Multiple Dispatch in Practice

by

Radu Muschevici

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2009

Abstract

Multiple dispatch uses the run time types of *more than one* argument to a method call to determine which method body to run. While several languages over the last 20 years have provided multiple dispatch, most object-oriented languages still support only single dispatch — forcing programmers to implement multiple dispatch manually when required. This thesis presents an empirical study of the use of multiple dispatch in practice, considering six languages that support multiple dispatch. We hope that this study will help programmers understand the uses and abuses of multiple dispatch; virtual machine implementors optimise multiple dispatch; and language designers to evaluate the choice of providing multiple dispatch in new programming languages.

Acknowledgments

First and foremost I would like to thank James Noble and Alex Potanin. I very much enjoyed having you both as supervisors and it's hard for me to imagine this relationship could have been any better than it was. Also thanks for COMP462, the most inspiring course I ever attended.

Many thanks to Ewan Tempero for his advice, guidance and dedication during our collaboration on our OOPSLA paper.

Thanks to Craig Chambers, Todd Millstein, Bruce Hoult, Daniel Bonniot, Jeffrey Hightower, Gary T. Leavens and the many friendly people on the #dylan and #lisp IRC channels. This research was possible not until they shared their knowledge with me.

Thanks to Craig, my office mate for answering all my questions, organising a whole bunch of things for me and coping with my stressed self as the submission of this thesis grew nearer.

Thanks to the School of Mathematics, Statistics and Computer Science and the Faculty of Science for funding my travel to OOPSLA.

Thanks to my family: my Dad for encouraging me to venture on this undertaking and for making it possible; my Mom for supporting my decision; my sister for keeping in touch. Thanks to Heidi for being my friend and soul mate. Thanks to Matthias and Barbara for their friendship and the support they gave me for my move to New Zealand.

Contents

1 Introduction **1**

2 Background **3**

2.1 Multiple Dispatch 3

2.1.1 Classes and Multiple Dispatch 4

2.1.2 Simulating Multiple Dispatch 5

2.2 Multiple Dispatch Research 7

2.2.1 Multiple Dispatch Languages 8

2.2.2 Multiple Dispatch Extensions 10

2.2.3 Multiple Dispatch Efficiency 11

2.2.4 Empirical Multiple Dispatch Studies 11

3 Model **13**

3.1 Modelling Dynamic Dispatch 13

3.2 Modelling Programming Languages 15

3.3 Metrics 18

3.3.1 Dispatch Ratio (DR) 19

3.3.2 Choice Ratio (CR) 20

3.3.3 Degree of Specialisation 21

Degree of Specialisation of a Concrete Method (DOS) 21

Degree of Specialisation of a Generic Function (DOS_G) . . . 21

3.3.4 Rightmost Specialiser 21

Rightmost Specialiser of a Concrete Method (RS) 21

Rightmost Specialiser of a Generic Function (RS_G) 22

3.3.5 Degree of Dispatch (DOD) 22

3.3.6 Rightmost Dispatch (RD) 23

3.4 Example 23

4 Methodology **27**

4.1 Corpus Analysis 27

4.2 Corpus 28

4.2.1 CMUCL (CLOS) 29
4.2.2 SBCL (CLOS) 30
4.2.3 McCLIM (CLOS) 30
4.2.4 Open Dylan (Dylan) 31
4.2.5 Gwydion (Dylan) 31
4.2.6 Vortex (Cecil) and Whirlwind (Diesel) 32
4.2.7 The Nice Compiler (Nice) 32
4.2.8 The Location Stack (MultiJava) 32
4.3 Analysis 33
4.3.1 Intermediate Format 34
4.3.2 Multiple Dispatch Corpus Analysis Tool (MuDiCAT) 35
Requirements 35
Design 36
Implementation 37
4.4 Language-specific Analysis Front-ends 37
4.4.1 Common Lisp Compilers – CMUCL and SBCL 37
4.4.2 Gwydion Dylan 39
4.4.3 Open Dylan 41
4.4.4 The Vortex Compiler Infrastructure for Cecil and Diesel . . 42
4.4.5 The Nice Compiler 43
4.4.6 The MultiJava Compiler 45
4.5 Summary 47

5 Results **49**
5.1 Ratios 49
5.1.1 Dispatch Ratio 51
5.1.2 Choice Ratio 51
5.2 Specialisation 52
5.2.1 Specialisation of Concrete Methods 52
5.2.2 Specialisation of Generic Functions 54
5.3 Dispatch 54
5.3.1 Degree of Dispatch 59
5.3.2 Dispatch Ratio and Degree of Dispatch 63
5.3.3 Specialisation and Dispatch 65
5.4 Power Laws 68

6 Conclusions **71**
6.1 Contributions 71
6.1.1 A Language-independent Model for Multiple Dispatch . . . 71

6.1.2 A Metrics Suite for Measuring Multiple Dispatch 72
6.1.3 An Evaluation of the Use of Multiple Dispatch in Practice . 72
6.2 Discussion 72
6.2.1 Monomorphic vs Polymorphic Functions 72
6.2.2 Style 73
Use of Specialisers 73
Specialisation Patterns 74
Dispatch 74
6.2.3 A Critical Perspective 75
6.3 Evidence Based Language Design 75
6.4 Future Work 76
6.4.1 Method Calls and Dynamic Aspects of Multimethods 76
6.4.2 Object-Orientation and Beyond 76
6.4.3 Java Method Overloading 77

A Results **79**

List of Figures

3.1 A Model for multiple dispatch . . . 14
3.2 Multimethods across languages . . . 17
3.3 Metrics . . . 19

4.1 Applications in multiple dispatch corpus. . . . 30
4.2 Relative size of applications in corpus . . . 31
4.3 Intermediate Format Grammar . . . 34
4.4 Class diagram for the Multiple Dispatch Corpus Analysis Tool . . . 36

5.1 Metrics: averages across applications . . . 50
5.2 DR distribution . . . 52
5.3 CR distribution . . . 53
5.4 DoS and RS distributions (stacked bars) . . . 55
5.5 DoS and RS distributions compared . . . 56
5.6 DoS vs. RS scatter plot . . . 57
5.7 DoS_G and RS_G distributions (stacked bars) . . . 58
5.8 DoD and RD distributions (stacked bars) . . . 60
5.9 DoD and RD distributions compared . . . 61
5.10 DoD vs. RD scatter plots . . . 62
5.11 DoD and DR compared . . . 64
5.12 DoD vs. DR scatter plots . . . 66
5.13 DoS_G and DoD distributions compared . . . 67
5.14 DR distribution (corpus), log-log scale . . . 68
5.15 DR distributions, log-log scale . . . 70

A.1 Metrics: Distributions (numbers) . . . 80

Chapter 1

Introduction

> *What is the difference between a Turing machine and the modern computer? It's the same as that between Hillary's ascent of Everest and the establishment of a Hilton hotel on its peak.*
>
> Alan Perlis, 1982.

High level programming languages provide programmers with powerful abstractions, making the process of developing a program simpler and more understandable. While the benefits of programming in a high level language are generally acknowledged, there is little empirical evidence of how much programmers actually use particular high level language features. We believe there are clear advantages to informing the design of future languages about the use of those features in the real world. Similarly, programming, maintenance and debugging practice, and even teaching about programming paradigms, would surely benefit from being based on evidence about how programs are written in practice.

Multiple dispatch is a programming language feature that several object-oriented languages in the last 20 years have provided. Still, most mainstream languages used today lack support for multiple dispatch. Our research goal is to collect evidence about the use of multiple dispatch in practice. We believe this evidence will be useful for programmers to understand the benefits of multiple dispatch; virtual machine implementors to optimise multiple dispatch; and language designers to evaluate the choice of providing multiple dispatch in new programming languages.

All object-oriented languages provide single dispatch: when a method is called on an object, the actual method executed is chosen based on the dynamic type of the first argument to the method (the method receiver, generally self, or **this**). Some object-oriented languages provide multiple dispatch, where methods can be chosen based on the dynamic types of more than one argument.

The goal of this thesis is to understand how programmers write programs that use multiple dispatch when it is available. We ask the question: ***how much is multiple dispatch used?*** — what proportion of method declarations dispatch on more than one argument.

To that end, we describe a corpus analysis of programs written in six languages that provide multiple dispatch (CLOS, Dylan, Cecil, Diesel, Nice and MultiJava). While there are a range of other multiple dispatch languages (e.g. Slate (Salzman and Aldrich, 2005), Groovy (Subramaniam, 2008), Clojure (Hickey, 2008)), we focus on these six languages here because we were able to obtain a corpus for each of these languages.

Contributions

In this thesis, we make the following contributions:

- We design a language independent model of multiple dispatch.
- We develop a suite of language independent metrics, measuring the use of multiple dispatch;
- We conduct a corpus analysis study using those metrics on a collection of programs in six multiple dispatch languages.

Outline

This thesis is organised as follows:

- **Chapter 2** presents the brief history and an overview of multiple dispatch including related work.
- **Chapter 3** describes a language-independent model of multiple dispatch, and defines a set of six metrics in terms of that model.
- **Chapter 4** presents the corpus we analyse and describes the methodology we apply to measure multiple dispatch across this corpus.
- **Chapter 5** presents the results of our corpus analysis study in multiple dispatch languages.
- **Chapter 6** puts our results in perspective, shows directions for future work and concludes.

Chapter 2

Background

In this chapter we introduce the concept of multiple dispatch and contrast it to techniques used to simulate multiple dispatch in languages that support only single dispatch. We also present an overview of the research surrounding multiple dispatch by surveying programming languages that include multiple dispatch, efforts targeted at optimising multiple dispatch, and studies related to the use of multiple dispatch.

2.1 Multiple Dispatch

Multiple dispatch uses the run time types of all arguments in a method call to determine which method body to run. In contrast, in single dispatch languages, such as SIMULA, Smalltalk, C++, Java, and C♯, only the first argument of a method call can participate in dynamic method lookup. In Java, for example, the first argument of a method call is called the *receiver* object, is written "before the dot" in a method call (receiver.method(arguments)), and is called "**this**" inside a method body. The class of this first argument designates the method body to be executed. We will refer to a method body as being *specialised* on the class where it is defined, and to the class of that first formal parameter as the parameter's *specialiser*.

In Java, as in most single dispatch languages, a method's specialiser is implicitly defined by the class enclosing the method definition, for example:

```
class Car extends Vehicle {
  void drive () { print("Driving a car"); }
  void collide (Vehicle v) { print("Car crash"); }
}
```

In single dispatch languages, every dynamically dispatched method is specialised on precisely one class so it is easy to think of methods as operations on classes. Of course, some languages may also have non-dispatched methods (such

as Java static methods) that are not dynamically dispatched at all. Following C++, Java and C♯ also support method overloading, where methods may be declared with different formal parameter types, but only the receiver (the distinguished first argument) is dynamically dispatched. Given this definition of the Vehicle class:

```
abstract class Vehicle {
  void drive () { print("Brmmm!"); }
  void collide (Vehicle v) { print("Unspecified vehicle collision"); }
}
```

the following code will involve the Car class's collide(Vehicle) method shown above, and print "Car crash".

```
  Vehicle car = new Car();
  Vehicle bike = new Bike();
  car.collide(bike);
```

The method defined in Car is called instead of the method defined in Vehicle, because of the dynamic dispatch on the first argument — the receiver — of the message.

Now, in a single dispatch language, the method that prints "Car crash" will *still* be invoked even if the Car class overloaded the collide method with a different argument:

```
class Car extends Vehicle {
  // ... as above
  void collide (Bike b) { print("Car hits bike"); }
}
```

but in a multiple dispatch language, the "Car hits bike" message would be printed.

Getting to the Car.collide(Bike) method from a call of Vehicle.collide(Vehicle) requires *two* dynamic choices: on the type of the first "**this**" argument and on the type of the second (Vehicle or Bike) argument — this is why these semantics are called multiple dispatch. A method that uses multiple dispatch is often called a *multimethod*.

2.1.1 Classes and Multiple Dispatch

Methods in single dispatch languages are usually defined in classes, and the receiver.method(arguments) syntax for method calls supports the idea that methods are called on objects (or that "messages are sent to objects" as Smalltalk would put it). This does not apply to multiple dispatch languages, however, where a concrete

method body can be specialised on a combination of classes, and so methods are not necessarily associated with a single class. Some multiple dispatch languages declare methods separately, outside the class hierarchy, while others consider them part of none, one or several classes, depending on the number of specialised parameters. Since method bodies no longer have a one-to-one association with classes, all parameter specialisers have to be stated explicitly in method body definitions, as this example in the *Nice* programming language (Bonniot et al., 2008) shows:

```
abstract Class Vehicle;
class Car extends Vehicle {}
class Bike extends Vehicle {}

void drive (Car c) {
  /* a method specialised on the class Car */
  print("Driving a car");
}

void collide (Car c, Bike b) {
  /* a method specialised on two classes */
  print("Car hits bike");
}
```

Similarly, while Java method call syntax follows Smalltalk by highlighting the receiver object and placing it before the method name: myCar.drive(), multiple dispatch languages generally adopt a more symmetrical syntax for calls to functions: collide(myCar, yourBike); or drive(myCar);, often while also supporting Java-style receiver syntax.

2.1.2 Simulating Multiple Dispatch

Multiple dispatch is more powerful and flexible than single dispatch. Any single dispatch idiom can be used in a multiple dispatch language — multiple dispatch semantics are a superset of single dispatch semantics. On the other hand, implementing multiple dispatch idioms will require specialised hand-coding in a single dispatch language.

Binary methods (Bruce et al., 1995), for example, operate on two objects of related types. The Vehicle.collide(Vehicle) method above is one example of a binary method. Object equality: Object.equals(Object), object comparisons, and arithmetic operations are other common examples.

In a single dispatch language, overriding a binary method in a subclass is

not considered safe because it violates the contravariant type checking rule for functions (Cardelli, 1988). In the example above it is unsafe to override the collide(Vehicle) method in the Vehicle class with collide(Bike) in the Car class. Because a single dispatch language only considers the runtime type of the first (receiver) argument in a method call, nothing would prevent dispatching a call to Car.collide(Bike) when the second argument supplied in the method call is in fact not of type Bike (but of a supertype, such as Vehicle). This would cause a type error at runtime. Note that Java avoids such runtime errors by statically overloading the Vehicle.collide(Vehicle) method; this can cause the incorrect program behaviour shown in Section 2.1.

To avoid violating the contravariant type checking rule for functions, single dispatch languages like Smalltalk generally use the Double Dispatch pattern to implement binary methods, encoding multiple dispatch into a series of single dispatches (Ingalls, 1986). Double Dispatch is also at the core of the Visitor pattern (Gamma et al., 1994) that decouples operations from data structures.

For example, we could rewrite the collision example to use the Double Dispatch pattern in Java as follows:

```
class Car {
  void collide(Vehicle v) { v.collideWithCar(this); }

  void collideWithCar(Car c) { print("Car hits car"); }
  void collideWithBike(Bike b) { print("Bike hits car"); }
}

class Bike {
  void collide(Vehicle v) { v.collideWithBike(this); }

  void collideWithCar(Car c) { print("Car hits bike"); }
  void collideWithBike(Bike b) { print("Bike hits bike"); }
}
```

Calling a collide method provides the first dispatch, while the second call to a collideWithXXX method provides the second dispatch. The arguments are swapped around so that each argument gets a chance to go first and be dispatched upon. External clients of these classes should only call the collide method, while actual implementations must be placed in the collideWithXXX methods.

The double dispatch idiom is common in languages like Smalltalk where single dispatch is the preferred control structure. Java's **instanceof** type test provides an alternative technique for implementing multiple dispatch. The idiom here is a cascade of **if** statements, each testing an argument's class, and the body of each **if**

corresponding to a multimethod body. To return to the Car and Bike classes:

```
class Car {
  void collide(Vehicle v) {
    if (v instanceof Car) { print("Car hits car"); return; }
    if (v instanceof Bike) { print("Car hits bike"); return; }
    throw Error("missing case: should not happen");
  }
}

class Bike {
  void collide(Vehicle v) {
    if (v instanceof Car) {print("Bike hits car"); return; }
    if (v instanceof Bike) {print("Bike hits bike"); return; }
    throw Error("missing case: should not happen");
  }
}
```

Compared with directly declaring multimethods, either idiom for double dispatching code is tedious to write and error-prone. Code to dispatch on three or more arguments is particularly unwieldy. Modularity is compromised, since all participating classes have to be modified upon introducing a new class, either by writing new dispatching methods or new cascaded **if** branches. The cascaded **if** idiom has the advantage that it doesn't pollute interfaces with dispatching methods, but the methods with the cascades become increasingly complex, and it is particularly easy to overlook missing cases.

2.2 Multiple Dispatch Research

Despite its advantages over single dispatch (§ 2.1), multiple dispatch is not present in current mainstream object-oriented languages such as Smalltalk, C++, C♯ and Java.

The object concept supported by the languages mentioned above views methods as operations on particular (classes of) objects. Such an operation always depends on the type of one single object: it is a property of that type, and it can be encapsulated inside the object. The single dispatch idiom, where functions dispatch on a distinct receiver argument, consequently models this approach to object-orientation. In contrast, multiple dispatch allows operations to depend on multiple different types of objects, thus defying object-based encapsulation. This partly explains why traditional object-oriented languages do not support multiple dispatch.

Other reasons might be related to the early state of multiple dispatch research at the time when the above languages were designed. For example, the issue of independent static type checking of separate code modules has been tackled only during the late 1990s. Multiple dispatch is less time-efficient than single dispatch, due to the more complex lookup mechanism which involves evaluating the types of several arguments instead of just a single one. Therefore, the additional run time cost of multiple dispatch is acceptable only when multimethods are actually invoked (following the principle "you don't pay for what you don't use"), and that expense should not exceed the cost of hand-coded double dispatch (simulated multiple dispatch). Finally, the space efficiency of virtual dispatch tables has only been the subject of more recent research (§ 2.2.3).

Bjarne Stroustrup, the designer of C++, offers a first-hand account of the difficulties encountered when evaluating multiple dispatch for C++ (Stroustrup, 1994, Section 13.8). The two main obstacles which, he regrets, kept him from including multimethods in C++ were that, for one, he couldn't come up with a "calling mechanism that was simple and efficient as the table lookup used for virtual functions [, the C++ equivalent of single dispatch methods]." and second, difficulties in resolving method ambiguity at compile time. Further he offers the following thought, which we cite here as it reflects the driving force behind our research, almost 30 years later: "Multi-methods is one of the interesting what-ifs of C++. Could I have designed and implemented them well enough at that time? Would their application have been important enough to warrant the effort?"

2.2.1 Multiple Dispatch Languages

Multiple dispatch was pioneered by CommonLoops (Bobrow, 1983; Bobrow et al., 1986) and the Common Lisp Object system (CLOS) (Bobrow et al., 1988), both aimed at extending Lisp with an object-oriented programming interface. The extensions were meant to integrate "smoothly and tightly with the procedure-oriented design of Lisp" (Bobrow et al., 1986) and facilitate the incremental transition of code from the procedural to the object-oriented programming style.

The basic idea is that a CLOS *generic function* is made up of one or more methods. A CLOS method can have *specialisers* on its formal parameters, describing types (or individual objects) it can accept. At run time, CLOS will dispatch a generic function call on any or all of its arguments to choose the method(s) to invoke — the particular methods chosen generally depend on a complex resolution algorithm to handle any ambiguities.

Several more recent programming languages aim to provide multimethods in more object-oriented settings. Dylan (Feinberg, 1997) is based on CLOS. Dy-

lan's dispatch design differs from CLOS in that it features optional static type declarations which can be used to type generic functions, that is, to constrain their parameters to something more specific than <object>, the root of all classes in Dylan. Dylan also omits much of the CLOS's configurability, treating all arguments identically when determining if a generic function call is ambiguous.

Cecil (Chambers, 1992; Chambers and Leavens, 1995) is a prototype-based programming language that was the first to implement a modularly checked static type system for multimethods. Cecil treats each method as encapsulated within every class upon which it dispatches. This way a method is given privileged access to all objects of which it is a part. This is different from, for example Java, where methods are part of precisely one class and also unlike CLOS or Dylan in which methods are not part of any class.

Diesel (Chambers, 2006) is a descendant of Cecil and shares many of its multiple dispatch concepts. The main differences to Cecil are Diesel's module system (unlike Cecil, Diesel method bodies are separate from the class hierarchy and encapsulated in modules) and explicit generic function definitions (which bring it closer to CLOS). As in Dylan and Cecil, message passing is the only way to access an object's state. Diesel uses a modular type system initially designed by Millstein and Chambers (1999) for the Dubious language.

The Nice programming language (Bonniot et al., 2008) strives to offer an alternative to Java, enhancing it with multimethods and open classes. In Nice, operations and state can be encapsulated inside modules, as opposed to classes. Message dispatching is based on the first argument and optionally on any other arguments.

MultiJava (Clifton et al., 2000) extends Java with multimethods and open classes. MultiJava retains the concept of a privileged *receiver* object to associate methods with a single class for encapsulation purposes, however, the runtime selection of a method body is no longer based on the receiver's type alone. Rather, any parameter in addition to the receiver can be specialised by specifying a true subtype of the corresponding static type or a constant value. The MultiJava compiler mjc, translates MultiJava source code into standard Java bytecode. For methods that specialise additional parameters, it introduces cascaded sequences of **instanceof** tests (or equality comparisons, for value dispatch).

There are of course many other multiple dispatch languages. Kea (Mugridge et al., 1991) was the first statically typed language with multiple dispatch. Slate (Salzman and Aldrich, 2005) integrates Self-like prototype-based programming with multiple dispatch to propose a new object model. Some more recent programming languages are designed with multiple dispatch already on-board, among them Perl 6 (Randal et al., 2004, Section 8.6.2), Clojure (Hickey, 2008) and Groovy (Subra-

maniam, 2008, Section 4.7). Scala (Odersky et al., 2004) supports a form of pattern matching that can be used to dispatch on arbitrary predicates.

2.2.2 Multiple Dispatch Extensions

Several popular single dispatch languages (Perl (Conway, 2000), Python (Mertz, 2003), Ruby (Cyll, 2005), C++ (Pirkelbauer et al., 2007)) have been extended to support multiple dispatch, often by means of libraries. Smalltalk has been extended with multiple dispatch (Foote et al., 2005) using its reflective facility. Fickle (Drossopoulou et al., 2001), a statically typed, class-based object-oriented language with support for object reclassification has been extended with multiple dispatch and first class relationships (Sinha, 2005).

Java has been augmented with multiple-dispatch and similar facilities using several different approaches. Parasitic Multimethods (Boyland and Castagna, 1997) is an earlier extension to Java that provides multiple dispatch. Methods which are defined using the parasitic keyword override less specific methods. A modified compiler translates code that uses the extended semantics into standard bytecode by introducing type testing statements (**instanceof**) to determine the runtime types of all arguments in a method call, thereby dispatching to the most specific parasite.

The Walkabout (Palsberg and Jay, 1998) uses the reflection interface of Java 1.1 and later to simplify the implementation of the Visitor pattern. Walkabouts greatly improve extensibility by eliminating the need for visitable classes to implement a visit() method and allowing the addition of visitable classes without modification of existing visitors. As the authors note, however, the use of reflection to invoke the appropriate visit method makes this approach impractically slow.

The Runabout (Grothoff, 2003) improves upon the Walkabout approach in terms of performance: Where the Walkabout uses reflection to invoke visit methods, the Runabout dynamically generates (and caches) bytecode that will invoke the appropriate visitor. This makes the dispatch performance of the Runabout comparable to that of the classic visitor pattern and typically exceeds that of **instanceof** tests.

Dutchyn et al. (2001) modified the Java virtual machine to treat static overloading as dynamic dispatch in classes that implement the provided MultiDispatchable marker interface.

Millstein et al. (2003) have evolved MultiJava into Relaxed Multijava (RMJ) which essentially allows the programmer to write code in a more flexible style, without sacrificing static type checking.

Predicate dispatching generalises multiple dispatch to include field values and

pattern matching (Chambers and Chen, 1999), while aspect-oriented programming (Kiczales et al., 1997, 2001) is based around pointcuts that can dispatch on almost any combination of events and properties in a program's execution.

2.2.3 Multiple Dispatch Efficiency

Most language implementations summarised in the previous section include efficiency evaluations of the respective implementation. Additionally, space and time efficiency of method dispatch has been the subject of a large body of research (Chambers and Chen, 1999; Driesen et al., 1995; Naik and Kumar, 2000; Kidd, 2001; Zibin and Gil, 2002). Cunei and Vitek (2005) include a recent comparison of the efficiency of a range of multiple dispatch implementations such as the Visitor pattern, the Runabout and MultiJava.

2.2.4 Empirical Multiple Dispatch Studies

Studies investigating the practical use of multiple dispatch are less widespread than multiple dispatch implementations — Kempf, Harris, D'Souza, and Snyder's early 1987 study of the CommonLoops language (a CLOS predecessor) is one notable exception. One of that study's goals is to assess how useful generic functions and multimethods are for developers, by measuring how often these constructs are used in the implementations of CommonLoops itself and a window library called BeatriX.

Finally, this thesis is part of a larger project that compares the use of multiple dispatch to the use of techniques that simulate multiple dispatch (§ 2.1.2) in Java. In their project, Muschevici, Potanin, Tempero, and Noble (2008) ask a complementary question: ***how much could multiple dispatch be used?*** — that is, what proportion of methods that simulate multiple dispatch could be refactored to use multiple dispatch if it was provided by the language.

Chapter 3

Model

In this chapter we introduce a language-independent model for dynamic dispatch. We then describe six multiple dispatch languages in terms of that model. Finally, we use the model to define metrics for multiple dispatch. The model and metrics we designed allow us to reason about the application of the multiple dispatch paradigm in a language-independent way.

3.1 Modelling Dynamic Dispatch

The model shown in Figure 3.1, is designed to allow us to compare multiple dispatch consistently across different programming languages. The model's terminology has been chosen to match general usage, rather than following any particular programming language. Section 3.3 will use the model to define the metrics that can be used across a range of programming languages. We now present the main entities of the model in turn.

Generic function A generic function is a function that may be dynamically dispatched, such as a CLOS generic function, a Smalltalk message, a MultiJava method family, or Java method call. Each generic function will have one or more *concrete methods* associated with it: calling a generic function will invoke one (or more) of the concrete methods that belong to that function. Generic functions are identified by a *name* and a *signature*. Some languages allow a generic function to be defined explicitly (e.g. CLOS's `defgeneric`, Dylan's `define generic` and Diesel's `fun`), whereas in other languages (such as MultiJava or Cecil) they are implicit and must be inferred from method definitions.

Some languages also automatically generate generic functions as accessors to all field declarations. Because we wish to focus on programmer specified multiple

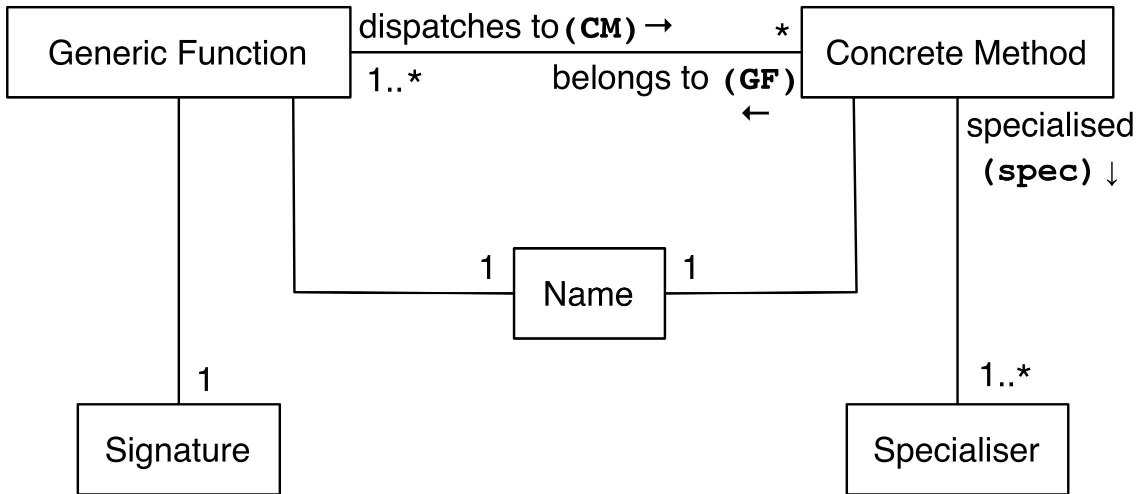

Figure 3.1: A Model for multiple dispatch. GF refers to generic function, CM refers to concrete method, and **spec** refers to specialiser.

dispatch methods, we omit automatically generated accessors from our analysis.

Name Generic functions and concrete methods are referred to by their names. In our model, a name is always "fully-qualified", that is, if a namespace is involved then that information is part of the name. To avoid ambiguity, our analyses always compute fully-qualified names where necessary.

Signature The permissible arguments to a generic function are defined by that function's signature, and all the concrete methods belonging to a generic function must be compatible with that signature. In languages with only dynamic typing, a generic functions signature may be simply the number of arguments required by the function: some language's signatures additionally support refinements such as variable length argument lists or keyword arguments. In languages with (optional or mandatory) static type systems, a generic function's signature will also define static types for each formal argument of the function.

Some languages have implicit parameters (such as the "receiver" or "**this**" parameter in traditional object-oriented languages such as SIMULA, Smalltalk, Java, C++, C#). In our model, these parameters are made explicit in the signature (hence our use of the term "function"). In the case of traditional object-oriented languages, the receiver is the first formal parameter position.

Concrete method A concrete method gives one code body for a generic function — roughly corresponding to a function in Pascal or C, a method in Java or Smalltalk, or CLOS method. As well as this code, a concrete method will have a *name* and

an argument list — the argument list must be compatible with the signature of its generic function (as always depending on the rules of a particular language). A concrete method may also have a *specialiser* for each formal argument position. The rules of each language determine the generic function(s) to which a given concrete method belongs.

Specialiser Formal parameters of a concrete method can have specialisers. Specialisers are used to select which concrete method to run when a generic function is called. When a generic function is called, the actual arguments to the call are inspected, and only those concrete methods whose formal specialisers match those arguments can be invoked in response to the call. Specialisers can describe types, singleton objects, or sets of objects and types (details depend on the language in question).

Some concrete method parameters may have no specialiser (they are *unspecialised*) — the method is applicable for any argument values supplied to those parameters. In contrast, in a class-based object-oriented language, every instance method will belong to a class, and its distinguished first "receiver" argument will be specialised to that class. For example, this is true for every non-static, non-constructor method in Java; Java statics and constructors are not specialised.

Dynamic specialisers are closely related to generic function signatures in statically typed languages: whenever a generic function is called, its actual arguments must conform to the types described by its signature. Depending on the language, specialisers may or may not be tied into a static type system.

Dispatch When a generic function is called at run time, it must select the concrete method(s) to run. In our model, this is a *dynamic dispatch* from the generic function to its concrete methods. If this dispatch is based on the type of one argument, we call it *single dispatch*; if on the type of more than one argument, *multiple dispatch*. If a generic function has only a single concrete method, then no dynamic dispatch is required: we say the function is *monomorphic* or statically dispatched.

3.2 Modelling Programming Languages

To ground our study, we now describe how the features of each of the languages we analyse are captured by the model. The crucial differences between the languages can be seen as whether they offer static typing, dynamic typing, or optional (static) typing; the number of generic functions per method name; and whether a concrete method can be in more than one generic function. These details are summarised

in Figure 3.2, which also gives an overview of terminology used by each language, with Java and Smalltalk for comparisons.

CLOS CLOS (Bobrow et al., 1988) fits quite directly into our model. CLOS *generic functions* are declared explicitly, and then (concrete) *methods* are declared separately; both generic functions and methods lie outside classes. Each generic function is identified by its name (within a namespace), so all methods of the same name belong to the same generic function. CLOS requires "lambda list congruence": all methods must agree on the number of required and optional parameters, and the presence and names of keyword parameters (Lamkins, 2004).

Dylan Dylan's dispatch design (Feinberg, 1997) is similar to CLOS in most respects, including concrete *methods* being combined via explicit *generic function* definitions, and similar parameter list congruency conditions. Dylan supports optional static type checking, and specialisers and static type declarations are expressed using the same syntax. When defining a concrete method, the type declarations serve as dynamic specialisers if they are more specific than the types declared by the generic function.

Cecil Cecil (Chambers, 1992) generic functions (*multimethods*) are declared implicitly, based on concrete method definitions, and each concrete method is contained within one generic function. Unlike CLOS, a generic function comprises concrete methods of the same name and number of arguments: generic functions with the same name but different parameter counts are independent. Like Dylan, Cecil supports optional static type declarations, but unlike Dylan, different syntactic constructs are used to define static type declarations and dynamic specialisers. A parameter can incur a static type definition, specialisation, or both.

Diesel Diesel (Chambers, 2006) is a descendant of Cecil, however generic functions are declared explicitly (called *functions*). Each Diesel function can have a default implementation, which in our model corresponds to a concrete method with no specialised parameters. Additional concrete methods (simply called *methods*) can augment a function by specialising any subset of its parameters.

Language	Typing	GF term	GF def.	CM term	CM grouped in GF by	Multi	Autogen. acc.
CLOS	dyn	generic function	explicit	method	name [a]	no	yes
Dylan	opt	generic function	explicit	method	name [a]	no	yes
Cecil	opt	method	implicit	method body	name+#args	no	yes
Diesel	opt	function	explicit	method	name+#args	no	yes
Nice	static	method declaration	implicit	method implementation	name+#args+types	yes	no
MultiJava	static	method family	implicit	method	name+#args+types	no	no
Java	static	method call	implicit	method body	name+#args+types	yes [b]	no
Smalltalk	dyn	message	implicit	method	name(+#args) [c]	no	no

[a] All argument lists (lambda lists) must be congruent.

[b] from Muschevici et al. (2008)

[c] Smalltalk message selectors encode the number of arguments to the message.

Figure 3.2: Multimethods across languages. Columns describe language name; static, dynamic, or optionally static typing; the terminology used for “generic function” (GF); whether generic function definitions are explicit or implicit; the term used for “concrete method” (CM); how concrete methods are grouped into generic functions (i.e. how a generic function signature is defined); whether one concrete function can be part of multiple generic functions; and whether the language automatically generates accessor messages (which we elide from our analysis).

Nice Nice (Bonniot et al., 2008) is a more recent multiple dispatch language design based on Java. A Nice generic function (*method declaration*) supplies a name, a return type and a static signature. Different concrete methods (*method implementations*) can exist for a declaration. When defining a concrete method, the parameter type declarations serve as dynamic specialisers if they are different to (that is more specific than) the types stated in the method declaration.

MultiJava MultiJava (Clifton et al., 2006) is an extension of Java that adds the capability to dynamically dispatch on other arguments in addition to the receiver object. A generic function (also called *method family*) consists of a *top method*, which overrides no other methods, and any number of methods that override the top method. Any method parameter can be specialised by specifying a true subtype of the corresponding static type or a constant value.

Multiple dispatch subsumes single dispatch: where in multiple dispatch languages, any argument to a function can be used to dispatch, in single dispatch languages it is only the first parameter that can be dispatched on. To compare the multiple dispatch languages described above to mainstream languages, we use our model to additionally describe Smalltalk and Java.

Smalltalk Smalltalk is a single dispatch language which introduced the terms *message*, roughly corresponding to implicitly defined generic function, and *method* for concrete method. Smalltalk is dynamically typed, and every message is single dispatched (even the equivalent of constructors and static messages, which are sent dynamically to classes). Every method name (or *selector*) defines a new generic function, and the names encode the number of arguments to the message.

Java Java is a single dispatch statically typed class-based language; it uses the term “method” for both generic functions (*method call*) and concrete methods (*method bodies*). Generic functions are defined implicitly, and depend on the names and the static types of their arguments.

3.3 Metrics

Our study approaches multimethods and multiple dispatch from a programmer’s point of view by analysing source code available publicly, mostly under open-source licenses. We focus on method definitions which we examine statically.

Abbr.	Name	Basis	Description
DR	Dispatch Ratio	generic function	number of methods in the generic function
CR	Choice Ratio	concrete method	number of methods in the same generic function
DoS DoS$_G$	Degree of Specialisation	concrete method generic function	number of specialisers
RS RS$_G$	Rightmost Specialiser	concrete method generic function	rightmost specialised argument position
DoD	Degree of Dispatch	generic function	number of specialisers required to dispatch to a concrete method
RD	Rightmost Dispatch	generic function	Rightmost Specialiser required to dispatch to a concrete method

Figure 3.3: Metrics

To study multiple dispatch across languages we define metrics based on our language independent model. Figure 3.3 summarises the metrics we define in this section.

3.3.1 Dispatch Ratio (DR)

We are most interested in measuring the relationships between generic functions and concrete methods. We define $CM(g)$ as the set of concrete methods belonging to a given generic function g. The number of concrete methods that belong to a generic function g gives the basic metric *Dispatch Ratio* $\text{DR}(g) = |CM(g)|$. DR measures, in some sense, the amount of choice offered by a generic function: monomorphic functions will have $\text{DR}(g) = 1$, while polymorphic functions will have $\text{DR}(g) > 1$.

We are usually not interested in the measurements from the above metrics (Figure 3.3) for individual generic functions or concrete methods, but rather we want to know about their distribution over a given application, or even collection of applications. We can report the measurements as a frequency distribution, that is, for a value dr, what proportion of generic functions g have $\text{DR}(g) = dr$.

Frequency distributions provide information such as: what proportion of generic functions have exactly one concrete method.

We use the basic DR metric to define an average Dispatch Ratio across each application. The average Dispatch Ratio DR_{ave}, that is the average number of concrete methods that a generic function would need to choose between is:

$$\mathrm{DR}_{ave} = \frac{\sum_{g \in \mathcal{G}} \mathrm{DR}(g)}{|\mathcal{G}|}$$

where $\mathcal{G}$ is the set of all generic functions. The intuition behind DR_{ave} is that if you select a *generic function* from a program at random, to how many concrete methods could it dispatch?

3.3.2 Choice Ratio (CR)

Because a generic function with a DR > 1 necessarily contains more methods than a monomorphic generic function, we were concerned that DR_{ave} can give a misleading low figure for programs where some generic functions have many more concrete methods than others.

For example, consider a program with one generic function with 100 concrete methods, $\mathrm{DR}(g_1) = 100$, and another 100 monomorphic methods $\mathrm{DR}(g_{2..101}) = 1$. For this program, $\mathrm{DR}_{ave} = 1.98$, even though *half* the concrete methods can only be reached by a 100-way dispatch.

To catch these cases, we define the *Choice Ratio* of a concrete method m to be the total number of concrete methods belonging to all of the generic functions to which m belongs:

$$\mathrm{CR}(m) = |\bigcup_{g \in GF(m)} CM(g)|$$

Note that this counts each concrete method only once, even if it belongs to multiple generic functions. An application-wide average, CR_{ave} can be defined similarly:

$$\mathrm{CR}_{ave} = \frac{\sum_{m \in \mathcal{M}} \mathrm{CR}(m)}{|\mathcal{M}|}$$

where $\mathcal{M}$ is the set of concrete methods. The intuition behind CR_{ave} is that if you select a concrete method from a program at random, then how many other concrete methods could have been dispatched instead of this one? For the example above $\mathrm{CR}_{ave} = 50.5$.

3.3.3 Degree of Specialisation

Degree of Specialisation of a Concrete Method (DoS)

The *Degree of Specialisation* of a *concrete method* simply counts the number of specialised parameters:

$$\text{DoS}(m) = |\textbf{spec}(m)|$$

where **spec**(m) is the set of argument positions of all specialisers of the method m (we will later write $\textbf{spec}_i(m)$ for the i'th specialiser). DoS can also be extended to an average, DoS_{ave} in the obvious manner, over all concrete methods.

Dynamically specialising multiple method parameters is a key feature of multiple dispatch: DoS measures this directly. Functions without dynamic dispatch, like Java static methods, C functions, or C++ non-virtual functions, will have DoS $= 0$. Singly dispatched methods like Java instance methods, C++ virtual functions, and Smalltalk methods will have DoS $= 1$. Methods that are actually specialised on more than one argument will have DoS > 1.

Degree of Specialisation of a Generic Function (DoS_G)

The *Degree of Specialisation* of a *generic function* counts the specialisers of that generic function.

$$\text{DoS}_\text{G}(g) = |P|, \quad \text{where } i \in P \text{ iff } \exists m \in CM(g) \text{ such that } i \in \textbf{spec}(m)$$

Specialisers for formal parameters of a generic function are inferred from the concrete methods that belong to it: if at least one concrete method in the generic function has a dynamic specialiser at a certain position, then the same parameter position of the generic function is considered specialised.

We define this metric in addition to the *per concrete method* DoS metric for accuracy: we compare DoS_G measurements with those furnished by the Degree of Dispatch (DoD) metric (§ 3.3.5), which is also defined in terms of generic functions.

DoS_G can also be extended to an average, $\text{DoS}_{\text{G},ave}$ over all generic functions.

3.3.4 Rightmost Specialiser

Rightmost Specialiser of a Concrete Method (RS)

Programmers read method parameter lists from left to right. This means that a method with a single specialiser on the last (rightmost) argument may be qualita-

tively different to a method with one specialiser on the first argument. To measure this we define the *Rightmost Specialiser*:

$$\text{RS}(m) = \mathbf{max}(\mathbf{spec}(m))$$

If a method has some number of specialised parameters (perhaps none) followed by a number of unspecialised parameters, then RS = DOS; where a method has some unspecialised parameters early in the list, and then some specialised parameters, RS > DOS. The capability to specialise a parameter other than the first distinguishes multiple dispatch languages from single dispatch languages. RS can, for example, identify methods that use single dispatching (DOS=1) but where that dispatch is not the first method argument. Once again, we can define a summary metric RS_{ave} by averaging RS over all concrete methods.

Rightmost Specialiser of a Generic Function (RS_G)

Similarly to the Rightmost Specialiser of a concrete method, we define the *Rightmost Specialiser* of a *generic function* as the rightmost specialised parameter position of that function:

$$\text{RS}_\text{G}(g) = max(\text{RS}(m)),\ \text{where } m \in CM(g)$$

By averaging RS_G over all generic functions, we can define $\text{RS}_{\text{G},ave}$.

3.3.5 Degree of Dispatch (DOD)

The *Degree of Dispatch* is the number of parameter positions required for a generic function to select a concrete method. The key point here is that specialising concrete method parameters does not by itself determine whether that parameter position will be required to dispatch the generic function. This is because all the concrete methods in the generic function could specialise the same parameter position in the same way. Similarly, if only one concrete method specialises a parameter position, that position could still participate in the method dispatch even if no other concrete method specialises that parameter — the other concrete methods acting as defaults.

The DOD metric counts the number of parameter positions where two (or more) concrete methods in a generic function have different dynamic specialisers. In general, these are the positions that must be considered by the dispatch algorithm.

$$\text{DoD}(g) = |P|, \quad \begin{array}{l}\text{where } i \in P \text{ iff } \exists m_1, m_2 \in CM(g)\\ \text{such that } \mathbf{spec}_i(m_1) \neq \mathbf{spec}_i(m_2)\end{array}$$

Where the Degree of Specialisation (DoS_G) measures the number of parameters of a generic function which the programmer has designed dispatch-able (by specialising them), the Degree of Dispatch counts only those parameters on which the generic function actually has the potential to dispatch upon.

We can once again define a summary metric DoD_{ave} as the average over all generic functions. If DR_{ave} and CR_{ave} measure the amount of choice involved in dispatch, then DoD_{ave} measures the complexity of that choice.

3.3.6 Rightmost Dispatch (RD)

Finally, by analogy to RS, we can define RD as the rightmost parameter a generic function actually dispatches upon.

$$\text{RD}(g) = \mathbf{max}(P), \quad \begin{array}{l}\text{where } i \in P \text{ iff } \exists m_1, m_2 \in CM(g)\\ \text{such that } \mathbf{spec}_i(m_1) \neq \mathbf{spec}_i(m_2)\end{array}$$

RD is to RS as DoD is to DoS: the "Do" versions count specialisers of methods, or dispatching positions of generic functions, while the "R" versions consider only the rightmost position. RD for a generic function will usually be the maximum RS of that function's methods, unless every concrete method in the generic function specialises the rightmost parameter in the same way. For a whole application, we can report RD_{ave} as the average RD across all generic functions.

3.4 Example

To illustrate the metrics, consider the following simple multiple dispatch example written in Dylan, which defines a range of binary methods for the type hierarchy <sports−car> extends <car> extends <vehicle>.

This example defines two generic functions (collide and pileup) with two and four concrete methods respectively. The values for the metrics relevant to each declaration are in the comments above them.

```
define class <vehicle> (<object>) ... ;
define class <car> (<vehicle>) ... ;
define class <sports-car> (<car>) ... ;
```

```
// DR = 2, DoD = 1, RD = 2, DoS(g) = 2, RS(g) = 2
define generic collide(v1 :: <vehicle>, v2 :: <vehicle>);
// CR = 2, DoS = 1, RS = 1
define method collide(sc :: <sports-car>, v :: <vehicle>) ... ;
// CR = 2, DoS = 2, RS = 2
define method collide(sc :: <sports-car>, c :: <car>) ... ;

// DR = 4, DoD = 3, RD = 3, DoS(g) = 3, RS(g) = 3
define generic
  pileup(v1 :: <vehicle>, v2 :: <vehicle>, v3 :: <vehicle>);
// CR = 4, DoS = 2, RS = 3
define method
  pileup(sc :: <sports-car>, v :: <vehicle>, c :: <car>) ... ;
// CR = 4, DoS = 2, RS = 2
define method
  pileup(sc :: <sports-car>, c :: <car>, v :: <vehicle>) ... ;
// CR = 4, DoS = 3, RS = 3
define method
  pileup(c :: <car>, c :: <car>, c :: <car>) ... ;
// CR = 4, DoS = 0, RS = 0
define method
  pileup(v :: <vehicle>, v :: <vehicle>, v :: <vehicle>) ... ;
```

DR is 2 for collide and 4 for pileup because that is the number of concrete methods each of these generic functions contains. Obviously, each of the concrete methods has a respective CR of 2 and 4. However the difference can be observed if we try and count the DR_{ave} and CR_{ave} for this Dylan example. $DR_{ave} = (2+4)/2 = 3$ is the Dispatch Ratio for this program that examines each generic function. $CR_{ave} = (2+2+4+4+4+4)/6 = 3.33$ is the Choice Ratio for this program that examines each concrete method. This means that the choice of alternative concrete methods for each method is larger than the average number of methods per generic function.

DOS is calculated for each concrete method by examining the number of specialisers, while RS records the position of the rightmost specialiser (accounting in particular for the second concrete method collide that does a single dispatch on *a second* argument). Averages for DOS and RS give us $(1+1+2+2+3+0)/6 = 1.5$ and $(1+2+3+2+3+0)/6 = 1.83$ respectively.

Finally, DOS_G, RS_G, DOD and RD are measured at the level of generic functions. DOS_G measures the number of a generic function's parameter positions that are specialised by at least one of its concrete methods. RS_G captures the rightmost of these positions. DOD records the number of a generic function's parameters

that can be potentially used to dispatch upon. Finally, RD records the rightmost position used by such a parameter. Their averages are: $\text{DOS}_{\text{G},ave} = (2+3)/2 = 2.5$; $\text{RS}_{\text{G},ave} = (2+3)/2 = 2.5$; $\text{DOD}_{ave} = (1+3)/2 = 2$; and $\text{RD}_{ave} = (2+3)/2 = 2.5$.

Chapter 4

Methodology

In this Chapter we first motivate Corpus Analysis, the research methodology underlying our study (§ 4.1). We then present our corpus, a collection of applications written in multiple dispatch languages (§ 4.2). We proceed to describe the analysis techniques and tools we developed to apply the metrics presented in Section 3.3 to the data collected from the corpus (§ 4.3). Finally, we demonstrate the customised approaches we use to extract and store the information relevant to our research goals from the corpus into an intermediate processing format (§ 4.4).

4.1 Corpus Analysis

The Software Corpus Analysis approach uses an automated process to measure certain artifacts in a body of programs (in the form of their source code or any compilation stage). Assuming that the studied body (corpus) is representative of how the language is used in general, one can derive a set of rules or patterns governing that language.

A large amount of research in software engineering has focused on developing models and methodologies of how software should be written to meet certain quality criteria such as re-usability, maintainability and cost-effectiveness. Corpus Analysis is an approach that measures attributes of software as it actually is. By understanding the shape of existing software, we can learn about the characteristics of good software.

Software Corpus Analysis is a widely used empirical research method. There are many recent examples addressing program topology (Potanin et al., 2005; Baxter et al., 2006), mining patterns (Fabry and Mens, 2004; Gil and Maman, 2005), object initialisation (Unkel and Lam, 2008), aliasing (Potanin et al., 2004; Ma and Foster, 2007), dependency cycles (Melton and Tempero, 2007), exception handling (Cabral and Marques, 2007), inheritance (Tempero et al., 2008), non-nullity (Chalin

and James, 2007) and visualisation (Noble and Biddle, 2002).

We adopt the corpus analysis approach to study the use of multiple dispatch. Given that, using a multiple dispatch language, programmers can freely choose whether to use either multiple dispatch or another technique, we want to know how much multiple dispatch *is* used in practice in a set of languages. Corpus analysis can answer this question by *measuring* — across a given corpus — the proportion of method declarations that dispatch on more than one argument.

There are of course alternative approaches to studying the use of multiple dispatch. They range from manually analysing selected source code artifacts, surveying practitioners about their perceived usage of multiple dispatch to using multiple dispatch languages in teaching and as part of class projects. These approaches are valuable and we acknowledge their potential to complement corpus analysis: they can help verify our results and answer further questions, such as *when* and *why* is multiple dispatch used. For some examples, corpus analysis can draw attention to interesting code portions in a program, which can then be examined manually in order to understand the kind of problems they solve. A study involving in-field interviews could reveal the level of awareness practitioners have of multiple dispatch and its potential. Any discrepancies between the findings from our study and the subjective value attributed to multiple dispatch by practitioners could help improve software engineering pedagogy. Class projects involving a multiple dispatch language and a single dispatch language as a control could provide further insight into the relative value of multiple dispatch in practice.

Our Corpus Analysis of multiple dispatch is a first step in a line of research aimed at understanding how much value is gained from the use of multiple dispatch in practice.

4.2 Corpus

For this study we have gathered a corpus of nine applications written in six languages that offer multiple dispatch[1] (Figure 4.1).

Most applications in this corpus are compilers for the respective language — they are all too often the only applications of significant size that we could obtain. This fact certainly introduces a degree of bias to our analysis. An additional reason that limits the generalisability of our results comes from the nature of these compilers being mostly academic projects: the development processes undertaken

[1] The multiple dispatch corpus is available from `http://homepages.mcs.vuw.ac.nz/~muscheradu/mdip/`.

at universities cannot be regarded as representative for software development at large.

CLOS is notably distinct with respect to the availability of large projects and the corpus could be expanded by several CLOS projects. We opted to cover a broad spectrum of languages rather than weighting this study towards one language because we are interested in measuring multiple dispatch *across* languages. We note that this corpus contains a single application per language (with exceptions of CLOS with three and Dylan with two applications): this largely reflects the fact that multiple dispatch languages are not in wide use today.

All of the languages studied here come with more or less extensive standard libraries. Our measurements of each application include the contribution due to the libraries.

As is often the case when measuring real code, we have to make assumptions about exactly what to measure. One assumption is with respect to the auto-generated field accessors some languages provide (see Figure 3.2). As our interest is in how programmers interact with language features, we do not measure these accessors, nor do we measure other compiler-generated artefacts.

The following paragraphs briefly present each application in our corpus, giving a rough idea about its scope, development history and size. Figure 4.2 shows the relative sizes of applications at a glance, measured in terms of generic functions and concrete methods. We do not give the lines of code (LOC) measurement as it can be misleading for applications written in languages that can mix object-oriented and functional code (such as CLOS and Dylan).

4.2.1 CMUCL (CLOS)

The first of three Common Lisp applications in our corpus, CMUCL is a free implementation of the Common Lisp programming language as defined by ANSI (American National Standards Institute, 1996). It has been in development since the early 1980's at the Computer Science Department of Carnegie Mellon University. The package includes a compiler and an extensive library (CMUCL Maintainers, 2008). CMUCL auto-generates a large number of generic functions, which we exclude from our analysis (§ 4.4.1). We examined version 19d and found it to define 271 generic functions and 550 concrete methods.

Language	Application	Domain	Version	Concrete methods	Generic functions
CLOS	CMUCL	compiler	19d	550	271
CLOS	SBCL	compiler	0.9.16	861	363
CLOS	McCLIM	toolkit/ library	0.9.5	4419	1906
Dylan	OpenDylan	compiler	1.0beta5, SVN rev. 11779	5389	2143
Dylan	Gwydion	compiler	2.5.0pre3, SVN rev. 11733	6621	3799
Cecil	Vortex	compiler	3.3	15212	6541
Diesel	Whirlwind	compiler	3.3	11871	5737
Nice	NiceC	compiler	0.9.13	1615	1184
MultiJava	LocStack	framework	0.8	735	491

Figure 4.1: Applications in multiple dispatch corpus.

4.2.2 SBCL (CLOS)

Steel Bank Common Lisp (SBCL) is an open source/free compiler and runtime system for ANSI Common Lisp. SBCL was forked from CMUCL in 1999 and has evolved independently since then. The system is described to be simpler than CMUCL, partly because some extensions were removed, partly due to ample refactoring (SBCL Maintainers, 2008). We found this fact reflected in SBCL's smaller size, compared to CMUCL (before excluding auto-generated class predicates): the version we examined (0.9.16) has 363 generic functions and 861 concrete methods.

4.2.3 McCLIM (CLOS)

McCLIM is an open source implementation of the Common Lisp Interface Manager specification (McKay and York, 2001), a toolkit for writing graphical user interfaces in Common Lisp. It runs on top of several Lisp implementations, including CMUCL and SBCL. The project initiated in 2000 by merging several developers' individual efforts to write a free implementation of CLIM; by the end of 2002 it was considered "almost feature complete" by its developers (Strandh and Moore, 2002). McCLIM is the largest Common Lisp project in our corpus, containing 1906 generic functions and 4419 concrete methods.

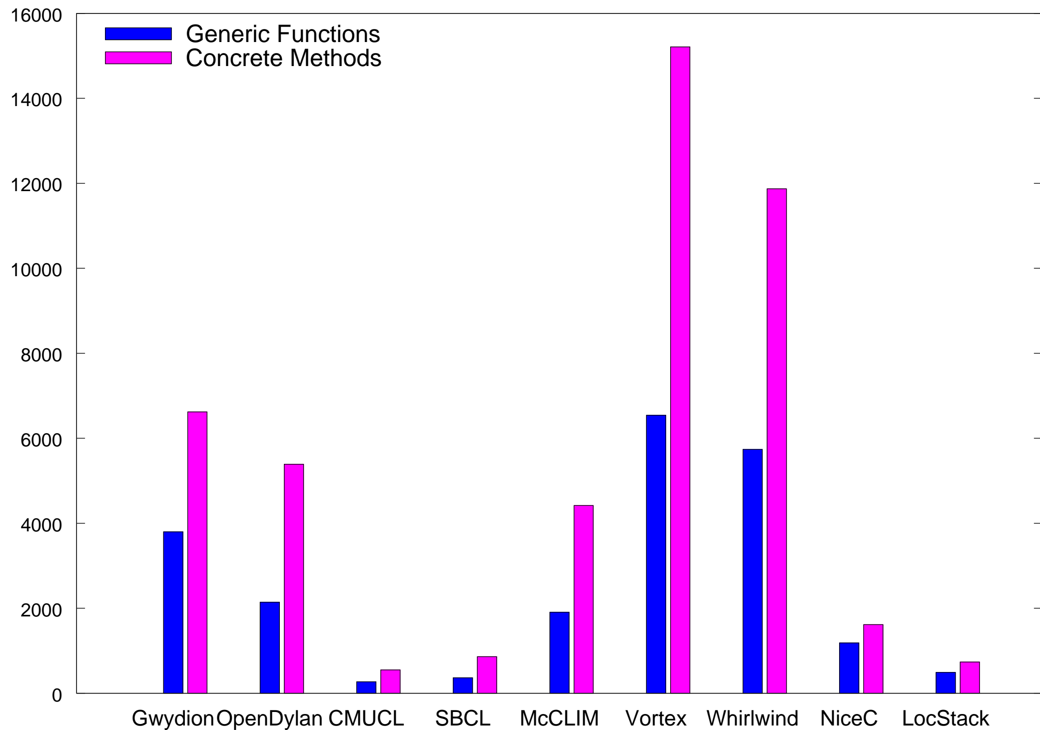

Figure 4.2: Relative size of applications in corpus

4.2.4 Open Dylan (Dylan)

OpenDylan, also known as Functional Developer, is a fully-featured development environment originally created in the early 1990's at Apple with the objective of combining the best qualities of Common Lisp with the advantages offered by static languages like C++ (Feinberg, 1997). Only the command line version of the compiler, known as the *minimal console compiler* currently runs on GNU/Linux, the platform we used for our analysis. The version we inspected (1.0 beta5; obtained from the SVN repository on 27-April-2008) has 2143 generic functions and 5389 concrete methods.

4.2.5 Gwydion (Dylan)

Gwydion Dylan includes a Dylan-to-C (d2c) compiler originally created at Carnegie Mellon University, mostly by developers who had previously worked on the CMU Common Lisp (CMUCL) project. d2c uses gcc as the back-end for generating native code. The current release 2.4.0 is considered a technology preview by its maintainers (Gwydion Dylan Maintainers, 2008) due to limitations that make it hard to use for production development, such as the lack of incremental compilation ("d2c generates fast code slowly"), uncomfortable debugging and slightly incomplete

support for the Dylan language specification. We analysed a pre-release of version 2.5 which we obtained from the SVN repository on 12-March-2008. Including contributions of various libraries, this version has 3799 generic functions and 6621 concrete methods.

4.2.6 Vortex (Cecil) and Whirlwind (Diesel)

Vortex and Whirlwind are compiler front-ends developed by the Cecil Group (Cecil Group, 2008) at the University of Washington's Department of Computer Science and Engineering. They are written in Cecil and Diesel respectively and are designed to plug into the Vortex compiler back-end. We use release 3.3 of the Vortex compiler infrastructure and count 6541 generic functions and 15212 concrete methods in the Cecil code, versus 5737 generic functions and 11871 concrete methods in Diesel source.

4.2.7 The Nice Compiler (Nice)

The Nice language and compiler was developed as part of an academic project on object-orientation at Institut National de Recherche en Informatique et en Automatique (INRIA) in Rocquencourt, France(Bonniot et al., 2008). Presently (version 0.9.13), it is not considered feature complete. The compiler and Nice standard library itself are written partly in Java (such as the ML-Sub type system (Bourdoncle and Merz, 1997)) and partly in Nice. For a crude idea about proportions it is useful to remark that the project includes some 35,000 lines of Java code and 23,000 lines of Nice code. For our analysis we only consider the contribution made by native Nice code, which we found to define 1184 generic functions and 1615 concrete methods.

4.2.8 The Location Stack (MultiJava)

The MultiJava-based Location Stack (Hightower, 2002) is a framework for processing measurements from a heterogenous network of geographical location sensors. Its development began as part of the Portolano project on ubiquitous computing at the University of Washington; the efforts have been since carried forward into the PlaceLab project at Intel Research Seattle, though PlaceLab has abandoned MultiJava in favour of more mainstream Java. We base our analysis on the latest release 0.8, counting 491 generic functions and 735 concrete methods.

4.3 Analysis

The first step towards measuring the use of multiple dispatch across a corpus is to use the model from Chapter 3 and identify how generic functions and concrete methods are defined in each of the six languages.

For each language, the task is to collect and store the information related to all concrete entities found within applications in our corpus. In practice, we need to develop a language-specific technique to identify generic functions, their concrete methods and the specialisers of each method, as found in the source code. This information has to be stored in a unified, language independent format and made available for further analysis. We refer to the information we collect and store for each application as a sample.

All this information is already available to a compiler after parsing the source, since the compiler itself is in charge of assigning a particular generic function to each call-site, and possibly optimising dynamic dispatch.

Consequently, we focus our attention on the compilers for the given languages. By relying on a compiler's parser to identify generic functions, we are able to obtain the most accurate information. This kind of information is generally not made available externally, which is why, in some cases, we had to modify the compilers in order to supplement their output during the compilation process.

Given the diversity of programming languages (and hence compilers) used in our corpus, we needed to develop a customised approach for each of them. Fortunately, all compilers we use come with source code and their authors were very helpful in directing us to the sections in the source code we needed to change.

The compilers for CLOS, OpenDylan, Cecil and Diesel provide an interface for introspection of the project being compiled. For these languages, it proved to be most practical to write small utilities that use this interface. We run these in the context of a project being compiled to query information built in memory by the compiler. The CLOS metaobject protocol (MOP) (Kiczales et al., 1991) falls in this category. The MOP provides a particularly elegant way to inspect the structure of a loaded program with respect to static information including class inheritance relationships, generic function and methods, as well as dynamic information about instances and data.

The next section describes the intermediate format used to store the raw data we obtain by compiling each application in the corpus, followed by a section presenting the tool we have developed for multiple dispatch corpus analysis.

```
sample ::= "{", { generic-function-entry, "," }, "};"          ;

generic-function-entry ::= string, "=>", "{", body, "}"         ;

body ::=        "specialisers", "=>", specialiser-vectors,
         [ ",", "static types", "=>", string-vector ]           ;

specialiser-vectors ::= "[", { string-vector, "," }, "]"        ;
string-vector       ::= "[", { string, "," }, "]"               ;
string              ::= "'", name, "'"                          ;
name                ::= { all-characters - "'" }                ;
all-characters      ::= ? all visible characters ?              ;
```

Figure 4.3: Grammar for the intermediate format used to collect data from applications in the corpus.

4.3.1 Intermediate Format

We modify each compiler to save the compilation output to a file in an intermediate, language independent format. Each output file contains structural data describing an application's generic function definitions: their names, the static types of their formal parameters and the specialisers defined by concrete methods.

The intermediate format uses nested Perl hash tables. The context-free grammar for our intermediate format is specified in Figure 4.3. Note that for legibility, the grammar presented here does not allow whitespace, yet our format allows whitespace under the rules of the Perl language.

Each generic function is identifiable inside the outer hash table by a unique key consisting of the function's qualified name and (when the language requires it) the number of formal parameters. In turn, each generic function element defines a hash table that stores arrays of specialiser vectors ('specialisers' key). Each of these specialiser vectors stores strings corresponding to the specialisers defined by each concrete method. All specialiser vectors have the same size, with empty strings for non-specialised argument positions. Some languages support static type definitions; we record these in an array referenced by the 'static types' key.

The following example illustrates the format described above by showing the data obtained by compiling the Dylan example discussed in Section 3.4. Note that the Dylan language does not allow generic functions with the same name but a different number of parameters, therefore the function name alone can serve as a unique key.

```
{
'vehicle-simulation:collide' => {
  'static types' => [ '<vehicle>', '<vehicle>' ];
  'specialisers' => [
    [ '<sports-car>', '<vehicle>' ],
    [ '<sports-car>', '<car>' ]
  ],
},
'vehicle-simulation:pileup' => {
  'static types' => [ '<vehicle>', '<vehicle>', '<vehicle>' ];
  'specialisers' => [
    [ '<sports-car>', '<vehicle>', '<car>' ],
    [ '<sports-car>', '<car>', '<vehicle>' ],
    [ '<car>', '<car>', '<car>' ],
    [ '<vehicle>', '<vehicle>', '<vehicle>' ]
  ],
}
};
```

4.3.2 Multiple Dispatch Corpus Analysis Tool (MuDiCAT)

We developed a program to analyse the raw data we obtain through the language-specific front-ends (§ 4.4). This section states the requirements of the analysis tool, presents our design and overviews some implementation aspects.

Requirements

Overall, the Multiple Dispatch Corpus Analysis Tool serves the purpose of measuring the data recorded from applications using the metrics introduced in Section 3.3. The results from these measurements have to be stored and visualised.

MuDiCAT reads in records (produced by the various language-specific front-ends) in the format defined in Section 4.3.1 into an in-memory object which we refer to as a *sample*. A sample has to be measured in terms of all metrics defined in Section 3.3.

A metric based on generic functions computes a value for each generic function in the sample. Similarly, metrics based on concrete methods compute a value for each concrete method. A metric makes these results available and also provides them as a *distribution*, that is a collection of tuples containing all measured values along with their frequency across the entire sample. Additionally, a metric

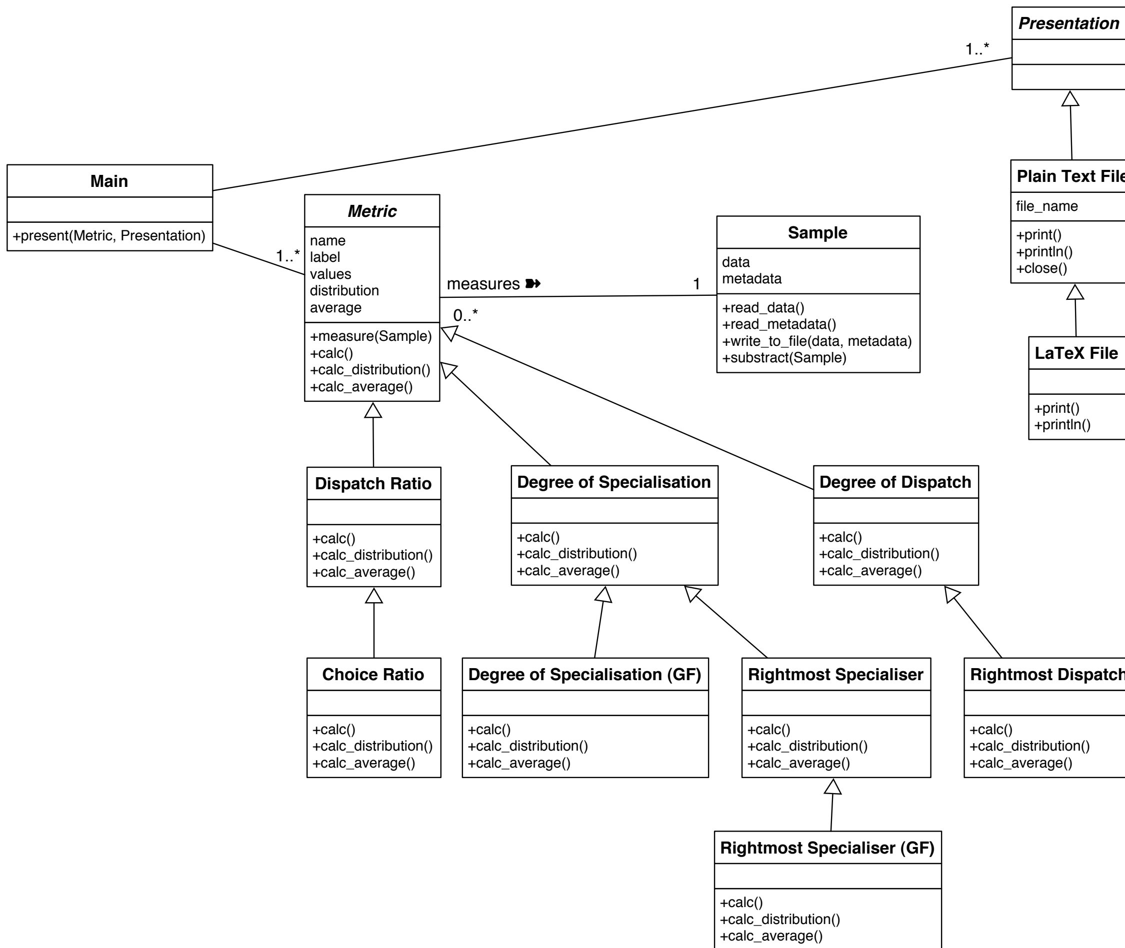

Figure 4.4: Class diagram for the Multiple Dispatch Corpus Analysis Tool

computes an *average* from the values measured across the sample.

Distribution and average of each processed sample have to be stored to plain-text files suitable for visualisation via gnuplot[2] as well as in files conforming to LaTeX table syntax, suitable for inclusion into documents.

Design

Figure 4.4 shows the class diagram of our analysis tool. A Sample contains a table with the raw data collected through the respective language-specific front-end from one application in the corpus, as well as meta-information such as the name and version of the application this data was collected from. Sample also implements a subtraction operation that subtracts a set of generic functions from a given superset, as this is required for the McCLIM application using the technique described in Section 4.4.1.

A Metric operates upon a Sample to compute a value for each generic function

[2]Gnuplot plotting utility: `http://www.gnuplot.info/`

or concrete method. These values are used to calculate a distribution and an average value. Each Metric subclass individually implements these methods according to the definition for the respective metric. We save computed values, distributions and averages through different Presentation objects to plain text files, suitable for further processing (such as visualisation) through, e.g. gnuplot, as well as to LaTeX tables suitable for inclusion into documents.

Implementation

We chose to implement the analysis back-end in Perl for practical reasons: on one hand, we are familiar with this language; on the other hand, as a dynamic language, Perl allows the rapid prototyping/exploratory programming style we deem most suitable with the research nature of this project.

The raw data recorded by language-specific front-ends is saved to files using a Perl hash table syntax (described in Section 4.3.1). We choose this approach instead of inventing our own record format so that we can rely on the Perl compiler to read and parse the recorded data.

At this stage, the project contains 20 Perl modules, 14 classes, and around 100 unit test cases for the Test::More Perl testing framework. We use about a dozen auxiliary modules from the Perl library (CPAN) for various tasks such as unit testing, I/O, logging/tracing (Log::Log4perl, Data::Dumper), extended regular expressions (Regexp::Common), and multiple dispatch methods (Class::Multimethods).

4.4 Language-specific Analysis Front-ends

This section goes into the details of each language and the customised approaches we use with the respective compilers to collect the raw data for our analysis.

4.4.1 Common Lisp Compilers – CMUCL and SBCL

The metaobject protocol is part of the Common Lisp Object System (CLOS) (Bobrow et al., 1988), which in turn is part of the Common Lisp ANSI standard. We use the metaobject protocol to collect information about all generic function, method and specialiser objects in a compiled project. Our implementation, which is based on an example in Kiczales et al. (1991, Chapter 2), walks the class hierarchy tree starting at the root class (Common−Lisp:T) and collects the generic function objects that specialise (via method definitions) on each class. Calling (method (generic−function−methods gf)) produces the methods defined by the given generic function gf. Subsequent calls to (method−specializers method) for

each method returns its specialisers. Accessor methods which are auto-generated are detected as they subclass standard−accessor−method and are excluded from counting.

Using this technique in CMUCL, we found that a large proportion of generic functions have a remarkably symmetrical shape. Our investigations showed that CMUCL auto-generates generic function predicates for every class in the system. These predicates are used internally to test whether an object is an instance of the corresponding class, and each of them implements two concrete methods: one is unspecialised and the other is specialised on the particular class. The following function recorded in intermediate format (§ 4.3.1) is such an example:

```
'common-lisp-user:(class-predicate number)' => {
  'specialisers' => [
    [ '' ],
    [ 'common-lisp:number' ],
  ],
}
```

Generic functions of the kind shown above are single-dispatch — they have a Dispatch Ratio DR $= 2$ and a Degree of Dispatch DoD $= 1$. As they are auto-generated by the CMUCL compiler, we exclude them from our analysis, which brings the number of generic functions in our CMUCL sample down by about 40%.

To measure McCLIM we load it into the SBCL compiler environment and use the metaobject protocol as described above to recount generic functions in the system. We separate generic functions defined in the McCLIM source from the rest of the SBCL system by subtracting the set of generic functions we had recorded before loading McCLIM from the current set. For the case of overlapping functions, that is when SBCL and McCLIM both contribute concrete methods to the same generic function, we exclude the method contributions due to SBCL from accounting. As a side note, we found that McCLIM adds new concrete methods to 33 generic functions already defined in SBCL; among these are 67 additional implementations of common−lisp−user:print−object and 160 of common−lisp−user:initialize−instance.

Consider now this example of a generic function containing two concrete methods, taken from the McCLIM source.

```
(defgeneric display-drei-contents (stream drei syntax) (...))
(defmethod display-drei-contents (
  (stream clim-stream-pane)
  (drei drei)
  (syntax fundamental-syntax)) (...))
(defmethod display-drei-contents (
```

```
  (stream clim-stream-pane)
  (drei drei)
  (syntax lisp-syntax)) (...))
```

According to the documentation, the above generic function is designed to contain methods that display the buffer contents of an object of type Drei to some output surface (stream) using a certain syntax. A Drei instance in general can be any type of editor (e.g. an input-editor, a text editor gadget or a simple pane). In this example the generic function contains two concrete methods which specialise all three parameters. Note that in this case only the third parameter (syntax) is specialised in different ways across the two methods.

The CLOS introspection front-end records the relevant information from the previous code example in a Perl data structure as shown below. All names are recorded fully-qualified, with the name of the module prepended. According to the rules of the language, the two methods are recognised as part of the given generic function because they are defined as having the same name. Hence, the qualified name of the generic function is enough to uniquely identify this generic function and its methods during further processing.

```
'common-lisp-user:display-drei-contents' => {
  'specialisers' => [
    [
      'clim:clim-stream-pane',
      'drei:drei',
      'drei-lisp-syntax:lisp-syntax'
    ],
    [
      'clim:clim-stream-pane',
      'drei:drei',
      'drei-fundamental-syntax:fundamental-syntax'
    ]
  ]
};
```

4.4.2 Gwydion Dylan

Gwydion's d2c compiles each Dylan source module into a C source file. Each generic function, whether implicit or explicitly declared, is mapped to a C function (labelled <original name>_ROOT). Methods of that same generic function are mapped to a couple of C functions labelled METH_GENERIC_<number> and METH_<number> where each METH_GENERIC includes a call to a METH.

At runtime, a lookup function (gf_call_lookup_FUN) will be called with a pointer to the ROOT function whenever there is need to dispatch dynamically. The ROOT function will then select and call the appropriate METH_GENERIC function, which in turn will call its corresponding METH function.

We extract information about generic functions and methods by parsing the generated C code. This turned out to be the easiest approach since generated code has by its nature a very systematic layout. Gwydion does generate getters and setters for all slots defined in a class; these are annotated "internal" in the C code and we exclude them from counting.

The following example taken from the Gwydion Dylan library shows the three stages: Dylan source code, C code and the data we extract from the C code.

```
define sealed generic binary-logxor (x :: <general-integer>, y :: <general-integer>)
  => res :: <general-integer>;

// in num.dylan
define inline method binary-logxor (a :: <integer>, b :: <integer>)
  => res :: <integer>;
  ...
end;

// in bignum.dylan
define inline method binary-logxor
  (a :: <extended-integer>, b :: <extended-integer>)
  => (res :: <extended-integer>);
  ...
end;
define inline method binary-logxor
  (a :: <extended-integer>, b :: type-union(<integer>, <double-integer>))
  => res :: <extended-integer>;
  ...
end;
```

As described above, d2c compiles each of the concrete methods defined above into two C functions, one calling the other. Here are the C functions for the method binary−logxor(a :: <integer>, b :: <integer>).

```
/* generic-entry for binary-logxor{<integer>, <integer>} */
descriptor_t * dylanZdylan_visceraZbinary_logxor_METH_GENERIC_7(...) {
  ...
  L_result0 = dylanZdylan_visceraZbinary_logxor_METH_7(...);
}
```

```
/* Define Method binary−logxor{<integer>, <integer>} */
/* binary−logxor{<integer>, <integer>} */
long dylanZdylan_visceraZbinary_logxor_METH_7(...) {...}
```

The Gwydion-specific analysis front-end extracts the information contained in the C code and saves it in a Perl data structure, as shown below.

```
'dylan::dylan_viscera::binary_logxor' => {
  'specialisers' => [
    [ '<integer>', '<integer>' ],
    [ '<extended−integer>', '<extended−integer>' ],
    [ '<extended−integer>', 'type−union(<double−integer>, <integer>)' ]
  ],
  'static types' => [ '<general−integer>', '<general−integer>' ]
};
```

Note that, in contrast to CLOS, Dylan allows programmers to specify types for parameters in a generic function definition; these are checked statically. We record them in an additional static types array as part of the generic function data structure (see § 4.3.1). When assessing whether a method specialises a particular parameter, we compare the static type and specialiser – only if they differ (as is the case with all methods in this example), we consider the method to specialise that parameter. Also notable in this example is how Dylan allows to specialise parameters on non-class types, such as type unions.

4.4.3 Open Dylan

We use the "environment protocol", a library of the OpenDylan language that provides a convenient interface for meta-level introspection at compile-time. A small Dylan tool loads the "minimal console compiler" project into memory and triggers parsing of its source code. We then use environment protocol API calls to obtain a list of all generic functions along with their parameter types for each module. Subsequently, we query each generic function for its methods and each method for the list of its specialisers.

OpenDylan generates accessor methods for all slots of a class. We detect these accessors by first walking the class hierarchy to collect references to all slot objects. Then, on each slot object, we call slot−getter and slot−setter; these functions return accessor method objects, which we exclude from counting.

The approach we use with OpenDylan is notably different from how we handle code written for the Gwydion Dylan compiler. We have considered using a single

approach to compile both Dylan applications but this was not successful because of the differences in the libraries each language variant includes. It is however possible to compile simpler projects, such as the example discussed in Section 3.4 using either compiler. We did this and were able to verify that, when using the same input, both Gwydion and OpenDylan approaches produce equivalent outputs. We therefore omit giving an additional code example for OpenDylan.

4.4.4 The Vortex Compiler Infrastructure for Cecil and Diesel

Vortex is a compiler back-end that supports different language front-ends as plug-ins. By coincidence, the Cecil front-end is also named Vortex; Whirlwind is the Diesel front-end. The Vortex back-end includes the Cecil evaluator, an interactive environment similar to a Lisp read-eval-print loop (REPL). Through the Cecil evaluator, various kinds of introspection are possible on the global data structures describing the program being compiled, such as browsing the class hierarchy, and inspecting methods and fields.

To collect the information from Cecil and Diesel applications (Vortex and Whirlwind) we use the Cecil evaluator. In the course of developing the Vortex back-end, its authors have written a small utility that dumps the class structure and method groups of the project being compiled after the source code has been processed by the language front-end and transformed into the Vortex RTL intermediate language. The Vortex developers have provided us with this utility, which we modify to output information in our simple language independent format (§ 4.3.1) and to exclude auto-generated field accessor methods based on the fact that these method objects inherit from field_method_decl.

Note that we can use the same approach for collecting data from both Cecil and Diesel applications, as the information available at the Cecil evaluator has already been translated from each language into a shared internal format, which we translate into our Perl-based format.

For an example, consider the implementation of the division operator in the Cecil standard library (which in reality is spread over seven Cecil source files):

```
implementation / (l@:num,r@:num):float {...}
implementation / (l@:int, r@:int):int {...}
implementation / (l@:integer,r@:integer):integer {...}
method / (l@:big_int, r@:big_int):big_int {...}
method / (l@:float,r@:float):float {...}
method / (l@:single_float, r@:single_float):single_float {...}
method / (l@:double_float, r@:double_float):double_float {...}
```

Using the **method** keyword declares a method signature and implementation while **implementation** only introduces the declaration of a method implementation whose signature has been introduced earlier.

Now compare the above code example to the implementation of the division operator in the Diesel standard library, again spread across seven different Diesel source files: Note that Diesel supports explicit generic functions; they are defined using the keyword **fun**.

```
public fun / (:‘T <= num, :T):T (** no_itc **);
method / (l@num, r@num):float {...}
method signature / (l@int, r@int):int {...}
method signature / (l@integer, r@integer):integer {...}
method / (l@big_int, r@big_int):big_int {...}
method signature / (l@float, r@float):float {...}
method signature / (l@single_float, r@single_float):single_float {...}
method signature / (l@double_float, r@double_float):double_float {...}
```

Since Cecil and Diesel standard libraries appear to define the exact same concrete methods for the division operator (a fact that makes us speculate that the Diesel stdlib has been, at least in parts, automatically translated to Diesel from the Cecil library), this is how we (separately) record the information for each of the generic functions shown above:

```
'/(a,b)' => {
  'specialisers' => [
    [ 'num', 'num' ],
    [ 'int', 'int' ],
    [ 'integer', 'integer' ],
    [ 'big_int', 'big_int' ],
    [ 'float', 'float' ],
    [ 'single_float', 'single_float' ],
    [ 'double_float', 'double_float' ],
  ]
}
```

4.4.5 The Nice Compiler

Nice has the notion of *method declarations*, which corresponds to the generic function entity in our model (§ 3.1). A Nice method declaration has a name and a signature which consists of a return type and a list of parameter types. A method declaration can be associated with a set of *implementations*, (corresponding to concrete methods) which have no return type, but bear a name (the same as the

declaration's), a list of (optionally typed) parameters (of the same length as the declaration) and a block of code. Parameter types specified by a method declaration act as static types, while the types declared for the parameters of a method implementation act as specialisers.

A method declaration can override a previous method declaration by specifying a more specific return type and/or more specific parameter types. In such a case, all implementations of the overriding declaration will be also implementations of the overridden declaration.

The overriding declaration has to be explicitly declared as such (using the **override** keyword) in order to avoid a compiler warning. While the method overriding concept allows Nice concrete methods to be part of multiple generic functions, we found this feature used only in two cases in the Nice compiler source; one of them is shown in the following example taken from the syntax analysis module.

```
?Expression analyse(?Expression, Info);
analyse(?Expression e, info) {...}
override Expression analyse(Expression, Info);
analyse(TypeConstantExp e, info) {...}
analyse(ConstantExp c, info) {...}
analyse(TupleExp e, info) {...}
```

The method analyse is first declared to return an object of type ?Expression and takes two arguments of types ?Expression and Info. In Nice, a ? indicates that a type may be **null** — by default, Nice types are non-null. A second declaration of analyse overrides the first by specifying a (more specific) non-null type for the return value and first argument. As a consequence, of the four method implementations (concrete methods) shown here, the three who satisfy the parameter types of both method declarations are assigned to both method declarations (generic functions).

We modified the Nice compiler to write out the desired information as a Perl data structure. Here is the output for this example:

```
'analyse(?bossa.syntax.Expression, bossa.syntax.Info)' => {
  'static types' => [ '?bossa.syntax.Expression', 'bossa.syntax.Info' ],
  'specialisers' => [
    [ '?bossa.syntax.Expression', 'bossa.syntax.Info' ],
    [ 'bossa.syntax.TypeConstantExp', 'bossa.syntax.Info' ],
    [ 'bossa.syntax.ConstantExp', 'bossa.syntax.Info' ],
    [ 'bossa.syntax.TupleExp', 'bossa.syntax.Info' ]
  ]
}
```

```
'analyse(bossa.syntax.Expression, bossa.syntax.Info)' => {
  'static types' => [ 'bossa.syntax.Expression', 'bossa.syntax.Info' ],
  'specialisers' => [
    [ 'bossa.syntax.TypeConstantExp', 'bossa.syntax.Info' ],
    [ 'bossa.syntax.ConstantExp', 'bossa.syntax.Info' ],
    [ 'bossa.syntax.TupleExp', 'bossa.syntax.Info' ]
  ]
}
```

The two declared generic functions are identified by their name and static parameter types. The first generic function has a set of four specialiser vectors while the second one has only three vectors — these are a subset of the specialiser vectors of the first generic function. Note that the concrete methods shown in this example only ever specialise the first of two parameters of the generic functions.

As Nice does not generate accessors for fields, we do not need to worry about these.

4.4.6 The MultiJava Compiler

MultiJava shares with Cecil and Nice the property of having implicit generic functions, that is, no generic function construct exists as a first-class element of the language. A MultiJava generic function (sometimes referred to as a *method family*) consists of a *top method* (a possibly abstract method which overrides no other method) and all methods that override the top method (Clifton et al., 2006). The Multijava compiler groups methods together into families to form generic functions for dynamic dispatch.

We found that the most feasible approach for analysing MultiJava code was to *hack* mjc, the MultiJava compiler, to gather the information we need and output it in the intermediate format expected by our analysis tool (§ 4.3.1).

MultiJava, like Java, does not generate getters and setters for fields, which means that we don't need to treat accessor methods any differently than other methods.

The following example illustrates MultiJava programming practice and shows how method definitions are grouped together into method families by mjc.

```
public class Species {
  public void encounter (Species s) {}
}

public class Bunny extends Species {
  public void encounter (Species@Lion l) { System.out.println("run away"); }
```

```
  public void encounter (Species@Bunny b) { System.out.println("mate"); }
}

public class Lion extends Species {
  public void encounter (Species@Bunny b) { System.out.println("eat"); }
  public void encounter (Species@Lion l) { System.out.println("fight"); }
  public void encounter (Bunny b) { System.out.println("guzzle"); } // unsafe!
}

public static class Main {
  public static void main() {
    Species s1 = new Bunny();
    Species s2 = new Bunny();
    Species s3 = new Lion();
    Species s4 = new Lion();

    s1.encounter(s2);  // mate
    s1.encounter(s3);  // run away
    s3.encounter(s1);  // eat
    s3.encounter(s4);  // fight
  }
}
```

The methods defined above are recorded in intermediate format as shown below.

```
'Species/encounter(Species,Species)' => {
  'static types' => [ 'java.lang.Object', 'Species' ],
  'specialisers' => [
    [ 'Species', 'Species' ],
    [ 'Bunny', 'Lion' ],
    [ 'Bunny', 'Bunny' ],
    [ 'Lion', 'Bunny' ],
    [ 'Lion', 'Lion' ]
  ]
}

'Main/main(Main)' => {
  'static types' => [ 'Main' ],
  'specialisers' => [ ['Main'] ]
}

'Lion/encounter(Lion,Bunny)' => {
```

```
  'static types' => [ 'java.lang.Object', 'Bunny'],
  'specialisers' => [ ['Lion', 'Bunny'] ]
}
```

The above example shows how the encounter binary methods are grouped together into a single generic function and the parameter specialisers (specified after the @ keyword) are recorded to two-element arrays. Note that the first formal parameter to a method in (Multi)Java is implicitly the class enclosing the method's definition — our language-independent format makes this explicit.

The static main method is being assigned to a generic function with a single concrete method and thus a Degree of Dispatch of 1. The method's vector of specialisers is identical to the generic function's vector of static types, leading to a Degree of Specialisation of 0 for the concrete method. For having a single unspecialised concrete method, the generic function incurs a Degree of Dispatch of 0, which effectively means that associated method calls are not dynamically dispatched.

The third encounter method defined in the Lion class uses standard Java method overloading semantics to *statically* overload encounter in the Species class. This practice is unsafe, as it violates the contravariant typechecking rule for functions (§ 2.1.2). MultiJava's ability to statically overload methods, due to being backward-compatible with standard Java, prevents the above method from being part of the same generic function with methods defined using multimethod syntax. We note that this behaviour is different from other languages we study: where Dylan, Cecil and Diesel multimethods subsume statically overloaded functions, in MultiJava (and Nice) there are two distinct features programmers can use in parallel. The MultiJava compiler does however warn that statically overloaded methods can be rewritten as multimethods.

4.5 Summary

This Chapter describes the analysis methodology at the core of our study. The approach we adopt is one of corpus analysis, an approach that focuses on assessing the usage of language features by analysing a corpus of existing software. Our corpus includes nine programs, written in six different languages; this corpus limits the generalisability of our analysis by consisting mostly of compilers.

For the analysis we developed a set of tools which we use to measure the programs in our corpus in two phases: language-specific front-ends gather raw data from each program and store it in an intermediate format. The Multiple Dispatch Corpus Analysis Tool then measures the data obtained in the first phase

using the metrics introduced in Section 3.3. The results from these measurements are stored and visualised in ways that facilitate their comparison and discussion.

Chapter 5

Results

In this chapter we present the results we obtain from the metrics we introduced in Chapter 3 using the methodology described in Chapter 4 to the nine applications which make up our corpus.

The presentation is structured into three sections: Ratios (§ 5.1) summarises the results from the basic Dispatch Ratio (DR) and Choice Ratio (CR) metrics; Specialisation (§ 5.2), presents the results we obtain from the Degree of Specialisation (DoS, DoS_G) and Rightmost Specialiser (RS, RS_G) metrics; and Dispatch (§ 5.3), which covers the Degree of Dispatch (DoD) and Rightmost Dispatch (RD) metrics.

The result averages for the metrics we use are summarised in Figure 5.1. Figure A.1 of the appendix presents all values we measured across the corpus in a single table.

The final section (§ 5.4) explores the hypothesis that some of our results might be distributed according to power laws, thus making these distributions scale free. We investigate the possibility and explain the implications.

5.1 Ratios

In this Section we present the results from the basic Dispatch Ratio (DR) and Choice Ratio (CR) metrics, which measure basic relations between generic functions and concrete methods.

	Gwydion	OpenDylan	CMUCL	SBCL	McCLIM	Vortex	Whirlwind	NiceC	LocStack
DR_{ave}	1.74	2.51	2.03	2.37	2.32	2.33	2.07	1.36	1.50
CR_{ave}	18.27	43.84	6.34	26.57	15.43	63.30	31.65	3.46	8.92
DOS_{ave}	2.14	1.23	1.17	1.11	1.17	1.06	0.71	0.33	1.02
$DOS_{G,ave}$	2.23	1.19	1.11	1.10	1.05	0.85	0.35	0.16	0.85
RS_{ave}	2.24	1.34	1.24	1.23	1.39	1.10	0.78	0.34	1.08
$RS_{G,ave}$	2.37	1.29	1.17	1.29	1.26	0.88	0.40	0.16	0.85
DoD_{ave}	0.20	0.39	0.42	0.42	0.45	0.36	0.32	0.15	0.08
RD_{ave}	0.24	0.48	0.45	0.45	0.54	0.41	0.37	0.15	0.11

Figure 5.1: Metrics: averages across applications

5.1.1 Dispatch Ratio

The Dispatch Ratio distribution for all applications is shown in Figure 5.2. All applications follow a similar distribution with 58–93% of generic functions having a single concrete method. The shares for generic functions with two (2–24%), three (1–8%), and more methods decrease rapidly.

The generic function with the most concrete methods (print_string(a)) has 825 definitions in Vortex. This concrete method shows up among the top polymorphic functions in several other applications (e.g. Whirlwind: 462, SBCL: 136, OpenDylan: 128, LocStack: 61). Other generic functions with a high degree of polymorphism include binary operators such as the equality test for two objects (Vortex: 257 definitions, Whirlwind: 213, OpenDylan: 58, Gwydion:42), 'less than' and addition operators.

The DR_{ave} values for the applications in our corpus are shown in Figure 5.1. Six of the applications have a DR_{ave} measurement of at least 2, indicating that for every generic function, on average a dispatch decision must be made between two concrete functions.

5.1.2 Choice Ratio

The results for CR_{ave} in Figure 5.1 show considerable variance. On average, any concrete method in Vortex is part of a dispatch decision with 60 or so other methods, whereas for the NiceC it would be only with 3.5 other methods. This is somewhat surprising for NiceC, which is the only language model we study with a concept of overridable method declarations that allows the same concrete method to be part of multiple generic functions. In reality we found only two Nice generic functions that take advantage of this feature. We discuss Nice's method overriding concept and show an example in Section 4.4.5.

Figure 5.3 shows the Choice Ratio distribution for each application separately. The first thing to note is that most concrete methods (25 – 63%) have a Choice Ratio of 1, indicating that each is the sole method in a monomorphic generic function.

We note further that a relatively high proportion of concrete methods have choice ratios of nine and higher. This factor can be attributed to some generic functions with a high degree of polymorphism, that is, a large number of concrete methods. However rare these *generic functions* are, the *concrete methods* they contain are not as rare, as they occur in large numbers. For example, if a single generic function has a DR of 100, then there will be at least 100 concrete methods with CR ≥ 100. The strength of the Choice Ratio metric is the ability to expose this

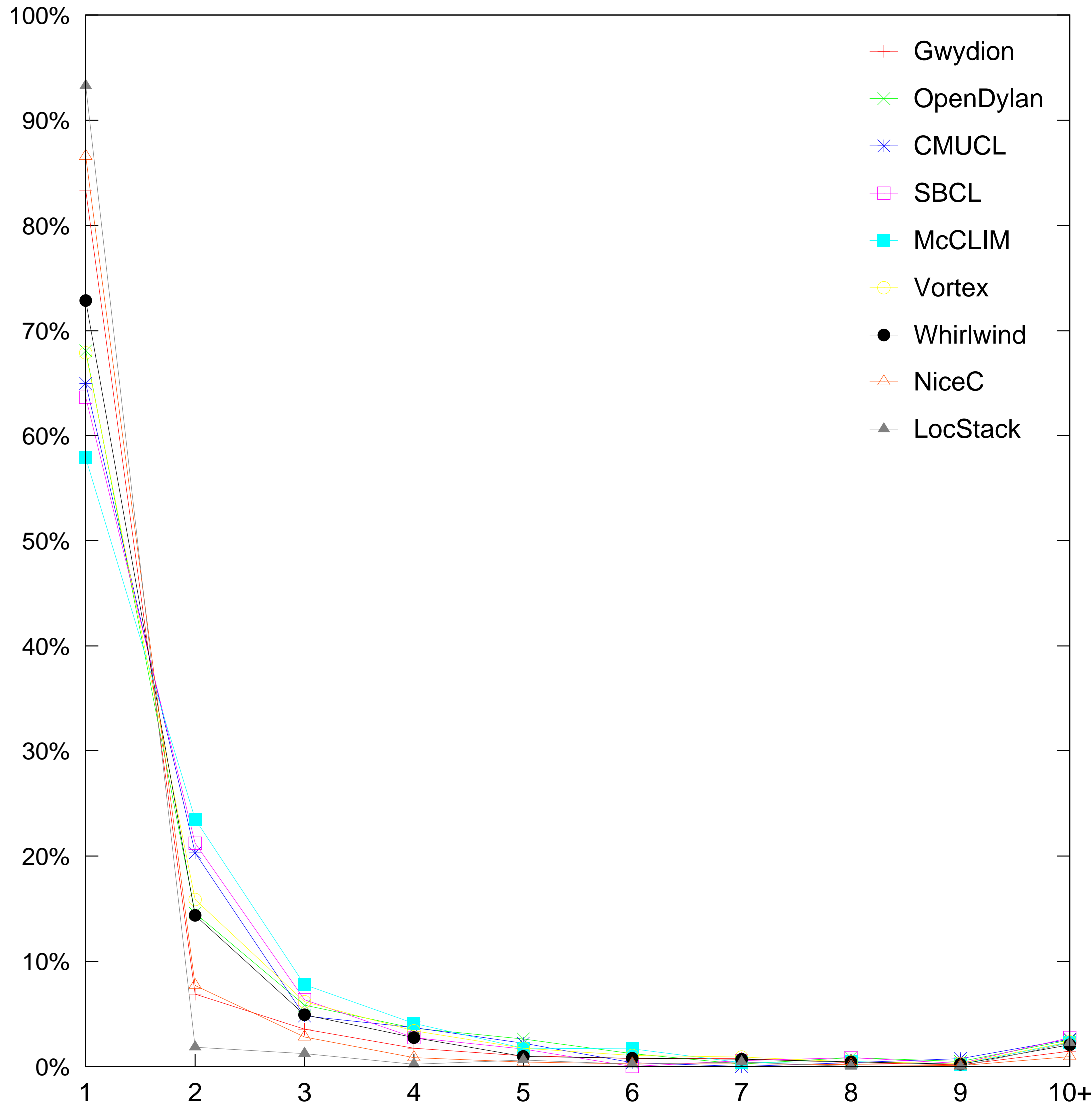

Figure 5.2: Dispatch Ratio (DR) frequency distribution, expressed as a percentage of all generic functions with a given DR measurement.

correlation.

5.2 Specialisation

This section presents the results from the Degree of Specialisation metrics, measured per concrete method (DOS) and per generic function (DOS_G).

5.2.1 Specialisation of Concrete Methods

Figure 5.4a shows, for each application, what proportion of concrete methods have a given DOS measurement. At the top are the highest DOS values mea-

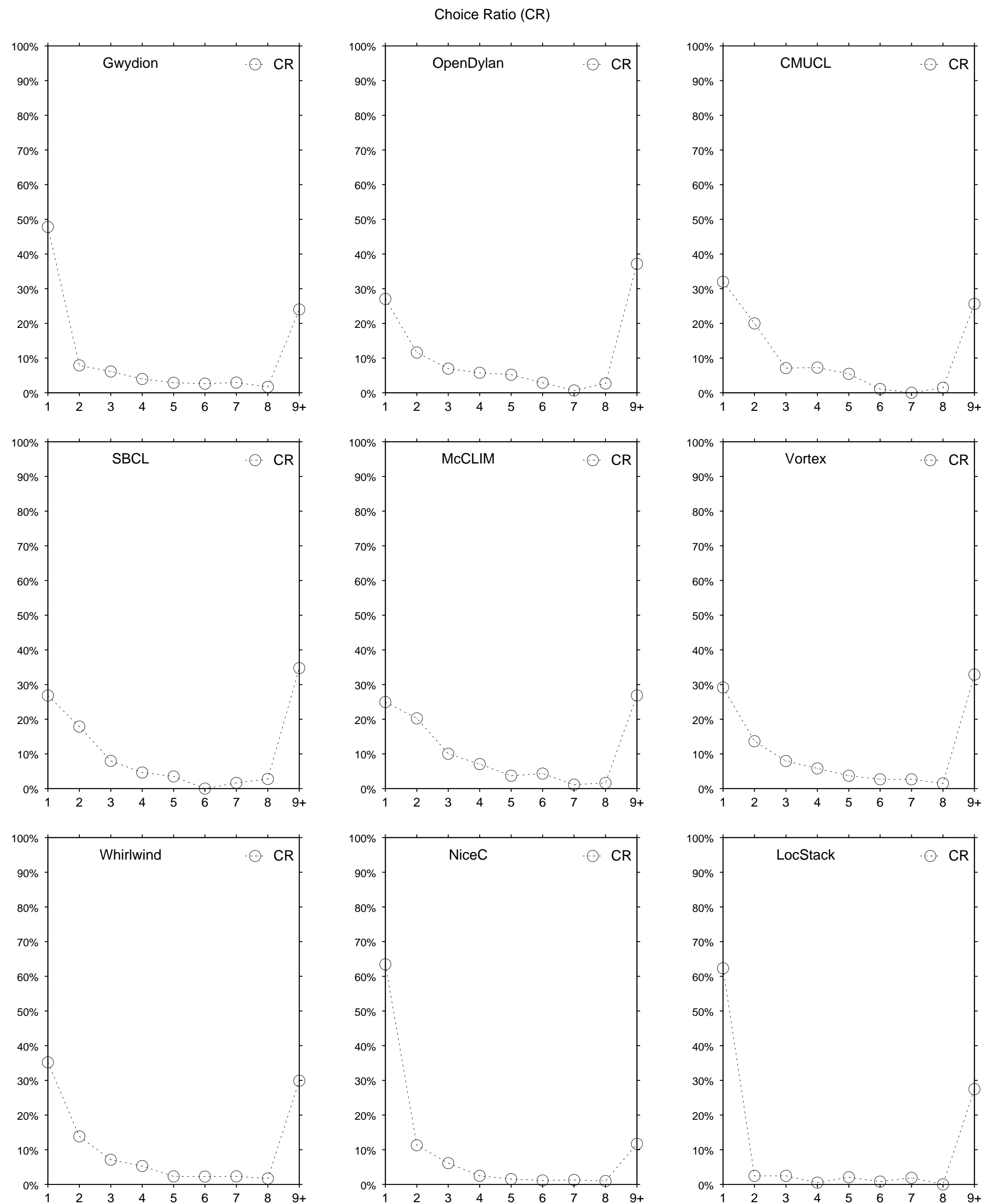

Figure 5.3: Choice Ratio frequency distribution, expressed as a percentage of all concrete methods with a given CRmeasurement.

sured for the respective application. While it is quite common for methods to specialise up to three parameters, we found generic functions that specialise seven (OpenDylan, make−source−location), eight (Whirlwind, resolve8) and 20 (Gwydion, parser:production_113) parameters. The range of proportions of generic functions with no specialisation varies considerably across the applications.

The results for the RS metric are shown in Figure 5.4b. Comparing RS to DOS in Figure 5.5, we see that some applications (OpenDylan, SBCL, McCLIM) have significantly higher values for Rightmost Specialiser RS than they do for Degree of Specialisation DOS. This means that some methods' parameter list must have some non-specialised parameters "to the left of" specialised parameters. Since these methods' parameters are not ordered based on their specialisation, we hypothesise that programmers make use of the greater flexibility in ordering parameters to enhance code readability. In other applications (Gwydion, CMUCL, Vortex, NiceC), RS and DOS values seem to mostly equal each other. This reflects the fact that for most methods, specialised parameters come first and are then followed by unspecialised parameters. The RS distribution for McCLIM shows the most pronounced deviation from the DOS distribution, as we measure 15% less methods with RS $= 1$ than with DOS $= 1$. For RS $= 2$ the relation is inverted, 12% more methods having RS $= 2$ than DOS $= 2$.

Figure 5.6 shows scatter plots that account for the relation between DOS and RS for each concrete method. It reveals certain differences between applications: While the values for RS in CMUCL, SBCL and LocStack only exceed DOS values by 1, at least one concrete method in McCLIM has DOS $= 1$ and RS $= 6$, meaning that the method only specialises its sixth parameter. We can find a similar case for Whirlwind, with a method that specialises its seventh parameter only.

5.2.2 Specialisation of Generic Functions

The Degree of Specialisation and Rightmost Specialiser distributions of generic functions (DOS_G and RS_G) show similar pictures as their per concrete method siblings (Figure 5.7).

5.3 Dispatch

In this section we first present the results from the Degree of Dispatch (DOD) and Rightmost Dispatch (RD) metrics. We then compare DOD with Dispatch Ratio

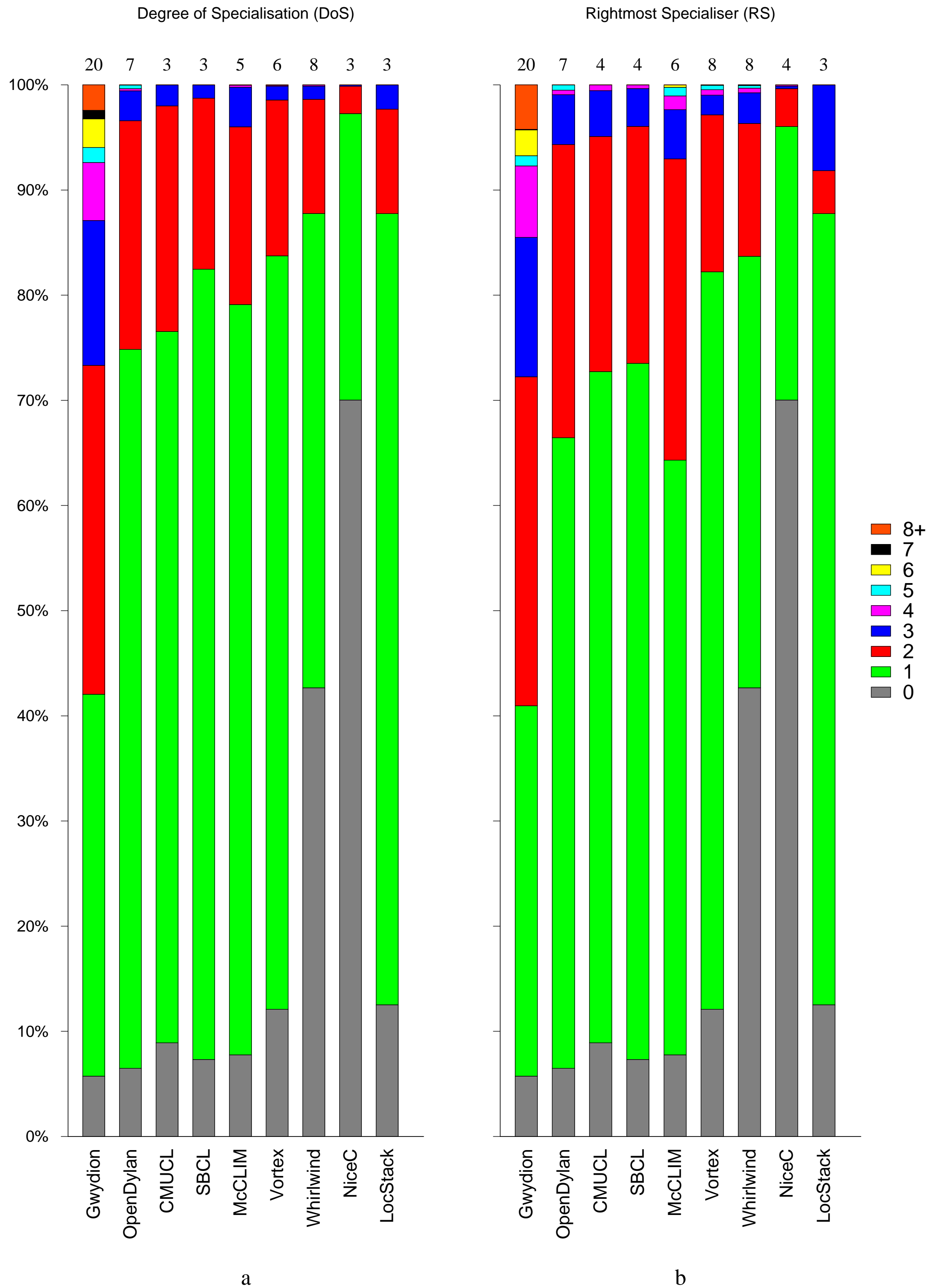

Figure 5.4: a: Degree of Specialisation (DoS), and b: Rightmost Specialiser (RS) frequency distributions of concrete methods across applications. The lowest block in each stack (grey) is the proportion with DoS/RS measurement of 0, the next (green) is the proportion with 1, and so on. The value at the top of each stack indicates the highest DoS/RS measured for this application.

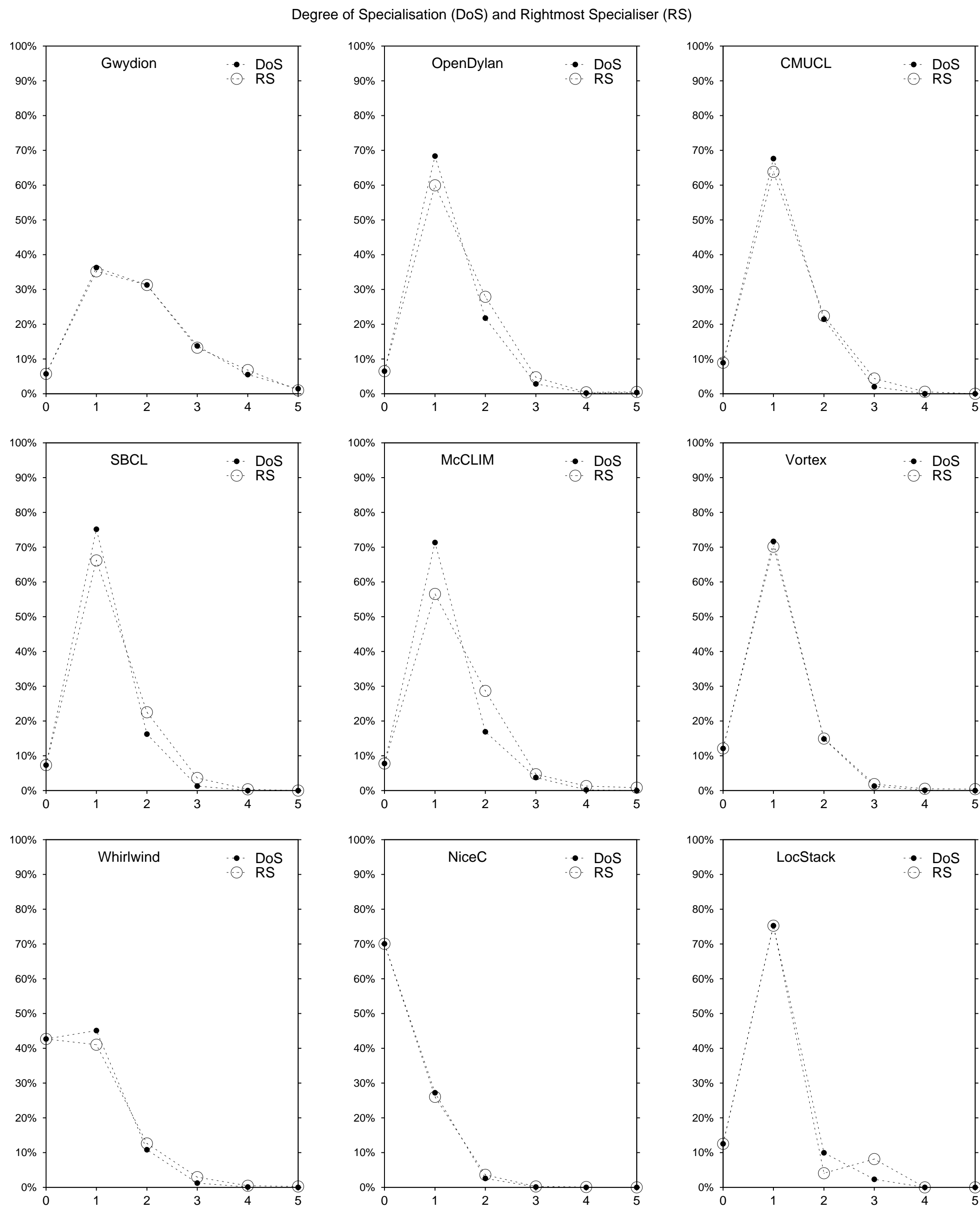

Figure 5.5: Degree of Specialisation (DOS) and Rightmost Dispatch (RS) distributions compared for $0 \leq \text{DOS}|\text{RS} \leq 5$.

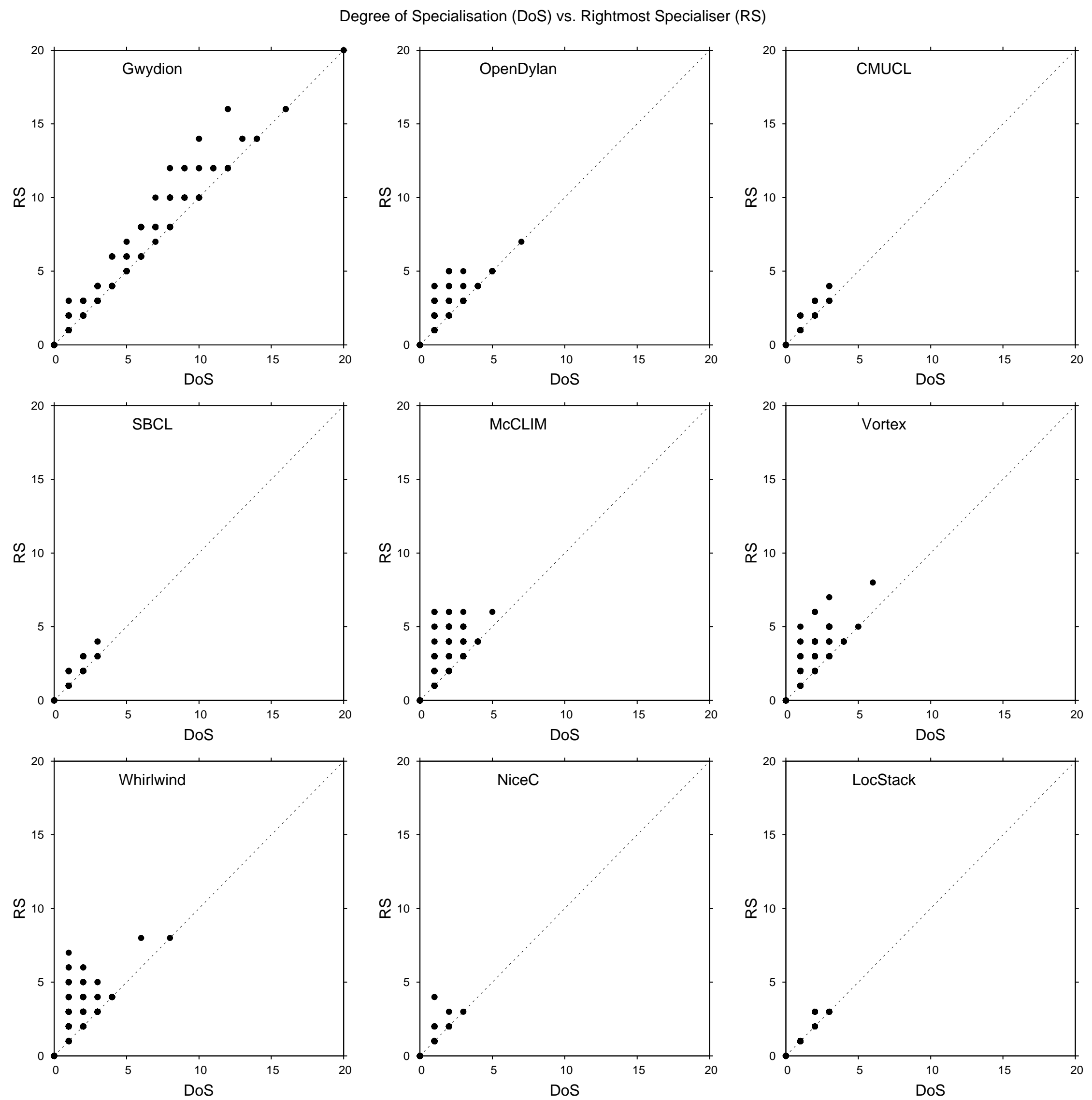

Figure 5.6: Degree of Specialisation (DoS) values plotted against Rightmost Specialiser (RS) values for all concrete methods in each application.

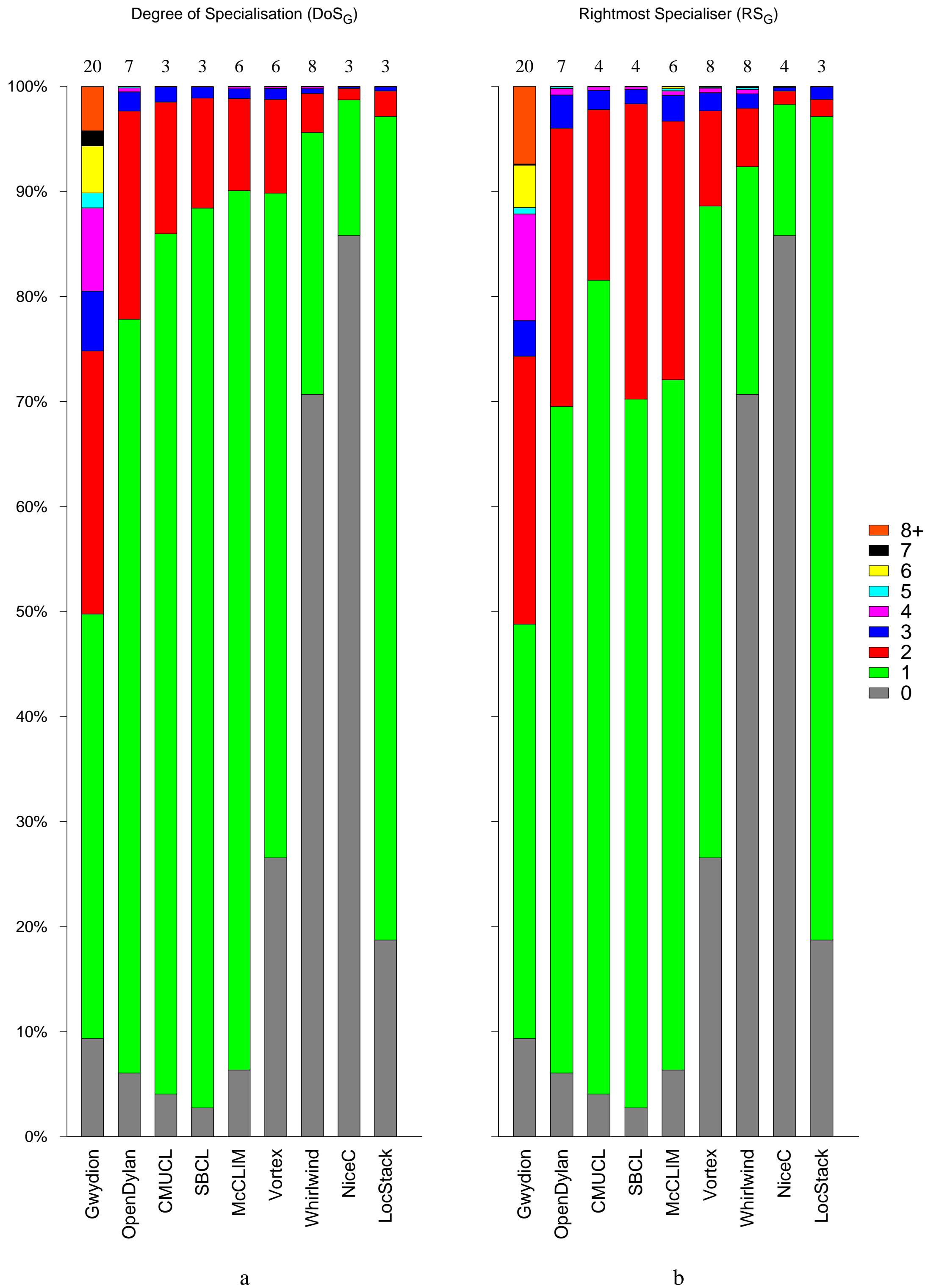

Figure 5.7: a: Degree of Specialisation ($\mathrm{DoS_G}$), and b: Rightmost Specialiser ($\mathrm{RS_G}$) frequency distributions of generic functions across applications. The lowest block in each stack (grey) is the proportion with $\mathrm{DoS_G}/\mathrm{RS_G}$ measurement of 0, the next block (green) is the proportion with 1, and so on. The value at the top of each stack indicates the highest $\mathrm{DoS_G}/\mathrm{RS_G}$ measured for this application.

and Degree of Specialisation values and demonstrate their relationships.

5.3.1 Degree of Dispatch

Figure 5.8a shows the Degree of Dispatch (DOD). Excluding NiceC and LocStack, most applications have similar levels (2.7–6.5%) of multiple dispatch (DOD $>$ 1), and single dispatch (14–32%). NiceC has the lowest proportion of multiple dispatch (1.0%) among the analysed applications, even though we have excluded that part of the source written in Java. LocStack closely follows with 1.4%.

Across the entire corpus, the share of generic functions that are not required to dispatch dynamically ranges from 62% to 93%; this corresponds nicely with the proportions of generic functions having a single concrete method and thus a Dispatch Ratio of 1.

On average, across all measured applications, we found that around 4% of generic functions utilise multiple dispatch (DOD > 1) and around 22% utilise single dispatch (DOD $= 1$).

Figure 5.8b shows the Rightmost Dispatched parameter (RD). This generally follows DOD, although the proportions are often a little higher for RD ≥ 2 (see Figure 5.9). This shows that a significant number of single-dispatched generic functions have their dispatch decision made on the second or beyond argument supplied in the call.

We repeat a portion of Figure 5.1 below to show the averages of the Degree of Dispatch and Rightmost Dispatch metrics. As can be seen, RD is generally a little larger than the Degree of Dispatch (DOD): RD $\geq$ DOD by definition (because dispatch must occur on the RD'th argument, but there could be arguments to the left of it that do not dispatch). RS is higher than DOS for the same reason.

	Gwydion	OpenDylan	CMUCL	SBCL	McCLIM	Vortex	Whirlwind	NiceC	LocStack
DOD_{ave}	0.20	0.39	0.42	0.42	0.45	0.36	0.32	0.15	0.08
RD_{ave}	0.24	0.48	0.45	0.45	0.54	0.41	0.37	0.15	0.11

The scatter plots in Figure 5.10 show the relation between DOD and RD measurements for all generic functions. These plots clearly demonstrate the value of the RD metric, as it is able to capture conditions when functions dispatch on a single parameter found as far right as the sixth (McCLIM) or seventh (Vortex, Whirlwind) position in the functions formal parameter list. Such generic functions, although they are technically single dispatch, cannot be implemented as is in a single dispatch language, because in single dispatch languages functions only dispatch upon the *first* argument in a call.

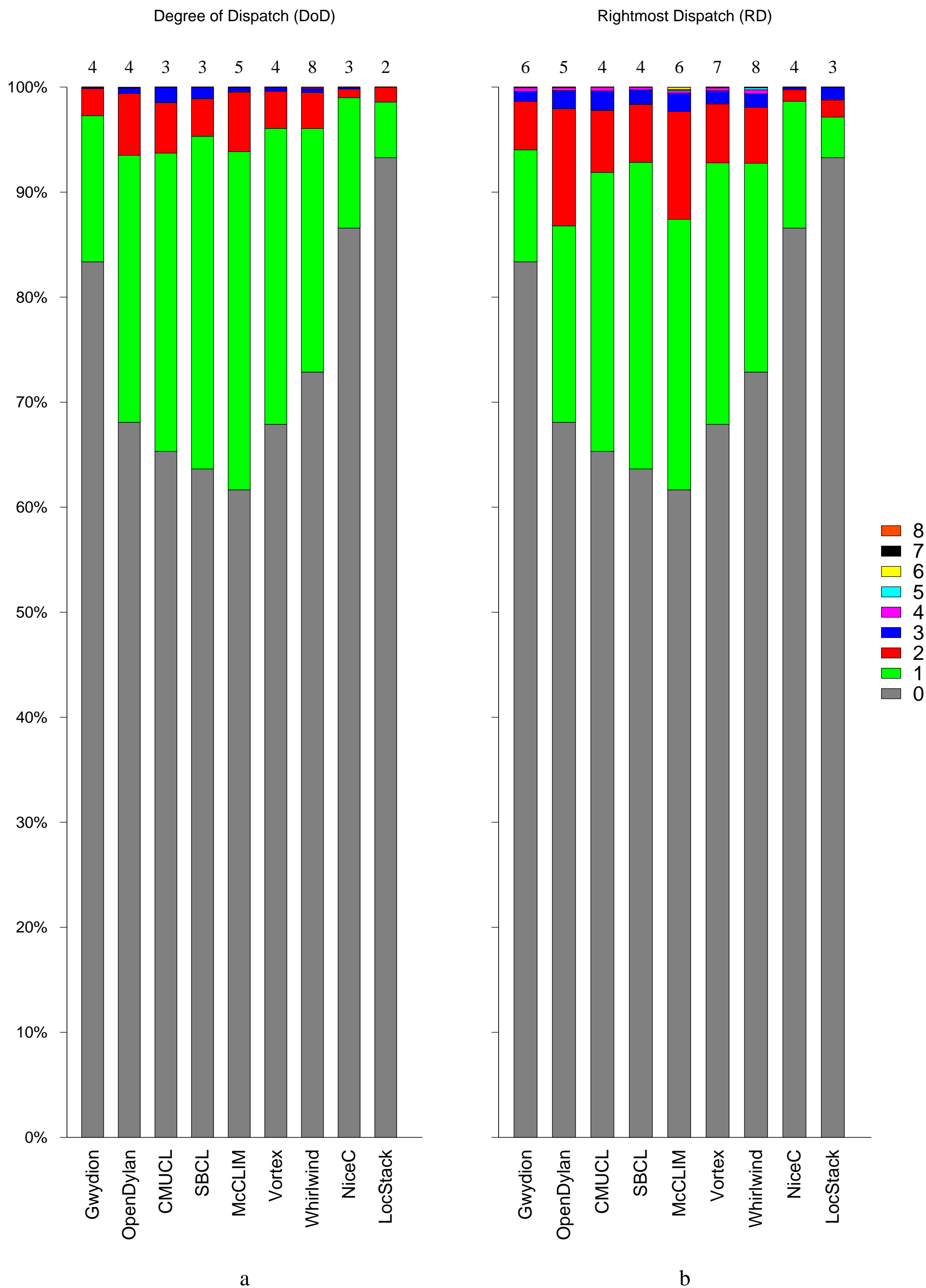

Figure 5.8: a: Degree of Dispatch (DoD), and b: Rightmost Dispatch (RD) frequency distributions of generic functions across applications. The value at the top of each stack indicates the highest DoD/RD measured for this application.

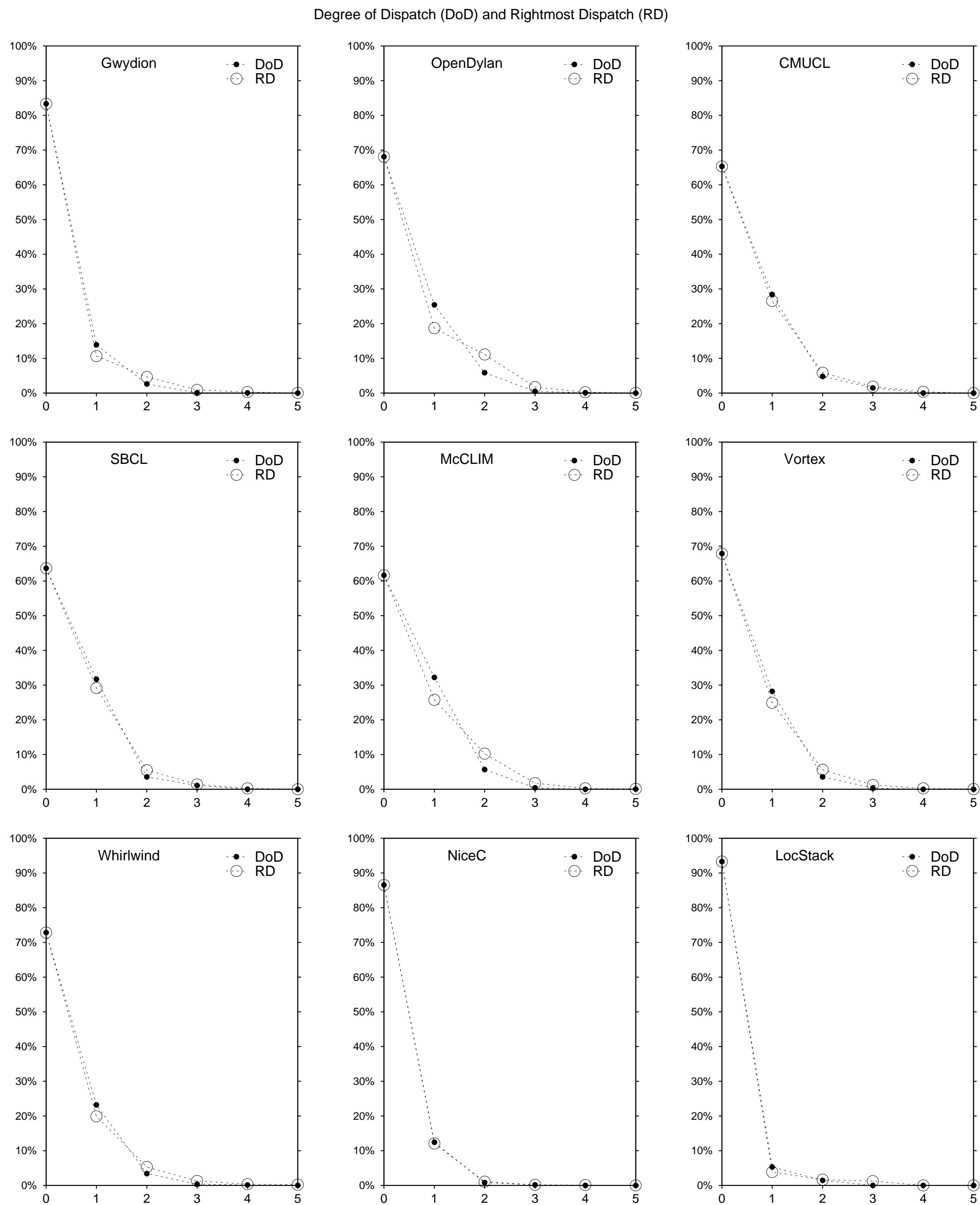

Figure 5.9: Degree of Dispatch (DoD) and Rightmost Dispatch (RD) distributions compared for $0 \leq \text{DoD}|\text{RD} \leq 5$.

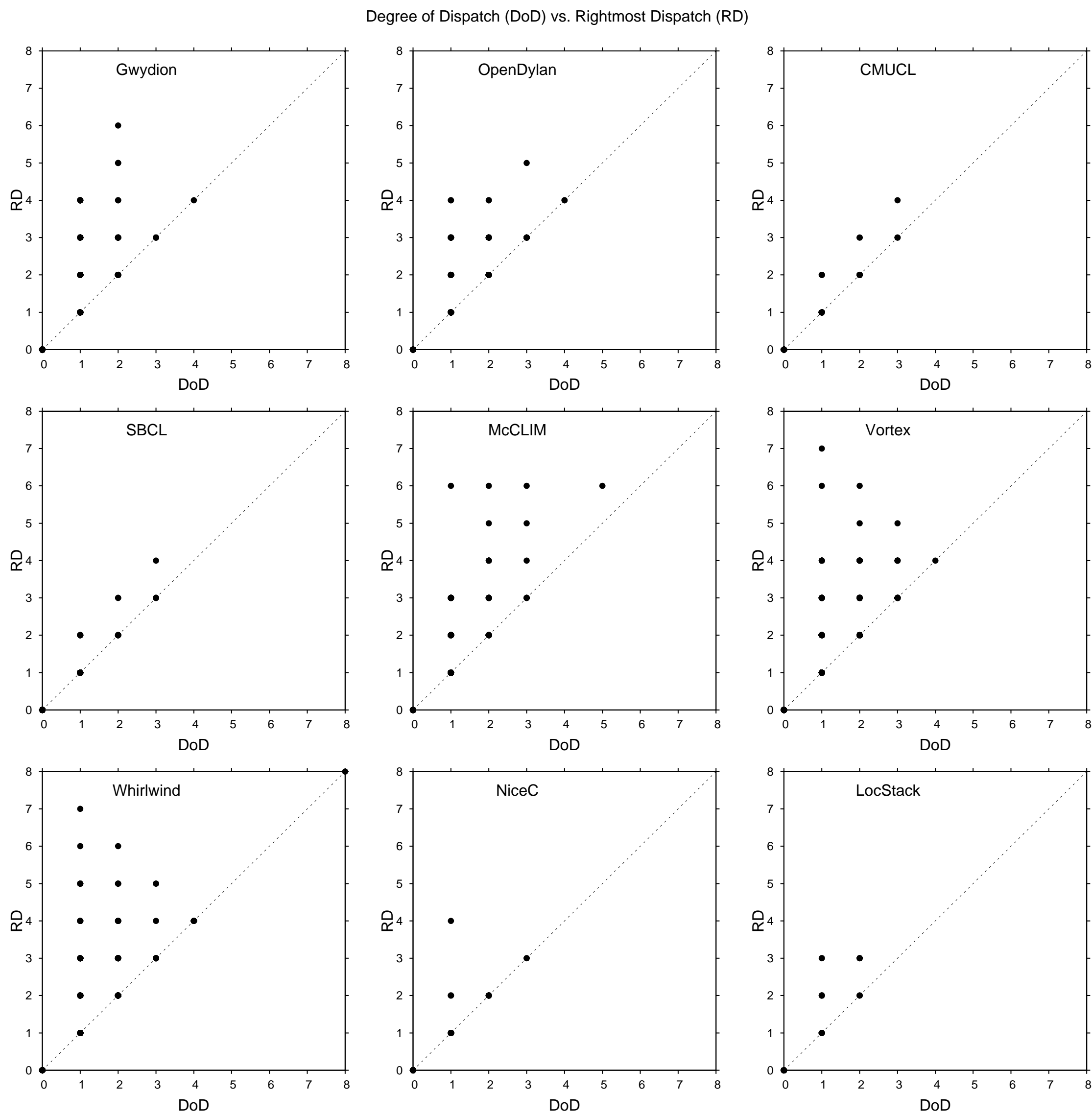

Figure 5.10: Degree of Dispatch (DoD) values plotted against Rightmost Dispatch (RD) values for all generic functions in each application.

5.3.2 Dispatch Ratio and Degree of Dispatch

The proportion of generic functions with DoD $= 0$ exactly matches the proportion of generic functions with DR $= 1$ for seven out of the nine applications (see leftmost data points in Figure 5.11). A monomorphic function will never need to dispatch dynamically (its DoD is always 0) because there are no alternative implementations to choose from. We infer that the set of monomorphic generic functions is contained in the set of non-dispatched generic functions (these have DR $= 1$).

Conversely, all polymorphic generic functions will have a Degree of Dispatch of at least 1. This seems logical, since most of the languages we analyse will not allow two concrete methods of the same generic function to have the same set of specialisers; in other words, two concrete methods have to specialise at least one parameter position differently, to allow the dispatch mechanism of the language to choose the *single*, most applicable method from the two given definitions.

The exception here is Common Lisp, where it is perfectly legal to have several most applicable methods, given that these have different qualifiers (none, one or more of BEFORE, AFTER, AROUND). Common Lisp qualifiers prescribe how methods can be combined together to one runnable entity. CMUCL and McCLIM make some use of this method combination capability, which explains why the proportion of functions with DoD $= 0$ is slightly lower than the proportion of monomorphic generic functions (DR $= 1$): a relatively small number of generic functions have DoD $= 0$ even though they have multiple concrete methods. These methods' specialiser vectors are identical, but their qualifiers differ — which technically means that they are not designed as alternatives to each other. Rather, these methods are designed to be combined to a single runnable entity — meaning that there is no need for the dispatch mechanism to select one most-applicable code body from a set of alternatives. CMUCL has one generic function with this property (representing 0.2% of total), and McCLIM has 72 (3.2%).

For Dispatch Ratios higher than 1 and Degrees of Dispatch higher than 2, the correlation is less obvious, with the share for DR $= 2$ being consistently and considerably lower than the share for DoD $= 1$ (as can be seen from Figure A.1: for example, Gwydion has roughly 7% of generic functions with a DR of 2 while double that proportion have DoD $= 1$).

The above relationship is reversed for DR > 3 and DoD > 2, from which point on the Degree of Dispatch stays consistently below the Dispatch Ratio (see Figure 5.11). In other words, the Degree of Dispatch distributions consistently lean to the left of the Dispatch Ratio distributions; also, the "tail" of the Dispatch Ratio distribution curve is much longer (which Figure 5.11 doesn't show). The

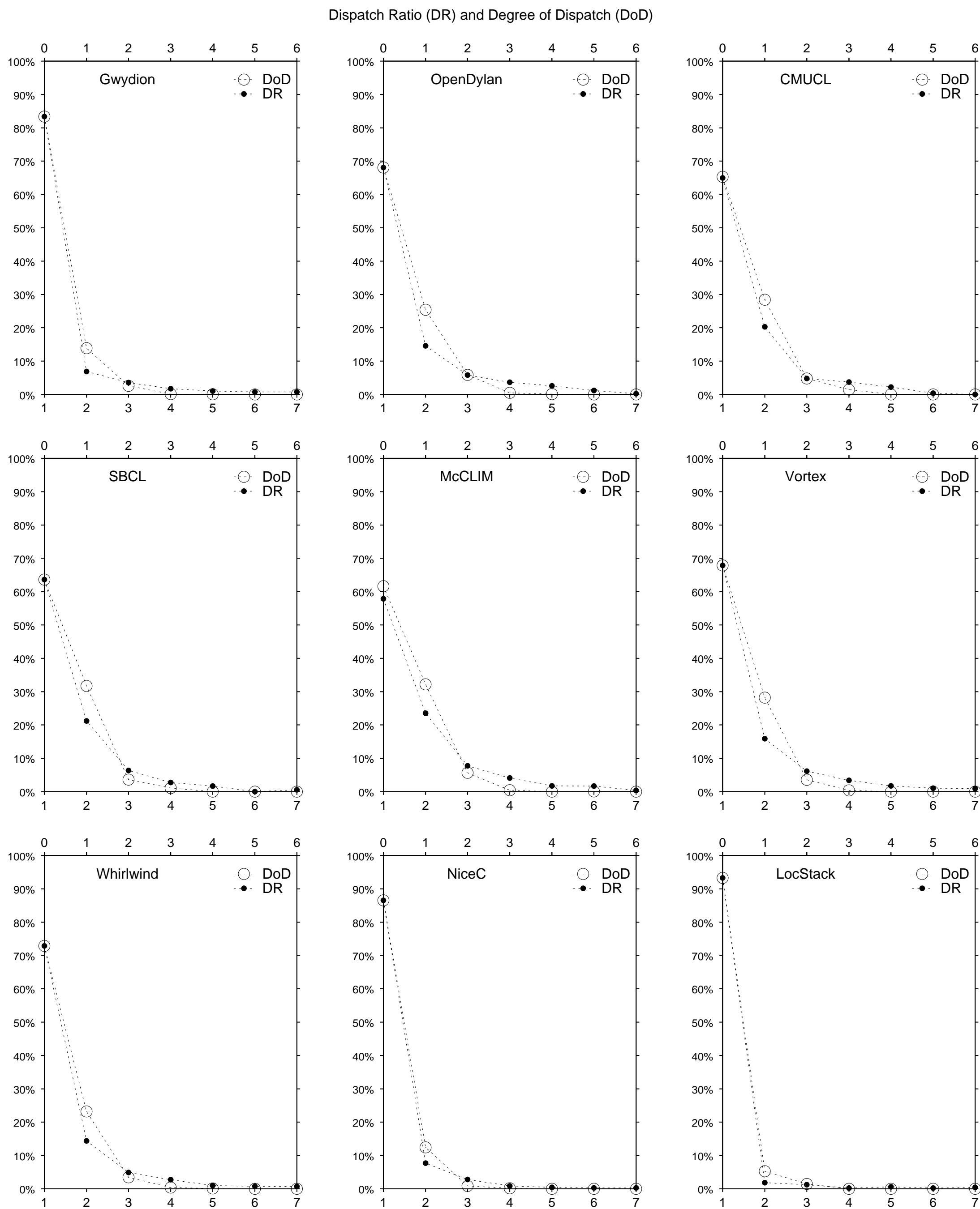

Figure 5.11: Degree of Dispatch (DoD) and Dispatch Ratio (DR) distributions compared, with DR vertically aligned with DoD $=$ DR $- 1$ to reflect the fact that monomorphic generic functions do not dispatch dynamically.

explanation for this phenomenon is simple: although we found generic functions with a large number of concrete methods (i.e a DR as large as many hundreds, see "Max" column in Figure A.1), the corresponding DOD is much lower (the highest DOD measured across the whole corpus is 8). This shows to some extent that two guidelines for good program design are being followed by a) using polymorphic functions to confine the complexity of a program and b) keeping a function's list of parameters short to prevent losing their count[1].

Figure 5.12 shows the relation between DR and DOD metrics from a different perspective. The dots represent the intersection between DR and DOD values for each generic function across the corpus, on a per-application basis. The resulting plots reflects the fact that, although both of these metrics measure generic functions, they do so along two different dimensions: DR measures the degree of polymorphism, expressed by the number concrete methods in a generic function, while the DOD's upper boundary is set by the length of the list of parameters to that function.

5.3.3 Specialisation and Dispatch

Figure 5.13 compares DOS_{G} and DOD distributions. It shows that the dispatch metrics DOD and RD are generally below the specialiser metrics DOS_{G} and RS_{G}, because generic functions dispatch on specialised positions, but not all specialised positions will dispatch if all concrete methods specialise the same argument position in the same way. Indeed, this appears to be the case in Gwydion Dylan leading to the large values in Figure 5.7, such as a maximum 20 specialisers: many of these specialisers are common to all the methods in the generic function, and are in effect acting as static (non-dispatching) type declarations for those method arguments. At the other extreme range Whirlwind and NiceC, where it appears (Figure 5.13) that dispatch occurs on nearly each specialiser that is defined. In other words, if methods of a generic function specialise a parameter position, then they nearly always do it using different specialisers. This again hints at different programming styles. What is surprising is that Vortex (written in Cecil) rather resembles Gwydion Dylan in specialising many argument positions of a generic function in the same way, while Whirlwind (written in Diesel) strictly specialises only those parameter positions it needs to dispatch on.

[1] Alan Perlis pointedly addressed both these concerns in his *Epigrams on Programming*: "6. Symmetry is a complexity-reducing concept (co-routines include sub-routines); seek it everywhere." and "11. If you have a procedure with ten parameters, you probably missed some." (Perlis, 1982)

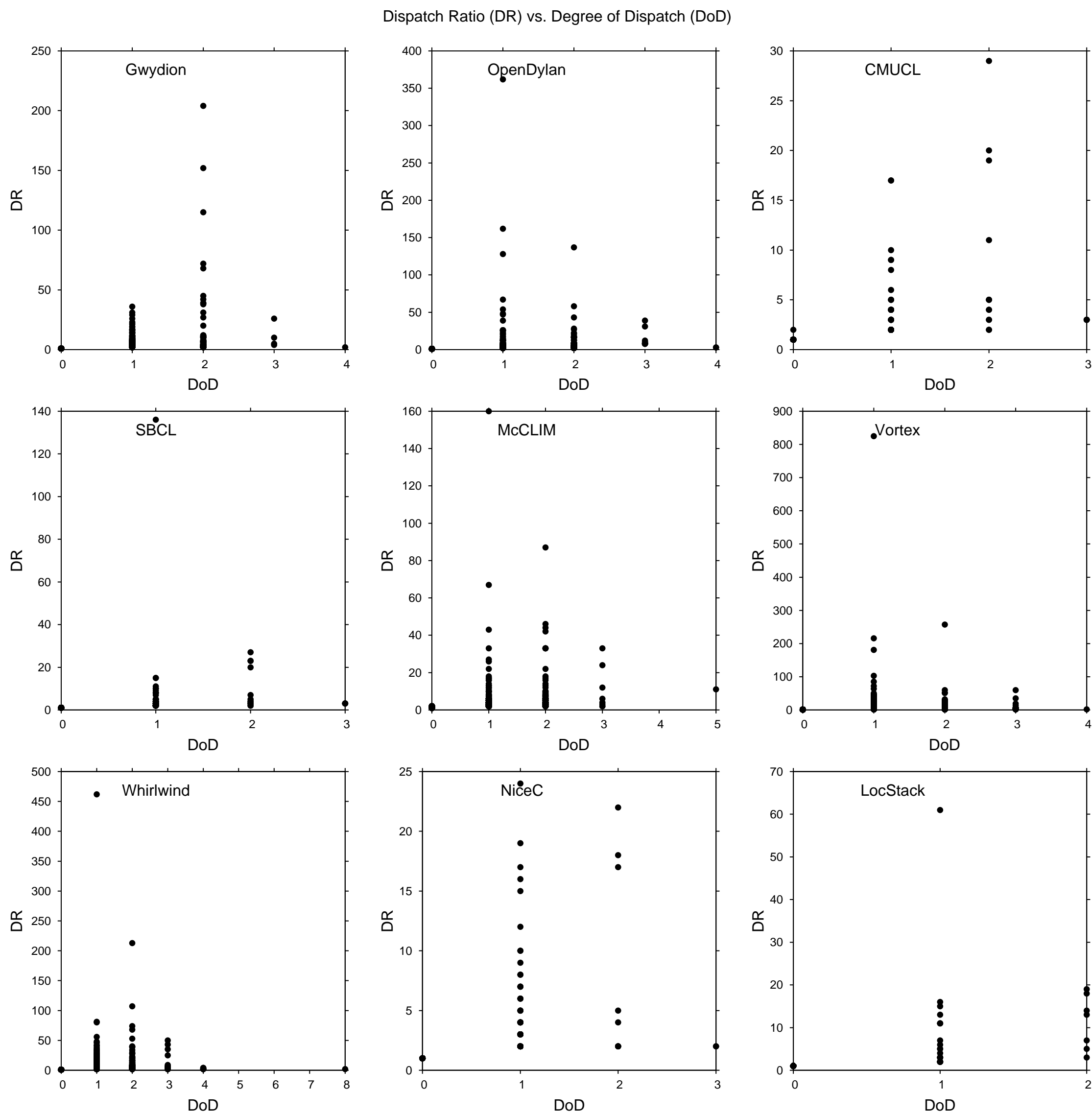

Figure 5.12: Degree of Dispatch (DoD) values plotted against Dispatch Ratio (DR) values for all generic functions in each application.

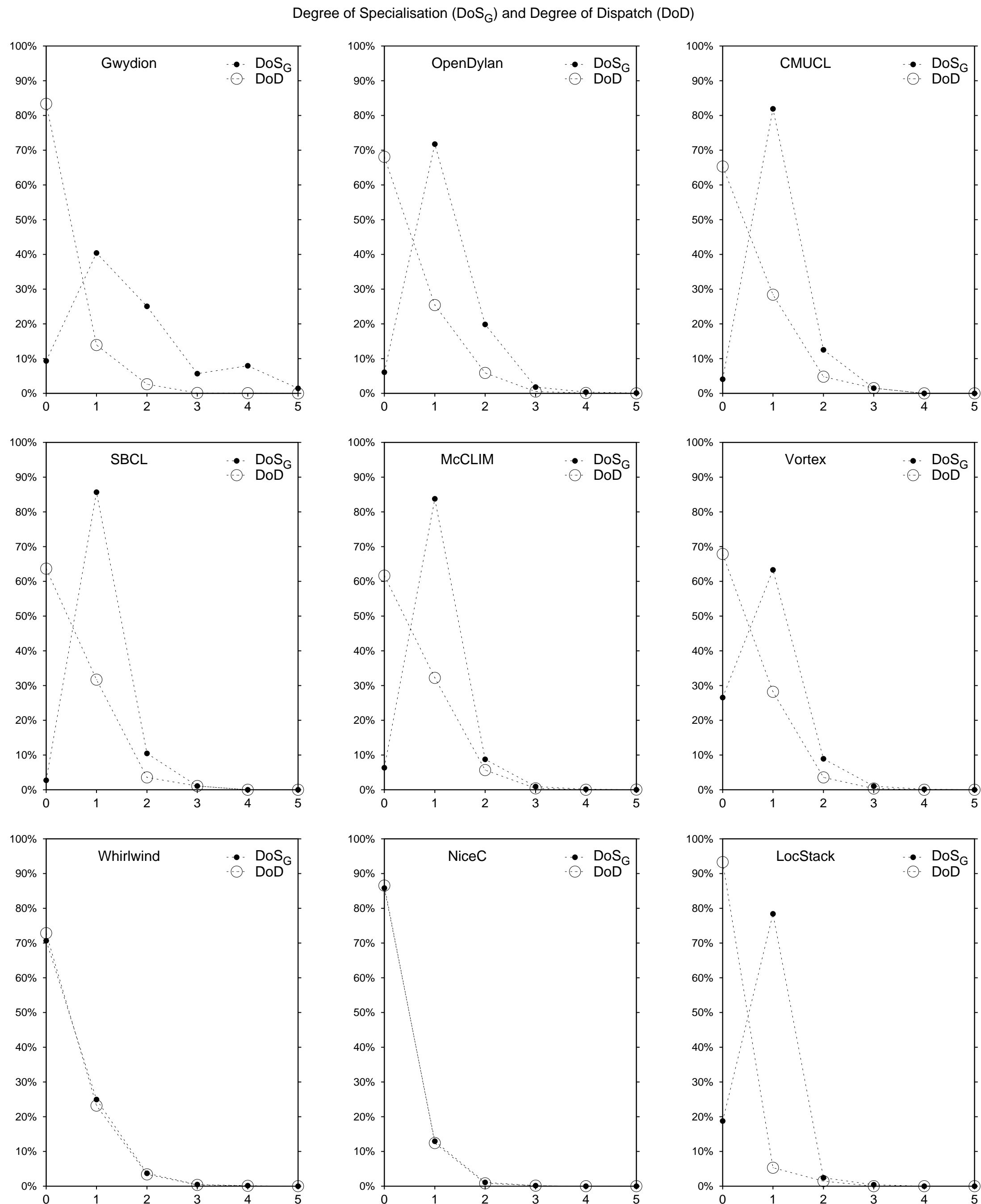

Figure 5.13: Degree of Specialisation of a generic function (DoS_G) and Degree of Dispatch (DoD) distributions compared for $0 \leq \text{DoS}_\text{G}|\text{DoD} \leq 5$.

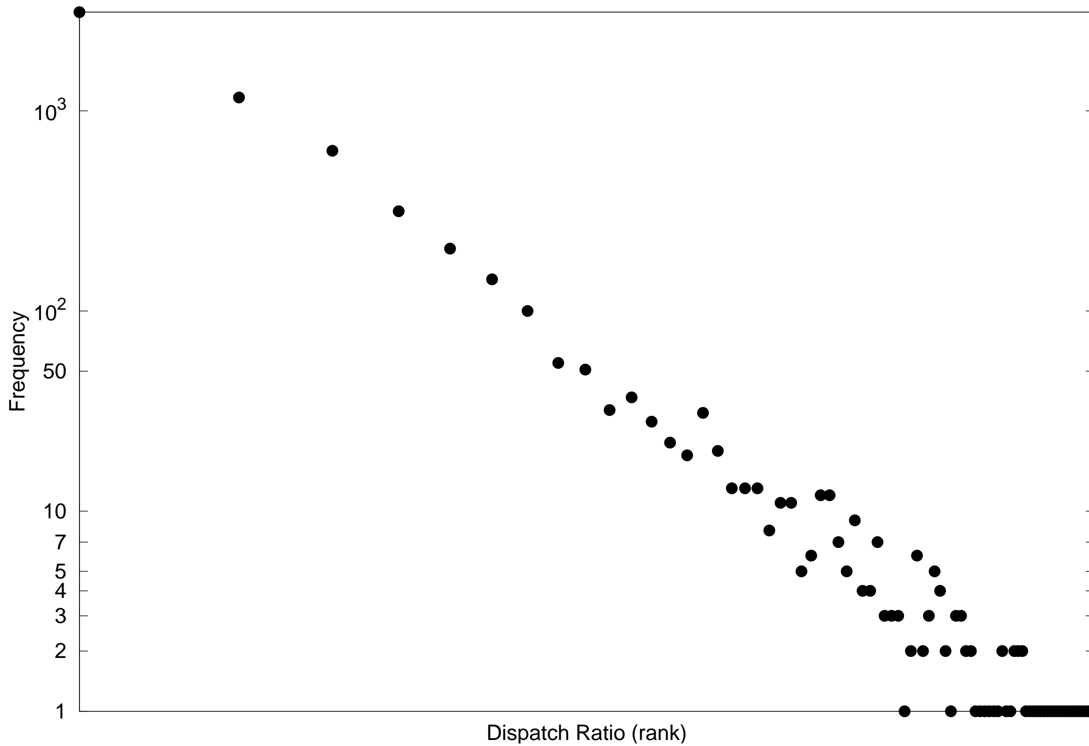

Figure 5.14: Dispatch Ratio (DR) distribution of all generic functions in the corpus, plotted on logarithmic scales.

5.4 Power Laws

Among related studies of corpus analysis, some have sought to provide evidence that many important relationships between software artifacts follow a power law distribution (Wheeldon and Counsell (2003); Potanin et al. (2005); Baxter et al. (2006)). The lines shown in Figure 5.2 for the Dispatch Ratio distribution are reminiscent of logarithmic curves. This observation prompts us to investigate the possibility of the Dispatch Ratio following a power law distribution.

As the curves seem fairly close to each other, we first plot all values on the same log-log scale (Figure 5.14). To reduce the noise caused by high DR values, which are, by their nature, very rare, the array of measured DR values is first transformed into a vector of consecutive ranks. The strong indication of a straight line is further evidence of the possibility that power laws are being followed.

Figure 5.15 shows separate logarithmic plots for each application's Dispatch Ratio distribution. While most distributions are indeed reminiscent of straight lines, there is considerable spread between how close the applications follow a power law: Gwydion, Vortex and Whirlwind fit the power law distribution quite closely, while NiceC and LocStack hardly follow a straight line. Baxter et al. (2006) came to similar results in their large-scale analysis of Java code. They surmise

that certain attributes of the application's code (such as its design or domain) may affect the resulting distribution.

All samples show a considerable spread for higher ranks, which reflects the fact that many generic functions with a high Dispatch Ratio have a low frequency of one, two or three. In other words, generic functions with a certain very high number of concrete methods (i.e. several hundred) rarely occur more than once in an application.

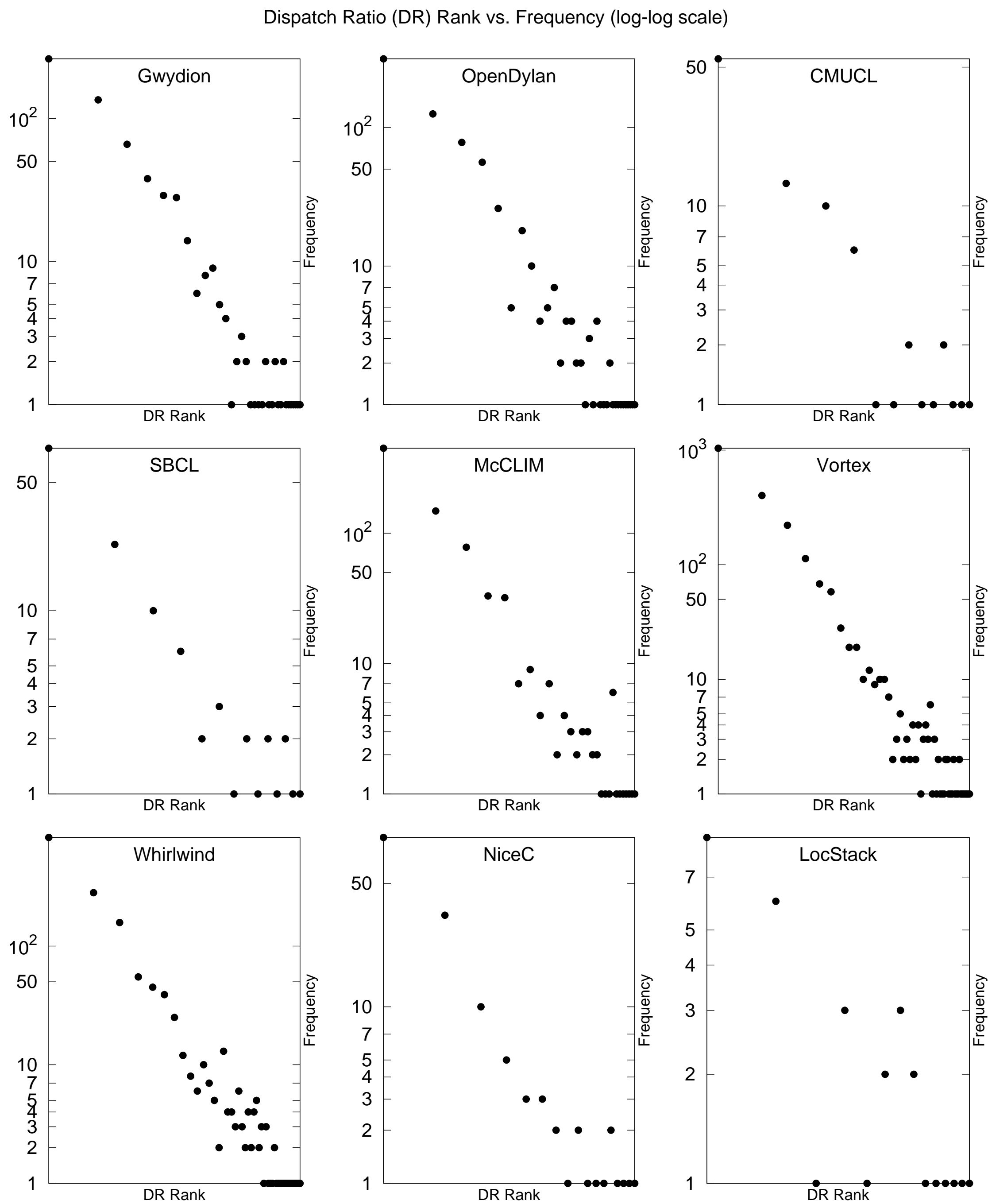

Figure 5.15: Dispatch Ratio (DR) distributions of each application, plotted on logarithmic scales.

Chapter 6

Conclusions

In this Chapter we summarise the contributions of our research, discuss our findings and look at potential directions for future work.

6.1 Contributions

This thesis presents an empirical study of multiple dispatch in object-oriented languages. To our knowledge it is the first cross-language corpus analysis of multiple dispatch. It is at the same time part of a larger research project that compares the use of multiple dispatch to the use of techniques that simulate multiple dispatch in Java (Muschevici et al., 2008). The main contributions of this study are a language-independent model for multiple dispatch; a suite of metrics to measure the use of multiple dispatch; and finally the results we obtain by measuring a corpus in six multiple dispatch languages, which together provide a quantitative assessment of the use of multiple dispatch in practice. The following three sections summarise these contributions.

6.1.1 A Language-independent Model for Multiple Dispatch

We have developed a model of multiple dispatch that unifies language-specific concepts to support reasoning about multiple dispatch in a language-independent way. The model (represented in Figure 3.1) defines five main entities: A *generic function* is a function that may be dynamically dispatched. Each generic function dispatches to one or more *concrete methods*. Concrete methods specify their applicability by defining *specialisers* for any of the function's formal parameters. A generic functions and its associated concrete methods are referred to by a *name* and share a *signature* defining the number of permissible arguments and, if applicable, the static types for each formal argument of that function.

6.1.2 A Metrics Suite for Measuring Multiple Dispatch

We defined six metrics to measure multiple dispatch: the Dispatch Ratio and Choice Ratio measure basic relations between generic functions and concrete methods. The Degree of Specialisation and Rightmost Specialiser both capture the use of formal parameter specialisers in methods and generic functions. The Degree of Dispatch and Rightmost Dispatch determine the number and position of formal parameters needed for a generic function to dispatch to a concrete method.

6.1.3 An Evaluation of the Use of Multiple Dispatch in Practice

We presented the values measured according to each metric for a corpus of nine programs written in six multiple dispatch languages: CLOS, Dylan, Cecil, Diesel, Nice and MultiJava, thus providing quantitative evidence of the use of multiple dispatch in practice.

In answer to our introductory question ***how much is multiple dispatch used?***, we found that around 4% of generic functions utilise multiple dispatch and around 22% utilise single dispatch. Determining how much these results generalise — i.e., how well these measurements represent the use of multiple dispatch in other applications and languages — necessarily requires further study, but we expect these results to provide a benchmark for comparison.

6.2 Discussion

There are a number of inferences which can be drawn from the results presented in Chapter 5. Perhaps the most obvious is that many of the metric values are low. Every language we measured had more than 55% monomorphic generic functions; less than 7% of functions dispatch on two or more arguments (Figure 5.8). This is reflected in Dispatch Ratio DR_{ave} values: no language had more than 2.5 concrete methods for each generic function; on average, a function dispatches only on 0.1 to 0.7 arguments (Figure 5.1, DOD_{ave} values).

6.2.1 Monomorphic vs Polymorphic Functions

It seems that in all of our studied languages, there will be many generic functions that do not dispatch: static methods, constructors, but also auxiliary methods, methods that provide default argument values in languages without variable argument lists or keyword arguments. On the other hand, there will be a significant number of generic functions that do dispatch to three or more different concrete

methods — and the methods belonging to those functions make up a substantial fraction of the program's *methods*.

The Template Method pattern (Gamma et al., 1994), for example, will contribute to this effect, as only "hook methods" should be overridden in subclasses, while methods providing abstract, concrete, and primitive operations will not be overridden.

Our metrics cannot say anything about how important multiple dispatch (or even single dispatch) is to program design: simply that many methods are monomorphic, and most of the remainder are single dispatch. Those dispatching methods may be crucial to the functioning of a particular program — as well as Template method, many other patterns (Visitor, Observer, Strategy, State, Composite) are about scaffolding a well-chosen dynamic dispatch with lots of relatively straightforward non-dispatching code.

Another point here is that a language specification does not dictate a programming style. Just supporting multiple (or even single) dispatch in a programming language doesn't mean it will be used in programs, the Nice compiler being a prime example.

6.2.2 Style

Our metrics, besides furnishing absolute numbers, can be interpreted together to identify attributes of programming style in the applications we study.

Use of Specialisers

By comparing $\mathrm{DoS_G}$ and DoD metrics in Figure 5.13, we can single out two applications that use specialisation strictly for the purpose of dynamic dispatch (Whirlwind and NiceC); across the rest of the corpus, there seem to be much less generic function parameters that are dispatched upon than specialised parameters. We surmise that in general, specialisers are used descriptively: programmers specify specialisers for each parameter for documentation purposes, even though these parameters are not required for method dispatch.

From various discussions with users of CLOS and Dylan[1], specialising parameters is often seen as the way to document the scope of a method by stating the types of objects it is designed to handle. How and which specialisers are used to dispatch is not the programmers focus of attention. This is regarded as the key advantage of multiple dispatch over hand-coded alternatives (**instanceof**-tests, double dispatch pattern). Those alternatives are low-level techniques, as they force

[1] in person and on the #Lisp and #Dylan IRC channels

developers to focus on how to simulate multiple dispatch. Multiple dispatch is a high-level language feature because it doesn't deflect the programmers attention from the actual programming problem.

Specialisation Patterns

Comparing RS and DOS distributions in Figure 5.5 we find that for some applications (mainly Gwydion, CMUCL, Vortex) these distribution closely follow each other, while for OpenDylan, SBCL and McCLIM there is a larger proportion of methods with RS > DOS. From these observations we can tell that Gwydion, CMUCL and Vortex tend to specialise parameters left-to-right, then following with unspecialised parameters; the applications that fall into the second group do not order parameters based on specialisation.

Dispatch

Comparing RD and DOD metrics in Figures 5.1 (averages), 5.9 (distributions) and 5.10 (values) we see that some applications (primarily McCLIM, and OpenDylan, but also Gwydion, Vortex and Whirlwind) have significantly higher values for Rightmost Dispatched parameter RD than they do for Degree of Dispatch DOD. This means that some generic functions' argument lists must have some non-dispatching parameters "to the left of" the dispatching parameters — a contrast to single-dispatch languages where the dispatch is always on the single leftmost parameter. For example, programs could contain two-argument generic functions which dispatch on the second argument but not on the first.

In the case of McCLIM, this must partly be explained by the fact that the CLIM Standard (McKay and York, 2001) explicitly requires some types of generic functions to dispatch on their second arguments (setfs and mapping functions). More generally, multiple dispatch gives more options to API designers, who can choose argument order to reflect application semantics rather than be restricted by having to place a dispatching argument first. In single dispatch languages, code can fall into a "Object Verb Subject" order: `rectangle.drawOn(window)`. Here, `Rectangle` must come first, purely because the code needs to dispatch on `Rectangle` to draw different kinds of figures. In multiple dispatch languages, this could equally be written `window.draw(rectangle)` matching the "Subject Verb Object" word order commonly used in English, or perhaps "Verb Subject Object" `draw(window,rectangle)`. Multiple dispatch languages offer this flexibility, even where only single dispatch is required, and our metrics demonstrate that programmers take advantage of this flexibility.

6.2.3 A Critical Perspective

The corpus we use in this study may appear relatively small when compared to corpora used in studies taking similar methodological approaches (Baxter et al. (2006); Tempero et al. (2008); the Java study of Muschevici, Potanin, Tempero, and Noble (2008)). This largely reflects the fact that multiple dispatch languages are not in wide use today. For most languages covered, we only analyse a single application (the exceptions are CLOS with three and Dylan with two applications). Furthermore, most applications are compilers and their standard libraries. Therefore we cannot claim that our results are representative for the use of multiple dispatch in any of these languages.

Our study provides "hard evidence" in terms of concrete numbers that account for the use of multiple dispatch in existing programs. Still, these numbers only express measurements based on the information furnished by compilers through their respective interfaces.

Finally, the strength of our approach comes from its diversity. Our study analyses a broad range of six languages and eight compilers, yet when comparing the results among each other, their similarity is striking. Function polymorphism, as measured by the Dispatch Ratio metric, is distributed in all applications according to the same logarithmic curve (Figure 5.2); methods generally specialise one single parameter (Figure 5.4); and the proportions of static (non-dispatched), single-dispatch and multiple dispatch functions uniformly follow the same distribution across the whole corpus (Figure 5.8). This common result across a range of languages, libraries, applications, implementations and analysis instruments increases our confidence in the large-scale findings of our study.

6.3 Evidence Based Language Design

> *I have always remark'd, that the author proceeds for some time in the ordinary ways of reasoning…when all of a sudden I am surpriz'd to find, that instead of the usual copulations of propositions,* is, *and* is not, *I meet with no proposition that is not connected with an* ought, *or an* ought not.

David Hume, *A Treatise of Human Nature*, 1739.

Hume's Law states that normative (prescriptive) statements — in this case, statements about how programs *ought* to be written — cannot be justified exclusively by descriptive statements. Our corpus analysis is descriptive: it tells us about how programs are written, but cannot (on its own) tell us about whether that is a "good" way to write programs, or whether language designers should

consider multiple dispatch (or even single dispatch) as a language feature worth retaining. In this thesis, we do not try to make any of these claims — we do not even claim whether high or low values for metrics are desirable: our metrics characterise program structures: they do not attempt to measure program quality.

Nonetheless, there seem to be clear advantages to informing the design of future languages with evidence drawn by something other than anecdote, personal experience or small-scale observational studies. Similarly, maintenance and debugging tasks — and even teaching about programming paradigms — would surely benefit from being based in evidence about the world as it is, as well as the world as we would like it to be!

6.4 Future Work

Our study is but a beginning in a line of research aiming at the question ***how much value is gained in practice by the use of multiple dispatch?***, and we hope our work will inspire more studies. This section lists possible directions for future research in this area.

6.4.1 Method Calls and Dynamic Aspects of Multimethods

Our quantitative study of multiple dispatch approaches the use of multimethods from a static point of view. We focus on method definitions which we examine at compile-time. We do not examine method calls or dynamic aspects of a program, such as the frequency of method calls through a call site or the frequency of invocations per method. Consequently, we would like to see these aspects covered in future studies.

6.4.2 Object-Orientation and Beyond

While single dispatch naturally fits the class- or object-based method encapsulation approach of mainstream programming languages, multiple dispatch seems to be the natural solution for a different category of problems. These problems occur whenever a behaviour depends on the types of multiple objects being used together (combined), and cannot be regarded as an invariant property of a single type of object, as are methods according to the object-oriented paradigm (Booch, 1982).

Clifton et al. (2006, Section 5.1) present anecdotal evidence for some programming scenarios where multiple dispatch is used in MultiJava projects. These include binary methods, event handling, tree traversals, and finite state machines. Our study complements their findings by providing quantified evidence that

multiple dispatch is used for solving existing programming problems. However, further qualitative studies are needed to identify and classify the real-world problems for which multiple dispatch provides an adequate solution — and for which traditional object-oriented languages do not.

6.4.3 Java Method Overloading

One aspect to consider about this study is that most languages examined here are primarily dynamic, in the sense that even if they support static type declarations, these are purely optional. Java, C++ and C♯, on the other hand, have a mandatory prescriptive static type system which can be used to overload method definitions. Dispatch to these overloaded methods occurs at compile time, which, as the examples in Section 2.1 show, can lead to incorrect behaviour. Due to its error potential, method overloading is generally considered bad programming style and discouraged (Bloch, 2001; Bloch and Gafter, 2005; Lea, 2008).

Because the multiple dispatch facility of the languages covered in this study subsumes static overloading of methods, we cannot tell how many programming problems that are solved using multiple dispatch in these languages are customarily solved for example in Java using static method overloading. We note however that Nice and MultiJava, the two languages which support both Java-like static overloading and dynamically dispatched multimethods, exhibit a significantly lower proportion of multiple dispatch (1.0–1.4%) than applications in other languages (on average 4.9%), as measured by the Degree of Dispatch metric.

We hope to see studies of method overloading in Java. Such studies could reveal the extent to which method overloading is used in existing programs and strengthen the case for multiple dispatch as a safe, upward compatible alternative.

Appendix A

Results

Figure A.1 contains the frequency distributions measured by each metric. The "Max." column gives the highest value measured for the given application. Subsequent columns show the absolute number and proportion (in percent) of measured subjects (generic functions or concrete methods) with the measurement given in the corresponding column header. The last column shows the sum for subjects with a metric value of 9 or higher.

Application	Metric	Max.	0	%0	1	%1	2	%2	3	%3	4	%4	5	%5	6	%6	7	%7	8	%8	9+	%9+
Gwydion	DR	204	0	0.00	3167	83.36	261	6.87	135	3.55	66	1.74	38	1.00	29	0.76	28	0.74	14	0.37	61	1.61
Gwydion	CR	204	0	0.00	3167	47.83	522	7.88	405	6.12	264	3.99	190	2.87	174	2.63	196	2.96	112	1.69	1591	24.03
Gwydion	DoS	20	380	5.74	2404	36.31	2071	31.28	912	13.77	365	5.51	95	1.43	180	2.72	54	0.82	84	1.27	76	1.15
Gwydion	DoS_G	20	355	9.34	1536	40.43	952	25.06	216	5.69	301	7.92	54	1.42	171	4.50	54	1.42	84	2.21	76	2.00
Gwydion	RS	20	380	5.74	2331	35.21	2072	31.29	877	13.25	450	6.80	64	0.97	162	2.45	4	0.06	185	2.79	96	1.45
Gwydion	RS_G	20	355	9.34	1499	39.46	970	25.53	129	3.40	385	10.13	23	0.61	153	4.03	4	0.11	185	4.87	96	2.53
Gwydion	DoD	4	3167	83.36	528	13.90	99	2.61	4	0.11	1	0.03	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
Gwydion	RD	6	3167	83.36	405	10.66	176	4.63	36	0.95	12	0.32	2	0.05	1	0.03	0	0.00	0	0.00	0	0.00
OpenDylan	DR	362	0	0.00	1459	68.08	312	14.56	125	5.83	78	3.64	56	2.61	26	1.21	5	0.23	18	0.84	64	2.99
OpenDylan	CR	362	0	0.00	1459	27.07	624	11.58	375	6.96	312	5.79	280	5.20	156	2.89	35	0.65	144	2.67	2004	37.19
OpenDylan	DoS	7	349	6.48	3684	68.36	1172	21.75	153	2.84	11	0.20	19	0.35	0	0.00	1	0.02	0	0.00	0	0.00
OpenDylan	DoS_G	7	130	6.07	1538	71.77	425	19.83	39	1.82	8	0.37	2	0.09	0	0.00	1	0.05	0	0.00	0	0.00
OpenDylan	RS	7	349	6.48	3231	59.96	1503	27.89	255	4.73	23	0.43	27	0.50	0	0.00	1	0.02	0	0.00	0	0.00
OpenDylan	RS_G	7	130	6.07	1360	63.46	568	26.50	68	3.17	13	0.61	3	0.14	0	0.00	1	0.05	0	0.00	0	0.00
OpenDylan	DoD	4	1459	68.08	545	25.43	126	5.88	11	0.51	2	0.09	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
OpenDylan	RD	5	1459	68.08	401	18.71	239	11.15	37	1.73	6	0.28	1	0.05	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	DR	29	0	0.00	176	64.94	55	20.30	13	4.80	10	3.69	6	2.21	1	0.37	0	0.00	1	0.37	9	3.32
CMUCL	CR	29	0	0.00	176	32.00	110	20.00	39	7.09	40	7.27	30	5.45	6	1.09	0	0.00	8	1.45	141	25.64
CMUCL	DoS	3	49	8.91	372	67.64	118	21.45	11	2.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	DoS_G	3	11	4.06	222	81.92	34	12.55	4	1.48	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	RS	4	49	8.91	351	63.82	123	22.36	24	4.36	3	0.55	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	RS_G	4	11	4.06	210	77.49	44	16.24	5	1.85	1	0.37	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	DoD	3	177	65.31	77	28.41	13	4.80	4	1.48	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
CMUCL	RD	4	177	65.31	72	26.57	16	5.90	5	1.85	1	0.37	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	DR	136	0	0.00	231	63.64	77	21.21	23	6.34	10	2.75	6	1.65	0	0.00	2	0.55	3	0.83	11	3.03
SBCL	CR	136	0	0.00	231	26.83	154	17.89	69	8.01	40	4.65	30	3.48	0	0.00	14	1.63	24	2.79	299	34.73
SBCL	DoS	3	63	7.32	647	75.15	140	16.26	11	1.28	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	DoS_G	3	10	2.75	311	85.67	38	10.47	4	1.10	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	RS	4	63	7.32	570	66.20	194	22.53	31	3.60	3	0.35	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	RS_G	4	10	2.75	245	67.49	102	28.10	5	1.38	1	0.28	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	DoD	3	231	63.64	115	31.68	13	3.58	4	1.10	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
SBCL	RD	4	231	63.64	106	29.20	20	5.51	5	1.38	1	0.28	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
McCLIM	DR	160	0	0.00	1103	57.87	448	23.50	148	7.76	78	4.09	33	1.73	32	1.68	7	0.37	9	0.47	48	2.52
McCLIM	CR	160	0	0.00	1103	24.96	896	20.28	444	10.05	312	7.06	165	3.73	192	4.34	49	1.11	72	1.63	1186	26.84
McCLIM	DoS	5	343	7.76	3153	71.35	747	16.90	166	3.76	9	0.20	1	0.02	0	0.00	0	0.00	0	0.00	0	0.00
McCLIM	DoS_G	6	121	6.35	1596	83.74	167	8.76	18	0.94	3	0.16	0	0.00	1	0.05	0	0.00	0	0.00	0	0.00
McCLIM	RS	6	343	7.76	2499	56.55	1266	28.65	207	4.68	58	1.31	36	0.81	10	0.23	0	0.00	0	0.00	0	0.00
McCLIM	RS_G	6	121	6.35	1253	65.74	469	24.61	47	2.47	8	0.42	4	0.21	4	0.21	0	0.00	0	0.00	0	0.00
McCLIM	DoD	5	1175	61.65	614	32.21	108	5.67	8	0.42	0	0.00	1	0.05	0	0.00	0	0.00	0	0.00	0	0.00
McCLIM	RD	6	1175	61.65	491	25.76	196	10.28	33	1.73	5	0.26	2	0.10	4	0.21	0	0.00	0	0.00	0	0.00
Vortex	DR	825	0	0.00	4441	67.89	1038	15.87	402	6.15	221	3.38	113	1.73	68	1.04	58	0.89	28	0.43	172	2.63
Vortex	CR	825	0	0.00	4441	29.19	2076	13.65	1206	7.93	884	5.81	565	3.71	408	2.68	406	2.67	224	1.47	5002	32.88
Vortex	DoS	6	1839	12.09	10900	71.65	2255	14.82	200	1.31	15	0.10	2	0.01	1	0.01	0	0.00	0	0.00	0	0.00
Vortex	DoS_G	6	1737	26.56	4140	63.29	584	8.93	68	1.04	10	0.15	1	0.02	1	0.02	0	0.00	0	0.00	0	0.00
Vortex	RS	8	1839	12.09	10668	70.13	2271	14.93	285	1.87	79	0.52	64	0.42	4	0.03	1	0.01	1	0.01	0	0.00
Vortex	RS_G	8	1737	26.56	4060	62.07	592	9.05	114	1.74	28	0.43	5	0.08	3	0.05	1	0.02	1	0.02	0	0.00
Vortex	DoD	4	4441	67.89	1843	28.18	232	3.55	24	0.37	1	0.02	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
Vortex	RD	7	4441	67.89	1629	24.90	367	5.61	81	1.24	18	0.28	2	0.03	2	0.03	1	0.02	0	0.00	0	0.00
Whirlwind	DR	462	0	0.00	4180	72.86	823	14.35	282	4.92	158	2.75	55	0.96	45	0.78	39	0.68	25	0.44	130	2.27
Whirlwind	CR	462	0	0.00	4180	35.21	1646	13.87	846	7.13	632	5.32	275	2.32	270	2.27	273	2.30	200	1.68	3549	29.90
Whirlwind	DoS	8	5065	42.67	5354	45.10	1288	10.85	149	1.26	13	0.11	0	0.00	1	0.01	0	0.00	1	0.01	0	0.00
Whirlwind	DoS_G	8	4055	70.68	1432	24.96	213	3.71	26	0.45	9	0.16	0	0.00	1	0.02	0	0.00	1	0.02	0	0.00
Whirlwind	RS	8	5065	42.67	4871	41.03	1499	12.63	347	2.92	52	0.44	30	0.25	4	0.03	1	0.01	2	0.02	0	0.00
Whirlwind	RS_G	8	4055	70.68	1244	21.68	319	5.56	79	1.38	24	0.42	11	0.19	2	0.03	1	0.02	2	0.03	0	0.00
Whirlwind	DoD	8	4180	72.86	1331	23.20	196	3.42	22	0.38	7	0.12	0	0.00	0	0.00	0	0.00	1	0.02	0	0.00
Whirlwind	RD	8	4180	72.86	1141	19.89	305	5.32	74	1.29	22	0.38	11	0.19	2	0.03	1	0.02	1	0.02	0	0.00
NiceC	DR	24	0	0.00	1025	86.57	91	7.69	33	2.79	10	0.84	5	0.42	3	0.25	3	0.25	2	0.17	12	1.01
NiceC	CR	24	0	0.00	1025	63.47	182	11.27	99	6.13	40	2.48	25	1.55	18	1.11	21	1.30	16	0.99	189	11.70
NiceC	DoS	3	1131	70.03	440	27.24	42	2.60	2	0.12	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NiceC	DoS_G	3	1016	85.81	153	12.92	13	1.10	2	0.17	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NiceC	RS	4	1131	70.03	420	26.01	58	3.59	4	0.25	2	0.12	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NiceC	RS_G	4	1016	85.81	148	12.50	15	1.27	4	0.34	1	0.08	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NiceC	DoD	3	1025	86.57	147	12.42	10	0.84	2	0.17	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
NiceC	RD	4	1025	86.57	143	12.08	13	1.10	2	0.17	1	0.08	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	DR	61	0	0.00	458	93.28	9	1.83	6	1.22	1	0.20	3	0.61	1	0.20	2	0.41	0	0.00	11	2.24
LocStack	CR	61	0	0.00	458	62.31	18	2.45	18	2.45	4	0.54	15	2.04	6	0.82	14	1.90	0	0.00	202	27.48
LocStack	DoS	3	92	12.52	553	75.24	73	9.93	17	2.31	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	DoS_G	3	92	18.74	385	78.41	12	2.44	2	0.41	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	RS	3	92	12.52	553	75.24	30	4.08	60	8.16	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	RS_G	3	92	18.74	385	78.41	8	1.63	6	1.22	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	DoD	2	458	93.28	26	5.30	7	1.43	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00
LocStack	RD	3	458	93.28	19	3.87	8	1.63	6	1.22	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00	0	0.00

Figure A.1: Metrics: frequency distributions

Bibliography

American National Standards Institute. *American National Standard for Information Technology: Programming Language – Common LISP*, 1996. Also available as "The HyperSpec", `http://www.lispworks.com/reference/HyperSpec/`. 29

Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan Tempero. Understanding the Shape of Java Software. In *OOPSLA*, pages 397–412, Portland, OR, USA, 2006. ACM Press. 27, 68, 75

Joshua Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, June 2001. ISBN 0201310058. 77

Joshua Bloch and Neal Gafter. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Addison-Wesley, July 2005. ISBN 032133678X. 77

Daniel G. Bobrow. *The LOOPS Manual*. Xerox Parc, 1983. 8

Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. *SIGPLAN Not*, 21:17–29, 1986. 8

Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *SIGPLAN Not*, 23:1–142, 1988. 8, 16, 37

Daniel Bonniot, Bryn Keller, and Francis Barber. The Nice user's manual, 2008. URL `http://nice.sourceforge.net/manual.html`. 5, 9, 18, 32

Grady Booch. Object-Oriented Design. *ACM SIGADA Ada Letters*, 1:64–76, 1982. 76

François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–315, New York, NY, USA, 1997. ACM Press. 32

John Boyland and Giuseppe Castagna. Parasitic Methods: An implementation of multi-methods for Java. In *OOPSLA*, pages 66–76. ACM Press, 1997. 10

Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Jonathan Eifrig, Scott F. Smith, Valery Trifonov, Gary T. Leavens, and Benjamin C. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1:221–242, 1995. 5

Bruno Cabral and Paulo Marques. Exception Handling: A Field Study in Java and .NET. In *ECOOP*. Springer-Verlag, 2007. 27

Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2/3):138–164, 1988. 6

Cecil Group. The University of Washington Cecil group. World Wide Web electronic publication, 2008. URL `http://www.cs.washington.edu/research/projects/cecil/`. [accessed 4-September-2008]. 32

Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *ECOOP*, volume 4609, pages 227–247. Springer-Verlag, 2007. 27

Craig Chambers. The Diesel Language, Specification and Rationale, 2006. URL `http://www.cs.washington.edu/research/projects/cecil/www/Release/doc-diesel-lang/diesel-spec.pdf`. 9, 16

Craig Chambers. Object-oriented Multi-methods in Cecil. In *ECOOP*, volume 615, pages 33–56. Springer-Verlag, 1992. 9, 16

Craig Chambers and Weimin Chen. Efficient multiple and predicated dispatching. In *OOPSLA*, pages 238–255, Denver, CO, USA, 1999. ACM Press. 11

Craig Chambers and Gary T. Leavens. Typechecking and modules for multimethods. *TOPLAS*, 17(6):805–843, 1995. 9

Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *OOPSLA*, pages 130–145, Minneapolis, MN, USA, 2000. ACM Press. 9

Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *TOPLAS*, 28: 517–575, 2006. 18, 45, 76

CMUCL Maintainers. CMUCL: a high-performance, free Common Lisp implementation. World Wide Web electronic publication, 2008. URL `http://www.cons.org/cmucl/`. [accessed 3-September-2008]. 29

Damian Conway. Multiple Dispatch in Perl. *The Perl Journal*, 5(17), 2000. URL `http://www.foo.be/docs/tpj/issues/vol5_1/tpj0501-0010.html`. 10

Antonio Cunei and Jan Vitek. PolyD: a flexible dispatching framework. In *ACM Press*, pages 487–503, San Diego, CA, USA, 2005. 11

Topher Cyll. Multiple Dispatch Library. World Wide Web electronic publication, 2005. URL `http://rubyforge.org/projects/multi/`. [accessed 6-October-2008]. 10

Karel Driesen, Urs Hlzle, and Jan Vitek. Message dispatch on pipelined processors. In *ECOOP*, volume 952, pages 253–282, 1995. 11

Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *ECOOP*, pages 130–149. Springer-Verlag, 2001. 10

Christopher Dutchyn, Paul Lu, Duane Szafron, Steven Bromling, and Wade Holst. Multi-Dispatch in the Java Virtual Machine: Design and Implementation. In *USENIX*, pages 6–6, San Antonio, Texas, United States, 2001. USENIX Association. 10

Johan Fabry and Tom Mens. Language-independent detection of object-oriented design patterns. *Computer Languages, Systems and Structures*, 30(1–2):21–33, 2004. 27

Neal Feinberg. *Dylan Programming: An Object-Oriented and Dynamic Language*. Addison-Wesley, 1997. ISBN 0201479761. 8, 16, 31

Brian Foote, Ralph E. Johnson, and James Noble. Efficient Multimethods in a Single Dispatch Language. In *ECOOP*, volume 3586, pages 337–361. Springer-Verlag, 2005. 10

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns*. Addison-Wesley, November 1994. ISBN 0201633612. 6, 73

Yossi Gil and Itay Maman. Micro patterns in Java code. In *OOPSLA*, pages 97–116, San Diego, CA, USA, 2005. ACM Press. ISBN 1-59593-031-0. 27

Christian Grothoff. Walkabout Revisited: The Runabout. In *ECOOP*, volume 2743, pages 99–108. Springer-Verlag, 2003. 10

Gwydion Dylan Maintainers. Dylan Overview. World Wide Web electronic publication, 2008. URL `http://www.opendylan.org/`. [accessed 2-September-2008]. 31

Rich Hickey. Clojure Multimethods. World Wide Web electronic publication, 2008. URL `http://clojure.org/multimethods`. [accessed 23-September-2008]. 2, 9

Jeffrey Hightower. The Location Stack: A Layered Model for Location in Ubiquitous Computing. *In Proceedings of the 4th IEEE Workshop on Mobile Computing Systems & Applications (WMCSA2002)*, pages 22–28, 2002. 32

David Hume. *A Treatise of Human Nature*. Printed for John Noon, London, 1739. 75

Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *OOPSLA*, pages 347–349, Portland, OR, USA, 1986. ACM Press. 6

James Kempf, Warren Harris, Roy D'Souza, and Alan Snyder. Experience with CommonLoops. In *OOPSLA*, pages 214–226, Orlando, FL, USA, 1987. ACM Press. 11

Gregor Kiczales, Jim Des Rivières, and Daniel Gureasko Bobrow. *The Art of the Metaobject Protocol*. NetLibrary, Incorporated, 1991. ISBN 0585358125. 33, 37

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997. 11

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP*, volume 2072, pages 327–355. Springer-Verlag, 2001. 11

Eric Kidd. Efficient Compression of Generic Function Dispatch Tables. Technical Report TR2001-404, Dartmouth College, Hanover, NH, USA, 2001. 11

David B. Lamkins. *Successful Lisp: How to Understand and Use Common Lisp*. bookfix.com, December 2004. ISBN 3937526005. 16

Doug Lea. Java Coding Standard (Draft). World Wide Web electronic publication, 2008. URL `http://g.oswego.edu/dl/html/javaCodingStd.html`. [accessed 4-October-2008]. 77

Kin-Keung Ma and Jeffrey S. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, pages 423–440, Montreal, Quebec, Canada, 2007. ACM Press. 27

Scott McKay and William York. *Common Lisp Interface Manager: CLIM II Specification*, 2001. 30, 74

Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415, August 2007. 27

David Mertz. Charming Python: Multiple Dispatch. World Wide Web electronic publication, 2003. URL `http://www.ibm.com/developerworks/library/l-pydisp.html`. [accessed 6-October-2008]. 10

Todd Millstein and Craig Chambers. Modular Statically Typed Multimethods. In *ECOOP*, volume 1628, page 668. Springer-Verlag, 1999. 9

Todd Millstein, Mark Reay, and Craig Chambers. Relaxed MultiJava: balancing extensibility and modular typechecking. In *OOPSLA*, pages 224–240, Anaheim, California, USA, 2003. ACM Press. ISBN 1-58113-712-5. 10

Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-Methods in a Statically-Typed Programming Language. In *ECOOP*, volume 512, page 307. Springer-Verlag, 1991. 9

Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple Dispatch in Practice. In *OOPSLA*, Nashville, TN, USA, 2008. ACM Press. 11, 17, 71, 75

Mayur Naik and Rajeev Kumar. Efficient message dispatch in object-oriented systems. In *OOPSLA*, volume 35, pages 49–58, 2000. 11

James Noble and Robert Biddle. Program visualisation for visual programs. In *Proceedings of the Australasian Conference on User Interfaces (AUIC)*, pages 29–38, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc. ISBN 0-909925-85-2. 28

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, Ecole Polytechnique Fèdèrale de Lausanne, Lausanne, Switzerland, 2004. 10

Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *COMPSAC '98: Proceedings of the 22nd International Computer Software and Applications Conference*, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8585-9. 10

Alan J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9): 7–13, 1982. 1, 65

Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. Open multi-methods for C++. In *Proceedings of the 6th international conference on Generative programming and component engineering*, pages 123–134, Salzburg, Austria, 2007. ACM Press. 10

Alex Potanin, James Noble, and Robert Biddle. Checking ownership and confinement. *Concurrency and Computation: Practice and Experience*, 16(7):671–687, 2004. 27

Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in Object-Oriented programs. *CACM*, 48:99–103, May 2005. 27, 68

Allison Randal, Dan Sugalski, and Leopold Toetsch. *Perl 6 and Parrot Essentials*. O'Reilly Media, Inc., 2nd edition, June 2004. ISBN 059600737X. 9

Lee Salzman and Jonathan Aldrich. Prototypes with multiple dispatch: An expressive and dynamic object model. In *ECOOP*, volume 3586, pages 312–336, Glasgow, Scotland, 2005. Springer-Verlag. 2, 9

SBCL Maintainers. Steel Bank Common Lisp. World Wide Web electronic publication, 2008. URL `http://www.sbcl.org/`. [accessed 3-September-2008]. 30

Asim Anand Sinha. Multiple Dispatch and Roles in OO Languages: $Fickle_{MR}$. Master's thesis, Imperial College, London, UK, 2005. 10

Robert Strandh and Timothy Moore. A Free Implementation of CLIM. In *International Lisp Conference*. Franz Inc., 2002. 30

Bjarne Stroustrup. *The Design and Evolution of* C++. AW, 1994. 8

Venkat Subramaniam. *Programming Groovy: Dynamic Productivity for the Java Developer*. Pragmatic Bookshelf, April 2008. ISBN 1934356093. 2, 9

Ewan Tempero, James Noble, and Hayden Melton. How do Java programs use inheritance? In *ECOOP*, pages 667–691. Springer-Verlag, 2008. 27, 75

Christopher Unkel and Monica S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *POPL*, volume 43, pages 183–195, New York, NY, USA, 2008. ACM Press. 27

Richard Wheeldon and Steve Counsell. Power law distributions in class relationships. *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation*, cs.SE/0305037, 2003. 68

Yoav Zibin and Joseph Yossi Gil. Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching. In *OOPSLA*, pages 142–160, Seattle, Washington, USA, 2002. ACM Press. 11