Multiple Dispatch in Practice

Radu Muschevici
Alex Potanin
Victoria University of Wellington, New Zealand
{radu|alex}@mcs.vuw.ac.nz

Ewan Tempero
University of Auckland, New Zealand
ewan@cs.auckland.ac.nz

James Noble
Victoria University of Wellington, New Zealand
kjx@mcs.vuw.ac.nz

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 paper presents an empirical study of the use of multiple dispatch in practice, considering six languages that support multiple dispatch, and also investigating the potential for multiple dispatch in Java programs. 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.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Procedures, functions, and subroutines; D.1.0 [*Programming Techniques*]: General

General Terms Design, Experimentation, Languages, Measurement

Keywords double dispatch, empirical software engineering, instanceof, multimethods, multiple dispatch

1. Introduction

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 paper is to understand how programmers write programs that use multiple dispatch when it is available — and to investigate what programmers do when it is not. We ask two complementary questions. For multiple dispatch programs we ask ***how much is multiple dispatch used?*** — what proportion of method declarations dispatch on more than one argument. For single dispatch programs, we ask ***how much could multiple dispatch be used?*** — that is, what proportion of methods hand-code idioms to provide multiple dispatch, or what proportion of methods could be refactored to use multiple dispatch if it was provided by the language.

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)), we focus on these six languages here because we were able to obtain a corpus for each of these languages. We use Java as a control subject in our study. We present the result of a second analysis of a large corpus of Java programs that do not use explicit multiple dispatch.

The contributions of this paper are as follows: (1) a language independent model of multiple dispatch; (2) a suite of language independent metrics, measuring the use of multiple dispatch; (3) the corpus analysis study using those metrics on a collection of programs in six multiple dispatch languages; and (4) a comparison with hand-coded multiple dispatch in a large corpus of Java programs.

Outline. Section 2 presents the brief history and overview of multiple dispatch including related work. Section 3 presents a language-independent model of multiple dispatch, and defines the metrics we will use in terms of that model. Section 4 then presents the results of our study in multiple dispatch languages, and section 5 presents the results for Java programs. Section 6 puts our results in perspective and Section 7 concludes.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
Copyright © 2008 ACM 978-1-60558-215-3/08/10…$5.00

2. Multiple Dispatch

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 *"Car crash"* method 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 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 m) {
  /* 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 generic functions: "collide(myCar, yourBike);" or "drive(myCar);", often while also supporting Java-style receiver syntax.

2.2 Single vs 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. For this reason, 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"); }
}
```

for the Car class, and

```
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"); }
}
```

for the Bike class.

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 class:

```
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");
  }
}
```

and the Bike class:

```
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.

2.3 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. Dylan'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) is a prototype-based programming language that features symmetric multimethods and an optional static type system. 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, e.g. 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.

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.

2.4 Related Work

There are of course many other multiple dispatch languages, which we have not been able to include in our study: space does not permit us to describe them all here. Parasitic Multimethods (Boyland and Castagna 1997) is an earlier extension to Java that provides multiple dispatch. Kea (Mugridge et al. 1991) was the first statically typed language with multiple dispatch. Smalltalk has been extended with multiple dispatch (Foote et al. 2005) while Dutchyn et al. (2001) modified the Java virtual machine to treat static overloading as dynamic dispatch. Slate (Salzman and Aldrich 2005) provides multiple dispatch in a Self-like setting.

Alternatives to multiple dispatch range from classical double dispatch (Ingalls 1986) and the Visitor pattern (Gamma et al. 1994) to visitor-oriented programming (Palsberg and Drunen 2004) and dispatching on tuples of objects (Leavens and Millstein 1998). 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.

Multiple dispatch studies are less widespread than multiple dispatch implementations — Kempf, Harris, D'Souza, and Snyder's early 1987 study of CLOS is one notable exception. The efficiency of implementation has been evaluated (Kidd 2001; Foote et al. 2005) as part of larger projects: Cunei and Vitek (2005) include a recent comparison of the efficiency of a range of multiple dispatch implementations.

Corpus analysis is a widely used empirical software engineering 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 (Ma and Foster 2007), dependency cycles (Melton and Tempero 2007), exception handling (Cabral and Marques 2007), and non-nullity (Chalin and James 2007).

3. Methodology

In this section we describe the methodology underlying our studies. We begin by introducing a language-independent model for multiple dispatch, describe each of the multiple dispatch languages in terms of that model, and give a Java example as a control. We then use the model to define metrics for multiple dispatch.

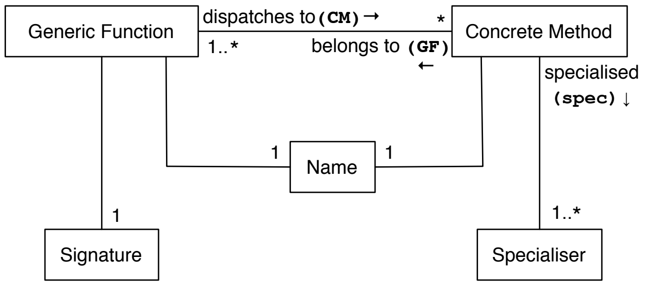

Figure 1. A Model for Multimethod Analysis. GF refers to generic function, CM refers to concrete method, and **spec** refers to specialiser.

3.1 Modelling Dynamic Dispatch

We begin by describing a language-independent model of dynamic dispatch. The model, shown in Figure 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, 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), whereas in other languages (such as Java) 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 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

language	typing	GF term	GF dfn	CM term	CM grouped in GF	multi	accessor
CLOS	dyn	generic function	explicit	method	name [a]	no	auto
Dylan	opt	generic function	explicit	method	name [a]	no	auto
Cecil	opt	method	implicit	method body	name+#args	no	auto
Diesel	opt	function	explicit	method	name+#args	no	auto
Nice	static	method declaration	implicit	method implementation	name+#args+types	yes	–
MultiJava	static	method family	implicit	method	name+#args+types	no	–
Java	static	method call	implicit	method body	name+#args+types	yes	–
Smalltalk	dyn	message	implicit	method	name(+#args) [b]	no	–

[a] All argument lists (lambda lists) must be congruent. [b] Smalltalk message selectors encode the number of arguments to the message.

Figure 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).

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 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":

Abbrev	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	Degree of Specialisation	concrete method	number of specialisers
RS	Rightmost Specialiser	concrete method	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. Metrics

all methods must agree on the number of required and optional parameters, and the presence and names of keyword parameters (Lamkins and Gabriel 2005).

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.

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.

Java Java is of course a single dispatch, statically typed class-based language that we include as a control. Java 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.

Smalltalk Smalltalk is not part of our study but we include it in the table as a comparison. Smalltalk 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.

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 will examine statically. We do not examine method calls or dynamic aspects of a program (e.g. frequency of method calls through a call site, frequency of invocations per method) although we would like to see these aspects covered in future studies.

To study multiple dispatch across languages we define metrics based on our language independent model. Figure 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. Any number of concrete methods can belong to a given generic function, giving the basic metric *dispatch ratio* $\mathrm{DR}(g) = |CM(g)|$ — the number of concrete methods that belong to the generic function g. DR measures, in some sense, the *amount of choice* offered by a generic function: monomorphic functions will have $\mathrm{DR}(g) = 1$, while polymorphic functions will have $\mathrm{DR}(g) > 1$.

We are usually not interested in the measurements from the above metrics 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 $\mathrm{DR}(g) = dr$. Frequency distributions provide information such as: what proportion of generic functions have exactly 1 concrete method.

Across whole applications or corpora, we use the basic DR metric to define an average dispatch ratio across each corpus. The average dispatch ratio $\mathrm{DR}_{\mathit{ave}}$ — the average number of concrete methods that a generic function would need to choose between is:

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

where $\mathcal{G}$ is the set of all generic functions. The intuition behind $\mathrm{DR}_{\mathit{ave}}$ 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 $\mathrm{DR} > 1$ necessarily contains more methods than a monomorphic generic function, we were concerned that $\mathrm{DR}_{\mathit{ave}}$ 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, $\mathrm{DR}_{\mathit{ave}} = 1.98$, 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 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. A corpus-wide average, $\mathrm{CR}_{\mathit{ave}}$ can be defined similarly:

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

where $\mathcal{M}$ is the set of concrete methods. The intuition behind $\mathrm{CR}_{\mathit{ave}}$ 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?

3.3.3 Degree of Specialisation (DoS)

The *degree of specialisation* of a concrete method simply counts the number of specialised parameters:

$$\mathrm{DoS}(m) = |\mathbf{spec}(m)|$$

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

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

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

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

If a method has some number of specialised parameters (perhaps none) followed by a number of unspecialised parameters, then $\mathrm{RS} = \mathrm{DoS}$; where a method has some unspecialised parameters early in the list, and then some specialised parameters, $\mathrm{RS} > \mathrm{DoS}$. 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 ($\mathrm{DoS}{=}1$) but where that dispatch is *not* the first method argument. Once again, we can define a summary metric $\mathrm{RS}_{\mathit{ave}}$ by averaging RS over all concrete methods.

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.

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

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: the rightmost parameter a generic function actually dispatches upon.

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

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 the whole corpora, 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 Gwydion Dylan:

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

// DR = 2, DoD = 1, RD = 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 = 1, RS = 2
define method collide(v :: <vehicle>, c :: <car>) ... ;

// DR = 4, DoD = 3, RD = 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>) ... ;
```

These are 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.

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, DoD and RD are measured at the level of generic functions. DoD records the number of generic function's arguments that can be potentially specialised by one or more of the concrete methods and RD records the rightmost position used by a specialiser. Their averages are $(1+3)/2 = 2$ for the DoD_{ave} and $(2 + 3)/2 = 2.5$ for the RD_{ave}.

4. Multiple Dispatch Languages

For this study we have gathered a corpus of 9 applications written in 6 languages that offer multiple dispatch (Figure 4). Most are compilers for the respective language — they are all too often the only applications of significant size that we could obtain. CLOS is notably distinct in this respect 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. The MultiJava-based Location Stack (Hightower 2002) is a framework for processing measurements from a network of geographical location sensors.

We applied the metrics defined in Section 3.3 to our corpus: the results are summarised in Figure 11. As is often the case when measuring real code, we had to make assumptions about exactly what to measure. One assumption was with respect to the auto-generated field accessors some languages provided (see Figure 2). As our interest is in how programmers interact with language features, we did not measure these accessors. All of the languages studied here come with standard libraries. Our measurements of each application included the contribution due to the libraries (in contrast with the Java measurements, see Section 5). The Nice language compiler compiles both Nice and Java source code into Java bytecode. The compiler itself is written partly in Java, partly in Nice. For this study, we only consider

Language	Application	Domain	Version	Concrete methods	Generic functions
Dylan	Gwydion	compiler	2.5 svn:12/03/2008	6621	3799
Dylan	OpenDylan	compiler	1.0beta5 svn:27/04/2008	5389	2143
CLOS	SBCL	compiler	0.9.16	861	363
CLOS	CMUCL	compiler	19d	1031	512
CLOS	McCLIM	toolkit/library	0.9.5	5400	2222
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. Size of applications in corpus

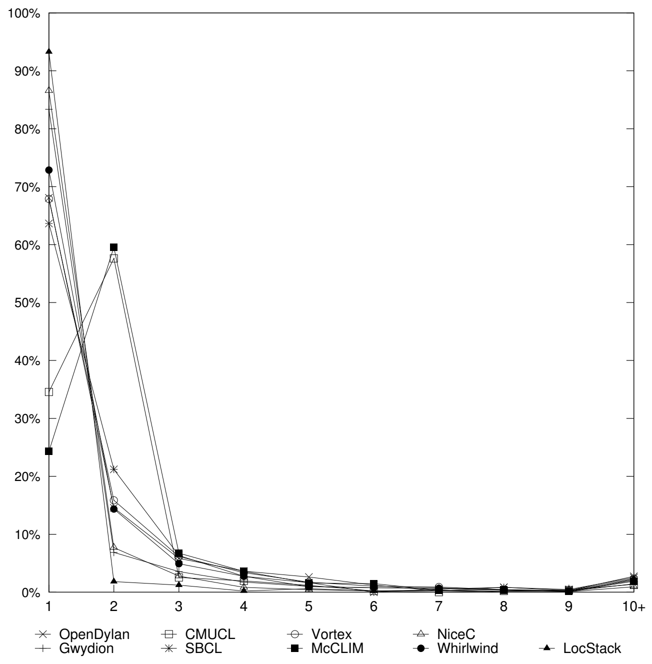

Figure 5. Dispatch ratio (DR) frequency distribution, expressed as a percentage of all generic functions with a given DR measurement.

native Nice methods, since only these have multiple dispatch potential.

4.1 Dispatch Ratio

The Dispatch Ratio distribution for all applications is shown in Figure 5. Seven applications in six different languages follow a similar distribution with 65%–93% of generic functions having a single concrete method. The shares for generic functions with two (2%–20%), three (3%–6%) and more methods decrease rapidly. The exceptions here are CMUCL and McCLIM (both Common Lisp projects), which have a 60%-share of generic functions with 2 alternative implementations that is roughly double the proportion of generic functions with one single concrete method, but otherwise have a similar shape to the other 7 applications. We are not measuring the use of *non-generic* functions in CLOS applications, so we

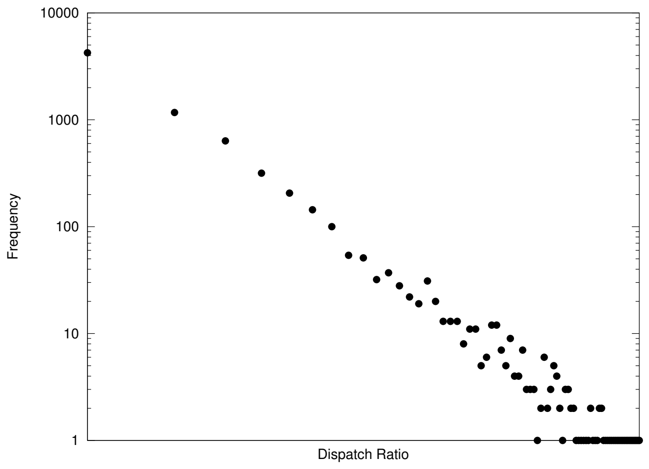

Figure 6. Dispatch ratio (DR) distribution, log-log scale

hypothesize that some monomorphic functions in these applications may be implemented by non-generic functions. It is also notable that these two projects are quite different in size, McCLIM being roughly 5 times the size of CMUCL in terms of concrete methods, yet their distributions are very similar.

The curves shown in Figure 5 are reminiscent of power law distributions. As the curves are fairly close to each other, we show all values on the same log-log scale (Figure 6). The strong indication of a straight line is further evidence of the possibility that power laws are being followed.

The DR_{ave} and CR_{ave} values for the applications in our corpus are shown in Figure 11. 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. The results for CR_{ave} show considerable variance. On average, any concrete method in Vortex is part of dispatch decision with 60 or so other methods, whereas for NiceC it would be only with 3.5 other methods.

4.2 Specialisation

Figure 7 shows, for each application, what proportion of generic functions have a given DoS measurement. At the top are the highest DoS values measured for the respective appli-

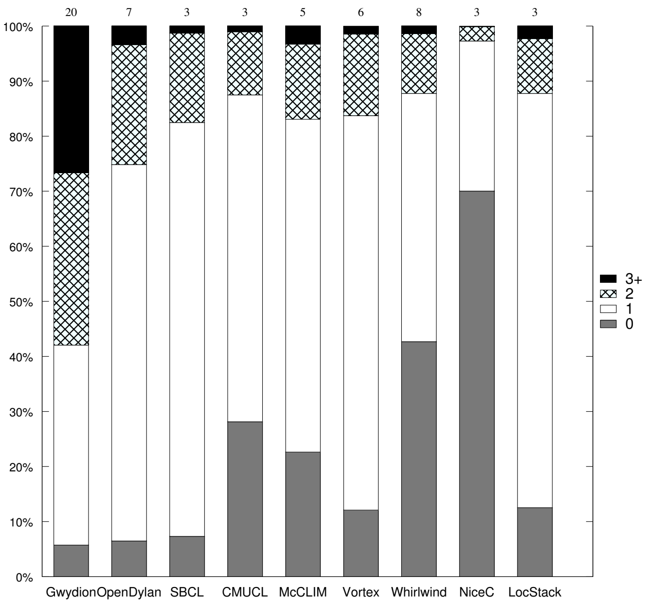

Figure 7. Degree of specialisation (DoS) distribution of concrete methods across applications. The lowest block in each stack is the proportion with DoS measurement of 0, the next, the proportion with 1, and so on. The value at the top of each stack indicates the highest DoS measured for this application.

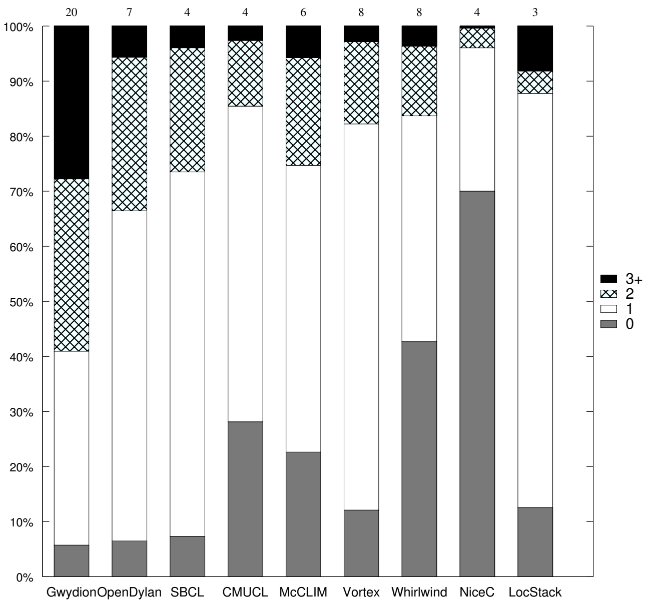

Figure 8. Rightmost specialiser (RS) distribution of concrete methods across applications. The value at the top of each stack indicates the highest RS measured for this application.

cation. While it is quite common for methods to specialise up to 3 parameters, we found generic functions that specialise 7 (OpenDylan, "make−source−location"), 8 (Whirlwind, "resolve8") and 20 (Gwydion, "parser:production_113") parameters. There is also a considerable range for the proportion of generic functions with no specialisation across the applications.

The results for the RS metric are shown in Figure 8. Since they are not significantly different from DoS numbers, we conclude that programmers generally specialise parameters left-to-right, then follow with unspecialised parameters.

4.3 Dispatch

Figure 9 shows the degree of dispatch (DoD). Again excluding CMUCL and McCLIM, most applications have similar levels (2.7–6.5%) of multiple dispatch (DoD > 1), and single dispatch (13–32%). The share of generic functions that are not required to dispatch dynamically ranges from 64% to 93%; this corresponds nicely with the 65%–93% of generic functions having a single concrete method and thus a dispatch ratio of 1. The Nice compiler has the lowest proportion of multiple dispatch (1%) among the analysed applications, even though we have excluded that part of the source written in Java. On average, across all measured applications, we found that around 3% of generic functions utilise multiple dispatch (DoD > 1) and around 30% utilise single dispatch (DoD $= 1$).

Figure 10 shows the rightmost dispatched parameter (RD). This generally follows DoD, although the proportions are often a little higher for RD ≥ 2. 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.

Figure 11 provides the averages of each of the metrics for all the multimethod applications to show relationships between the 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. The specialiser metrics DoS and RS will also generally be below the dispatch metrics DoD and RD, because generic functions dispatch on positions where methods are specialised, 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 figures 7 and 8, 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. (Strictly, specialiser and dispatch metrics are not comparable, as dispatch metrics average over generic functions while specialisation metrics average over concrete methods).

	Gwydion	OpenDylan	SBCL	CMUCL	McCLIM	Vortex	Whirlwind	NiceC	LocStack
DR_{ave}	1.74	2.51	2.37	2.01	2.43	2.33	2.07	1.36	1.50
CR_{ave}	18.27	43.84	26.57	4.31	7.61	63.30	31.65	3.46	8.92
DoS_{ave}	2.14	1.23	1.11	0.85	0.98	1.06	0.71	0.33	1.02
RS_{ave}	2.24	1.34	1.23	0.89	1.11	1.10	0.78	0.34	1.08
DoD_{ave}	0.20	0.39	0.42	0.69	0.78	0.36	0.32	0.15	0.08
RD_{ave}	0.24	0.48	0.45	0.71	0.86	0.41	0.37	0.15	0.11

Figure 11. Metrics: averages across applications

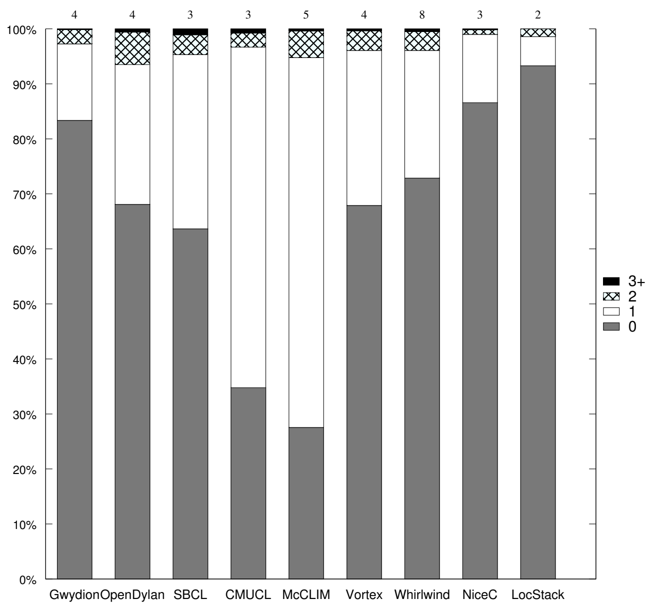

Figure 9. Degree of dispatch (DoD) distribution of generic functions across applications. The value at the top of each stack indicates the highest DoD measured for this application.

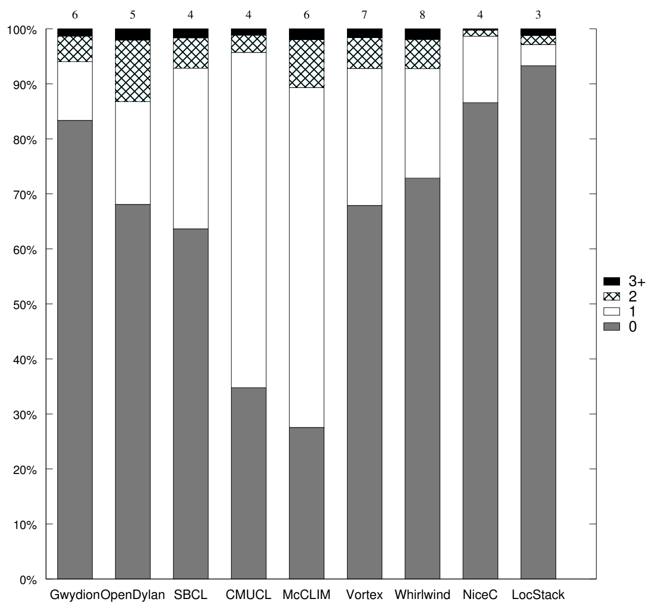

Figure 10. Rightmost dispatch (RD) distribution of generic functions across applications. The value at the top of each stack indicates the highest RD measured for this application.

5. Multiple Dispatch in Java

In order to understand how representative our results from the previous section are, it is useful to determine to what degree multiple dispatch is needed. We examine a mainstream language, namely Java, and determine how often programmers use some mechanism to simulate multiple dispatch. Our methodology is to establish the common idioms, and then measure the use of these idioms in a standard corpus. The release of the corpus that we used for this study has 100 applications in it (Qualitas Research Group 2008). The measurements we present here are for just the latest release of each application in the corpus. We also measure these applications using the applicable metrics from Section 3.

As with the multimethod languages, we do not measure any code that is automatically generated, but unlike the multimethod languages we could only measure that code that is distributed as an application independent of the standard library (JRE) and any third-party libraries. This is due to the fact that the JRE is significantly larger that many of the applications, and we felt its measurements would mask those of the application.

5.1 Double Dispatch

As illustrated in Section 2, a common approach to providing multiple dispatch in single dispatch languages is the *double dispatch* pattern described by Ingalls (1986). We can get an idea of to what degree double dispatch is used by measuring the occurrence of the double dispatch pattern. We refer to methods that match this pattern as double dispatch candidates. The characteristics of the double dispatch pattern that we use to identify its use are:

1. The **this** object is passed as an actual parameter to a method invoked on one of the formal parameters to the double dispatch candidate.
2. The type of the formal parameter of the invoked method is different from the actual parameter passed.

3. There is more than one child (either through **extends** or **implements**) of the formal parameter of the invoked method containing the same method.

The first characteristic by itself will produce many false positives. This pattern occurs very frequently when setting up a mutual relationship between two objects. For example, in antlr the class antlr.preprocessor.Hierarchy has a method **public void** addGrammar(Grammar gr) whose first statement is gr.setHierarchy(**this**). Inspection of Grammar class indicates that all that is happening is a "setter" is being used to set up a mutual relationship between a Grammar and a Hierarchy object.

The key to avoiding the "setter" situation is to realise that a true double dispatch pattern applies to hierarchies of types (Shape and Port in the case of the example). The second and third characteristics provide heuristics for establishing that the hierarchies exist. We also rule out the use of java.lang.Object as the formal parameter type as meeting the double dispatch pattern.

We do our measurements on bytecode rather that source, in part due to the difficulty in getting high-fidelity parsers (Irwin 2007) and in part due to having existing bytecode analysis tools available. We only examined methods written for the application for which double dispatch could take place, that is, we did not examine synthetic methods, native methods, constructors, static methods, and abstract methods. We do examine private (and generally non-public) methods to allow for the possibility that the double dispatch has been factored into a private method.

5.1.1 Results for double dispatch

Of the 100 applications we measured, 30 have at least one candidate method, that is, at least one method that has the pattern described above. Figure 12 shows those applications, the number of candidate methods, and the number of methods examined. We have checked each case and in the fourth column give our assessment as to whether or not one of the candidates is indeed intended to provide double dispatch. Note that "Yes" only means that at least one candidate could be considered use of double dispatch (sometimes by a very generous interpretation), but not necessarily all do.

In some cases (azureus) the appearance of the pattern does not seem to be due to deliberate use of double dispatch, whereas in others (eclipse) it does (in this case an example of the Visitor pattern). The number of candidate methods is not a useful indicator of the use of double dispatch — a relatively high number does not indicate its use (azureus again), and nor does a lower number indicate non-use (emma is a Visitor pattern).

Our results clearly have a number of false positives — many candidates are not in fact an actual use of double dispatch. False negatives are also possible. Because characteristic 1 requires that the invoked method be on a formal parameter to a candidate, if the parameter is assigned to a local

Application	DDC	M	DD
aoi	2	5122	No
aspectj	1	9647	No
azureus	14	17553	No
colt	1	2783	No
derby	10	17224	Yes
drjava	1	9491	No
eclipse	77	102231	Yes
emma	2	943	Yes
freecol	1	3625	No
gt2	26	15980	Yes
informa	6	832	No
itext	1	4931	No
jedit	1	4361	No
jhotdraw	1	1672	No
jre	28	77563	Yes
jrefactory	80	1939	Yes
jruby	19	6681	Yes
jtopen	1	21360	Yes
jung	4	2456	No
megamek	4	4515	No
nakedobjects	9	7581	Yes
pmd	3	2126	Yes
poi	14	6239	Yes
pooka	4	3426	Yes
proguard	26	3306	Yes
quartz	2	1575	No
sandmark	1	5400	No
squirrel	1	6465	No
velocity	5	1296	Yes
xalan	10	7935	Yes

Figure 12. Number of double dispatch candidate methods (DDC), Number of methods examined (M), Manual assessment of whether at least one candidate is an actual use of double dispatch (DD).

variable and the invocation done on the local, such methods will not be considered candidates. We have not seen an example of this, and it seems unlikely that such situations will occur when actually doing double dispatch. We believe the results presented represent upper bounds on the actual use of double dispatch.

5.2 Cascaded **instanceof**

An alternative to the use of the double dispatch pattern is to "manually" do the dispatch through the use of the **instanceof** operator. Again an example was given in Section 2. In this case we consider a method to be a cascaded **instanceof** candidate if it contains two applications of **instanceof** to the *same* formal parameter of a method. We require two applications because we have found many uses of single uses of **instanceof** within a method that do not appear to be

Application	%	CIC	M	DDC,DD
junit	0.51	2	391	
myfaces	0.57	27	4779	
jpf	0.58	5	867	
jedit	0.62	27	4361	1,No
freecol	0.63	23	3625	1,No
jsXe	0.65	3	465	
displaytag	0.65	5	769	
gt2	0.65	104	15980	26,yes
jung	0.65	16	2456	4,No
aspectj	0.65	63	9647	1,No
eclipse	0.69	707	102231	77,Yes
jchempaint	0.75	27	3624	
quartz	0.76	12	1575	2,No
megamek	0.82	37	4515	4,No
colt	0.93	26	2783	1,No
antlr	0.96	19	1987	
jruby	1.00	67	6681	19,Yes
axion	1.32	32	2419	
argouml	2.28	216	9484	

Figure 13. Applications with more than 0.5% methods being cascaded **instanceof** candidates (CIC). The last column repeats the relevant double dispatch data from figure 12.

simulating multiple dispatch. We examined the same set of methods as we did for the double dispatch pattern.

5.2.1 Results

All but 16 applications show at least some use of the **instanceof** pattern described above. Figure 13 shows those 19 applications that have more than 0.5% of their methods being cascaded **instanceof** candidates (the remaining results are omitted for space reasons). As in the case of the double dispatch pattern, there are some that do not appear to be simulating multiple dispatch (jsXe for example) but others (antlr for example) that clearly could be rewritten to use double dispatch (and more generally multimethods). Of particular interest is argouml. Not only does it have the highest proportion of its methods with the **instanceof** pattern, it also has a considerable number of uses of **instanceof** that don't match the pattern we measure and also apparently also has no use of double dispatch. We suspect much of this could be reduced through use of double dispatch but it would require considerable refactoring.

It would seem that multiple dispatch is more often being simulated in Java using cascading **instanceof** rather than double dispatch, although we note two (xalan, jruby) that appear to use both. In both those cases the double dispatch pattern are associated with the use of the visitor pattern.

As with double dispatch candidates, the cascading **instanceof** candidates must have the application of **instanceof** to a parameter of the candidate. Sampling of the code in the corpus suggests that it is possible that **instanceof** be applied to local variables. Such situations would not be considered candidates in our measurements. We also require at least two applications of **instanceof**. It is likely that even a single use of **instanceof** may correspond to a crude double dispatch. This means we are likely to have more false negatives for the cascading **instanceof** results than for the double dispatch results. Nevertheless we believe the results we have are upper bounds on the use of cascading **instanceof** as a means to provide multiple dispatch.

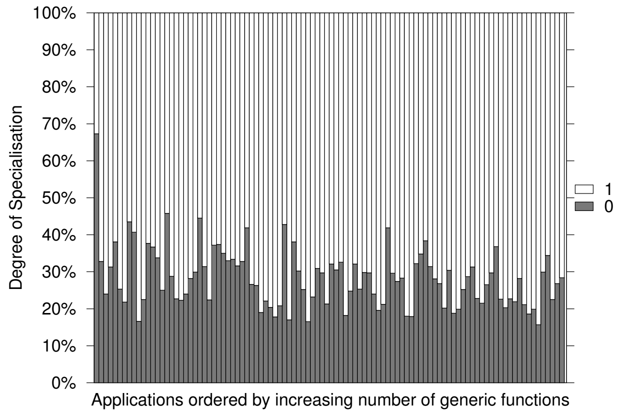

Figure 14. Degree of specialisation (DoS) for Java applications; measurements as a proportion of total concrete methods.

5.3 Metrics Results

We now present the measurements from the metrics presented in Section 3.3 for our Java corpus. For those metrics based on the presence of specialisers, in Java the only parameter that can be specialised is the "**this**" parameter. It is possible that no parameters are specialised, namely in the case of static methods and constructors. This means that in the standard interpretation of Java, rightmost specialiser (RS) will be either 0 or 1, and the rightmost dispatch (RD) will be 0 or 1 exactly when RS is 0 or 1. So the proportion of functions having RS and RD measurements of 0 is the proportion of generic functions that are either static methods or constructors, measurements we give below.

Unlike the double dispatch and cascading **instanceof** measurements, for the metrics discussed here we must measure static methods and constructors. We also must measure abstract methods, which do have a specialiser. We do not measure synthetic and native methods, and we do not measure private methods.

We can consider the use of either double dispatch or cascaded **instanceof** as providing specialisation on a second parameter, giving a RS of 2. However, as our results above show, even if we consider presence of the double dispatch or cascaded **instanceof** pattern as actually simulating multiple dispatch (which we know is not the case), then it is rare that even 1% of the functions will have an RS measurement of

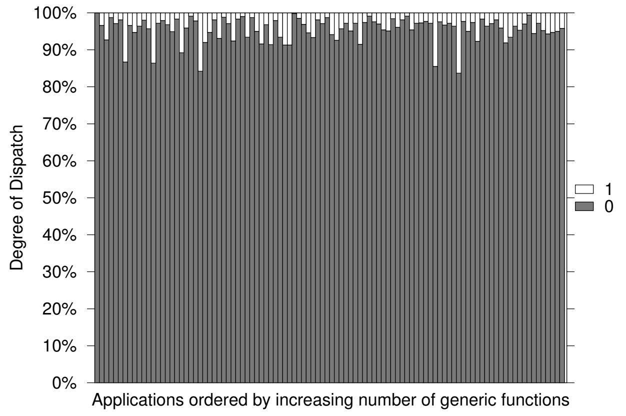

Figure 15. Degree of dispatch (DoD) for Java applications; measurements as proportion of total generic functions.

2. Accordingly, for clarity we will only give the results for measurements of 0 or 1.

Degree of specialisation (DoS) is also either 0 or 1 in the standard interpretation of Java, exactly when RS is 0 or 1. Figure 14 shows the results for DoS as applied to our Java corpus. The bars are ordered in increasing size of the applications, as measured by number of generic functions. What is somewhat surprising is how large the proportion of functions have DoS of 0. All applications have at least 15% of generic functions being constructors and static methods (the lowest being derby at 15.7%). The application with the largest proportion of DoS being 0 is jasml (67.3%). It seems that the presence of static methods accounts for many of these results. Half the applications have more than 40% of the DoS measurements due to static methods; the lowest is 11.7% (trove) and the highest is 80.5% (mvnforum).

As with DoS the degree of dispatch (DoD) metric will only provide measurements of 0 or 1. However, in this case non-static methods and non-constructors can have a measurement of 0 if there is not more than 1 concrete method belonging to the generic function. Figure 15 shows the DoD measurements as a proportion of the generic functions. It shows that rarely (6 of the 100 applications) does the proportion of generic functions with more than 1 concrete method get to even 10%, that is, usually less than 10% of methods are overridden. However, recall that user-defined methods that override standard library or third-party library are not counted in our measurements.

For the remaining metrics we need to determine what are generic functions in Java. Unlike the other languages there is no specific concept on which to base the decision, so we appeal to the definition given in Section 3, and identify generic functions with any possible method call. Figure 16 illustrates the consequences of this definition. Focusing only on the generic functions associated with A and B, there are seven generic functions — the two default constructors, and the 5 possible calls that can take place as shown in the body of Main#uses(A,B).

```
class A {
   public void methodA() {}
   public void inherited()  {}
}

class B extends A {
   public void methodB() {}
   public void inherited()  {} // overrides from A
}

class Main {
   public void uses(A anA, B aB) {
      anA.methodA();   // GF: methodA(A)
      anA.inherited();  // GF: inherited(A)
      aB.methodB();    // GF: methodB(B)
      aB.inherited();   // GF: inherited(B)
      aB.methodA();    // GF: methodA(B)
   }
}
```

Figure 16. Java example of generic functions. The generic functions being called at each callsite are shown in the comments.

For illustration, we will name generic functions with the "functional" form of the possible method calls, that is, making the implicit "**this**" argument type explicit, as shown in the comments in the figure. For example, there is the generic function methodB(B) and it contains the concrete method B.methodB(). All generic functions except inherited(A) contain only one concrete function. inherited(A) contains two concrete methods, A.inherited() and B.inherited(), as either of these could be executed at the second callsite.

The last callsite in the example requires more discussion. It is a legal call, and so by our definition it is a generic function. However, in the given example, the only concrete method it contains is A.methodA(), as that is the method inherited by B. At first glance this seems odd, however it must be this way. If we consider the generic function at this callsite to be methodA(A) then, if a new class inherits from B and overrides inherited(), the generic function at the callsite would have to change, despite neither B nor Main changing. So our conclusion is that the generic function called at the last callsite of the example must be methodA(B) (and this is in fact how it will be compiled, as an invokevirtual on B.methodA()). We have explored other possible definitions and these also have issues and as we had to make a choice, chose that presented here. However it does suggest that more work is needed to unify the concept of generic function (and specialiser) across all programming languages.

Figure 17 shows the measurements for DR$_{ave}$ with the definition of generic function as described above. All of the measurements are at least 1 or greater. The smallest is in fact 1 (jasml), indicating that no methods in this application are

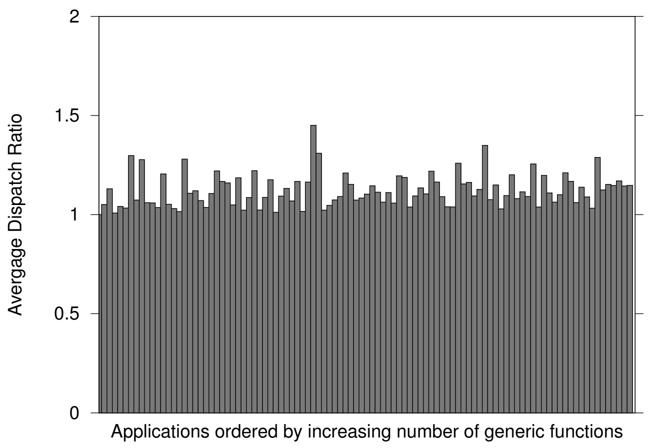

Figure 17. Average dispatch ratio (DR_{ave}) for Java applications.

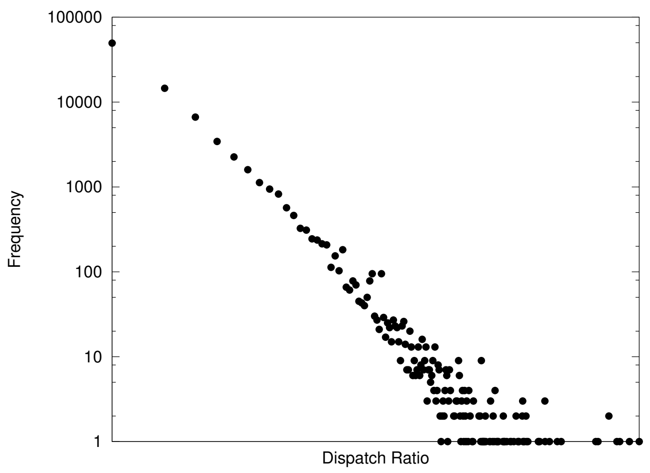

Figure 18. Dispatch ratio (DR) percentage distribution for Java applications, log-log scale.

overridden. The largest is 1.44 (proguard). The median is 1.11 (marauroa).

Figure 18 shows the distribution of dispatch ratio (DR) over all generic functions over all Java applications analysed shown on a log-log scale. In all, there are 1,927,036 generic functions represented, of which 4.97% have more than 1 concrete method. The generic function with the most concrete methods (926) is from eclipse. The distribution shows the classic power law shape (along the same lines as Figure 6 in Section 4).

Finally, the results for average choice ratio (CR_{ave}) are shown in Figure 19. The smallest value is 1.0 (for jasml, as we would expect from the previous results), the largest is 77.1 (jruby), and the median is 3.2 (jspwiki).

One point to note is that we have presented the Java results in order of application size as measured by number of generic functions. There is no obvious trend in any of the measurements with respect to size.

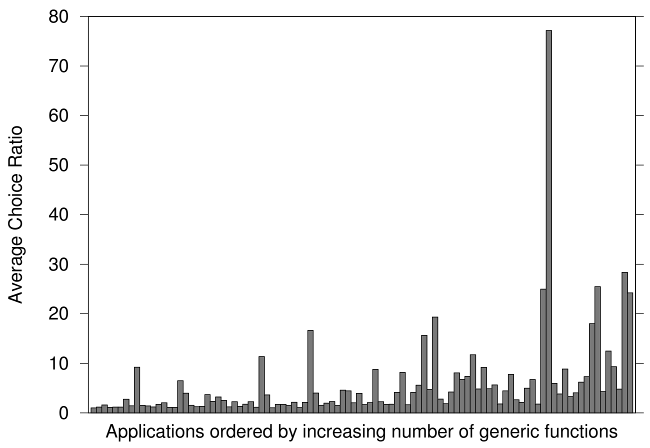

Figure 19. Average choice ratio (CR_{ave}) for Java applications.

6. Discussion

There are a number of inferences which can be drawn from the results presented in the last two sections. Perhaps the most obvious is that many of the metric values are low: other than CMUCL and McCLIM, every language we measured had more than 60% monomorphic generic functions; less than 10% of functions dispatch on two or more arguments (Figure 9). This is reflected in dispatch ratio DR_{ave} values: no language had more than 2.5 concrete methods for each generic function (Figure 11). Furthermore, the DR_{ave} values for the multimethod languages (1.50–2.51, except Nice, 1.36) exceed those for Java (median 1.11, max 1.44). We may also see some effects of the maturity of applications being measured. CMUCL and McCLIM are the most mature of the 9 multiple dispatch applications, and they exhibit the most dynamic dispatch (around 70% in Figure 8, whereas the next closest is less than 40%).

The choice ratio CR_{ave} provides an alternative view of the amount of dispatch in the systems we studied: counting how many alternative concrete methods could have been reached by a dispatch. Here again we see Java's values are consistently lower than those of the multimethod languages, although the figures are larger overall. The larger CR_{ave} values — even in Nice, every method on average could have dispatched to three alternative methods — not only demonstrates the skewed dispatch ratio distribution shown in Figure 5, but also demonstrates the value of the CR metric: while dispatching does not appear that important when measured by *generic functions*, it is more important measured by *methods* (because, of course, one dispatching generic function will have at least two methods to dispatch to).

Monomorphic vs Polymorphic methods It seems that especially in Java, but also in other 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. On the other hand, the multiple dispatch corpora generally exhibit more *single dispatch* than most of the Java corpus.

Style Comparing RD and DoD metrics in Figure 11 we see that some corpora (primarily McCLIM, Gwydion and OpenDylan, but also LocStack, 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.

Java Idioms Our detailed analysis of Java idioms shows that there is significantly more use of **instanceof** than we expected — recall that we only count methods with multiple applications of **instanceof** to a parameter, meaning that applications to a non-parameter, including fields, and single uses within a method are not counted.

Multiple dispatch is being simulated by use of **instanceof** rather than via explicit double dispatching, and when double dispatch is used, it is in implementations of the Visitor pattern. It is not clear if this is because double dispatch is largely unknown by most programmers, or whether concern over the performance of double dispatch has lead programmers to prefer use of **instanceof** — although double dispatch will often be faster than **instanceof** (Foote et al. 2005; Cunei and Vitek 2005).

We surmise (we cannot tell from just the corpus data) the reasons why programmers seem to prefer **instanceof** to double dispatch. Dispatch is in some sense an implementation issue, but especially in Java, where objects have explicit and documented interfaces, dispatching methods pollute their classes interfaces, reducing classes cohesion and increasing coupling. Although **instanceof** cascades may be slower than double dispatching, and are certainly less extensible, by being localised to a single class they are significantly more straightforward to code than double dispatching. This may account for the relative popularity of each idiom.

Multiple dispatch benefits Adding multiple dispatch to a programming language can help improve the expressiveness by providing a first class alternative to either double dispatch or cascaded **instanceof**. Multiple dispatch is considered one of the possible solutions to the *expression problem* (Wadler 1998; Zenger and Odersky 2005). Clifton et al. show how multimethods can be used to help with binary methods, event handling, tree traversals, and implementing finite state machines (Clifton et al. 2006, Section 5.1).

An Historical Perspective Figure 20 is taken directly from Kempf, Harris, D'Souza, and Snyder (1987) published at OOPSLA'87. This paper evaluates CommonLoops (the figures report on PCL CommonLoops and PCL's BeatriX graphics library). Our studies replicate the corpus analysis from that research, but 20 years later and across a number of languages and metrics. Comparing Figures 5, 7, and 20, the similarity of the distributions is striking. Although both systems share the same heritage — Lisp-based multiple dispatch languages and GUI libraries for those languages — there are also significant differences: we have analysed recent releases of CLOS and McCLIM, versions at least twenty years later than PCL and BeatriX as studied in 1987. The more recent programs are also much larger than their 1987 counterparts: where PCL has 91 generic functions, CLOS has 512; BeatriX has 143 while McCLIM has 2222. Taken together, these results show

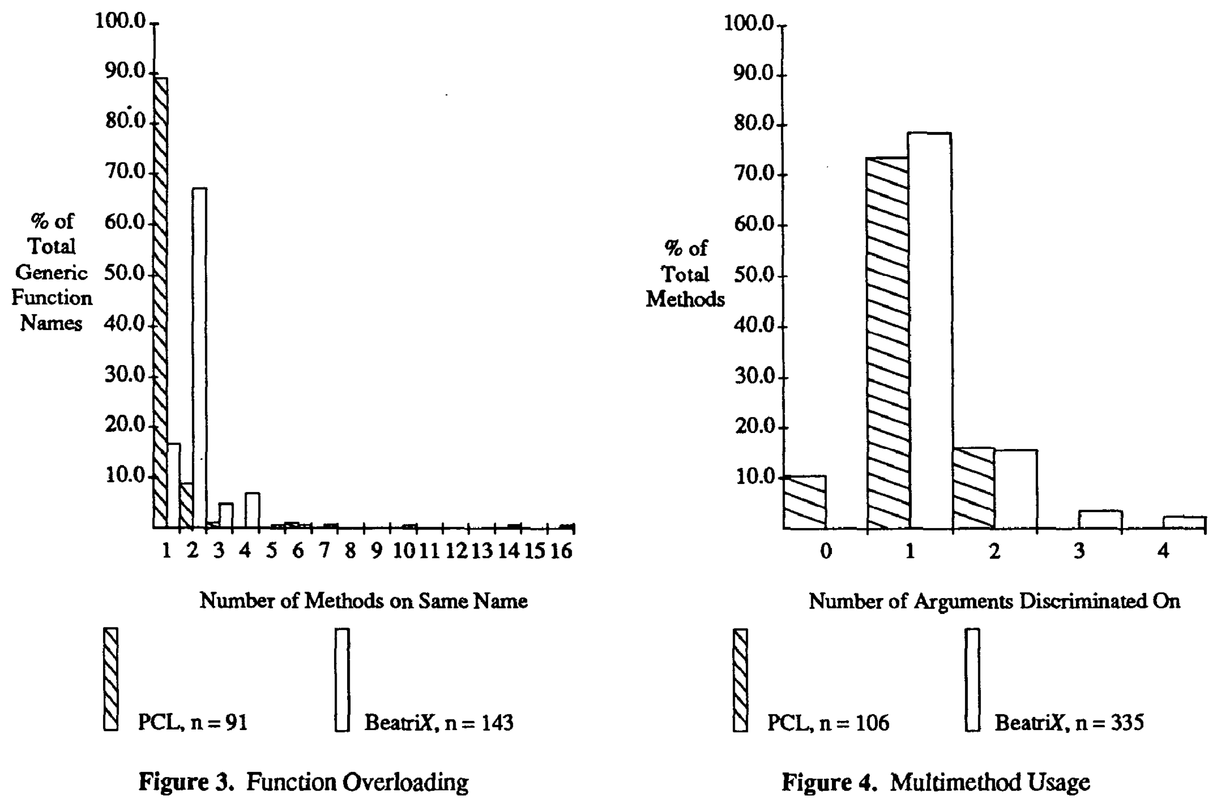

Figure 20. Equivalents of dispatch ratio (DR) and degree of specialisation (DoS) metrics for CommonLoops and BeatriX. Reprinted from Kempf, Harris, D'Souza, and Snyder, OOPSLA'87.

that, at least as far as generic functions are concerned, CLOS programming practice is consistent over the last 20 years.

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 analyses are descriptive: they tell us about how programs are written, but cannot (on their 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 paper, 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, small-scale observational studies, or personal morality (Dijkstra 1968). 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!

7. Conclusion

In this paper we present an empirical study of multiple dispatch in existing languages. To our knowledge it is the first cross-language corpus analysis of multiple dispatch. We define six metrics (Dispatch Ratio, Choice Ratio, Degree of Specialisation, Rightmost Specialiser, Degree of Dispatch, and Rightmost Dispatch) based on a language-independent model of multiple dispatch. We present the values of these metrics for a corpus of programs written in six multiple dispatch languages: CLOS, Dylan, Cecil, Diesel, Nice and MultiJava. We compare our results with an additional study on the use of the double dispatch pattern and cascaded **instanceof** expressions in Java.

In answer to our question ***how much is multiple dispatch used?***, we found that around 3% of generic functions utilise multiple dispatch and around 30% 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.

Considering our single dispatch study of Java programs, to answer ***how much could multiple dispatch be used?***, we found that cascaded **instanceof** expressions are used more often than double dispatch, but that both together are used much less than multiple dispatch in any of the multiple dispatch applications we studied. We consider that this result means that Java programs would have scope to use more multiple dispatch were it supported in the language.

Finally, our study is but a beginning in this line of research. Our language independent model of multiple dispatch, and the definitions of the metrics, proved more difficult to develop that we initially expected; ensuring measurements were comparable across languages required particular care. This suggests there is considerable subtlety in the concepts we are trying to model. We hope this work will inspire more research, including quantitative and qualitative studies of multiple dispatch languages and applications, and design studies of languages supporting multiple dispatch, to further our understanding of multiple dispatch in practice.

A. Corpus

A.1 Applications in Multiple Dispatch Languages

Figure 21 presents the raw measurements used to generate Figures 5–10. It shows percentages of the total generic functions (DR, DoD, RD) or concrete methods (DoS, RS) for frequencies between 0 and 9 and the sum of frequencies equal to or higher than 10. It also mentions the sources where we obtained each application.

A.2 Java Applications

The complete list of Java applications measured in this study is listed below. The format is *application name-version id*. This is release 20080603 of the Qualitas Corpus (Qualitas Research Group 2008).

ant-1.7.0, antlr-2.7.6, aoi-2.5.1, argouml-0.24, aspectj-1.0.6, axion-1.0-M2, azureus-3.0.3.4, c_jdbc-2.0.2, checkstyle-4.3, cobertura-1.9, colt-1.2.0, columba-1.0, compiere-250d, derby-10.1.1.0, displaytag-1.1, drawswf-1.2.9, drjava-20050814, eclipse_SDK-3.1.2-win32, emma-2.0.5312, exoportal-v1.0.2, findbugs-1.0.0, fitjava-1.1, fitlibraryforfitnesse-20050923, freecol-0.7.3, freecs-1.2.20060130, galleon-1.8.0, ganttproject-1.11.1, gt2-2.2-rc3, heritrix-1.8.0, hibernate-3.3.0.cr1, hsqldb-1.8.0.4, htmlunit-1.8, informa-0.6.5, ireport-0.5.2, itext-1.4, ivatagroupware-0.11.3, jFin_DateMath-R1.0.0, jag-5.0.1, james-2.2.0, jasml-0.10, jasperreports-1.1.0, javacc-3.2, jchempaint-2.0.12, jedit-4.3pre14, jena-2.5.5, jext-5.0, jfreechart-1.0.1, jgraph-5.10.2.0, jgraphpad-5.10.0.2, jgrapht-0.7.3, jgroups-2.6.2, jhotdraw-5.3.0, jmeter-2.1.1, jmoney-0.4.4, joggplayer-1.1.4s, jparse-0.96, jpf-1.0.2, jrat-0.6, jre-1.5.0_14-linux-i586, jrefactory-2.9.19, jruby-1.0.1, jsXe-04_beta, jspwiki-2.2.33, jtopen-4.9, jung-1.7.1, junit-4.4, log4j-1.2.13, lucene-1.4.3, marauroa-2.5, megamek-2005.10.11, mvnforum-1.0-ga, myfaces_core-1.2.0, nakedobjects-3.0.1, nekohtml-0.9.5, openjms-0.7.7-alpha-3, oscache-2.3-full, picocontainer-1.3, pmd-3.3, poi-2.5.1, pooka-1.1-060227, proguard-3.6, quartz-1.5.2, quickserver-1.4.7, quilt-0.6-a-5, roller-2.1.1-incubating, rssowl-1.2, sablecc-3.1, sandmark-3.4, springframework-1.2.7, squirrel_sql-2.4, struts-1.2.9, sunflow-0.07.2, tomcat-5.5.17, trove-1.1b5, velocity-1.5, webmail-0.7.10, weka-3.5.7, xalan-j_2_7_0, xerces-2.8.0, xmojo-5.0.0.

Acknowledgments

Thanks to Todd Millstein, who suggested looking at the use of **instanceof** and the rightmost parameter; Bruce Hoult and the other maintainers of Gwydion Dylan and OpenDylan for their help; Daniel Bonniot for support on Nice; Craig Chambers for valuable discussions and help with Cecil and Diesel; and Jeffrey Hightower for providing support for the Location Stack. Thanks are also due to the anonymous reviewers for their comments and suggestions.

This work was funded in part by a Victoria University of Wellington Faculty of Science, Engineering and Architecture & Design Strategic Research Grant, and the New Zealand Foundation for Research Science and Technology supported Software Process and Product Improvement project. Some work was carried out while Tempero was a visiting researcher at the BESQ centre at the Blekinge Institute of Technology, whose support he gratefully acknowledges.

References

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.

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

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.

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.

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

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

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.

Bruno Cabral and Paulo Marques. Exception Handling: A field study in Java and .NET. In *ECOOP*, volume 4609, pages 151–175. Springer-Verlag, 2007.

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.

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

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`.

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

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.

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

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

Edsger W. Dijkstra. Go To statement considered harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

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.

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

Neal Feinberg. *Dylan Programming: An Object-Oriented and Dynamic Language*. Addison-Wesley, 1997.

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.

Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. AW, 1994.

Joseph (Yossi) Gil and Itay Maman. Micro patterns in Java code. In *OOPSLA*, pages 97–116, San Diego, CA, USA, 2005. ACM Press.

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.

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

Daniel H. H. Ingalls. A simple technique for handling multiple polymorphism. In *OOPSLA*, pages 347–349, Portland, OR, USA, 1986. ACM Press.

Warwick Irwin. *Understanding and Improving Object-Oriented Software Through Static Software Analysis*. PhD thesis, University of Canterbury, Christchurch, New Zealand, 2007.

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

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.

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.

Eric Kidd. Efficient compression of generic function dispatch tables. Technical Report TR2001-404, Hanover, NH, USA, 2001.

David B. Lamkins and Richard P. Gabriel. *Successful Lisp: How to Understand and Use Common Lisp*. bookfix.com, 2005.

Gary T. Leavens and Todd Millstein. Multiple dispatch as dispatch on tuples. In *OOPSLA*, pages 274–287. ACM Press, 1998.

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.

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

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

Warwick Mugridge, John Hamer, and John Hosking. Multi-methods in a statically typed programming language. In *ECOOP*, volume 512, pages 147–155. Springer-Verlag, 1991.

Jens Palsberg and J. Van Drunen. Visitor-oriented programming. In *FOOL*, Venice, Italy, 2004.

Alex Potanin, James Noble, Marcus Frean, and Robert Biddle. Scale-free geometry in object-oriented programs. *Communications of the ACM*, May 2005.

Qualitas Research Group. Qualitas corpus release 20080603. `http://www.cs.auckland.ac.nz/~ewan/corpus/` The University of Auckland, June 2008.

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.

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.

Philip Wadler. The expression problem. Discussion on the Java-Genericity mailing list (see 12 November 1998 post), November 1998.

Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. In *FOOL*, San Diego, USA, October 2005. Also available as Technical Report IC/2004/109, EPFL, Switzerland, December 2004.

Application	Metric	0	1	2	3	4	5	6	7	8	9	10+
Gwydion `http://www.opendylan.org//downloading.phtml`	DR	0	83.36	6.87	3.55	1.74	1.00	0.76	0.74	0.37	0.16	1.45
	DoS	5.74	36.31	31.28	13.77	5.51	1.43	2.72	0.82	1.27	0.30	0.85
	RS	5.74	35.21	31.29	13.25	6.80	0.97	2.45	0.06	2.79	0	1.45
	DoD	83.36	13.90	2.61	0.11	0.03	0	0	0	0	0	0
	RD	83.36	10.66	4.63	0.95	0.32	0.05	0.03	0	0	0	0
OpenDylan `http://www.opendylan.org/downloading.phtml`	DR	0	68.08	14.56	5.83	3.64	2.61	1.21	0.23	0.84	0.47	2.52
	DoS	6.48	68.36	21.75	2.84	0.20	0.35	0	0.02	0	0	0
	RS	6.48	59.96	27.89	4.73	0.43	0.50	0	0.02	0	0	0
	DoD	68.08	25.43	5.88	0.51	0.09	0	0	0	0	0	0
	RD	68.08	18.71	11.15	1.73	0.28	0.05	0	0	0	0	0
SBCL `http://www.sbcl.org/`	DR	0	63.64	21.21	6.34	2.75	1.65	0	0.55	0.83	0.28	2.75
	DoS	7.32	75.15	16.26	1.28	0	0	0	0	0	0	0
	RS	7.32	66.20	22.53	3.60	0.35	0	0	0	0	0	0
	DoD	63.64	31.68	3.58	1.10	0	0	0	0	0	0	0
	RD	63.64	29.20	5.51	1.38	0.28	0	0	0	0	0	0
CMUCL `http://www.cons.org/cmucl/`	DR	0	34.57	57.62	2.54	1.95	1.17	0.20	0	0.20	0.39	1.37
	DoS	28.13	59.36	11.45	1.07	0	0	0	0	0	0	0
	RS	28.13	57.32	11.93	2.33	0.29	0	0	0	0	0	0
	DoD	34.77	61.91	2.54	0.78	0	0	0	0	0	0	0
	RD	34.77	60.94	3.12	0.98	0.20	0	0	0	0	0	0
McCLIM `http://common-lisp.net/project/mcclim/`	DR	0	24.30	59.54	6.75	3.60	1.58	1.49	0.32	0.41	0.14	1.89
	DoS	22.63	60.44	13.67	3.07	0.17	0.02	0	0	0	0	0
	RS	22.63	52.06	19.56	3.83	1.07	0.67	0.19	0	0	0	0
	DoD	27.54	67.24	4.82	0.36	0	0.05	0	0	0	0	0
	RD	27.54	61.79	8.69	1.49	0.23	0.09	0.18	0	0	0	0
Vortex `http://www.cs.washington.edu/research/projects/cecil/www/Release/`	DR	0	67.89	15.87	6.15	3.38	1.73	1.04	0.89	0.43	0.29	2.34
	DoS	12.09	71.65	14.82	1.31	0.10	0.01	0.01	0	0	0	0
	RS	12.09	70.13	14.93	1.87	0.52	0.42	0.03	0.01	0.01	0	0
	DoD	67.89	28.18	3.55	0.37	0.02	0	0	0	0	0	0
	RD	67.89	24.90	5.61	1.24	0.28	0.03	0.03	0.02	0	0	0
Whirlwind `http://www.cs.washington.edu/research/projects/cecil/www/Release/`	DR	0	72.86	14.35	4.92	2.75	0.96	0.78	0.68	0.44	0.21	2.06
	DoS	42.67	45.10	10.85	1.26	0.11	0	0.01	0	0.01	0	0
	RS	42.67	41.03	12.63	2.92	0.44	0.25	0.03	0.01	0.02	0	0
	DoD	72.86	23.20	3.42	0.38	0.12	0	0	0	0.02	0	0
	RD	72.86	19.89	5.32	1.29	0.38	0.19	0.03	0.02	0.02	0	0
NiceC `http://nice.sourceforge.net/`	DR	0	86.57	7.69	2.79	0.84	0.42	0.25	0.25	0.17	0.08	0.93
	DoS	70.03	27.24	2.60	0.12	0	0	0	0	0	0	0
	RS	70.03	26.01	3.59	0.25	0.12	0	0	0	0	0	0
	DoD	86.57	12.42	0.84	0.17	0	0	0	0	0	0	0
	RD	86.57	12.08	1.10	0.17	0.08	0	0	0	0	0	0
LocStack `http://portolano.cs.washington.edu/projects/location/`	DR	0	93.28	1.83	1.22	0.20	0.61	0.20	0.41	0	0	2.24
	DoS	12.52	75.24	9.93	2.31	0	0	0	0	0	0	0
	RS	12.52	75.24	4.08	8.16	0	0	0	0	0	0	0
	DoD	93.28	5.30	1.43	0	0	0	0	0	0	0	0
	RD	93.28	3.87	1.63	1.22	0	0	0	0	0	0	0

Figure 21. Metrics distributions for each application in corpus: dispatch ratio (DR), degree of specialisation (DoS), rightmost specialiser (RS), degree of dispatch (DoD), rightmost dipatch (RD), expressed in percent.