).

At last, the reader and writer generator functions are overridden (only the reader is shown here, the writer is similar):

```
(define-method
  (ensure-slot-reader (class <hashing-class>)
                      (slot <slot-descriptor>)
                      slots
                      (reader <generic-function>))
          (let ((reader-method
                 (make-method
                  reader
                  (list class)
                  (lambda (next-method obj)
                   (hash-table/get (object-data obj)
                                   (slot-descriptor-name
                                     slot)
                                   #f)))))
            (add-method! reader reader-method)
            reader-method))
```

Here, a reader method is created, which accesses the hash table (with `hash-table/get`). It is subsequently added to the reader generic function, and returned.

These simple steps have given us a new meta-class `<hashing-class>`, which we can use as the meta-class for any new class and which modifies the slot storage behaviour of all instances of these classes.

The following code sequence shows that instances of these classes work as expected:

```
(define-class <hash-slots> (<object>)
  ((a (reader a) (writer a!))
   (b (reader b) (writer b!))
   (c (reader c) (writer c!)))
  (instance-of <hashing-class>))

(define-class <vector-slots> (<object>)
  ((a (reader a) (writer a!))
   (b (reader b) (writer b!))
   (c (reader c) (writer c!))))

(define hash-instance (make <hash-slots>))
(define vector-instance (make <vector-slots>))
```

```
(hash-table? (object-data hash-instance))
=> #t

(vector? (object-data hash-instance))
=> #f

(vector? (object-data vector-instance))
=> #t
```

We define two classes here, one that uses the normal vector slot storage (which is called `<vector-slots>` and one that uses the hash table slot storage (`<hash-slots>`). For testing purposes, we use the undocumented internal primitive `object-data` to access the slot storage and see that the expected results emerge. For the user of these classes, there is no discernible difference, the slot accessors and the instance generator hide the fact that the internal slot storage is different:

```
(a! hash-instance 4)
(a hash-instance)
=> 4

(a! vector-instance 5)
(a vector-instance)
=> 5
```

6.2. Tracing Function Calls

When developing or debugging a software system, we often want to trace function entry and exit. We want to see which parameters were passed to a function, and which results it returned. One way to achieve this is to add some form of output statements right after the function start and before every return point. However, this is error-prone, and tedious. The following section shows how to achieve this effect by defining a new meta-class for traced generic functions.

In order to trace every function entry and exit, we need to intercept a generic function right at the point of its call. For this, we can override the discriminating function with a custom function, which prints the parameters, then calls the original discriminating function, prints the results, and then returns the original results.

6.2.1. Implementation

The following code section shows how to achieve this. We assume here the existence of two functions `enter-function` and `exit-function` which deal with nicely printing

and indenting the function parameters and return values.

```
(define-class <tracing-generic-function>
              (<generic-function>)
  ())

(define-method (compute-discriminating-function
                (gf <tracing-generic-function>) . args)
  (let ((dp (next-method)))
    (lambda args
      (enter-function gf args)
      (let ((res (apply dp args)))
        (exit-function gf (list res))
        res))))
```

It now suffices to declare a generic function (via the `instance-of` option) as an instance of the meta-class `<tracing-generic-function>` (instead of the default meta-class `<generic-function>`) to get this tracing behaviour for all methods. As an example, we trace the execution of the Ackermann function (appropriately named [1]) by adding `(instance-of <tracing-generic-function>)` as a generic function option specifying the class that `ackermann` should be an instance of:

```
(define-generic-function (ackermann m n)
  (instance-of <tracing-generic-function>))

(define-method (ackermann (m <integer>) (n <integer>))
  (if (= m 0)
      (+ n 1)
      (if (and (> m 0)
               (= n 0))
          (ackermann (- m 1) 1)
          (ackermann (- m 1) (ackermann m (- n 1))))))

(ackermann 2 2)
=>
[Enter (ackermann 2 2)\n  [Enter (ackermann 2 1)
    [Enter (ackermann 2 0)
      [Enter (ackermann 1 1)
        [Enter (ackermann 1 0)
          [Enter (ackermann 0 1)
           Leave ackermann 2]
         Leave ackermann 2]
        [Enter (ackermann 0 2)
```

```
        Leave ackermann 3]
      Leave ackermann 3]
    Leave ackermann 3]
  [Enter (ackermann 1 3)
    [Enter (ackermann 1 2)
      [Enter (ackermann 1 1)
        [Enter (ackermann 1 0)
          [Enter (ackermann 0 1)
           Leave ackermann 2]
         Leave ackermann 2]
        [Enter (ackermann 0 2)
         Leave ackermann 3]
       Leave ackermann 3]
      [Enter (ackermann 0 3)
       Leave ackermann 4]
     Leave ackermann 4]
    [Enter (ackermann 0 4)
     Leave ackermann 5]
   Leave ackermann 5]
 Leave ackermann 5]
[Enter (ackermann 1 5)
  [Enter (ackermann 1 4)
    [Enter (ackermann 1 3)
      [Enter (ackermann 1 2)
        [Enter (ackermann 1 1)
          [Enter (ackermann 1 0)
            [Enter (ackermann 0 1)
             Leave ackermann 2]
           Leave ackermann 2]
          [Enter (ackermann 0 2)
           Leave ackermann 3]
         Leave ackermann 3]
        [Enter (ackermann 0 3)
         Leave ackermann 4]
       Leave ackermann 4]
      [Enter (ackermann 0 4)
       Leave ackermann 5]
     Leave ackermann 5]
    [Enter (ackermann 0 5)
     Leave ackermann 6]
```

```
     Leave ackermann 6]
    [Enter (ackermann 0 6)
     Leave ackermann 7]
   Leave ackermann 7]
 Leave ackermann 7]
```

6.3. Class-allocated Slots

Many object-oriented programming languages discern per-instance (or dynamic) fields from per-class (or static) fields. By default, all slots in our object system are allocated per instance, i.e. each instance storage location for each slot is distinct from the storage locations of the same slot in other instances. If class-allocated slots are necessary, we can achieve this as follows:

- Define a new class for slot descriptors that support an allocation type
- Extend instance initialisation to support slots of the new class correctly (by not allocating storage in each instance)
- Extend class initialisation to support slots of the new class correctly (by allocating storage for each class-allocated slot)
- Override reader and writer generation to automatically define accessor functions that access the class object, not the instance, when accessing class-allocated slots

Note that it would be possible to create other types of allocation (not just per-instance and per-class).

We will store all class-allocated slots in a vector, which is a member of the class object.

6.3.1. Implementation

The following code shows how to implement these requirements. First we define a new meta-class that serves as a base class for all classes which should support class-allocated slots. This class also includes a slot for the values of all class-allocated slots:

```
(define-class <class-slots-class> (<class>)
  ((slots (reader class-allocated-slots)
          (writer class-allocated-slots!))))
```

We override the slot storage allocator, so that our instances only allocate slots with a slot allocation of type *instance*:

```
(define-method (calculate-slot-storage-allocator
                (class <class-slots-class>))
  (lambda ()
    (let ((nr-slots
           (length (filter
                    (lambda (slot)
                      (eq? (slot-allocation slot)
                           'instance))
                    (effective-slot-descriptors class)))))
      (make-vector nr-slots #f))))
```

We define a new class `<allocated-slot>` to describe slots with an allocation type (either *instance* or *class*) and set this class as the class to use for slot descriptors used in class `<class-slots-class>`:

```
(define-class <allocated-slot> (<slot-descriptor>)
  ((allocation (reader class-slot-allocation)
               (writer class-slot-allocation!)))
  (predicate allocated-slot?))

(class-slot-descriptor-class! <class-slots-class>
                              <allocated-slot>)
```

Then we override the initialisation function of instances of class `<class-slots-class>`, in order to correctly initialise the slot storage for class-allocated slots by creating a new vector of the correct size:

```
(define-method (initialize (class <class-slots-class>)
                           params)
  (let* ((obj (next-method))
         (class-slots (filter-class-allocated-slots class))
         (class-slot-storage (make-vector
                              (length class-slots)
                              #f)))
    (class-allocated-slots! obj class-slot-storage)
    obj))
```

We also override the generation of reader and writer functions (only readers are shown here, writers work similarly) to access the class object if the slot allocation of a given slot is *class*:

```
(define-method
  (ensure-slot-reader (class <class-slots-class>)
                      (slot <allocated-slot>)
```

```
                    slots
                    (reader <generic-function>))
  (if (eq? (slot-allocation slot) 'class)
      (let* ((slot-index (position
                          slot
                          (filter-class-allocated-slots
                            class)))
             (reader-method
              (make-method reader
                           (list class)
                           (lambda (next-method obj)
                             (vector-ref
                               (class-allocated-slots
                                 (class-of obj))
                               slot-index)))))
        (add-method! reader reader-method)
        reader-method)
      (next-method)))
```

We can now define new classes (with meta-class `<class-slots-class>`) which have an explicit allocation type:

```
(define-class <static> (<object>)
  ((private (reader private) (writer private!))
   (shared (reader shared)
           (writer shared!)
           (allocation 'class)))
  (instance-of <class-slots-class>))
```

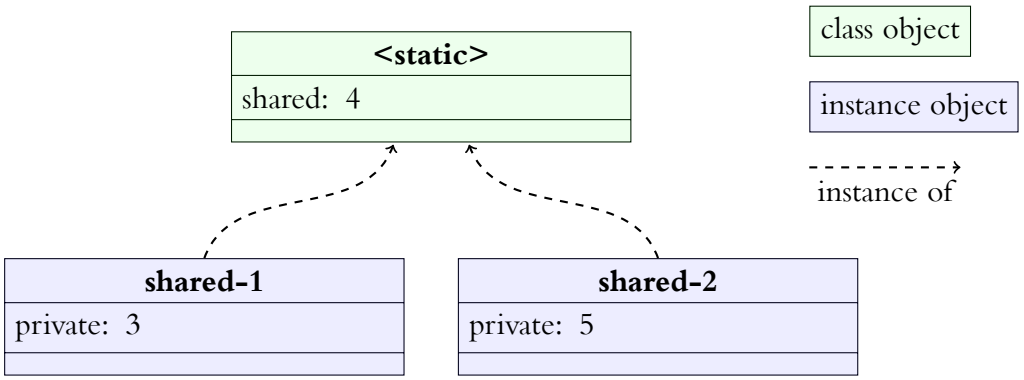

Figure 6.2.: Class-allocated slots example

This class has two slots, one with no explicit allocation type (slot `private`, defaulting to *instance*), and one with allocation type *class* (slot `shared`). All objects of this class share a single slot `shared`, but have their own versions of slot `private`:

```
(define static-1 (make <static>))
(define static-2 (make <static>))

(private! static-1 3)
(private! static-2 5)
(private static-1)
=> 3
(private static-2)
=> 5
(shared! static-1 4)
(shared static-2)
=> 4
(shared! static-2 7)
(shared static-1)
=> 7
```

6.4. Predicate- and Singleton-based dispatch

The normal dispatch mechanism of methods of generic functions takes the classes of the parameters into account. This is only one view onto the object hierarchy, defined by static considerations. The meta-object protocol shown in chapter 4 offers extensibility, many other schemes can be implemented. An example shown here is predicate dispatch: The specialisers of methods can be predicates, if the predicate matches, this is seen as more specific than a class type match. If multiple methods match in the same predicates, a runtime error is raised. This scheme can be used to implement primitive pattern matching for separating recursive base cases from the generic case.

A further extension of this mechanism is singleton-based dispatch, i.e. a method is applicable if and only if an exact object is passed as a parameter. This enables us to write methods that are only applicable to the object `<class>`, for example, or to write recursive base cases more easily (by specialising on the object 0 or the empty list etc.).

To implement predicate-based dispatch as an extension to the normal class-based dispatch mechanism, we need to do the following:

- Override the applicability check for methods to take into account predicates
- Override the check for method specificity, so that predicate-specialiser methods are more specific than class-based methods

For singleton-based dispatch, the changes in the same places are necessary. Singleton-based dispatch is always the most specific (as there is only exactly one element that matches).

6.4.1. Implementation

The following code section shows how this can be achieved:

First, we define new meta-classes for predicate-specialiser methods and generic functions:

```
(define-class <predicate-specializer-method> (<method>)
  ())
(define-class <predicate-specializer-generic-function>
              (<generic-function>)
              ())
```

Then we override the generic function `specializer-more-specific?` for predicate-specialiser methods. Whether a specialiser is a predicate can be checked with `procedure?`:

```
(define-method (specializer-more-specific?
                (method-1 <predicate-specializer-method>)
                (method-2 <predicate-specializer-method>)
                (s1 <object>)
                (s2 <object>))
               (if (and (procedure? s1)
                        (procedure? s2))
                (error "ambiguous␣predicate␣specializers")
                (if (or (procedure? s1)
                        (procedure? s2))
                    (procedure? s1)
                    (next-method))))
```

This implements the error check that there cannot be multiple predicates on the same parameter for different methods. Methods with a predicate-specialiser are more specific than those with other specialisers.

Lastly, we need to override `method-applicable?`, to take into account predicate matches:

```
(define-method (method-applicable?
                (method <predicate-specializer-method>)
                arguments)
  (let loop ((remaining-arguments arguments)
             (remaining-specializers
               (method-specializers method)))
```

```
    (if (pair? remaining-specializers)
        (if (not (pair? remaining-arguments))
            (error "too␣few␣arguments␣"
                   method
                   arguments)
            (if (or (and (procedure?
                           (car remaining-specializers))
                         ((car remaining-specializers)
                          (car remaining-arguments)))
                    (instance?
                      (car remaining-arguments)
                      (car remaining-specializers)))
                (loop (cdr remaining-arguments)
                      (cdr remaining-specializers))
                #f))
        #t)))
```

We can now define generic functions which allow predicate specialisers:

```
(define-generic-function (factorial n)
  (instance-of <predicate-specializer-generic-function>))

(define-method (factorial (n zero?))
  1)
(define-method (factorial (n <integer>))
  (* n (factorial (- n 1))))
```

Here the base case of the recursion is defined as a separate method, specialising via the predicate `zero?`. The recursive case is defined for all (other) integers.

6.5. Persistence

Many applications need a way to store data between runs. In object-oriented programs, there is already a structure to the data, namely the objects. It would be nice if we could just tell our system to store this structure as it is, without explicitly writing storage and retrieval code by hand, which is error-prone and easily forgotten.

The meta-object protocol can be used to define a new class `<persistent>`, which can be used as the superclass of new classes. This class takes care of automatically persisting data in a database. The programmer need only load a root object (which automatically loads all other referenced objects) upon startup, every modification of slots of persistent classes is automatically persisted to the database.

The following functionality is automatically generated for all subclasses of `<persistent>`:

- Overridden writers that write to the database in addition to memory
- Automatic and transparent handling of unique object ids
- Correct storing and loading of objects referred from a persistent object
- Code to serialise and deserialise instances to a binary format
- Code to read objects from and write them to the database

In this example, we use a key-value database (Kyoto Cabinet [18]). Each object is stored under a unique id (the key) in a serialised form (the value), which contains the object id, the class and all slot values. The serialisation mechanism is independent of the persistence mechanism, it would not be complicated to exchange this for another database (e.g. SQLite). For reasons of simplicity, we assume that a single global database is used for all persistent objects.

6.5.1. Implementation

To implement this, we proceed as follows:

- Create a meta-class `<persistent-class>` which overrides slot access (in order to save to the database) and overrides class initialisation (in order to generate automatic serialisation and deserialisation methods)
- Create a class `<persistent>` (with meta-class `<persistent-class>` which has an object id slot. This is the base class for all persistent objects.

First we define the class `<persistence-class>`:

```
(define-class <persistent-class> (<class>)
  ())
```

Then we override the slot writer generator, in order to first call the actual writer, but then to also persist the object:

```
(define-method
  (ensure-slot-writer (class <persistent-class>)
                      (slot <slot-descriptor>)
                      slots
                      (writer <generic-function>))
  (let* ((old-writer-method (next-method))
         (old-writer-lambda (method-lambda
                             old-writer-method)))
    (let ((writer-method
```

```
          (make-method writer
                       (list class <object>)
                       (lambda (next-method obj value)
                         (old-writer-lambda
                           next-method
                           obj
                           value)
                         (persist-object! obj)))))
      (add-method! writer writer-method)
      writer-method)))
```

Finally, we also create appropriate serialisation methods when a new subclass is initialised (the code for serialisation deals with internals and is not shown here):

```
(define-method (initialize (class <persistent-class>)
                           params)
  (add-persistence-methods class)
  (next-method))
```

We can now define an instance class of `<persistent-class>` as a base class for all persistent objects:

```
(define-class <persistent> (<object>)
  ((object-id (reader object-id) (writer object-id!)))
  (instance-of <persistent-class>))
```

When a new object of class `<persistent>` is initialised, it is given a new object id:

```
(define-method (initialize (obj <persistent>)
                           params)
  (let ((obj (next-method)))
    (object-id! obj (next-object-id))
    obj))
```

The database also defines two methods `database-root` and `database-root!` to store an object under a special key that signifies that this is the root object that references all other objects. We can now use our persistence layer:

```
(define-class <person> (<persistent>)
  ((name
     (reader person-name)
     (writer person-name!))
   (age
     (reader person-age)
     (writer person-age!))
   (friends
```

```
     (reader person-friends)
     (writer person-friends!))))

(define-class <rolodex> (<persistent>)
  ((contacts
     (reader rolodex-contacts)
     (writer rolodex-contacts!))))

(database-root! (make <rolodex>
                      (list (make <person> "Peter" 15)
                            (make <person> "Suzy" 19)
                            (make <person> "Chuck"))))
```

The database now contains a root object (a rolodex), which contains three people. We can read this root object from the database (using `database-root`) which automatically loads all objects that are referenced from the database root, edit all these objects and our overridden slot writer ensures that each modification is instantly stored in the database. The default implementation persists basic Scheme datatypes (such as strings, numbers and lists), however changing a list (not storing a new or modified list in a slot of an object) is not persisted. A custom class re-implementing a linked list would rectify this problem.

The serialisation is independent of word size or byte order, this database can be moved to another machine and will work there just as well.

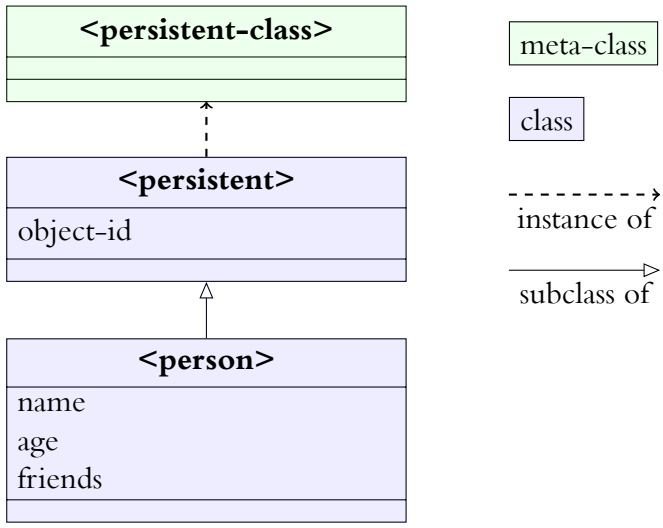

Figure 6.3.: Persistence class hierarchy

Figure 6.3 shows the class relationship defined above. The meta-class `<persistent-class>` is the hinge for all the meta-object protocol methods. The class `<persistent>` is the superclass of all classes that are supposed to be persistent. It adds the slot `object-id`, which

is needed to ensure object identity. One of its subclasses is `<person>`, which adds several slots for use by the programmer. All slots (including the ones defined in `<persistent>`) of `<person>` are persisted to the database.

7. Conclusions

Chapter 4 of this thesis presents an object system and based upon that a meta-object protocol for reflecting upon the structure of object-oriented programs and interceding in the operation in certain restricted ways. The meta-object protocol allows the programmer to challenge certain design decisions and adapt the program to his or her needs more easily than in traditional object systems which allow no adaptations.

7.1. Comparison to Existing Approaches

The meta-object protocol specified in chapter 4 is deemed more efficient than the other existing meta-object protocols by the author based on the measurements shown later in this chapter. This claim is mainly based on the reduced number of (costly) generic function calls in the resulting code. Whereas the CLOS MOP often employs generic functions directly in the call path of often-used functions, the design of the new meta-object protocol for Scheme has sought to avoid this as much as possible, instead relying on functional protocols that are called upon class or generic function creation, moving the calculation effort to compile- or load-time.

In the following sections, a closer comparison to the two main existing approaches – CLOS and TELOS – is made.

7.1.1. Comparison to the CLOS MOP and TELOS

Both the CLOS MOP and TELOS are semantically and syntactically similar to the meta-object protocol specified in this work. Both were developed for languages in the Lisp family, and thus share many traits with each other. The underlying object system is similar, though there are differences in detail (e.g. TELOS consists of several layers, which build upon each other).

CLOS

The Common Lisp Object System (detailed in section 3.3) was the first that received a meta-object protocol (see [15]). This protocol allows the user to intercede in many places, however it is often procedural, i.e. it mandates that generic functions be called in places where this costly call is detrimental to performance.

As an example, the default CLOS instance slot reader protocol specifies three generic functions that must be called nestedly:

```
<reader>
  slot-value
    slot-value-using-class
```

This means that every slot access must call at runtime two or three (depending on the exact implementation) generic functions (each entailing at least a minimal form of multiple dispatch). As the generic function `slot-value-using-class` contains multiple methods, this results in dispatch overhead.

In contrast, the protocol described in section 4.4 specifies that only a single generic function is invoked: the actual reader (even this can be avoided if the reader is made a primitive function, not a generic function).

Similarly instance allocation in CLOS requires a generic function call (of the generic function `allocate-instance`), whereas the protocol described in section 4.4 requires none.

Table 7.1 shows these differences.

Case	CLOS	Proposed MOP
Slot access	3	1
Instance allocation	1	0
Generic function invocation	3	1-3

Table 7.1.: Number of Generic Function calls

TELOS

TELOS has only specified the slot accessor protocol, which is similar to the one described in section 4.4. As TELOS remained unfinished, the other protocols were just taken from CLOS unchanged.

Comparison with the slot access protocol (detailed in section 3.4) shows that the protocol is similar to the one specified in figure 4.6:

- Both are functional protocols that are executed once upon class definition
- Both can be used to compute arbitrary readers and writers, which need not be generic functions

TELOS is an ancestor to the proposed meta-object protocol in spirit, focusing on efficient and powerful protocols.

7.1.2. Other languages

The other languages have less in common with the proposed meta-object protocol, yet have offered unique influences on the design.

Smalltalk and Dylan

Both Smalltalk and Dylan in their standard versions only include a reflective meta-object protocol. These protocols are at the same level of power as the one proposed in chapter 4. All three offer a way to inspect classes, find out about slots, their accessors, sub- and superclass relationships, etc. Generic functions do not exist in Smalltalk, so instead of retrieving the methods of a generic function Smalltalk allows access to a classes defined methods. Dylan is interesting, because it used to have a more powerful meta-object protocol, which was withdrawn in later versions of the language due to concerns about implementation speed. This meta-object protocol was very similar to the one in CLOS.

OpenC++

As figure 3.6 shows, OpenC++ fulfils all the criteria except for a Runtime Component. OpenC++ offers limited customisation (due to its need to deal with the very complex semantics of C++), yet seems to work well in practice. It integrates well with C++ (having a slightly extended syntax, that meshes well with "normal" C++). However, as it excludes all runtime components, it does not directly support any reflexion or modification of intercedence at runtime. It is not possible to modify the meta-objects' behaviour after compile-time.

Ruby

Ruby is the most dynamic of all the investigated languages, offering ways to override all behaviour and intercede at any point. It also has full reflexive capabilities. However, precisely due to this dynamism, it does not support pre-runtime optimisations or code analysis (as the resulting code could potentially be modified at runtime in any way).

7.2. Applicability to Other Languages

In principle, the meta-object system specified in chapter 4 could be modified and adapted for other languages and object systems, however it relies on a few specific concepts:

- First-order functions

  Many of the protocols depend on the ability to create functions and pass them as parameters or return values. Languages which do not support this directly need to find appropriate abstractions.

- Full object-orientation

  The protocols depend on the fact that *everything* is an object. If primitive data types exist which cannot be used in the same way as all other objects, special care needs to be taken to ensure that this doesn't break the implementation (e.g. multiple dispatch).

- Syntactic abstraction

  The lower-level mechanisms for defining classes, generic functions and methods are perfectly usable, but more verbose than the higher-level macros (`define-class`, `define-generic-function` and `define-method`). If a language does not offer a powerful enough macro system, it might be difficult to ensure simple usability of the newly added concepts, though this can be alleviated by providing a pre-processor.

7.3. Prototypical implementation

The prototypical implementation described in chapter 5 is meant to demonstrate the feasibility of the proposed meta-object protocol. It consists of an implementation of the core protocols, and several applications (described in chapter 6).

Application	Object Instantiation	Class Creation	Generic Function Invocation
Hashtable-Storage	•	•	
Tracing Generic Functions			•
Class-allocated Slots		•	
Predicate Dispatch		•	•
Persistence	•	•	

Table 7.2.: Protocols used by the applications

These sample applications were implemented to show the usability and power of the proposed meta-object protocol in practise. Table 7.2 shows which applications use which protocols in their implementation.

Table 7.3 shows in detail the lines-of-code count (generated using David A. Wheeler's *SLOCCount*) for the various implementation artifacts. These counts include several test cases, apart from the Persistence application, all take less than 70 lines of actual code, showing the succinctness of Scheme as a language and the power of the meta-object protocol.

The core implementation consists of general bootstrapping code (241 lines), the implementation of generic functions (140 lines), the implementation of classes (179 lines) and the macro layer (108 lines).

Program	Lines of Code
Core implementation	668
Hashtable-Storage	55
Tracing Generic Functions	53
Class-allocated Slots	97
Predicate Dispatch	112
Persistence	210
Sum	1195

Table 7.3.: Lines of Code Count

The implementation was not optimised for speed in any way (see section 7.4 for ideas). Table 7.4 shows the running time of loops calling generic functions with one, two and three methods registered. It is expected that proper optimisation of generic function invocation would greatly improve these numbers.

Each loop was run 1.000,000 times, doing nothing but calling the generic function. The first entry (Apply-Hook only) shows the time for a loop that a loop which does nothing except call an apply-hook that forwards to the primitive `+` function. This is the fastest possible implementation that MIT GNU/Scheme offers, and is used as the baseline for the other comparisons. The optimised dispatch is a special case of `<generic-function>`, which optimises dispatch for generic functions with only one method. The other three methods use the default (entirely unoptimised) mechanism. No memoization or caching was implemented (which would speed up the dispatch mechanism).

Methods registered	Runtime	Factor
Apply-Hook only	0.077s	1.0
Optimised dispatch	0.376s	4.88
One method	3.344s	43.43
Two methods	5.309s	68.95
Three methods	7.425s	96.43

Table 7.4.: Generic Function Invocation Runtime

7.4. Future Work

The implementation of the prototype has focused on completeness, not on efficiency. Thus optimisation of generic dispatch (especially the multi-method case) would be one

point to focus on ([5], [9]).

The implementation of the applications detailed in chapter 6 has shown that the general design of the meta-object protocols is apt for the varying demands of very different problems, however more and larger applications should be developed to substantiate this claim.

Especially the language Dylan (see 3.7) offers a couple of efficiency-oriented extensions over traditional Lisp object systems, which it would be worthwhile to integrate:

- Sealed generic functions and classes, enabling specific optimisation
- Optimisation due to visibility of generic functions, methods and classes across modules

Concerning the implementation, many of the dynamic-compilation concepts from SELF [25] could be applied to further increase performance. These would however need some support from the actual Scheme implementation. The prototypical implementation of the object system and the meta-object protocol is portable at the moment, customisations involving dynamic compilation can probably not be implemented portably.

A. Function Reference

applicable-methods

Calculates which methods of a generic function are applicable to the arguments.

Full form: `(applicable-methods <generic-function> <arguments>)`

Parameters

- `<generic-function>`: the generic function of which the methods should be checked.
- `<arguments>`: the actual arguments for which the applicable methods should be found.

This generic function should return a list of all methods which are applicable to the arguments, and return it. The order is not important (they will be sorted later by the generic function `sort-applicable-methods`).

Called by: `sort-applicable-methods`

Calls:

apply-class-options

Apply the effects of the given class options to the class instance.

Full form: `(apply-class-options <class> <options>)`

Parameters

- `<class>`: The class to apply the options to.
- `<options>`: The list of options

This generic function should be used to implement custom class options. Its second parameter is a list of options. Each option is a list, where the first element is the symbol naming the option, and the rest of the list is option-dependent.

Called by: `initialize`

Calls:

`apply-generic-function`

Apply the correct methods of the generic function in the correct order to the given arguments.

Full form: `(apply-generic-function <generic-function> <arguments>)`

Parameters

- `<generic-function>`: The generic function to apply.
- `<arguments>`: The actual arguments this generic function is called with.

This is the main function that decides on how to apply a generic function to its arguments. It should calculate a list of sorted applicable methods, then invoke them on the arguments (calling the next method in line if necessary).

Called by:

Calls: `sort-applicable-methods`

`calculate-and-order-superclasses`

Calculate all effective superclasses of the class and sort them correctly.

Full form: `(calculate-and-order-superclasses <class> <superclasses>)`

Parameters

- `<class>`: The class to calculate the superclasses for.
- `<superclasses>`: The list of superclasses that the programmer passed when creating the class.

This function should return a list of classes which are all superclasses of the class passed as the first argument. They should be sorted according to their precedence.

Called by: `initialize`

Calls:

`calculate-slot-storage-allocator`

Calculate the allocator function for slot storage for instances of this class.

Full form: `(calculate-slot-storage-allocator <class>)`

Parameters

- `<class>`: The class for which the slot allocator for instances should be calculated.

This function should return a primitive or generic function that allocates slot storage for an instance of the class passed as the first parameter. The allocator has no parameters, and should return the slot storage.

Called by: `initialize`

Calls:

`ensure-slot-accessors`

Ensure that the slot accessors for one slot are correctly defined.

Full form: `(ensure-slot-accessors <class> <slot> <slots>)`

Parameters

- `<class>`: The class to which the slot belongs.
- `<slot>`: The slot for which the accessors are checked.
- `<slots>`: All slots of the class.

This function should ensure that correct slot accessors are present and defined for the slot passed as the second argument.

Called by: `ensure-slots-accessors`

Calls: `ensure-slot-reader`, `ensure-slot-writer`

`ensure-slot-reader`

Ensure that a correct reader for one slot is defined.

Full form: `(ensure-slot-reader <class> <slot> <slots> <reader>)`

Parameters

- `<class>`: The class to which the slot belongs.
- `<slot>`: The slot which is checked for a reader.
- `<slots>`: All slots of the class.
- `<reader>`: The reader primitive or generic function.

This function should ensure that a correct reader for the slot value is present. This can be a primitive function or a generic function, in which case a correct method must be present.

Called by: `ensure-slot-accessors`

Calls:

`ensure-slot-writer`

Ensure that a correct writer for one slot is defined.

Full form: `(ensure-slot-writer <class> <slot> <slots> <writer>)`

Parameters

- `<class>`: The class to which the slot belongs.
- `<slot>`: The slot which is checked for a writer.
- `<slots>`: All slots of the class.
- `<writer>`: The writer primitive or generic function.

This function should ensure that a correct writer for the slot value is present. This can be a primitive function or a generic function, in which case a correct method must be present.

Called by: `ensure-slot-accessors`

Calls:

`ensure-slots-accessors`

Ensure that correct accessors for all slots are present.

Full form: `(ensure-slots-accessors <class> <slots>)`

Parameters

- `<class>`: The class which is checked.
- `<slots>`: All slots of the class.

This function should ensure that correct accessors are present for all slots.

Called by: `initialize`

Calls: `ensure-slot-accessors`

`initialize`

Initialise a new class instance.

Full form: `(initialize <object> <parameters>)`

Parameters

- `<object>`: The new object to be initialised.
- `<parameters>`: Any parameters passed to `make`.

This function must correctly call all initialisation code to calculate necessary functions and data.

Called by: `make`

Calls: `calculate-and-order-superclasses`, `calculate-slot-storage-allocator`, `apply-class-options`, `ensure-slots-accessors`

`make`

Create a new class instance.

Full form: `(make <class> <parameters>)`

Parameters

- `<class>`: The class of which an instance is created.
- `<parameters>`: Any parameters for initialisation.

This function must allocate storage for the object, the object's slots and perform any initialisation necessary.

Called by:

Calls: `initialize`

`method-more-specific?`

Check whether one method is more specific than another method.

Full form: `(method-more-specific? <method-1> <method-2>)`

Parameters

- `<method-1>`: First method
- `<method-2>`: Second method

This function returns true if the first method is more specific than the second method. The definition of specificity depends on the application, the default definition is that a method is more specific than another if a parameter specialises on a subclass of the other method's specialiser for that parameter (checked from first to last specialiser).

Called by: `method-more-specific?`

Calls: `specializer-more-specific?`

`sort-applicable-methods`

Sort all applicable methods in order of their specificity.

Full form: `(sort-applicable-methods <generic-function> <arguments>)`

Parameters

- `<generic-function>`: The generic function for which the applicable methods should be sorted.

- `<arguments>`: The actual arguments for which the applicable methods should be sorted.

This function should calculate the applicable methods, then sort them in order of their specificity.

Called by: `apply-generic-function`

Calls: `applicable-methods`, `method-more-specific?`

`specializer-more-specific?`

Check whether a specialiser of one method is more specific than a specialiser of another method

Full form: `(specializer-more-specific? <m-1> <m-2> <s-1> <s-2>)`

Parameters

- `<m-1>`: First method
- `<m-2>`: Second method
- `<s-1>`: First specialiser, belonging to the first method
- `<s-2>`: Second specialiser, belonging to the second method.

Return true if the first specialiser is more specific than the second specialiser. The definition of specificity depends on the application, the default definition is that a method is more specific than another if a parameter specialises on a subclass of the other method's specialiser for that parameter (checked from first to last specialiser).

Called by: `method-more-specific?`

Calls:

Bibliography

[1] Wilhelm Ackermann, *Zum hilbertschen aufbau der reellen zahlen*, Mathematische Annalen **99** (1928), 118–133.

[2] Alan Bawden and Jonathan Rees, *Syntactic closures*, Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming, 1988, pp. 86–95.

[3] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel, *Commonloops: merging lisp and object-oriented programming*, Conference proceedings on Object-oriented programming systems, languages and applications (New York, NY, USA), ACM, 1986, pp. 17–29.

[4] Harry Bretthauer, Jürgen Kopp, Harley Davis, and Keith Playford, *Balancing the eulisp metaobject protocol*, Lisp and Symbolic Computation **6** (1993), no. 1-2, 119–138.

[5] Craig Chambers and Weimin Chen, *Efficient multiple and predicated dispatching*, Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM, 1999, pp. 238–255.

[6] Shigeru Chiba, *A metaobject protocol for c++*, Proceedings OOPSLA'95, ACM, 1995, pp. 285–299.

[7] William Clinger, *Hygienic macros through explicit renaming*, SIGPLAN Lisp Pointers **IV** (1991), no. 4, 25–28.

[8] Iain D. Craig, *Programming in dylan*, ch. 7, pp. 196–200, Springer, 1997.

[9] Eric Dujardin, Eric Amiel, and Eric Simon, *Fast algorithms for compressed multimethod dispatch table generation*, ACM Trans. Program. Lang. Syst. **20** (1998), 116–165.

[10] David Flanagan and Yukihiro Matsumoto, *The ruby programming language*, O'Reilly Media, 2008.

[11] Brian Foote and Ralph E. Johnson, *Reflective facilities in smalltalk-80*, Proceedings OOPSLA '89, ACM, ACM Press, 1989, pp. 327–335.

[12] Guillaime Germain, Marc Feeley, and Stefan Monnier, *Concurrency oriented programming in termite scheme*, LaBRI, University of Bordeaux, ACM Press, 2006.

[13] Adele Goldberg and David Robson, *Smalltalk-80: The language and its implementation*, Addison-Wesley, Reading, Massachusetts, 1983.

[14] Paul Graham, *On lisp*, Prentice Hall, 1993.

[15] Jim Des Rivieres Gregor Kiczales, *The art of the metaobject protocol*, MIT Press Cambridge, MA, USA, 1991.

[16] Doug Hoyte, *Let over lambda – 50 years of lisp*, Lightning Source UK Ltd., 2010.

[17] Richard Kelsey, William Clinger, and Jonathan Rees (eds.), *The revised^5 report on the algorithmic language scheme*, ACM SIGPLAN Notices **33** (1998), 26–76.

[18] FAL Labs, *Kyoto cabinet*, Web, July 2011.

[19] Chamond Liu, *Smalltalk, objects, and design*, toExcel, 2000.

[20] Massachusetts Institute of Technology, *Mit/gnu scheme*, Web, July 2011.

[21] Julian Padget, *Programming language eulisp, version 0.99*, 1993.

[22] Christopher Burdorf Peter Broadbery, *Applications of telos*, Lisp and Symbolic Computation **6** (1993), no. 1-2, 139–158.

[23] Jonathan A. Rees and Norman I. Adams IV, *T: A dialect of lisp or, lambda: The ultimate software tool*, Conference Record of the 1982 ACM Symposium on LISP and Functional Programming, ACM, ACM, August 1982, pp. 114–122.

[24] Andrew Shalit, David Moon, and Orca Starbuck, *The dylan reference manual*, Addison Wesley Publishing Company, 1996.

[25] Randall B. Smith and David Ungar, *Self: The power of simplicity*, Tech. report, Sun Microsystems, Inc., 1994.

[26] Guy Steele, *Common lisp. the language. second edition*, 2nd ed., Digital Press, June 1990.

[27] W. Teitelman and L. Masinter, *The interlisp programming environment*, Computer **14** (1981), no. 4, 25–33.

Glossary

Class is an abstract type that describes the behaviour and structure of its instances.. 18

Direct Instance An instance of a class X which is not an instance of any subclass of X.. 57

Functional Protocol is a protocol where a generic function is not invoked for its effect but for its result. The generic function may be invoked multiple times, but it may also only be invoked once, and the result cached.. 27

Generic Function is an abstraction of a function, which can contain multiple methods applicable to different types of parameters. At runtime, the correct method is chosen from all available methods of a generic function.. 19

Hygienic Macro System is a macro system where it is not possible to introduce new identifiers that inadvertently capture the user's identifiers.. 12

Meta-Object is an object that describes how other objects are created and used.. 17

Meta-Object Protocol is a combination of generic functions and meta-classes that allows the user to interfere with the operation of object-oriented programs in well-defined ways.. 21

Meta-Recursion The problem that generic functions are implemented in terms of generic functions, thus leading to a potentially infinite recursion.. 39

Method is a concrete implementation of a generic function for a fixed set of argument types.. 19

Multiple Dispatch describes a function dispatch system where the method that is applied does not depend only on a single parameter (single-dispatch, as in Java or C++), but depends on all parameters' type.. 19

Non-Hygienic Macro System is a macro system where it is possible to introduce new identifiers that inadvertently capture the user's identifiers.. 12

Procedural Protocol is a protocol where a generic function must always be invoked for its effect. A result must not be cached.. 27

Protocol describes conventions about how classes and generic functions are to be used, overloaded and overridden to achieve specific effects.. 18

Slot is a field in an object that can be read from or written to.. 18

Special Form is a form that does not follow the usual rules for evaluation of parameters. Examples are `if` and `lambda`.. 5

Syntactic Extension is a source code transformation which is not based purely on replacing text, but which interacts correctly with and honours the the defined syntax of the language.. 12

Index

Peter Feigl

DI Peter Feigl
Lindengasse 5
4040 Linz
Austria

Cell: +43-650-2131517
E-Mail: peter.feigl@gmx.at

Born: January 3rd, 1980 in Vienna
Nationality: Austria

Current position

FREELANCE SOFTWARE-DEVELOPER
Working on incorporating a Start-Up

Areas of specialization

Programming languages • GNU/Linux • Server systems

Education

current — DOCTOR of Technical Sciences
JKU Linz
"A Meta-Object Protocol for Scheme"
Supervisor: Dr. Blaschek

2000-2006 — DIPLOM-INGENIEUR of Informatics
JKU Linz
"An Environment for the Programming Language Scheme"
Supervisor: Dr. Mössenböck

1990-1998 — HIGH SCHOOL
Bundesgymnasium Bad Ischl

Skills

Programming	C++, C#, C, Java, Scheme, Erlang, Haskell
Databases	PostgreSQL, MySQL, SQLite
Web	HTML, XML, CSS, JSON, JavaScript, Ruby
Version Control	Subversion, Git, CVS
Administration	Apache, Postfix, Tomcat, Dovecot, Dolibarr CRM
Organisational	Project management, Management of educational seminars
Social	Communicative, Work in a team, Accept criticism

Languages

Fluent	English, Spanish Numerous travels abroad, university exchange programs
Proficient	French, Japanese, Hungarian, Russian Travels abroad, university exchange programs
Intermediate	Czech, Danish, Portuguese, Italian, Irish Gaelic

Appointments held

2010-2011	Research Assistant – JKU Linz Optimisation of Routing Problems
2005-2011	Freelance IT Consultant Design and implementation of Web solutions
2006-2010	Teaching Assistant – JKU Linz Algorithms and Data Structures, Software Development, various seminars
2009-2010	Research Assistant – JKU Linz Optimisation of Scheduling Problems
2008-2009	Research Assistant – JKU Linz Parallel Optimisation
2006-2008	Research Assistant – JKU Linz Compiler design and implementation
2006	Software Developer – WebDynamite GmbH Web application development
2001-2002	Technician – JKU Linz Server administration
2000	Technician – Infox GmbH Server administration
2000	Technician – omnipräsent, Wimmer GmbH Server administration

Experiences abroad

2004-2005	Erasmus Student Exchange University of Edinburgh, Scotland
2003	CEEPUS Student Exchange Eötvös Loránd Tudományegyetem (ELTE) in Budapest, Hungary
2000-2011	Numerous Travels India, Poland, Portugal, France, Ireland...
1998-1999	Student Exchange 若柳高校 宮城県 日本 Wakayanagi Koukou, Miyagi-ken, Japan

Interests

Interests	Reading, Hiking, Typography, Languages
Hobbies	Boardgames, Computers, Guitar
Projects	GNU/MIT-Scheme, Lightfeather 3D Engine, GNU Emacs, LaTeX