Efficient Dynamic Method Dispatch on the Java Virtual Machine

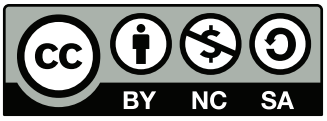
A thesis presented
by

Bastian Müller

in partial fulfillment of the requirements for the degree of
Master of Science

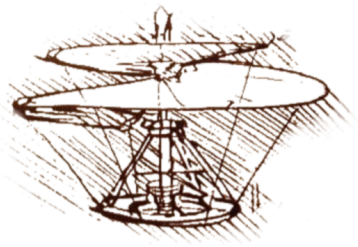
Supervisors:
Peter Sestoft
Hannes Mehnert

IT University of Copenhagen, Denmark
June, 2013



Simplicity is the ultimate sophistication.

— Leonardo da Vinci



Dynamic languages are a playground where one can do good science and have an impact on the real world.

— Jan Vitek

Abstract

Polymorphic operations (methods) provide a common interface to multiple implementations of a particular behavior. Dynamic method dispatch determines at run-time which specific method implementation should be invoked at a particular call site. While the majority of programming languages perform single dispatch, i.e., the selection of a method implementation based on one type, more advanced dispatch mechanisms exist. Multiple dispatch considers multiple argument types, and predicate dispatch is based on arbitrary predicates.

The Java Virtual Machine (JVM) is a popular target for many language implementations. However, most dynamic method dispatch implementations suffer from poor performance. This problem also prevented the adoption of advanced dynamic dispatch variants.

This thesis is about the efficient implementation of advanced variants of dynamic method dispatch. It investigates current implementation techniques, introduces current implementation approaches on the JVM, and explores the usage of new features added to the JVM, namely the new late-binding instruction `invokedynamic` and support for function pointers (method handles).

The thesis contributes an efficient general-purpose implementation of dynamic multiple and predicate dispatch, that does not require any modification to the JVM specification or implementations. The solution is powerful and is especialy suitable for dynamic languages. An evaluation shows that the new multiple dispatch implementation performs better than existing implementations in other dynamic languages, especially for polymorphic call sites. For the purpose of this thesis the dynamic language Lila was developed. It demonstrates the use of the new features of the JVM and serves as a testbed for the implementation and evaluation of the resulting dynamic method dispatch solutions.

Acknowledgments

First of all, I would like to thank my supervisors Peter Sestoft and Hannes Mehnert for their guidance and help during my studies and especially during the process of this thesis, providing me with excellent input and feedback.

I am grateful to Hannes and also Julian Stecklina for introducing me to the Lisp world and encouraging me to explore programming languages and their implementation.

Foremost and above all, I owe my deepest gratitude to my family. They always encouraged my decisions and provided me with invaluable love and support for my goals throughout the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Polymorphic operations (methods) provide a common interface to multiple implementations of a particular behavior. They improve code reuse, extensibility and maintainability. Method dispatch determines the specific method implementation that should be invoked at a particular call site. Static method dispatch selects the implementation at compile-time, while dynamic method dispatch is performed at run-time.

The majority of programming languages perform single dispatch, i.e., the selection of a method implementation based on one type. For example, in traditional object-oriented languages, the method dispatch is based on the class of the receiving object to which a message is sent. Some languages provide more advanced dispatch mechanisms, such as multiple dispatch, which considers multiple types, and predicate dispatch, which is based on arbitrary predicates.

The Java Virtual Machine (JVM) has been in development for almost two decades and implementations feature state-of-the-art just-in-time (JIT) compilation and garbage collection technologies, allowing it to run Java programs as fast as traditional compiled languages. It is widely deployed and programs only need to be compiled to a standardized bytecode. These advantages lead many language implementors to consider the JVM as a target platform. However, the method dispatch mechanisms in most implementations of dynamic languages suffer from poor performance. This problem also prevented the adoption of advanced dynamic dispatch variants.

This thesis investigates how advanced variants of dynamic method dispatch can be efficiently implemented on the JVM, in particular dynamic multiple dispatch and predicate dispatch. It includes an overview and evaluation of implementation techniques, and current implementation approaches on the JVM. It contributes an efficient implementation based on latest features added to the JVM that demonstrates that advanced dynamic dispatch can be efficiently implemented and is therefore practical. It also shows how existing implementations of advanced dynamic dispatch on the JVM can be improved significantly with little effort.

1.1 Motivation

A considerable amount of research has been conducted on the nature, usage and implementation of multiple and predicate dispatch, but they have not yet found their way into mainstream languages. It is commonly assumed that their implementation is difficult and inefficient, and therefore impractical.

The JVM was designed for the needs of Java, so method invocation instructions only perform single dispatch, are based on names, and contain static type information. Dynamically typed and dynamic languages often have different semantics, e.g., method binding may not be solely based on names. Implementing custom dynamic method dispatch

efficiently on the JVM is difficult. Implementations often use intermediate methods performing the dispatch and invocation, or Java's reflection mechanism, which enables the invocation of arbitrary methods at run-time. Both solutions perform poorly, as the JVM cannot apply optimizations, or checks have to be performed on each method invocation. This performance problem has therefore also prevented the adoption of advanced dynamic method dispatch variants.

To improve the support for dynamic language features, the late-binding instruction `invokedynamic` and method handles, typed method pointers, were added to Java 7. The new instruction allows linking a call site the first time it is invoked through a user-defined method, and also relinking it subsequently. JVM implementations are able to perform optimizations for such late-bound calls, so invocations are faster than reflection. An early evaluation of the new late-binding instruction showed it is well-suited for the purpose of implementing advanced dynamic method dispatch. Its practical use has not been widely investigated and documented yet, and dynamic method dispatch is one of the features benefiting most. Also, existing implementations of advanced dynamic dispatch only focus on static programming languages, mainly because the implementations in the setting of dynamism was impractical.

1.2 Purpose

The purpose of this thesis is to demonstrate that it is feasible to efficiently implement dynamic method dispatch on the JVM, in particular dynamic multiple and predicate dispatch, as dynamic single dispatch was already intensively researched, and many JVM implementations already feature very efficient dynamic single dispatch.

The goal of this thesis is to analyze and evaluate current approaches of implementing dynamic dispatch, and develop a new, efficient approach for advanced variants, primarily based on the new features added to the JVM. The main contribution of this thesis is a prototype implementation of dynamic multiple and predicate dispatch. This will serve as a demonstration of their practicality, and will hopefully improve and increase the adoption of advanced dynamic method dispatch for languages running on the JVM. There is no focus on the integration of these dispatch variants with Java's syntax and semantics.

1.3 Organization

The organization of the remainder of this thesis is as follows. Chapter 2 presents background information. This includes a description of a program's life-cycle, dynamism, typing, object orientation and inheritance. Chapter 3 describes the concept of method dispatch, common variants and discusses implementation considerations. Chapters 4 to 6 survey the three major variants of method dispatch, i.e., single, multiple and predicate dispatch. Each variant is described, and implementation techniques are discussed. In chapter 7 the Java Virtual Machine is presented. It contains an overview over the current features and implementation approaches for dynamic dispatch. Following that, new features and their possibilities are explained. Lila, the programming language that was developed for the purpose of this thesis, is presented in Chapter 8. An overview describes its features, followed by a description of their implementation. Chapter 9 contributes and evaluates an efficient implementation of dynamic multiple and predicate dispatch. In Chapter 10 related work is discussed. Finally, Chapter 11 concludes the thesis and suggests future work.

1.4 Practical

The source code of this thesis is available at `http://github.com/turbolent/lila`.

Listings containing JVM bytecode are output of the Java class file disassembler `javap`. For reasons of readability, the containing class and the package name were shortened. In Java listings, the access modifiers were intentionally omitted.

The experiments were performed on a PC with a dual core 2.67 GHz Intel i5 processor and 8 GB of memory, running Ubuntu 13.04 (Linux 3.8.0). For the experiments the following software was used: The latest preview build (1.8.0-ea-b91) of the Java Development Kit and Java Runtime Environment from Oracle (64-bit), OpenDylan 2012.1 (64-bit), SBCL 1.1.1.0, ABCL 1.1.1, Groovy 2.1.4, and JPred 2008-01-25.

Chapter 2

Background

This chapter presents fundamental concepts which are central to this thesis. It includes a description of a program's life-cycle, dynamism, typing, object orientation and inheritance. It also attempts to clarify concepts whose terminology may be used inaccurately or contradictory in literature.

2.1 Program life-cycle

The life-cycle of a program consists of several phases. During compile time, the compiler translates the program from source code to machine code. This may also be an abstract machine. Also, some virtual machines perform just-in-time compilation, i.e., the program is compiled during the execution. Hence, compile time does not not necessarily occur only before the execution. During link time, the compiler resolves symbolic references between separate program components. For example, compilers of static programming languages with support for separate compilation generate object files. Load time occurs when a program or a part of a program is loaded into memory for execution. Again, this does not necessarily occur before execution, but may also occur during execution if the programming language supports dynamic loading. Run-time is the phase when a program is executed. This should not be confused with the runtime, the component that provides fundamental functionality for the execution of the program.

The life-cycle of a program does not necessarily consist of all these phases, and they do not necessarily occur in sequence. For example, a interpreted languages may not have a compile-time phase, and linking may be performed during loading or even at run-time.

2.2 Static and dynamic languages

Dynamic languages allow the modification of the program during run-time. For example, new operations or types might be added. Some languages also allow reflection, the ability of the program to examine itself, i.e., accessing defined operations and types. For instance, programming languages such as JavaScript, Ruby, Python and Common Lisp are dynamic. Even though Java supports reflection and dynamic loading, i.e., adding new classes during run-time, it is generally considered as a static language.

2.3 Static and dynamic typing

Static types are available during compile time. The compiler may use the type information to perform type checking. It is common that variables are typed. Run-time types are available during run-time. Here, the runtime may use the information to perform type checks.

Such run-time types are associated with values, not variables. Types may be program values, enabling reflection, or the runtime may only associate values with ”tags” internally. A language that supports static types is generally considered statically-typed, and a language that supports dynamic types is considered dynamically-typed. However, some languages support both static and dynamic types, so a clear distinction is not always possible.

2.4 Object orientation

Object orientation is is a programming language paradigm where concepts are represented as “objects”, which have data associated with them (e.g., in “fields”, “attributes”, “properties”, etc.). In the majority of languages supporting object-orientation, objects are also associated with behavior. For example, in languages such as Java and C#, operations (methods) belong to types (classes). However, in languages such as Common Lisp and Dylan, operations are separate, first-class objects.

Objects may be typed. Type polymorphism allows using objects of type S safely in a context where an object of type T is expected, if S is a subtype of T ($S <: T$).

Another important feature is inheritance, the reusage of ancestor's behavior. Some object-oriented languages are based on the concept of classes and instances. Objects are instances and are constructed from classes, which contain constituent state and behavior. Some languages are based on prototypal inheritance, cloning, traits or mixins.

In a single inheritance hierarchy each type has at most one direct supertype, while a type in a multiple inheritance hierarchy can have one or more direct supertype.

Literature often uses the terms “type” and “class” interchangeably, especially in the context of method dispatch and type hierarchies, even though method dispatch is not restricted to class-based languages, and languages may have non-class-based type hierarchies. The thesis therefore uses the term “type” wherever possible.

Chapter 3

Method Dispatch

This chapter introduces and explains the concept of method dispatch. The overview includes a description of how and when method dispatch is performed, and presents the three most common variants, i.e., single, multiple, and predicate dispatch. Further, problems related to method dispatch are identified and possible solutions and implementation considerations are discussed.

3.1 Introduction

3.1.1 Polymorphic operations

In contrast to monomorphic constant operations, polymorphic operations (methods) [9] have a multitude of different implementations. Methods improve code reuse, extensibility and maintainability. Different implementations can be invoked implicitly through one interface instead of referring to specific implementations explicitly. Another advantage of methods is extensibility. New behavior can be added to the polymorphic operation and is automatically selected in cases where applicable, without requiring modifications to existing code, such as adjusting names. For example, a rendering engine can provide the method `render`, which automatically selects the appropriate rendering code for a variety of different objects, such as texts, shapes, and images, instead of providing individual monomorphic operations for each type (e.g., `renderText`, `renderShape`, etc.). A call site invoking (calling) the method m with n arguments is of the form:

$$m(a_1, ..., a_n)$$

The method m is also commonly called selector or message, and is usually a name or a first-class value. The definition of a particular implementation has a signature, which consists of the selector and a parameter list. In case of object-oriented programming, method dispatch determines which method should be invoked when sending a particular message m to a receiver (target) o. Hence, traditional object-oriented languages commonly use the following notion:

$$o.m(a_1, ..., a_n)$$

A call of a method involves the following phases:

- **Naming:** The call site has a reference to the method, such as a symbolic name. For example, the Java compiler specifies the method name, the name of the receiver class, and the names of the actual parameter types in the invocation instruction.

- **Linking:** The process of resolving the symbolic name to a specific method is also known as binding. It should not be confused with determining a specific method implementation. For example, even though the Java compiler statically checks the invocation of a method at compile time, it only generates a symbolic name, which is resolved by the Java Virtual Machine at run-time by loading the specified class and looking up the referenced method.

- **Selecting:** Method dispatch (method resolution) is the mechanism of determining the specific method implementation that should be invoked at a particular call site. The most specific applicable method implementation is selected based on information provided by the given set of arguments, the type system, call site context and system state. The dispatch is performed by first determining the set of applicable implementations and then choosing the most specific one, if any.

- **Adapting:** Once the method implementation is selected, a particular calling convention needs to be followed, e.g., arguments need to be passed in a particular order, and type conversion may be necessary. For example, if the method is variadic, i.e., may accept additional arguments beyond the required ones, the additional arguments may need to be passed inside an array, instead of being passed like normal arguments.

Method dispatch may occur in two different phases of the program life-cycle. Static method dispatch is performed at compile-time, while dynamic method dispatch is performed at run-time. A programming language featuring methods may support either static or dynamic dispatch, or both.

3.1.2 Static dispatch

Static dispatch performs the method selection at compile time, based on static information, such as the static type of the receiver object and the static types of the arguments. It is the primary variant of method dispatch in most statically typed languages, such as Java, C++ and C#.

If the static dispatch mechanism is unable to determine a suitable method implementation for a particular method invocation and the runtime supports no dynamic linking, the compiler usually reports an error, as it derives that the invocation is illegal. In case it supports dynamic linking, an implementation may be available at run-time and the compiler may accept the invocation, but possibly warn the programmer about a potential issue with the invocation.

In case the dispatch determines multiple implementations that are equally specific, the invocation is ambiguous. A compiler performing static dispatch may either automatically apply a resolution algorithm, e.g., based on a particular ordering, or produce a compile-time error. The language may support explicit disambiguation in such cases.

As most variants of method dispatch are primarily guided by the language's type system, its expressiveness has a large influence and is a major source for ambiguity. In the case the object system supports multiple inheritance, the declaration order of superclasses could be used to resolve ambiguity automatically.

In Java, static dispatch is available in the form of method overloading. Methods implementations may share the same name, but are distinguished by the number (arity) and types of the formal parameters. The Java compiler statically determines which method

implementation should be invoked based on these two factors. For instance, the following two implementation of the method `print` are distinguishable by their static argument type, even though their arity is equal.

```
void print(String s) { /* ... */ }
void print(int i) { /* ... */ }
```

In the following example, the Java compiler statically dispatches the first invocation to the method `print` with the formal parameter type `int`. The second invocation results in a compile-time error, as no suitable method is defined for the static type `Object`.

```
print(23);

Object x = 23;
print(x);
```

3.1.3 Dynamic dispatch

Dynamic dispatch performs the method selection at run-time. It is useful in cases where the appropriate method implementation depends on run-time information, such as the type of the receiver or the types of the arguments. It is supported in some statically typed and most dynamically typed languages. Dynamic dispatch may also be necessary in cases where the appropriate method cannot be determined statically. For example, in dynamic languages types and method implementations might be modified at runtime.

For instance, Java, C++ and C# support dynamic method dispatch on the run-time type of the receiver object. In the following example written in Java, the run-time type `B` of the value referenced by variable `a` is used to dispatch to the suitable implementation of method `getValue`, instead of the static type `A` given in the variable declaration.

```
abstract class A {
  abstract int getValue();
}

class B extends A {
  int getValue() { return 23; }
}

class C extends A {
  int getValue() { return 42; }
}

// ...
  A a = new B();
  a.getValue(); // ⇒ 23
```

If the dynamic dispatch is ambiguous, the runtime of the language may signal an error by throwing an exception. Another option is the invocation of a special global or method-specific ambiguity method that may be used to perform disambiguation, e.g., by using a custom resolution algorithm to break the tie.

If the dynamic dispatch mechanism is unable to determine an applicable implementation, an error may be signaled. Some languages that support dynamic dispatch invoke a special "not implemented" method instead. For example, Ruby invokes the `method_missing` method of the receiver, passing the message name the receiver was unable to handle. In combination with its dynamic nature, this behavior is often used for the implementation of domain-specific languages.

3.2 Variants

In most programming languages, one of the following three variants of method dispatch is used:

- **Single dispatch** is based on one type, e.g., in object-oriented languages the type of the receiver, which can be considered the first implicit argument. For example, Java, C#, C++, and Smalltalk use single dispatch when performing dynamic dispatch.

- **Multiple dispatch** considers multiple types, usually the types of all arguments (including the receiver). Multiple dispatch is available in languages such as Common Lisp, Dylan and Cecil. MultiJava is an extension to the Java language that supports multiple dispatch.

- **Predicate dispatch** is based on arbitrary user-provided predicates and logical implications between them [27]. Predicates may consider the types of arguments, just like single and multiple dispatch, but may also consider the arguments' state and relationships between them. JPred is an extension to the Java language that supports predicate dispatch.

These variants are increasingly expressive and powerful, but less widely used and more difficult to implement efficiently. Each variant is presented in detail in the following chapters, along with possible implementation techniques. The remaining part of this chapter discusses the implementation considerations for all variants in general, focusing on dynamic method dispatching.

3.3 Implementation considerations

3.3.1 Dispatch time and dispatch structure size

Dynamic dispatch has an apparent runtime overhead, which efficient implementations attempt to reduce. For example, the instruction count of the call site should be minimal to both reduce computation time and also permit optimizations such as inlining, which is commonly limited by a certain code size. Another issue that efficient implementations need to consider is the space required for the representation of the dispatch structure, which grows with the number of types and messages. Naive approaches may have good dispatch time characteristics, but produce prohibitively large representations unsuitable in real-world scenarios.

However, smaller dispatch structures lead to more complex dispatch code and thus increased dispatch time. Hence, a trade-off between the size of the dispatch structure and the dispatch time needs to be made. Also, an increase in expressiveness of the dispatch variant results in an increase in dispatch complexity and thus time and representation size. For example, single dispatch only needs to consider one type, optimally only requiring one dereferencing operation in the dispatch code, while multiple dispatch for a selector with n arguments and a hierarchy of c classes has a naive representation as a multidimensional dispatch table with a size of c^n and requires n dereferencing operations. The size and time factors are impacted further if the language supports multiple inheritance.

3.3.2 Extensibility

Implementation techniques are considered static if their dispatch structure is precomputed at compile-time to aid the dispatching process and minimize the dispatching time at run-time. In contrast, dynamic techniques precompute no or only some information, and incrementally update the dispatching structures at run-time.

Some programming languages are run-time extensible, i.e., the language is dynamic or it supports dynamic loading, so new code with additional types and methods might be loaded or the program performs changes to the sets of classes and methods. For instance, the most common change is the addition of new types and methods. In that case, the compiler and runtime cannot assume they have knowledge of all types and all methods, or that these sets stay constant. Hence, a dynamic technique is necessary instead of a static technique, as compilers are not able to pre-compute dispatching structures at compile-time. Even if structures are partially pre-computed, they will be less optimized. For example, in a closed-world assumption, negative information can be used, which is not possible in an open-world assumption. However, static techniques are still applicable if they are adjusted to support run-time extensibility by not hard-coding information, like class and method identifiers, and loading it dynamically. As the dispatching structures have to be possibly recomputed in a dynamic environment, also the time required for the construction and update of the structures needs to be minimized.

3.3.3 Separate compilation

Separate compilation is a related problem that dynamic dispatching implementations need to consider. Many programming languages allow the compilation of individual source units (e.g., modules) into binary objects. For example, Java source files may be individually compiled into class files and the Java compiler only requires information about code the source file uses, but not the entirety of the resulting program. In that case, static techniques suffer from the same problems as mentioned before. Also, global structures cannot be used.

3.3.4 Optimization

A compiler may be able to optimize a dynamic invocation if the number of actual possible targets can be statically reduced. In some cases it may perform static dispatch, for example, if it can ensure that the call site will only encounter one particular type or only one method implementation will be invoked. This is for example the case if the call site's static type is also the only possible type, because the programmer declared that the class may have no further subclasses, or no other method implementations for subclasses are possible,

because the programmer declared that the method cannot be overridden in a subclass. This concept is commonly known as sealing.

For example, Java provides the `final` keyword for methods and classes. Similarly, C# provides the `sealed` keyword for this purpose. The dynamic language Dylan provides the `sealed` keyword to let programmers declare which extensions are possible after the program is compiled. The compiler may also use static analysis to infer types and make static assertion about the program. For example, if a dynamic method invocation is guarded by a type comparison, the compiler can reduce the dynamic dispatch to a static dispatch, as it can derive the invocation will only ever have to perform the dispatch for this particular type. If the method invocation is guarded by an type membership test, it can reduce the set of possible target method implementations.

```
1 void doSomething(Object x) {
2   // ...
3   if (x.getClass() == Person.class)
4     x.greet();
5   // ...
6   if (x instanceof Car)
7     x.repair();
8 }
```

An advantage of dynamic techniques is that they can exploit run-time information. For instance, the runtime may use type and method information to update and optimize the dispatch process and dispatch structures [40]. The information may also be determined by profiling the program while executing. Dispatch procedures can keep record of encountered types and dispatched targets and optimize call sites even more, potentially even inlining call sites. As many call sites commonly only encounter few types and target few method implementations, this optimization is especially useful.

The shape of the call site, which will change to a certain degree depending on how many actual method implementations it is targeting, is referred to as call site morphism. Monomorphic call sites only ever invoke one target and polymorphic call sites target few targets. A call site is considered megamorphic if it is invoking more than a predetermined number of different method implementations.

3.3.5 Invocation of next most specific method

Many languages support invoking the next most specific method implementation from a particular method implementation. For example, Java allows calling the method defined in the superclass through the `super` keyword. Common Lisp provides the function `call-next-method` to call the next most specific method implementation. To enable such behavior, the dispatching code not only needs to determine the most-specific implementation of the method, but also keep a full list of all applicable methods, ordered by specificity, or have a mechanism to generate it when requested.

Chapter 4

Single Dispatch

In this chapter the single-type variant of method dispatch is presented. The chapter focuses on the implementation considerations and techniques, as it mainly serves as a reference and background for the more advanced dispatch variants presented in the following chapters, whose implementation techniques are mainly based on those of single method dispatch.

4.1 Introduction

Single dispatch determines the most specific applicable method implementation for a given selector (method) based on a single type, i.e., that of the receiver on which the selector is invoked. In class-based languages, the selection is based on the inheritance hierarchy, which is created through subclassing. Method implementations provided for a particular class are also applicable for subclasses, and method implementations defined on subclasses are more specific than inherited ones. For example, single dispatch is used in many statically-typed languages, such as Java (single inheritance) and C++ (multiple inheritance), and dynamically typed languages, such as Ruby (single inheritance) and Python (multiple inheritance). It is also commonly used in prototype-based languages, such as the dynamically typed language JavaScript. For example, in Java the invocation of an instance method performs dynamic single dispatch on the receiver object by default. In C++ and C#, methods are invoked using static single dispatch by default, i.e., the static type of the receiver object is used for dispatch. Dynamic single dispatch is used if the method definition contains the `virtual` modifier.

4.2 Implementation

In general, there are two major categories of implementation techniques for single dispatch: table-based techniques and cache-based techniques. Table-based techniques record method implementations in a table and the dispatch code determines an index into the table based on the selection process. The dispatch code in cache-based techniques determines whether the method implementation for a specific call site has been resolved before, in which case the cache entry is used. Otherwise, the method implementation is computed, usually through a search, and the dispatching result is recorded in the cache for later reuse.

Multiple inheritance can be a source of ambiguity. For example, given the diamond-shaped class hierarchy in Figure 4.1 and the set of method implementations $\{m_1(B), m_2(C)\}$, the method invocation $m(D)$ is ambiguous, as neither of the two types B and C is more specific. In C++ the result is a compile-time error.

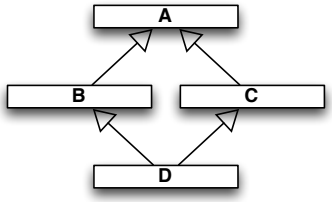
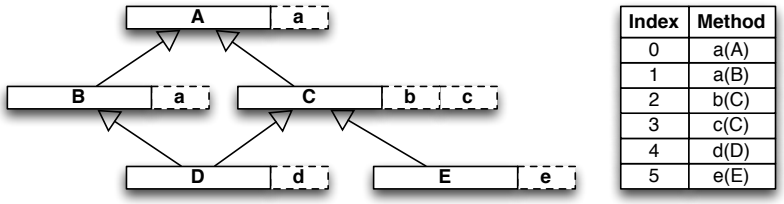


Figure 4.1: Diamond-shaped multiple inheritance hierarchy

The following examples are based on the hierarchy of classes and methods shown in Figure 4.2.



Index	Method
0	a(A)
1	a(B)
2	b(C)
3	c(C)
4	d(D)
5	e(E)

Figure 4.2: Example of a class hierarchy with various method implementations

4.2.1 Table-based techniques

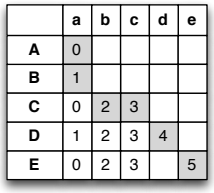
The most important implementation aspect for table-based techniques is the space requirement for the representation of the dispatch table. A naive representation of c classes and s selectors is very large ($c * s$). However, they are very sparse, as most methods are only defined on few classes, and contain duplicate entries. Hence, efficient implementations perform compression through empty entry (null) elimination and deduplication.

Dispatch Table Search

The most naive implementation of dynamic single dispatch is a simple search of the dispatch table. Each receiver type stores references to implementations of the methods it implements. At dispatch time the type of the receiver is searched for the given selector. If the search succeeds, the resulting method implementation is returned. Otherwise the search is continued in the superclasses. This may include aspects such as inheritance, propagation or delegation, e.g., the search continues in the superclass, or the object's prototype is used. This technique has a minimal memory usage, proportional to the number of methods, but is time inefficient. Hashing may be used to optimize the search.

Selector Table Indexing

The "Selector Table Indexing" (STI) technique uses a two-dimensional table to store references to method implementations. The table is indexed by unique identifiers of the classes and selectors. This technique is very time efficient, well suited to multiple inheritance and is also incremental. However, it has a very high memory usage, as no compression is performed. It is therefore not used in practical implementations, but other techniques provide space improvements. Figure 4.3 shows the dispatch table for the running example. Gray cells indicate actual method definitions and white cells inherited definitions.

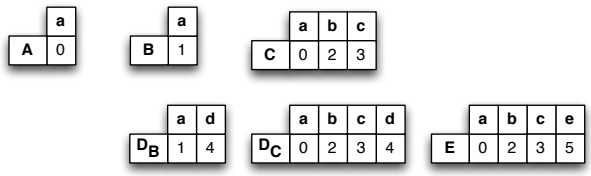


	a	b	c	d	e
A	0				
B	1				
C	0	2	3		
D	1	2	3	4	
E	0	2	3		5

Figure 4.3: Dispatch table for the running example

Virtual Function Table

Virtual function tables (VFT) [26], also known as virtual method tables, are commonly used in languages like C++ and C#, and is also used to implement interface dispatch in some Java Virtual Machines [3]. Each object has references to virtual method tables that contain references to the implemented methods for the particular object. As the set of method implementations is the same for a particular class, the virtual method table is commonly shared for all instances of a certain class. Dispatch tables of subclasses inherit the locations of method implementations, i.e., all inherited methods appear also in the table and are located at the same position as in the superclass's dispatch table. Thus, dispatch is simply performed by looking up the selector's location in the object's dispatch table. In the single inheritance version, the object only references one method table. In the multiple inheritance version, each object has references to multiple method tables, one for each of the superclasses of the object's class. Figure 4.4 shows the dispatch tables for all classes of the running example. Note that class D has superclasses B and C, and it exists a conflict for method a. Therefore, two dispatch tables are produced for class D: The dispatch table D_B is used when the receiver is viewed as an instance of class B, and table D_C is used when the receiver is viewed as an instance of class C.



	a
A	0

	a
B	1

	a	b	c
C	0	2	3

	a	d
D_B	1	4

	a	b	c	d
D_C	0	2	3	4

	a	b	c	e
E	0	2	3	5

Figure 4.4: Virtual function tables for the running example

The single inheritance version is both very time efficient (constant) and has low memory usage (optimal null elimination). In contrast, the multiple inheritance version has large space and time overheads. The technique supports separate compilation, as the dispatch tables of a class can be constructed only using information of its superclasses. However, it is only suitable for statically typed languages.

Selector Coloring

The "Selector Coloring" (SC) [51] technique compresses the two-dimensional table used in STI through null elimination. Two selectors are allowed to share the same index (color) as long as no class implements both selectors. The dispatch code is thus using the selector's shared color for indexing into the dispatch table, instead of its unique identifier.

Coloring is non-incremental and therefore not suitable for dynamic environments and languages supporting dynamic loading. An efficient implementation is difficult, as it is computationally hard to assign colors to selectors optimally – the problem is equivalent to graph coloring, which is NP-complete. A heuristic may still find a good approximation [22]. Compared to STI, the dispatch table is smaller, but it still contains null entries.

Selector coloring is suitable for dynamically typed languages, but due to the nature of the compression, dispatch may return a method implementation unrelated to the selector. Therefore, a prologue checking the selector needs to be added to each method implementation. The prologue adds to the code size and will thus possibly impact optimizations as mentioned earlier in the implementation considerations. Figure 4.5 shows how the previous example of a STI dispatch table can be compressed through SC.

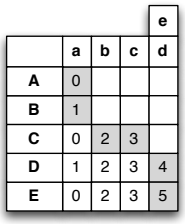


Figure 4.5: Selector coloring applied to the running example

Row Displacement

"Row displacement" (RD) [19] is another null-elimination technique aimed at compressing the two-dimensional dispatch table as generated by STI into a one-dimensional array. The row of each selector is displaced so that each column only contains one method, i.e. non-empty and empty entries overlap. Since the displacement offset is used as the class identifier, it needs to be unique. The offsets for all selectors are stored in an index array. By minimizing the amount of empty entries, the size of the final array is minimized. Displacing selectors instead of classes results in better compression rates [20].

The technique supports multiple inheritance and is also suitable for dynamically typed languages. Like in Selector Coloring, a prologue is necessary to ensure that the method implementation matches the selector. Memory usage is significantly reduced by null-elimination and the dispatch is as time-efficient as Selector Coloring. In a non-incremental setting, the technique is able to perform several optimizations. In Figure 4.6 RD is applied to the previous example of a STI dispatch table.

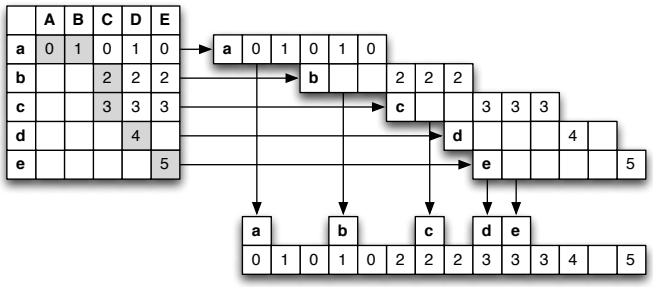


Figure 4.6: Row displacement applied to the running example

Compact Dispatch Tables

The "Compact Dispatch Tables" (CT) [68] technique improves the compression of STI dispatch tables compared to SC and RD and aims at both duplicate elimination and null entry elimination by performing a number of optimization steps. First, selectors are separated into two groups with separate dispatch tables: standard selectors, which are implemented only on a particular class and its subclasses, and conflict selectors, which are only implemented in unrelated classes, but not in a common class. Second, the table is trimmed by removing trailing null entries. As the size of the tables can now differ, the compiler needs to prevent indexing errors by producing appropriate code. Third, selectors that have a disjoint set of classes are aliased with the same offset. Like in the previous approaches, run-time type checks are needed for dynamically typed languages. After aliasing, tables can be shared. As many tables are highly similar and only differ in a small way, they are merged by overloading entries. Finally, post-processing orders entries depending on inheritance. The technique generates very compact dispatch tables and has thus very low memory usage. However, many different dispatch sequences need to be generated resulting in a large code size. Dispatch table size depends on the amount of overloading. With increased overloading, more optimal tables are generated, but dispatch performance decreases.

Compact Selector-Indexed Dispatch Table

The "Compact Selector-Indexed Dispatch Table" [68] technique is an improvement of the earlier developed CT technique. The overloading and trimming steps are replaced with a partitioning step that implements sharing of tables by sharing portions of the tables, i.e., common and different parts are separated into individual tables. The replacement results in constant dispatching time, and less specialized dispatch code needs to be generated. The dispatch tables increases slightly, but code size is reduced.

Incremental Table-Based Method Dispatch

Algorithms and data structures providing incremental dispatch table modification were introduced in [37]. Meta information is stored and used to efficiently update the data structures when the sets of methods or classes changes. The algorithms are general and can be applied to the previously presented implementation techniques.

Type Slicing

Two dispatching techniques [71] are based on type slicing [34], an algorithm based on interval containment. The CT_d dispatching techniques are a set of algorithms (CT_1, CT_2, etc.) that are improved and generalized versions of the CT technique. The index d can be considered a space/time trade-off parameter. The dispatching code is constant-time and each variant requires d memory dereferencing operations. Through a precise performance analysis of different variants of CT, optimal partitioning values were determined. It is suitable for both single and multiple inheritance. An incremental version is available that updates the dispatching data structures in optimal time. The TS dispatching algorithm generalizes the interval containment technique to multiple inheritance. It is very efficient, with better space requirements, faster dispatch structure creation time and faster dispatching time than the RD dispatching technique. However, dispatching time is not longer constant, but logarithmic to the number of method implementations.

4.2.2 Cache-based techniques

Inline Cache

Inline caching (IC) [17] takes advantage of the fact that dynamic dispatch lookups for a particular call site often result in the same target method implementation. The result is directly cached in the dispatch code at the call site and can be reused in subsequent invocations. The lookup is thus ideally replaced with a less expensive check, reducing the dispatch overhead significantly. If the call site is uninitialized or the cache is missed, a lookup needs to be performed, commonly through a dispatch table search. The technique was invented for and is the standard implementation technique of Smalltalk.

Polymorphic Inline Cache

Polymorphic inline caches (PIC) [39] extend inline caches by including more than one lookup result for each call site as a linear search. This reduces cache misses and improves polymorphic call sites significantly. Searches can be improved by taking the frequency distribution into account. This can be done by collecting dynamic profile information. The technique was initially developed for Self and is now used in several virtual machines, such as the JVM and JavaScript virtual machines (e.g. V8 [35]). An analysis [21] has shown that adaptive techniques such as inline caches are more efficient than static techniques such as STI, SC and RD. However, these techniques are still required as efficient lookup strategies, i.e., a combination of inline caches and compact dispatch tables improves dispatch performance even further.

An important implementation aspect is the size of the cache. A certain limit is required to prevent the cache from growing excessively. The size also influences the search and thus the dispatch time. Figure 4.7 shows a polymorphic inline cache for the call site of an invocation of method a of the running example.

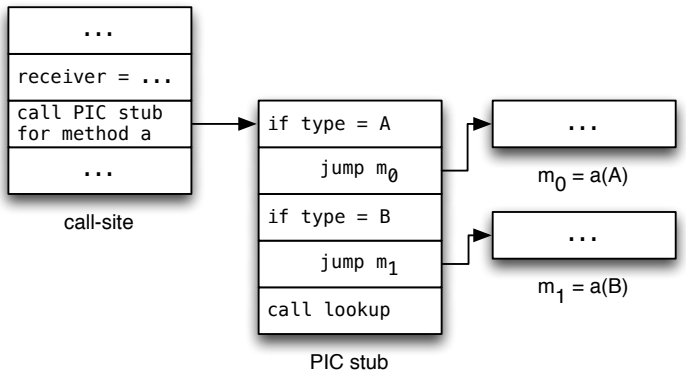


Figure 4.7: Polymorphic inline cache for invocation of method a of the running example

Chapter 5

Multiple Dispatch

Is multiple dispatch a good thing? – Yes.
— Bjarne Stroustrup,
On the Evolution of Languages [18]

This chapter describes the multiple type variant of method dispatch. First, a formal definition is given and possible ambiguity problems are presented. Further, the support of multiple dispatch in various programming languages is discussed in terms of usage, advantages and how the ambiguity problems are handled. Finally, implementation techniques for multiple dispatch are explained.

5.1 Introduction

Multiple dispatch determines the most specific applicable method implementation based on multiple types, commonly those of all arguments, i.e., the method has no single receiver object. The set $\mathcal{M}$ of n method implementations for the method m with a formal (declared) parameter types is defined as:

$$\mathcal{M} = \left\{ m_i \left(T_i^1, \ldots, T_i^a \right) \,\middle|\, 1 \leq i \leq n \right\}$$

The method implementation $m_i \left(T_i^1, \ldots, T_i^a \right)$ is applicable for a method invocation $m \left(T^1, \ldots, T^a \right)$, if and only if each actual parameter type T^k is a subtype of the formal parameter type T_i^k ($1 \leq k \leq a$).

The method implementation m_i is more specific than implementation m_j ($m_i \prec m_j$), if and only if each formal parameter type T_i^k of method implementation m_i is a subtype of each formal parameter type T_j^k of method implementation m_j.

$$m_i \left(T_i^1, \ldots, T_i^a \right) \prec m_j \left(T_j^1, \ldots, T_j^a \right) \Leftrightarrow T_i^k \prec T_j^k, 1 \leq k \leq a$$

If the most specific applicable method cannot be determined, i.e., if more than one method implementation is applicable and neither is more specific than the other, the method invocation is ambiguous. For example, given the type hierarchy $\{A, B\}, B \prec A$ and the set of method implementations $\{m_1 \left(A, B \right), m_2 \left(B, A \right)\}$, the invocation $m \left(B, B \right)$ is ambiguous, because neither of the method implementations m_1 and m_2 is more specific. Therefore, a method ordering is necessary.

Taking the ordering of arguments into account is one solution to this ambiguity problem. Multiple dispatch is asymmetric if the argument order is significant. For example, if a left-to-right precedence is used, implementation m_1 would be selected, as the first

parameter of m_1 is more specialized than that of m_2. Multiple dispatch is considered symmetric if the order of arguments is insignificant, i.e., all dispatched arguments are treated equally. Asymmetric dispatch resolves the ambiguity automatically, while symmetric dispatch leaves the ambiguity resolution to the programmer. This may be a design decision to avoid silent ambiguity and expose ambiguity problems as errors at compile time.

In cases where multiple dispatch is an extension to an existing single dispatch object system where methods are associated with classes, symmetric dispatch may be undesirable, as it does not fit well together with encapsulation [7].

Multiple inheritance can be another source of ambiguity. In a single-inheritance environment the specificity of classes is obvious, as a subclass is more specific than a superclass, but if multiple inheritance is supported, an ordering is required to determine specificity. For example, given a diamond-shaped class hierarchy, such as presented in the previous section in Figure 4.1, and the set of method implementations $\{m_1(B, A), m_2(C, A)\}$, the multiple dispatch invocation $m(D, A)$ is ambiguous, as neither of the two types B and C is more specific. Therefore, a class ordering is necessary to determine the order in which methods should be inherited. This is commonly known as "Method Resolution Order" (MRO).

A study [49] analyzed nine applications to estimate how commonly dynamic multiple dispatch is used in practice. Their results show that 13% – 32% of the methods utilize the dynamic type of a single argument, and 2.7% – 6.5% of the methods utilize the dynamic type of multiple arguments. The majority of the methods (the remaining 65% – 93%) only had one concrete method implementation, i.e., did not utilize any dynamic type, 2% – 20% had two, and 3% – 6% had three concrete method implementations.

5.2 Support in languages

Java provides static multiple dispatch through overloaded methods, which are differentiated by arity and argument types. The example shown in Listing 5.1 demonstrates both dynamic single dispatch and static multiple dispatch. The method implementation of method `draw` is determined by the dynamic type `Display` instead of the static type `Surface`, and the method implementation which has two arguments and the first argument is of the static type `Rectangle` is statically selected.

```
1  class Display extends Surface {
2      void draw(Rectangle rectangle, Color color) {
3          // render a rectangle with a given color
4      }
5
6      void draw(Circle circle, Color color) {
7          // render a circle with a given color
8      }
9  }
10 // ...
11     Surface display = new Display();
12     Rectangle rectangle = new Rectangle();
13     display.draw(rectangle, Color.GREEN);
```

Listing 5.1: Static multiple dispatch in Java through overloading

Double dispatch [41, 33] performs method implementation selection based on two types and is the most often encountered variant of multiple dispatch [49]. For example, it occurs in implementations of event handling systems, where a certain type of event needs to be handled for a particular type of object, or in rendering engines, where the painting of different types of shapes is implemented for different output surfaces. It is also common in mathematical code, where operators generally depend on the types of multiple arguments. For instance, the addition of an integer and a float is different from the addition of two integers.

Most object-oriented languages associate methods with classes and in each class a method implementation can be provided. Hence, these languages often only have support for dynamic single dispatch. This is regularly an intentional design decision, based on the fact that the approach feels natural for most programmers, as it emphasizes the importance of the receiver object in the method invocation. However, it is not always possible to decide on which class a certain method should be implemented. For example, in the case of addition, neither argument type is more important than the other. Also, multiple dispatch needs to be implemented by decomposing it into a series of single dispatch steps, one for each type.

One possible implementation approach is the use of the built-in dynamic single dispatch for the first type and using a cascade of type tests to execute appropriate code for the second type. For instance, in Java the dispatch of the second type can be implemented using a cascade of `instanceof` tests. This approach is commonly used when implementing equality and comparison, e.g., when implementing the methods `Object.equals` and `Comparable.compareTo` in Java. Listing 5.2 shows an example of this approach for a part of a rendering engine.

```
interface Surface {
    void draw(Shape shape);
}

class Display implements Surface {
    void draw(Shape shape) {
        if (shape instanceof Rectangle) {
            /* render a rectangle */
        } else if (shape instanceof Circle) {
            /* render a circle */
        }
    }
}

class EtchASketch implements Surface {
    void draw(Shape shape) {
        if (shape instanceof Rectangle) {
            /* sketch a rectangle */
        } else if (shape instanceof Circle) {
            /* sketch a circle */
        }
    }
}

// ...

```

```
27      Surface display = new Display();
28      Shape circle = new Circle();
29      display.draw(cicle);
```

Listing 5.2: Multiple dynamic dispatch in Java using `instanceof` cascades

A major disadvantage is of this approach is its closed nature: the system is not extensible, e.g., rendering of new types of shapes can only be implemented by modifying the source of all implementations of the `draw` method. This issue of enabling extensibility, i.e., adding support for new cases (classes) and adding new behaviour for them without modifying existing code, is known as the Expression Problem [69].

An alternative implementation approach for languages supporting both dynamic single dispatch and static dispatch through method overloading is the "Visitor" design pattern [33]. It is used to separate an algorithm from the object structure it is operating on, but moreover performs dynamic double dispatch. The instance of the first type is called the "element" and the second is called "visitor".

First, the "accept" method is invoked on the element, passing the visitor as an argument. This step performs dynamic single dispatch based on the type of the element and static dispatch on the visitor. Next, the implementation of the "accept" method invokes the "visit" method on the passed visitor object, passing the element as an argument. This step performs dynamic single dispatch based on the type of the visitor and static single dispatch on the type of the element. In this way dynamic dispatch is performed on both the type of the element and the visitor. Listing 5.3 shows how the previous example can be implemented using the visitor pattern. Here, the "visit" method is `drawOn`, and the "accept" method is `draw`.

```
1  abstract class Shape {
2      abstract void drawOn(Drawable drawable);
3  }
4
5  class Rectangle extends Shape {
6      void drawOn(Drawable drawable) {
7          drawable.draw(this);
8      }
9  }
10
11 class Circle extends Shape {
12     void drawOn(Drawable drawable) {
13         drawable.draw(this);
14     }
15 }
16
17 interface Drawable {
18     void draw(Circle polygon);
19     void draw(Rectangle rectangle);
20 }
21
22 abstract class Surface implements Drawable {}
23
24
```

```
25 class Display extends Surface {
26     void draw(Circle circle) {
27         /* render a circle */
28     }
29     void draw(Rectangle rectangle) {
30         /* render a rectangle */
31     }
32 }
33 // ...
34     Surface display = new Display();
35     Shape circle = new Circle();
36     circle.drawOn(display);
```

Listing 5.3: Multiple dynamic dispatch in Java using the visitor pattern

Still, using the visitor pattern for multiple dispatch has several problems. First of all the code is very verbose, a large part of it is only preparation for the use of the pattern, and many parts are boilerplate, e.g., the second dispatch step needs to be reimplemented in every "visit" method. The double dispatch is not obvious and recursive invocations are even more obscure. Also, the types for which dispatch is available have to be known and specified in advance. This approach therefore suffers from same extensibility problems as the previous one. In both cases, implementing double dispatch is complex and requires additional, manually written dispatch code. This provides a greater risk of coding errors and makes maintenance more difficult.

Some languages, such as Common Lisp, Dylan, Cecil and Julia, provide an expressive first class mechanism for multiple dispatch. For example, the object system of Common Lisp (CLOS) [8] and Dylan [64] have the concept of generic functions: Method implementations are not associated with particular classes, but with a generic function that represents the selector, i.e., it contains the set of all method implementations for specific type combinations. Generic functions are first-class values and can be invoked like normal functions. As a result, they can be used in higher-order programming and thus integrate function-oriented and object-oriented programming. Cecil provides a similar mechanism and uses the notion of multi-methods [10]. Listing 5.4 contains an implementation of the previous example using CLOS. The pair in the parameter list specifies the parameter name and the specializer, in this case a type.

```
1  (defmethod draw ((surface display) (shape rectangle))
2    #| render a rectangle |#)
3
4  (defmethod draw ((surface display) (shape circle))
5    #| render a circle |#)
6
7  (defmethod draw ((surface etch-a-sketch) (shape rectangle))
8    #| sketch a rectangle |#)
9
10 (defmethod draw ((surface etch-a-sketch) (shape circle))
11   #| sketch a circle |#)
```

Listing 5.4: Multiple dynamic dispatch in Common Lisp using CLOS

This solution is also more expressive, as the method specialization and dispatch is declarative, rather than imperative. As no boilerplate code is required, the code is smaller, more readable and better maintainable. It is also obvious for which type combinations method implementations exist. In comparison to the previous two implementation approaches, first-class multi-methods are extensible: new types can be added to the type hierarchy and method implementations for new types can be added without modifications to the existing code. Multi-methods are therefore a possible solution to the expression problem.

Both Common Lisp and Dylan totally order the inheritance hierarchy through a topological sort, using the local ordering of superclasses, i.e., the order in which superclasses are named in the subclass definition. An example illustrating this behavior is shown in Listing 5.5. Dylan uses a deterministic algorithm to compute the class precedence list. In case there is no consistent total ordering possible, a compile-time error is signaled. However, the algorithm is not consistent and may lead to counter-intuitive class linearizations. The C3 superclass linearization algorithm [6] produces consistent linearizations, which are monotonic and preserve the local precedence order, so that direct superclasses are not skipped over.

```
(defclass a () ())
(defclass b (a) ())
(defclass c (a) ())

(defmethod foo ((x b) (y a)) 1)
(defmethod foo ((x c) (y a)) 2)

(defclass d (b c) ())
(defvar x (make-instance 'd))
(foo x x) ;; ⇒ 1

(defclass d (c b) ())
(foo x x) ;; ⇒ 2
```

Listing 5.5: Ambiguity resolution in Common Lisp using MRO

CLOS performs asymmetric dispatch using a left-to-right argument precedence. In contrast, Dylan, Cecil and Julia perform symmetric dispatch. For example, Listing 5.6 shows an example of how method definitions and an ambiguous invocation in Julia results in a warning proposing to add a method for the ambiguous case.

```
julia> f(x::Int, y) = 1
julia> f(x, y::Int) = 2
Warning: New definition f(Any,Int) is ambiguous with f(Int,Any).
         Make sure f(Int,Int) is defined first.
```

Listing 5.6: Ambiguity warning in Julia

Common Lisp, Dylan and Cecil allow invoking further applicable method implementations. CLOS provides the `call-next-method` function, which invokes the next most specific applicable method with new arguments, or the current arguments if none are provided. In Dylan the same behavior is provided through the function `next-method`.

Cecil provides a resending mechanism similar to that of Smalltalk, allowing the programmer to explicitly state the type of certain arguments, so that the dispatch will select the specific method implementation for this particular case.

Common Lisp and Dylan extend typical multiple dispatch and also include a form of value dispatch. In Common Lisp the `eql` keyword can be used to specialize a parameter to a particular instance. As Dylan's type system is more expressive, parameters can be specialized in a number of different ways. The singleton type allows specialization for a specific instance, while the subclass type allows specializing for instances of subclasses of the given class. Lastly, using the union type, the parameter can be specialized for instances of one of the classes in a given set of classes.

Related to Java, a number of languages running on the JVM support multiple dispatch. MultiJava [15, 14] extends Java's syntax and semantics with symmetric multiple dispatch. Xtend is inspired by Java and provides the `dispatch` modifier to enable dynamic multiple dispatch for certain methods. Both languages are static and implement single-inheritance. They compile to Java, generating `instanceof` cascades like previously presented. The dynamic language Groovy supports asymmetric dynamic multiple dispatch and determines method specificity by calculating parameter distances.

Fortress [1] is a research language supporting multiple inheritance and symmetric multiple dispatch. Even though plans for optimizing compiler existed, it was only implemented as an interpreter. The project investigated type checking multiple inheritance in combination with modular multiple dispatch. It also came to the conclusion [65] that avoiding the need for the visitor pattern and providing symmetric multi-method dispatch and parametrically polymorphic methods makes code more flexible and easier to extend.

The design, implementation and evaluation of open multi-methods for C++ is presented in [57, 56]. Special attention is given to the integration with existing language features and rules. The solution resolves ambiguities through link-time analysis and provides constant time dispatch, with double dispatch performing faster than two single method invocations. The proposal is not standardized yet, but a prototype compiler exists.

5.3 Implementation

In general, there are two major categories of implementation techniques for multiple dispatch: table-based techniques, mostly adapted from table-based techniques for single dispatch, and tree-based techniques. A third approach to multiple dispatch is to transform it into a geometric problem on multi-dimensional integer grids [29]. Most of these algorithms are fast, performing dispatch in constant time, and have low space requirements. However, only few are incremental and maintain a list of all applicable methods. The following examples are based on the hierarchy of classes and methods shown in Figure 5.1.

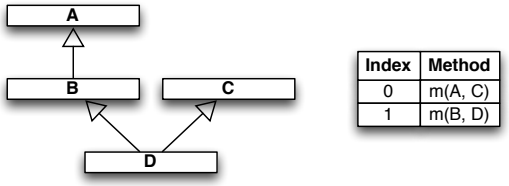


Figure 5.1: Example class hierarchy

5.3.1 Table-based techniques

Similar to single dispatch, it is possible to store method implementations in a table. In the case of multiple dispatch, this table has n dimensions, where n is the arity of the method. Again, the space requirement is the main concern, addressed by compression algorithms. The uncompressed dispatch table for the running example is shown in Figure 5.2.

	A	B	C	D
A			0	0
B			0	1
C				
D			0	1

Figure 5.2: Uncompressed dispatch table for the running example

Compressed N-Dimensional Tables

The "Compressed N-Dimensional Tables" (CNT) [4, 24] technique produces an n-dimensional dispatch table for a method of arity n. The dispatch table is compressed through null-entry elimination and deduplication.

Each column and row only consisting of null-entries is eliminated and identical rows and identical columns are grouped together. As the table is now indexed through type groups, rather than individual types, a mapping from types to type group index is kept in a separate table.

Dispatch is performed by looking up the group index for each type that is dispatched and indexing into the n-dimensional table to find the target method implementation, if any. The auxiliary data structure can be further compressed. Figure 5.3 shows the compressed dispatch table indexed by type groups, and the auxiliary table containing the group index for each type and argument position.

	{C}	{D}
{A}	0	0
{B,D}	0	1

	A	B	C	D
1	0	1		1
2			0	1

Figure 5.3: CNT technique applied to the running example

A naive algorithm produces an empty dispatch table, fills it with entries for all method implementations, and compresses it by scanning the whole table. As construction and compression are very space and time consuming, this approach is impractical. An efficient algorithm computes the final compressed dispatch structures directly from a given set of methods by first analyzing the types to determine empty entries, index groups and most-specific applicable methods.

For each argument position i, the types for which actual method implementations are defined and their influences are determined. In case of the running example, the 1-poles are A and B, D is in the influence of B, and C is neither a pole, nor under influence. The 2-poles are C and D, and A and B are not under influence of any of these poles. Entries for the type T can be eliminated from dimension i if T does not belong to the influence of any i-pole, and entries for type T and the i-pole of T can be grouped.

The algorithm efficiently determines poles and influences. An improved version of the pole analysis is available in [44]. The resulting dispatch tables have low memory usage and the dispatch is constant time. However, the technique lacks support for separate compilation and is not incremental, as all poles need to be recomputed when new methods are added.

Single-Receiver Projection

"Single-Receiver Projection" (SRP) [38] is a constant time dispatch technique that determines the common most specific applicable method of all argument types by maintaining a single-receiver dispatch table for each parameter. For example, for the method implementation $m(A, C)$, the first table specifies the implementation is applicable for type A and the second table for type C. For each type, the partially ordered set (poset) of applicable methods (most-specific first) is stored. The dispatch consists of determining the intersection of all individual applicable methods. For reasons of efficiency, the posets are stored as bitsets, so that the dispatch only requires logically anding the applicable vectors and returning the implementation for the left-most bit that is true.

As single dispatch tables are used, the technique is able to benefit from existing compression optimizations. For example, selector coloring (SRP/SC) can be applied. If instead of using separate tables, all argument positions are stored in a single table, row displacement can be used to efficiently compress it (SRP/RD). As a result, the technique is more space efficient than CNT, but has a slower dispatch time. The space requirement of the single dispatch stage can also be optimized through type slicing [70], which is incremental, but has slightly worse dispatch time.

The main advantage of the single-receiver projection technique is the maintenance of the set of all applicable methods. If no non-incremental compression optimizations are applied, it also has very good support for incremental changes. Figure 5.4 shows the SRP dispatch tables for each parameter (H_1 and H_2) both in poset and bitset representation.

	A	B	C	D
H_1	{0}	{1,0}		{1,0}
H_2			{0}	{1,0}

	A	B	C	D
H_1	10	11		11
H_2			10	11

Figure 5.4: Single-Receiver Projection applied to the running example

Multiple Row Displacement

"Multiple Row Displacement" (MRD) [55] is another technique for compressing n-dimensional dispatch tables. The previously presented row-displacement compression used for single dispatch tables is applied to the table of each dimension to produce one master array. The resulting shift indices are stored in auxiliary index arrays that are also compressed using row displacement.

The technique produces very compressed dispatch tables with comparable space, but better dispatch performance and call site size characteristics compared to SRP/RD and CNT. However, like the single dispatch variant it is based on, it has the problem of returning a wrong method for a type-invalid call site when used in a dynamically typed environment. Also, the technique is not incremental.

Figure 5.5 shows how the initial dispatch table for the running example is first compressed on each level, and finally displaced into the master array.

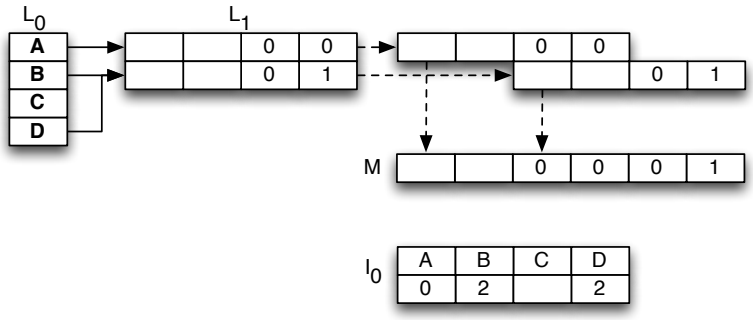


Figure 5.5: Multiple Row Displacement applied to the running example

5.3.2 Tree-based techniques

Compressed Dispatch Tree

The "Compressed Dispatch Trees" (CDT) [23] technique is similar to CNT in the sense that it performs a pole analysis, but instead of using the information to create a compressed dispatch table, a dispatch tree is generated. Each node in the tree performs a type test and the edges are labeled with the different possibilities. The order can either be uniform, i.e., the same order of tested positions is used in the whole tree, or local, i.e., different positions are tested in different subtrees.

The tree can be optimized through unification of vertices and storing the tree in mapping arrays instead of storing pole information inside vertices. The size of the dispatch tree is smaller than that of dispatch tables, as some lines in dispatch tables can still be shared, but this optimization is local, which is trivial in a tree form. However, the technique suffers from the same problems as CNT: it is not incremental and it requires run-time type checks in a dynamically-typed language.

Partial Dispatch

"Partial Dispatch" [5] is a dynamic technique constructing decision trees at run-time. For each method, one shared dispatch tree is created, and for each call site, a specific dispatch trees is constructed, that also acts as a cache and generates profiling information. Each node in the decision tree performs a type dispatch. Specialized nodes have efficient lookup mechanisms (linear search, hash table, etc.), depending on the number of dispatched types.

The type information (e.g., disjointness, sealing) that was statically inferred by the compiler and that is current available in the dynamic environment, as well as the profiling information, is used to optimize the decision trees: They are pruned by reducing and possibly eliminating discrimination steps. For example, based on the current state of the environment, a call site dispatch tree might be transformed into a normal function call performing no method dispatch at all. The incremental addition of new information results in incremental improvements in dispatch performance and size requirements. In addition, the profiling information can be fed back to the compiler, producing even further optimized dispatch trees in the next compilation phase.

Due to its dynamic nature, it solves the separate compilation and open world problems, and is especially suited for dynamic languages. However, it requires the runtime to track dependencies and update dispatch trees when changes in the dynamic environment occur, e.g. class additions or deletions. For example, the disjointness assumption might become invalid if a common subclass is added.

The technique was developed for Dylan in particular, so it also provides nodes for the specializers that go beyond standard type dispatch (subclass, singleton, union, etc., as mentioned before). The core principles however can be simplified and are suitable for any implementation of standard multiple dispatch.

Lookup Automata

The "Lookup automata" (LUA) [13] technique produces a method lookup automaton for each method. An implementation parameter determines dispatch performance and memory usage, which are inversely proportional. The automata can be further compressed by using nested transition arrays [12] as the representing dispatch structure. It is applicable for statically and dynamically-typed languages and is very time and space efficient as it combines the advantages of the lookup automaton (good compression) and compressed dispatch tables (fast dispatch). However, the technique is not incremental.

Chapter 6

Predicate Dispatch

This chapter presents predicate dispatch. First, a formal definition is given and use-cases are presented. Next, the ambiguity and complexity problems are discussed. Furthermore, several programming languages and their support for and approach to predicate dispatch are summarized in an overview. Finally, an implementation technique constructing efficient dispatch trees is described and evaluated.

6.1 Introduction

Predicate dispatch [28] determines the most specific applicable method implementation using predicate expressions. Method applicability can depend on the types of the arguments, similar to single and multiple dispatch, but also on arguments' runtime state, relationships between objects, or current execution context. This allows very fine-grained and dynamic dispatch. Predicate expressions are logical formula consisting of type tests and arbitrary boolean-valued expressions from the underlying programming language. Expressions may also destructure arguments and create bindings available inside method implementations. Logical implications between predicates are used to determine specificity, i.e., a method m_1 overrides another method m_2 if the predicate of m_1 predicate logically implies the predicate of m_2. Predicates are not ambiguous if they are disjoint, i.e., they are are mutually exclusive ($\neg\left(\mathit{pred}_1 \wedge \mathit{pred}_2\right)$). The dispatch mechanism may automatically reason about predicates to determine method specificity.

The set $\mathcal{M}$ of n method implementations for the method m with arity a is defined as:

$$\mathcal{M} = \left\{ m_i\left(\mathit{arg}_i^1, \ldots, \mathit{arg}_i^a\right), \mathit{pred}_i \;\middle|\; 1 \leq i \leq n \right\}$$

Predicate dispatch subsumes single and multiple dispatch, but also ML-style pattern matching and predicate classes.

- **Single dispatch** can be implemented by using a type test predicate for the first argument:

  $m_i\left(\mathit{arg}_i^1, \ldots, \mathit{arg}_i^a\right), \mathit{arg}_i^1 : T_i', T_i' \prec T_i$

- **Multiple dispatch** can be performed by using a type test predicate for each argument:

  $m_i\left(\mathit{arg}_i^1, \ldots, \mathit{arg}_i^a\right), \mathit{arg}_i^1 : T_i'^1, T_i'^1 \prec T_i^1 \wedge \ldots \wedge \mathit{arg}_i^a : T_i'^n, T_i'^n \prec T_i^n$

- **Pattern matching** can be accomplished by type testing and destructuring arguments, creating bindings for subcomponents:

  $m_i\left(\mathit{arg}_i^1, \ldots, \mathit{arg}_i^a\right), \mathit{arg}_i^1 : T_i'^1, T_i'^1 \prec T_i^1 \wedge \mathit{var}_1 \leftarrow \mathit{arg}_i^1.\mathit{sub}_1 \ldots$

Predicate dispatch also extends these dispatching variants. For example, multiple dispatch is normally expressed as a conjunction, but predicate dispatch also allows disjunction and negation. In pattern matching, normally ordering determines overriding and pattern matching expressions are fixed and datatype constructor patterns are used to discriminate values.

Predicate dispatch allows the same functionality, but is open, i.e, additional clauses can be added independently, and type tests support inheritance. Its declarative nature makes it also more expressive. Method applicability can also be ordered explicitly:

$$
\begin{aligned}
& m_1\left(arg_1^1, \ldots, arg_1^a\right), pred_1 \\
& m_2\left(arg_2^1, \ldots, arg_2^a\right), pred_2 \wedge \neg pred_1 \\
& \vdots \\
& m_n\left(arg_n^1, \ldots, arg_n^a\right), pred_n \wedge \neg pred_1 \wedge \ldots \wedge \neg pred_{n-1}
\end{aligned}
$$

Similar to multiple dispatch, some languages use the concept of generic functions, i.e., methods are not associated with a class, but a function holding all method implementations for a particular method. In case the language is an extension of an existing one, the semantics of the language are taken into account. For instance, if it is a class-based language, predicate methods are associated with a class.

Listing 6.1 contains the running example for the implementation section. It contains the definition of the predicate method `isomorphic`, which determines if two binary trees are isomorphic. Another example is presented in Listing 6.2, which shows how the Fibonacci function can be implemented using predicate dispatch.

```
type TreeNode;
class DataNode subtypes TreeNode { left:TreeNode, right:TreeNode };
class EmptyNode subtypes TreeNode;

method isomorphic(t1, t2)
  when t1@EmptyNode
   and t2@EmptyNode
{ return true; }

method isomorphic(t1, t2)
  when t1@EmptyNode
    or t2@EmptyNode
{ return false; }

method isomorphic(t1, t2)
  when t1@DataNode
   and t2@DataNode
{ return isomorphic(t1.left, t2.left)
      && isomorphic(t1.right, t2.right); }
```

Listing 6.1: Predicate method `isomorphic`

```
method fib(n) when n == 0 or n == 1
{ return n }

method fib(n) when n@Integer
{ return fib(n - 1) + fib(n - 2) }
```

Listing 6.2: Predicate method `fib`

However, the high expressiveness of predicate dispatch leads to complexity problems. Testing predicate implication is undecidable, if predicate expressions may contain arbitrary expressions of the underlying programming language. Even if the set of expressions is restricted, the complexity is still co-NP if arbitrary logical operators are allowed between predicates (conjunction, disjunction and negation) [36]. Hence, the predicate expressions and operators are often restricted in implementations. Also, predicate dispatch only determines the single most specific method implementation and therefore does not support invoking next most specific method implementations.

6.2 Support in languages

6.2.1 JPred

JPred [47, 48] extends Java with support for predicate dispatch. Its compiler statically checks all predicate method invocations and ensures all possible scenarios are covered and no ambiguity exists. The extended language is compiled to regular Java. Both steps are performed modularly. Instead of using special-purpose algorithms, JPred uses the automatic theorem prover CVC Lite to reason about predicates. It also supports predicate expressions with linear arithmetic inequalities. Like MultiJava, methods are associated with classes to fit in with the semantics of Java. JPred therefore performs asymmetric dispatch, as the receiver argument is treated specially.

The JPred compiler compiles the predicates of all method implementation for a particular method into a dispatch method, consisting of a cascade of predicate expressions. The compiler also performs exhaustiveness and ambiguity checking. The exhaustiveness checking ensures that for all method invocations the dispatch will result in a valid method implementation, i.e., no method invocation is inapplicable. The ambiguity checking ensures that the dispatch will never result in ambiguity. As it allows the use of n-ary predicates and arbitrary logical operators it suffers from co-NP-completeness.

6.2.2 Predicate-C

Predicate-C [36] is a lightweight C library that provides predicate dispatch. As it is written in a low-level rather than a higher-level language, it can be integrated into into a great range of language runtimes. It does not suffer from undecidability and co-NP-completeness by restricting the set of predicates to unary ones, e.g., comparing a variable with a constant, and logical conjunction, ensuring a polynomial complexity of the method dispatch. If two different method implementations have equivalent predicates, an error is signaled, and if a method call is ambiguous, the first method implementation is invoked. Predicate-C uses the concept of generic functions to group method implementations.

6.2.3 Filtered dispatch

Filtered dispatch [16] is a decidable version of predicate dispatch. Each method is associated with one or more filter functions. A filter function can use arbitrary predicates of the underlying language to map arguments to values which are used for dispatch. Method implementations specify the applicable filter function and specialize on the dispatch value. Dispatch is performed by first determining which of the filters are applicable, then determining which method implementations are associated to these filteres, and finally determining the most specific method implementation by using the value that the filter function produced. Ambiguity is avoided by taking the definition order of the filters into account.

The main concept, mapping arguments through one or more filter functions to one or more values that are used for dispatch using arbitrary predicates, is also used in Clojure. Here, multi-methods are associated with exactly one dispatch function that generates exactly one value that is used for the actual dispatch. Each method implementation specifies exactly one dispatch value for which it is applicable.

Clojure also provides an ad hoc hierarchy system consiting of derivation relationships between names, independent of the type system. The multi-method dispatch considers these relationships when determining applicable method implementations. In cases of ambiguity caused by multiple derivation, the function `prefer-method` can be used by the programmer to give an ordering between method implementations by specifying an ordering between dispatch values.

Even though both variants are decidable and allow the use of arbitrary predicates, they sufffer from the extensibility problem, as new cases cannot be added to the filter / dispatch functions.

6.3 Implementation

One implementation technique for predicate dispatch is a simple search. The method implementations are ordered from most- to least-specific. The dispatch process consists of testing each predicate of each method implementation and the first method implementation for which the predicate test is true is invoked. The search can for example be implemented as a sequence of if-statements. The compiler of JPred is using this approach. The predicates are not optimized, e.g. common expressions of all predicate expressions are not detected and are unnecessary re-evaluated, instead of being reduced to one single evaluation.

Another technique is constructing a dispatch tree. In [11] an algorithm for producing efficient dispatch trees is presented. First, the generic function is simplified. Second, a lookup DAG is constructed which performs the multiple dispatch as a sequence of single dispatches. Finally, a decision tree is constructed for each single dispatch node. The single decision trees first evaluate the node's expression and obtain the ID for the resulting value's type. Then, a binary decision trees consisting of equality and range tests (less-than) determines the target node.

In the canonicalization step, the generic function is translated into a dispatch function consisting of individual cases. All predicates of all methods implementations are transformed into disjunctive normal form (disjunction of conjunctions), only consisting of type tests. Type tests are left untranslated and boolean-valued expressions are translated to type tests against the (possibly conceptual) `True` class. For instance, the generic function of the example consists of the following predicates and method implementations:

```
t1@EmptyNode and t2@EmptyNode → m1
t1@EmptyNode or t2@EmptyNode → m2
t1@DataNode and t2@DataNode → m3
```

The resulting dispatch function consists of the following cases:

```
c1: t1@EmptyNode and t2@EmptyNode → m1
c2: t1@EmptyNode → m2
c3: t2@EmptyNode → m2
c4: t1@DataNode and t2@DataNode → m3
```

Disjunctions can be evaluated in any order, but conjunctions are short-circuting and the expressions need to be evaluated in order. Therefore, constraints between individual expressions need to be determined. In the example, the set of constraints is empty, as the evaluation of both expressions (`t1` and `t2`) are independent.

From the dispatch function, consisting of the individual cases, expressions and constraints, the lookup DAG is constructed through a recursive process that creates individual subtrees based on a set of candidate cases and expressions that are still left to be evaluated. A table memoizes the calls to this recursive construction function, so constructed internal nodes are automatically shared.

For each internal node of the lookup DAG, a single dispatch tree is constructed. First, a unique ID is assigned to all classes. Next, the individual subtrees of the binary decision tree are generated in a recursive manner, based on potentially available frequency data. If the most frequent ID has a relative frequency above a certain threshold, a node performing an equality test is created. Otherwise, the ID that starts an interval and divides the execution frequency in half is selected and a node performing a less-than test is created. This decision tree construction algorithm is similar to the Iterative Dichotomiser 3 (ID3) algorithm [58], as it also aims at equally distributing frequencies. Figure 6.1 shows the resulting lookup DAG and decision trees for the running example.

The algorithm employs a number of optimizations to construct very efficient dispatch. The amount of evaluated expressions is reduced, as two expressions are considered equal if their abstract syntax trees are isomorphic. Furthermore, it makes use of any available static information (explicit and inferred type information) to prune paths in the lookup DAG. The algorithm aims at building short paths by using heuristics to estimate the cost of expression evaluation and evaluating more-discriminating expressions first. Traversal time of the decision trees is also minimized by taking into account dynamic information such as class frequencies. By calculating the constraints between expressions, the evaluation of expressions in conjunctions can be safely reordered. The size of the lookup DAG is minimized by memoizing calls to the subtree construction function and post-processing the final DAG: leaf nodes with the same target methods are merged and preceding single-successor nodes are recursively eliminated.

The main problem of this technique is the non-incremental nature, as the dispatch function can be updated with additional cases, but the lookup DAG and the dispatch trees need to be regenerated.

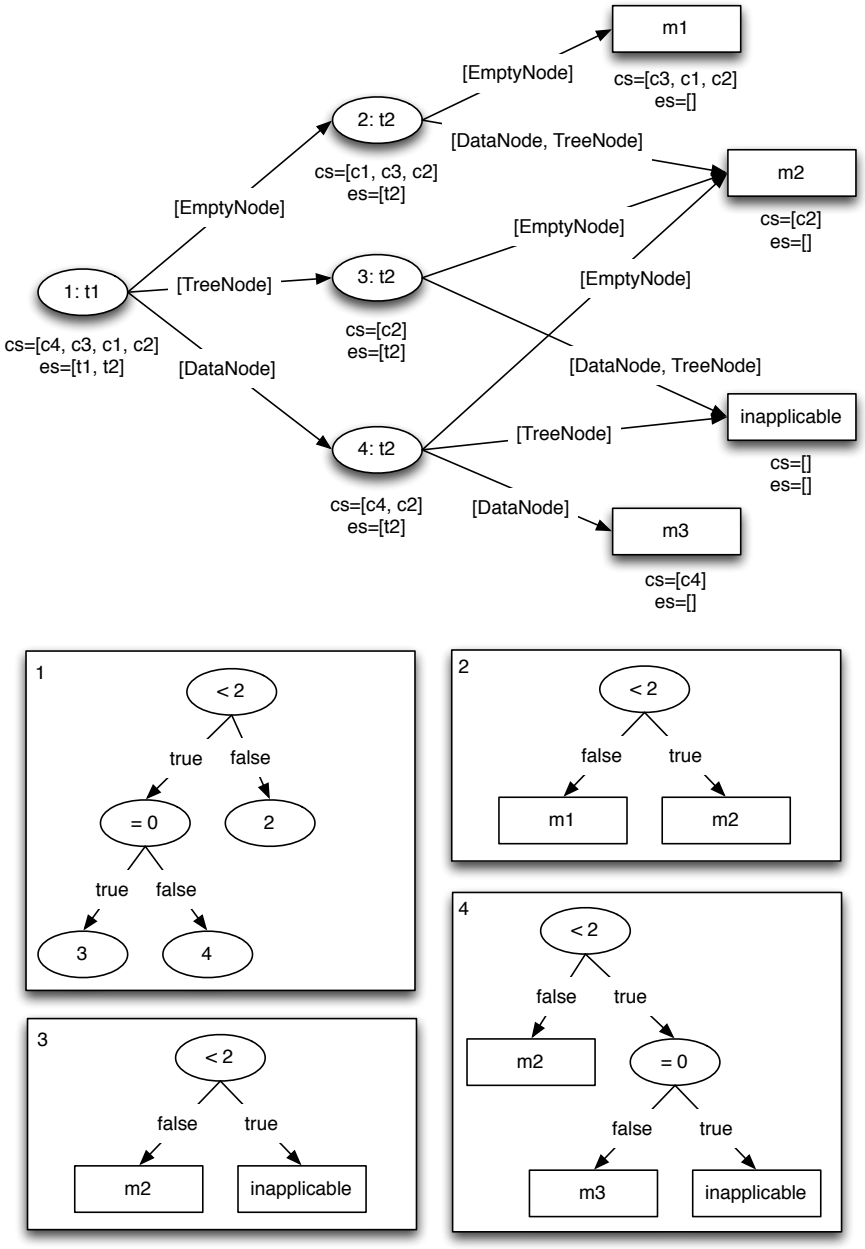


Figure 6.1: Lookup DAG and decision trees for to the running example

Chapter 7

Java Virtual Machine

This chapter presents the Java Virtual Machine. First, its functionality is described and its current method dispatch mechanisms are examined, with a focus on dynamic dispatch, function pointers and related support for other dynamic language features. Following this, the recent additions and improvements in regard to dispatch are described, such as the new method invocation instruction `invokedynamic`, which provides custom late-binding, and support for a form of function pointers (method handles).

7.1 Introduction

The Java Virtual Machine (JVM) [45] is the core of the Java Platform, one of the most popular programming platforms. It is available for devices ranging from embedded appliances to computing clusters and is used in areas such as web, mobile and desktop applications, games or transaction processing. The JVM is an abstract computing machine with a stack-based architecture, executing programs consisting of Java virtual machine instructions (JVM bytecode). The instruction set contains operations for loading and storing data; arithmetic; object instantiation, manipulation and conversion; stack management; control flow (unconditional and conditional branching); method invocation; signaling and handling exceptions. It was originally designed to run Java programs, but today implementations of several other programming languages, such as Ruby and Python, are available for the JVM.

As the JVM bytecode is also not tied to a particular architecture, the main advantage of using a virtual machine to execute programs is the hardware- and operating system-independence. Concrete implementations of the JVM abstract specification exist for all common operating systems (e.g., Windows, Linux, Mac OS X and Solaris) and architectures (e.g., x86, ARM and SPARC), such as Oracle OpenJDK, Oracle JRockit, Azul Zing JVM and IBM J9 VM. Compilers therefore only need one backend targeting one standardized instruction set, instead of having to implement specific backends for operating-system/architecture combinations. Backends are also easier to implement, as the instruction set is simpler than those of actual architectures.

The JVM consists of several components:

- **Class loader:** dynamically loads, verifies and links the bytecode contained in a class
- **Bytecode interpreter:** executes the bytecode
- **Memory manager:** automatically manages memory using a garbage collector
- **Runtime libraries**

Almost all implementations are dynamic translators instead of plain interpreters and feature state-of-the-art just-in-time (JIT) code generation, translating bytecode during execution into the native instruction set of the host system. Many implementations also feature latest garbage collection technologies. By producing highly efficient native code and also handling memory efficiently, this allows JVMs to run programs as fast as traditional compiled languages targeting native code directly. Other than speed, the virtual machines are also known for their high stability, security and scalable concurrency.

The Oracle OpenJDK is the official Java SE 7 Reference Implementation of the Java Platform. It is available under a free and open source license. It consists of the Java Class Library runtime, the `javac` Java compiler and the HotSpot Java virtual machine. HotSpot continually profiles the execution of the program, determines frequently executed parts ("hot spots", based on a count-threshold), and generates highly optimized code for them, yielding higher performance. For the different use case requirements, two execution modes are available, based on two different compilers sharing the same code base. They mostly differ in their compilation policy, heap defaults, and inlining behavior, i.e., how much called code is integrated into the call site and replaces the method invocation. This often increases performance by removing the invocation overhead at the cost of code size. It also produces opportunities for further optimizations.

- **Client** analyzes the code only to a limited degree and performs only simple optimizations. It also only generates a small amount of profiling data and inlines few functions. This results in a faster startup time, less runtime overhead and smaller memory consumption, but gives fewer opportunities for optimizations and generates less efficient code. It is mainly intended for interactive applications.

- **Server** analyzes the code extensively and performs complex optimizations. It tries to inline aggressively based on rich runtime profiling. This results in a slower startup and higher runtime overhead, but as better optimized code is generated, has better overall performance characteristics. It is intended for long-running applications.

Given these advantages and the rich ecosystem of libraries, applications, tools, and communities for the Java platform, there is an ongoing high interest in using the JVM as a target for languages other than Java. Many languages were either adapted or newly designed to run on the JVM. For example, JRuby is an implementation of the Ruby language, Jython an implementation of Python, and Rhino an implementation of JavaScript. Unlike Java, they are dynamic instead of static languages. Completely new languages exclusively targeting the JVM are for instance Scala, Groovy and Clojure.

As the JVM was originally designed and developed for the needs of running Java programs, the semantics of the JVM bytecode and class specifications reflect the semantics of Java to some degree. Thus, implementing languages close to Java, i.e., those that are static and statically typed, require less effort than implementing ones that are dynamic and are dynamically typed. The JVM is also missing many of the features and constructs that language implementors of the latter commonly rely on, such as direct support for method pointers or continuations. Some language specifications also have requirements for the execution model. For example, the standard definition of the language Scheme requires implementations to be properly tail recursive, i.e., tail calls are required to use constant stack space. This guarantee is not provided by the JVM [62].

7.2 Current Situation

7.2.1 Method invocation

The JVM provides method invocation instructions for both static dispatch (`invokestatic` and `invokespecial`) and dynamic dispatch (`invokevirtual`, `invokeinterface`).

Instruction	Receiver	Dispatch
`invokestatic`	—	—
`invokespecial`	class	—
`invokevirtual`	class	single
`invokeinterface`	interface	single

Table 7.1: Comparison of JVM method invocation instructions

Table 7.1 shows an overview of the four different method invocation instructions in regards to the receiving object, the object on which the method dispatch is performed, and the type of dispatch that is performed. The instructions behavior and use cases are as follows:

- **`invokestatic`** has no receiver, the method is looked up directly in the specified class, i.e., no dispatch is performed. Used in Java for calls of static methods.

- **`invokespecial`** performs no dispatch on the receiver, the method is looked up directly in the specified class. Used in Java for in certain special cases, such as calling the instance initialization method `<init>` after instantiation a new object using the `new` instruction, or calling a private method or method in a superclass of the `this` object.

- **`invokevirtual`** performs single dispatch based on the runtime class of the receiver, i.e., only the receiver type determines method selection. Recursively checks the class' superclass if the class does not implement the method. Used in Java for calls of instance methods.

- **`invokeinterface`** is similar to `invokevirtual`, but checks if the receiver implements the given interface. Used in Java for calls of interface methods.

Virtual function tables and inline caches are commonly used to implement the method dispatch of the `invokevirtual` and `invokeinterface` instructions [3, 2].

As mentioned earlier, Java provides static multiple dispatch through overloaded methods, which are differentiated by arity and argument types, i.e., a fixed type signature for the invocation is resolved at compile-time through the static types of the arguments.

The Java Virtual Machine constant pool table contains the literal constant values used throughout the class, such as the numbers (integers, floats, longs, doubles), Unicode strings (modified UTF-8), as well as string, class, interface, field and (interface) method references. The latter make use of a intermediate name and type descriptor field. Longs and doubles take up two consecutive slots. An exemplary constant pool table is shown in Table 7.2.

Index	Type	Value	Resolved
#1	Methodref	#2.#4	java/lang/Object.”<init>”:()V
#2	Class	#3	java/lang/Object
#3	Utf8	java/lang/Object	
#4	NameAndType	#5:#6	”<init>”:()V
#5	Utf8	<init>	
#6	Utf8	()V	
#7	Double	42.0d	
#9	String	#10	Test
#10	Utf8	Test	

Table 7.2: Example of a constant pool table

All method invocation instructions have an operand field referring to an entry in the constant pool that contains the symbolic type (class name), method name and descriptor (formatted string containing the argument and return type). As there is no instruction to provide custom dispatch mechanisms, currently they are either implemented by introducing generic intermediate dispatch code, often generic for all call sites, or extending the whole JVM implementation with the new dispatching variant, which is non-standard and thus unportable.

```
class Square extends Rectangle {
  boolean intersect(Shape@Rectangle r) {
    // method 1 body
  }
  boolean intersect(Shape@Square s) {
    // method 2 body
  }
}
```

Listing 7.1: Square intersection in MultiJava

```
class Square extends Rectangle {
  boolean intersect(Shape r) {
    if (r instanceof Square) {
      Square s_ = (Square) r;
      // method 2 body, substituting s_ for s
    } else if (r instanceof Rectangle) {
      Rectangle r_ = (Rectangle) r;
      // method 1 body, substituting r_ for r
    } else {
      return super.intersect(r);
    }
  }
}
```

Listing 7.2: Resulting dispatch code for Listing 7.1

For example, MultiJava [15] adds support for multiple dispatch to the Java programming language. The compiler [14] translates MultiJava programs to regular Java bytecode that will run on any JVM. Each generic function is translated into a dispatch function with fixed `instanceof` checks that is used for every call site, thus all checks need to be performed and no optimizations are performed. An example using MultiJava is shown in Listing 7.1. Listing 7.2 shows the resulting dispatch code code. Another project [25] avoids changes to Java itself, maintaining source and binary compatibility, but required JVM implementors to adopt the new non-standard dispatching variant.

Problems

Even though dynamic method dispatch is supported, all method invocation instructions require static type information for the argument and return types. In dynamically typed languages call sites often have unknown argument and return types, so it is only possible to provide the most generic type (e.g., `Object` in Java) as the static type of arguments or signature of the target method.

The dispatch is also only based on names and a single receiver object (in case of the dynamic dispatch instructions), i.e., functions are not first-class values and multiple dispatch is not available [59]. Implementing custom dispatch through indirections often incur a penalty on performance, as the VM is unable to detect intention and perform certain optimizations, esp. inlining.

Linkage is based on name resolution. The symbolic name and type are fixed in the bytecode, and are also constrained to certain 7-bit ASCII characters. Through dynamic linking classes and interfaces need to be statically specified, but are loaded and linked on demand. However, languages runtimes should be able to (re)decide when and how linkage is performed [61].

To improve the support for dynamic dispatch, the following enhancements would be necessary:

- **Naming:** Method invocation instructions should allow arbitrary strings, not just allow Java names or have character restrictions, imposing an encoding and decoding overhead. In some languages functions are first class objects and may be anonymous, therefore symbolic names should not be a requirement.

- **Linking:** Name resolution should be optional and not a requirement. Other linking rules than those of Java and the JVM should be possible. This would allow non-name based method invocation, e.g., in the case when functions are first-class objects and possibly anonymous. Currently it is not possible to let the runtime handle linking. Delegating this task would allow the runtime to link a call site to any callee and re-link it if necessary, e.g. in case of changes that are likely in dynamic languages.

- **Selecting:** The available instructions only allow static or receiver-based single dispatch. As some languages use more sophisticated dispatch mechanisms, it should be possible to implement custom dispatch logic in the runtime that is able to examine any argument or include other aspects relevant to the language. This should be possible without introducing an indirection, so optimizations are still applied.

- **Adapting:** The calling convention is currently very simple and requires explicit casting and (un)boxing. There is also no automatic handling of variable arity: The compiler needs to detect calls to variadic methods and emit specific code, i.e., trailing arguments are passed as an array. It should be possible to perform these steps automatically. There is also no mechanism to insert or delete extra language-specific arguments that may be required in the runtime.

7.2.2 Function pointers

As mentioned, method invocation is performed through names on classes or objects. Some languages however support first-class functions: functions are values, can be passed to other functions and can be invoked. Some languages also may have support for function objects implementing multiple behaviors (generic functions). Currently the JVM provides no means to directly treat methods as first-class values, i.e., there is no support for function pointers.

For this reason, runtimes of languages supporting such features currently use method definition reification through an object-based API: the method definition is reified as a method (simulator method) on an object (method simulator object). Additionally, the call site may also be reified as an object (call site simulator object).

The Java Reflection API inside the package `java.lang.reflect` provides means to examine classes and objects at runtime and includes support for the invocation of arbitrary methods by creating such simulator objects: The method simulator object class `java.lang.reflect.Method` provides the simulator method `invoke`.

```
public final class Method extends AccessibleObject
             implements GenericDeclaration, Member
{
    Object invoke(Object obj, Object... args);
    // ...
}
```

For example, the following two examples have the same effect:

```
System.out.println("hello, world");
```

```
Method println = PrintStream.class.getMethod("println", String.class);
println.invoke(System.out, "hello, world");
```

The biggest disadvantage of using the Java Reflection API is the poor performance compared to a native approach. Method lookup and argument access checks are performed on every invocation, and types are resolved dynamically. Thus, the JVM is unable to perform certain optimizations.

For this reason, and also because the calling convention of other languages may differ, many runtimes implement their own simulator objects, specific to the requirements of the language. The compiler generates a new class for each function, implementing the custom method simulator object interface / exending the custom simulator object class.

For example, in Clojure, functions implement the interface `clojure.lang.IFn`. The simulator method `invoke` is overloaded for different arities and has generic arguments and a return value of type `Object`:

```
public interface IFn extends Callable, Runnable {
    public Object invoke();
    public Object invoke(Object arg1);
    public Object invoke(Object arg1, Object arg2);
    // ...
}
```

The generated class for a function would be of the form:

```
public class FunctionN implements IFn {
    public Object invoke() {
      // ...
    }
    // ...
}
```

In Rhino, an implementation of JavaScript, a similar interface is used, additionally passing in the execution context, the scope, and the implicit `this` object of the function:

```
public interface Callable {
    public Object call(Context cx, Scriptable scope,
                       Scriptable thisObj, Object[] args);
}
```

In JRuby, the class `DynamicMethod` represents a method handle, providing an overloaded method `call` for different arities and an optionally passed block (anonymous function):

```
public abstract class DynamicMethod {
  IRubyObject call(ThreadContext context, IRubyObject self,
                   RubyModule klazz, java.lang.String name);
  IRubyObject call(ThreadContext context, IRubyObject self,
                   RubyModule klazz, java.lang.String name, Block block);
  IRubyObject call(ThreadContext context, IRubyObject self,
                   RubyModule klazz, java.lang.String name, IRubyObject arg0);
  IRubyObject call(ThreadContext context, IRubyObject self,
                   RubyModule clazz, java.lang.String name, IRubyObject[] args);
  abstract IRubyObject call(ThreadContext context, IRubyObject self, RubyModule clazz,
                            java.lang.String name, IRubyObject[] args, Block block);
  IRubyObject call(ThreadContext context, IRubyObject self, RubyModule klazz,
                   java.lang.String name, IRubyObject arg0, Block block);
  IRubyObject call(ThreadContext context, IRubyObject self, RubyModule klazz,
                   java.lang.String name, IRubyObject arg0, IRubyObject arg1);
  // ...
}
```

The abstract class `org.jruby.runtime.CallSite` is the call site simulator object, representing all call sites in JRuby. It has a method `call`, similarly overloaded for different arities and optionally passed block of the target:

```java
public abstract class CallSite {
  abstract IRubyObject call(ThreadContext context, IRubyObject caller,
                            IRubyObject self, Block block);
  abstract IRubyObject call(ThreadContext context, IRubyObject caller,
                            IRubyObject self, double flote);
  abstract IRubyObject call(ThreadContext context, IRubyObject caller,
                            IRubyObject self, IRubyObject... args);
  // ...
}
```

Problems

Using custom simulator methods has poor performance implications, due to execution overhead and added complexity. They do not express the intention of method invocation and their signatures are not standardized and vary between runtimes. Consequently, the JVM is unable to detect its intention and is missing the optimizations for native method invocation, such as inlining and inline caches. The indirection may also exceed the inlining threshold, resulting in unoptimized code.

Class metadata is stored in the permanent generation, a region of the heap from which data is never garbage collected. The generated simulator objects fill up this region.

Another problem is the missing high-level API to generate adapter methods, e.g., for accumulating variable arguments into a single parameter. Runtimes have to implement these again through code generation of additional classes.

7.2.3 Dynamic languages

Related to dynamic method invocation and first-class functions is the support for dynamic language features, i.e., allowing the modification of definitions at runtime, e.g., changing the sets of methods and classes. In general it is more difficult to generate efficient code at compile-time under an open-world assumption. The dynamic JIT code generation performed by most JVM implementations is already an improvement, as it enables optimizations using information as it becomes available at runtime.

Dynamically adding and modifying methods and classes requires dynamically generating code and linking it into the existing environment. For code generation, custom or third-party libraries (such as the popular ASM bytecode manipulation and analysis framework or the Apache Byte Code Engineering Library) can be used, and JVM class loaders allow dynamically loading new classes. However, changing existing classes is not possible, i.e., also changing or adding fields or methods to classes is not directly possible.

7.3 The Da Vinci Machine Project

The first edition of The Java Virtual Machine Specification [45] stated that, “In the future, we will consider bounded extensions to the Java virtual machine to provide better support for other languages.” This promise was only kept to certain extend, partially with the addition of the Reflection API, until the release of Java SE 7 platform, which introduced new features to improve the situation for other language, in particular for dynamic ones.

The effort was started in a project called Da Vinci Machine, also known as Multi Language Virtual Machine (MLVM), an extension of the existing JVM. The project's goal is to improve the support for non-Java languages, especially dynamic languages, and increase their performance. This is accomplished by adding features such as dynamic invocation, continuations, tail-calls and interface injection, and increasing the interoperability across languages, particularly Java, as generic purpose solutions suitable for as many languages as possible. Another means is removing current obstacles that prevent accomplishing these goals, mainly by working with language implementors and gathering feedback.

One first improvement that was developed inside the MLVM was standardized in the Java Specification Request (JSR) 292 (Supporting Dynamically Typed Languages on the Java™ Platform) [66] and introduced in the Java SE 7 platform. It introduced method handles, typed function pointers, accompanied with a high-level Java API to generate and transform them, and extended the existing instruction set with the bytecode instruction `invokedynamic`, which enables configurable method invocation [63]. Both features are explained in detail in the next two sections.

Several dynamic programming languages are currently in the process of improving their implementations with these new features, for example JRuby, Groovy (as of version 2.1) and Jython. With Nashorn, Oracle is implementing a new JavaScript engine solely based on mechanisms provided by JSR 292. Also, completely new languages based on these features are in development, for example Golo [42], a simple dynamic language. An unofficial patch adding support for JSR 292 on the Dalvik VM, the virtual machine running executing applications on the Android operating system, exists as well [30].

In the context of method handles, the Da Vinci Machine project is also working on extending the JVM with better support for closures and extending the Java language appropriately. Another aim is to improve class loading, so new extended versions of existing classes can be loaded, sharing structure and data with the old one. There are also plans to remove the permanent generation region of the heap in the HotSpot JVM [46], which will make its garbage collectors easier to tune.

7.3.1 Method handles

This section describes the newly introduced method handle mechanism and its high-level Java API [60] [52]. A method handle is an immutable method simulator object with a dynamically and strongly typed reference to behavior, for example in the form of an underlying method, constructor or field. The reference is verified at construction time. It is an instance of the class `MethodHandle` and can be directly executed using the method simulator method `invokeExact`. The type descriptor can be accessed through the `type` accessor. It is an instance of the class `MethodType`, and contains zero or more parameter types and an optional return type.

There are two different kinds of method handles:

- **Direct method handles** represent a specific named method, constructor, or field, without any transformations applied to it
- **Bound method handles** consist of a target method handle and one or more bound argument values, but has no visible state

The main difference from the existing method simulator object `Method` of the Java Reflection API is the moment when access checking is performed. Using the Reflection API

it is possible to create method simulator objects that access any target (e.g., field, method, constructor, etc.). On each invocation a check ensures the calling class has access to the target and an exception is thrown if the access is illegal. When creating a method handle, the access check is only performed once at construction time.

Lookup

Direct method handles are not instantiated directly, but created through a factory object referred to as a lookup, an instance of class `MethodHandles.Lookup` [53]. It in turn is also instantiated indirectly through the static method `MethodHandles.lookup`. It returns a lookup object that will only produces method handles to targets that the calling class has access to.

The lookup object can be used to create getter and setter method handles for fields through the methods `find(Static)Getter/Setter`. Method handles for (static) methods can be created using the methods `findVirtual/Static`. Handles for inherited methods (super calls) can be created using `findSpecial`, and for constructors using `findConstructor`. There are also methods available producing method handles for the simulator objects of the Reflection API

The following example demonstrates looking up the method `PrintStream.println`:

```
1 MethodType printlnType = MethodHandles.methodType(void.class, String.class);
2 MethodHandle println = MethodHandles.lookup()
3     .findVirtual(PrintStream.class, "println", printlnType);
```

Invocation

Method handles can be directly executed using the method `invokeExact`:

```
1 public final Object invokeExact(Object... args) throws Throwable
```

Even though the visible signature of the method has an `Object` return type and variable arity `Object` arguments, it is signature polymorphic, specifically the types and arity of the invocation must exactly match the target. For example, the previously looked up method handle for the `println` method has to be invoked with exactly one argument of type `String`:

```
1 println.invokeExact(System.out, "hello, world");
```

Like for any other virtual method invocation, the Java compiler is producing an `invokevirtual` instruction, but the symbolic type is not derived from the method definition (in this case `invokeExact`) as usual, but the actual argument and return types at the call site. Internally, the method handle invocation with `invokeExact` is only marginally more expensive than a regular one and can be compiled into a pointer check and jump. Instead of producing type conversion code according to the type descriptor, the Java compiler needs to use the supplied argument types and push the arguments unconverted.

The return type is determined through an explicit cast on the method invocation expression. In case there is none, the type `Object` is used if the invocation is used as an expression, or the type `void` otherwise. Thus, the previous example would lead to the return type `void`.

For inexact invocations the more permissive methods `invoke` and its variable arity counterpart `invokeWithArguments` are available. They are also signature polymorphic and require the invocation to have the same arity as the target, but also accept compatible types and perform type conversions on the arguments and return value if required, producing a new method handle which is exactly invokable.

Like the compiler, also the JVM is ignoring the type descriptor of signature polymorphic calls. When the method invocation is executed the first time, it is statically verified and linked. The virtual machine then checks if the type of the method handle matches that of the type descriptor of the invocation instruction. After the check, the method handle's underlying code is directly invoked.

Transformation

Method handles can be adapted and transformed into new method handles, allowing developers to build complex chains performing operations on argument and return values and their types. This reduces the need to generate specific classes performing these operations. The Java APIs are similar to those provided by programming languages treating functions as first-class objects, e.g. Lisp and Scheme.

Instances of the class `MethodHandle` have a set of core high-level methods to produce adapting method handles. The methods generally fall into the following categories:

- **Variable artity handling**

```
1 MethodHandle toString = MethodHandles.lookup()
2     .findStatic(Arrays.class, "toString",
3                 methodType(String.class, Object[].class));
```

  - **`asCollector`** produces a method handle collecting a given number of trailing positional arguments into an array, invoking the target with this single argument.

```
1 MethodHandle collect3 = toString.asCollector(String[].class, 3);
2 (String)collect3.invoke("a", "b", "c"); // ⇒ ["a", "b", "c"]
```

  - **`asVarargsCollector`** produces a method handle similar to `asCollector`, but for any number of trailing arguments.

```
1 MethodHandle collectAll = toString.asVarargsCollector(String[].class);
2 (String)collectAll.invoke("a", "b", "c", "d")); // ⇒ ["a", "b", "c", "d"]
```

  - **`asSpreader`** produces a method handle accepting an array argument of given size, invoking the target with the elements of the array as positional arguments.

```
1 MethodHandle spread3 = collect3.asSpreader(String[].class, 3);
2 (String)spread3.invoke(new String[] {"a", "b", "c"}); // ⇒ ["a", "b", "c"]
```

- **Type conversion**

  `asType` produces a method handle with the given type. When invoked, it will adapt the incoming argument list, possibly collecting arguments if the target method has variable arity, performs type conversion of arguments, if required, and will invoke the target with the new argument list. Once the target method handle returns, the generated method handle will perform type conversion to the new return type, if necessary. The type conversion handles casting between reference types, as well as reference and primitive types (boxing), and may involve runtime checks. Invocations through `invoke` may use `asType` if necessary, otherwise they are equivalent to invocations through `invokeExact`.

```
MethodType genericType = methodType(Object.class,
                                    Object.class, Object.class, Object.class)
MethodHandle ts = toString.asType(genericType);
ts.invokeExact((Object)"a", (Object)"b", (Object)"c")); // ⇒ ["a", "b", "c"]
```

- **Partial application**

  `bindTo` performs partial function application, fixing the first argument of the method handle to the given value, producing a bound method handle of smaller arity.

```
MethodHandle printArray = toString.bindTo(new String[] {"a", "b", "c"});
printArray.invoke(); // ⇒ ["a", "b", "c"]
```

The class `MethodHandles` provides further high-level operations for generating and transforming method handles, for example to perform argument insertion, deletion, and substitution. Some methods are equivalent to combinations of other operations, but as they express the intention explicitly, the resulting method handles may be more efficient than manually combined ones. The operations generally fall into the following categories:

- **Producing values**
  - **`constant`** produces a method handle for a given type and value that returns the value on each invocation (cf. `constantly` in Common Lisp (CL))

```
MethodHandles.constant(Integer.class, 23).invoke(); // ⇒ 23
```

  - **`identity`** produces a method handle for a given type that returns its argument of the type (cf. `identity` in CL).

```
MethodHandles.identity(Object.class).invoke(23); // ⇒ 23
```

- **Invocation**
  - **`spreadInvoker`** produces a method handle for a given a type and a leading argument count n. When the generated method handle is invoked with a method handle, n positional arguments, and an array of trailing arguments, it is in turn invoking the method handle with all arguments as positional arguments (cf. `apply` in CL)

- **Argument and return value manipulation**
  - **`permuteArguments`** produces a method handle with the given type, which invokes the target with a reordered argument list as specified by the given reordering list.

```
MethodHandle pair = MethodHandles.lookup()
    .findStatic(Arrays.class, "asList",
                methodType(List.class, Object[].class))
    .asCollector(Object[].class, 2);
MethodHandles.permuteArguments(pair, pair.type(), 1, 0)
    .invoke(1, 2); // ⇒ [2, 1]
```

  - **`dropArguments`** produces a method handle that, starting at the given position, ignores a range of arguments of given argument types.

```
MethodHandles.dropArguments(pair, 0, Object.class)
    .invoke("a", "b", "c"); // ⇒ ["b", "c"]
```

  - **`insertArguments`** produces a method handle that, starting at the given position, inserts a range of argument values into the argument list when invoking the target method handle. This is a more general version of `bindTo`, enabling partial application for any argument position.

```
MethodHandles.insertArguments(pair, 1, "b");
  .invoke("a"); // ⇒ ["a", "b"]
```

  - **`filterArguments`** produces a method handle which, starting at the given position, applies one unary filter function to each argument.

```
MethodHandle abs = MethodHandles.lookup()
    .findStatic(Math.class, "abs", methodType(int.class, int.class))
    .asType(methodType(Object.class, int.class));
MethodHandles.filterArguments(pair, 0, abs, abs)
    .invoke(-1, -2); // ⇒  [1, 2]
```

  - **`foldArguments`** produces a method handle aggregating the incoming argument list using the given combiner function and invokes the target with the argument list, which is preceded with the result of the combination, if it is non-void. For instance, it could be used to make all passed in arguments available as an array in the target method (cf. JavaScript's implicit `arguments` variable inside every function), by collecting the arguments into an array inside the combiner.
  - **`filterReturnValue`** produces a method handle filtering the return value. For example, it could be used to perform post-processing operations after the target method returns.

- **Guarding**
  - **`guardWithTest`** produces a method handle which invokes a test function, which has a less or equal amount of arguments and a boolean return value, with the incoming argument list. Depending on the outcome of the test call, the incoming argument list is passed to either the target method handle if the test succeeded or the fallback handle otherwise.

```
MethodHandle isInteger = MethodHandles.lookup()
    .findVirtual(Class.class, "isInstance",
                 methodType(boolean.class, Object.class))
    .bindTo(Integer.class);
MethodHandle absIfInterger = MethodHandles
    .guardWithTest(isInteger,
                   abs.asType(methodType(Object.class, Object.class)),
                   MethodHandles.identity(Object.class));
absIfInterger.invoke(-23); // ⇒ 23
absIfInteger.invoke(null); // ⇒ null
```

- **Exception handling**
  - **`catchException`** produces a method handle that will invoke a given exception handling method handle if the target method handle throws an exception of the given type (cf. `try-catch` statement).
  - **`throwException`** produces a method handle which ignores its one argument of the given return type, and throws an exception of the given exception type. (cf. `throw` statement).
- **Type conversion**

  `explicitCastArguments` produces a method handle that will perform type conversion on the received arguments to the target method type, if necessary.
- **Array manipulation**

  `arrayElementGetter/-Setter` produce method handles accepting an array of the given type, getting or setting elements in the array at the specified index.

With `SwitchPoint` another mean of guarding is available. It behaves like volatile boolean with a default value of true that can be switched off, i.e., set to false, once (through `SwitchPoint.invalidateAll`). The method `guardWithTest`, can be used to create one or more method handles that behave like guards created through `MethodHandles.guardWithTest`, i.e., delegate to a target or fallback, but they use the switch point's state as the test value. Invalidation is atomic and synchronized. The test is considered to unlikely change and the guard can be optimized.

Analysis

As shown, method handles and the accompanying transformation API are very useful additions for language implementors and solve many of the problems described in the previous section. They can be used as function pointers, but are also type safe and can be easily combined. They are not only useful when implementing dynamic languages, but also for static, functional languages.

However, method handles also have limitations. They are opaque and only expose their type. No method handle introspection (reflection) is available, i.e., it is impossible to examine if a given method handle is a direct or bound method handle, or even access bound arguments of bound method handle. This makes them difficult to transform further.

They also can not be subclassed and the underlying class hierarchy is not standardized. The specification discourages depending on concrete types of method handles.

Another problem is security. Method handles with non-public targets need to generally be kept secret. They should not be made unintentionally accessible by untrusted code. Also the complex implementation inside the JVM poses a risk. For example, the `MethodHandle` implementation in OpenJDK did not properly perform access control checks, which allowed any code use this flaw to bypass the Java sandbox restrictions[1].

7.3.2 Invokedynamic

With JSR 292 the `invokedynamic` instruction was added to the JVM instruction set to improve the method dispatching possibilities on the JVM. It is a simple, general purpose primitive, not inspired or tied to the semantics of Java, and can be used to implement a variety of method invocation types. It allows the linking of the call site target at invocation time through a bootstrap function. JVM implementations are able to perform optimizations (e.g., inlining), promising better performance than reflection and close to statically linked calls, while allowing for greater flexibility.

The `invokedynamic` instruction [54] has no receiver object. Instead of referring to a method reference in the constant pool, like all other invocation instructions, it refers to a constant pool entry of the new type `InvokeDynamic`. The entry refers to the bootstrap method through an index into the class file attribute `BootstrapMethods`; a symbolic reference, that is free to use any name; and the argument and return types.

For example, the following sequence of instructions performs a dynamic invocation and prints the result to the standard output. The invocation instruction refers to the custom name `getAnswer` that is unrelated to any actual method name. The call has no arguments and returns an integer. The resolved constant pool entries were added as comments.

```
1 public static void main(java.lang.String[]);
2   Code:
3      0: getstatic     #16
4         // Field java/lang/System.out:Ljava/io/PrintStream;
5      3: invokedynamic #27, 0
6         // InvokeDynamic #0:getAnswer:()I
7      8: invokevirtual #33
8         // Method java/io/PrintStream.println:(I)V
9     11: return
```

The class file attribute `BootstrapMethods` contains a reference to the bootstrap method as a reference to a constant pool entry of the new type `MethodHandle`, which specifies the method handle type and a reference to a method reference in the constant pool.

```
1 BootstrapMethods:
2     0: #23
3
4 Constant pool:
5  #23 = MethodHandle       #6:#22
6        //  invokestatic Life.bootstrap:
7        //    (Ljava/lang/invoke/MethodHandles\$Lookup;Ljava/lang/String;
8        //     Ljava/lang/invoke/MethodType;)Ljava/lang/invoke/CallSite;
```

[1] Java Applet Method Handle Remote Code Execution, CVE-2012-5088

Call sites are lazily linked the first time they are executed. The JVM will call the specified bootstrap method with the lookup object of type `MethodHandles.Lookup`, which has access permissions for the caller class in which the call site occurs; the symbolic reference, an arbitrary string; the method type, i.e., the resolved type descriptor, an instance of `MethodType`, and up to 251 additional static arguments that are stored in the constant pool. The arguments on stack are not passed to the bootstrap method.

An exemplary signature of such a bootstrap method could therefore be:

```
1 public static CallSite bootstrap
2     (MethodHandles.Lookup lookup, String name, MethodType type)
```

The bootstrap method is called via `invoke`, i.e. the argument types and arity are adapted if needed, so the specific type of the method is not important. For instance a valid definition of a bootstrap method could also be:

```
1 public static Object bootstrap(Object... args)
```

The bootstrap method is required to return a linked call site, reified as call site object, an instance of the class `CallSite`. It is linked by calling the `setTarget` method with a valid instance of a method handle, which may be either direct or bound and must be compatible with the required type. For example, it can do so by using the lookup object to resolve the given name or perform custom resolution and dispatch, but as notes, only based on the given name and type, not on the arguments on the stack. Depending on the use case, one of three concrete subclasses of the abstract class `CallSite` may be used or subclassed to embody further state, such as counters or profiled type information. It is not possible to directly subclass it.

- **`ConstantCallSite`** permanently binds the `invokedynamic` instruction to the target.
- **`MutableCallSite`** temporarily binds the `invokedynamic` instruction to the target. The target can be relinked by updating the target method handle on the reified call site. It may be unlinked by setting the target of the call site to `null`, which forces the recreation of the call site by calling the bootstrap method again.
- **`VolatileCallSite`** is mutable like a `MutableCallSite`, but is required when it has volatile variable semantics, i.e., when the target change must be immediately and reliably witnessed by other threads.

Once the bootstrap method returns a call site, it is permanently associated with the particular `invokedynamic` instruction. The JVM executes the invocation by obtaining the target method handle from the call site object and calling the `invokeExact` simulator method with the arguments on the stack. Any further execution will skip the initial bootstrapping of the call site and will directly perform this last step of the method invocation process.

The instruction cannot be directly used in Java, i.e., no Java source code currently compiles to an `invokedynamic` instruction.

The class `Life` referred to in the previous example contains the bootstrap method and a possible target method. It is shown in Listing 7.3.

```java
public class Life {
    static int getAnswer() {
        return 42;
    }

    public static CallSite bootstrap
        (MethodHandles.Lookup caller, String name, MethodType type)
        throws NoSuchMethodException, IllegalAccessException
    {
        MethodHandle target = MethodHandles.lookup()
            .findStatic(Life.class, name, type);
        return new ConstantCallSite(target);
    }
}
```

Listing 7.3: Class `Life` with bootstrap method

The bootstrap method uses the name and type passed from the invocation to find a static method inside the class, create a method handle referring to it, and create and finally return a constant call site targeting this method handle. The first execution of the instruction will thus produce the constant call site, it will be linked and subsequent executions will immediately invoke `Life.getAnswer`.

Performance

Early benchmarking results [43] showed that early implementations of `invokedynamic` using direct method handles were only 2 – 5 times slower than native static invocations. Rerunning these benchmarks for current implementations is not possible: they are based on an old version of the API (`java.dyn`), and even though the data set is available, the source code was not released. Because of this, a new benchmark was implemented to evaluate the performance of current implementations.

Table 7.3 shows a comparison of the running times of a simple number summation benchmark using the four presented method invocation approaches: Static, using the `invokestatic` instruction; Reflection, using the Java Reflection API as presented in the last section; Reification, using a custom method simulator object; and Invokedynamic, using the `invokedynamic` instruction and producing a mutable call site.

Name	Time (ms)
Static	1737
Reflection	148160
Reification	66160
Invokedynamic	1810

Table 7.3: Comparison of dispatch tests

The version using the Reflection API is two orders of magnitude slower compared to the version using the `invokestatic` instruction. The version using a custom simulator object performs twice as fast as the version using reflection, as no access checks are performed. The version using the `invokedynamic` instruction is almost as fast as the version using `invokestatic`. Thus, performance improved substantially in recent versions of the OpenJDK.

Invocation via an `invokedynamic` instruction is fast, as it is always exact and requires no runtime check, due to the fact that call sites are strongly typed and bootstrap methods are required to create call sites with a matching type. The code of the target method handle can often be inlined, which opens the possibility for even more optimizations. This is possible, as the call site target is usually stable and the method handles are immutable.

Examples

The `invokedynamic` instruction is generic and can be used to implement a variety of dynamic dispatch mechanisms. For instance, instead of creating only generic dispatch code, e.g., as one generic dispatch function as shown before, optimized call sites can be produced in the bootstrap method. Another possible use-case is the implementation of named parameters: the source language could allow specifying an argument for a particular parameter by name rather than the position in the method's parameter list. The method call would be compiled into an `invokedynamic` instruction and the arguments would be pushed on the stack in the order as specified in the source program, but additionally the parameter names would passed as options to the instruction. The bootstrap method could utilize the `permuteArguments` method to produce an adapter method handle that reorders the passed arguments according to the parameter list [31].

More importantly, the new the `invokedynamic` instruction allows the implementation of common dynamic dispatch techniques directly in the language runtime. The remaining part of this section will show how polymorphic inline caching can be implemented.

First, a mutable call site subclass is created, holding the lookup object and method name, as well as keeping track of the search depth. It is shown in Listing 7.4. As bootstrap methods have no access to arguments passed on the operand stack, but the type of the receiver (the first argument) needs to be examined, the bootstrap method (see Listing 7.5) creates an instance of the call site subclass and sets its initial target to an adapter collecting all arguments and passing it to the fallback method.

```
class InliningCacheCallSite extends MutableCallSite {
    static final int MAX_DEPTH = 3;
    final Lookup lookup;
    final String name;
    int depth = 0;
    // ...
}
```

Listing 7.4: Class representing the polymorphic inline cache call site

```
public static CallSite bootstrap(Lookup lookup, String name, MethodType type) {
    InliningCacheCallSite callSite =
        new InliningCacheCallSite(lookup, name, type);
    MethodHandle fallbackAdapter = FALLBACK.bindTo(callSite)
        .asCollector(Object[].class, type.parameterCount())
        .asType(type);
    callSite.setTarget(fallbackAdapter);
    return callSite;
}
```

Listing 7.5: The bootstrap method initializes the call site to the fallback method

The fallback method, responsible for handling search failures and constructing the search chain if the maximum depth is not yet reached, is shown in Listing 7.6. In case the maximum depth is reached, the target method is simply looked up and invoked. Otherwise, the receiver type is determined and the target method is looked up. The linear search is implemented by a chain of guard method handles produced using the `guardWithTest` utility method. Each test method handle for the guard is a method handle for the `checkClass` method, that tests if the receiver object has a specific type. It is partially applied with the type to test using `bindTo`.

```
static boolean checkClass(Class<?> clazz, Object receiver) {
    return receiver.getClass() == clazz;
}

static Object fallback(InliningCacheCallSite callSite, Object[] args)
    throws Throwable
{
    MethodType type = callSite.type();

    // inline class test depth limit reached?
    if (callSite.depth >= InliningCacheCallSite.MAX_DEPTH) {
        MethodHandle target =
            callSite.lookup.findVirtual(type.parameterType(0),
                                        callSite.name,
                                        type.dropParameterTypes(0, 1));
        return target.invokeWithArguments(args);
    } else {
        Object receiver = args[0];
        Class<?> receiverClass = receiver.getClass();
        MethodHandle target = callSite.lookup
            .findVirtual(receiverClass, callSite.name,
                         type.dropParameterTypes(0, 1))
            .asType(type);

        MethodHandle classTest = CHECK_CLASS.bindTo(receiverClass);
        MethodType classTestType = classTest.type()
            .changeParameterType(0, type.parameterType(0));
        classTest = classTest.asType(classTestType);

        // wrap target with new guard for current class test,
        // fall back to previous class tests or initially this fallback method
        callSite.setTarget(MethodHandles.guardWithTest(classTest, target,
                                                       callSite.getTarget()));
        // increase class test depth
        callSite.depth += 1;

        return target.invokeWithArguments(args);
    }
}
```

Listing 7.6: Generic fallback method of the polymorphic inline cache

Analysis

The `invokedynamic` instruction is a general purpose primitive and can be used to implement a variety of dynamic dispatch mechanisms. It is not tied to the invocation semantics of any particular language, unlike the existing instructions. The major difference from existing solutions is the standardization of the indirection through bootstrap methods. It is only performed once at the first call. As the indirection is explicit, JVMs are able to perform corresponding optimizations. Custom dispatch mechanisms can be implemented in an easy and high-level manner.

However, the solution also has problems. For instance, mutable call sites are not thread-safe. It is only guaranteed that the next call right after the target change observes the new target - other threads may observer the new target later.

The JVM JIT compilation is concurrent with the execution of the application. If mutable call sites change very frequently, they can be considered megamutable and code is repeatedly generated but discarded. The unlinking and the resulting rebootstrapping is serialized across the whole instance of the JVM, which might degrade performance. It is considered rare and is thus unoptimized.

The introduction of `invokedynamic` is a building block, however it does not improve the interoperability between languages. A possible solution could be the introduction (and possible standardization) of a Metaobject Protocol (MOP). It would mediate the naming, linking, selecting, adapting and invoking methods on a higher level across different languages. The major initiative in this direction is the Dynalink project [67], an high-level linking and metaobject protocol library based on `invokedynamic`. It makes it easy for language implementors to develop interoperability with Java and other languages.

Chapter 8

Lila

This chapter introduces Lila (***Li**ttle **la**nguage*), the programming language that was developed for the purpose of this thesis. An overview of the language presents the features, followed by a description of their implementation. This includes a discussion of the object system and functions. A particular focus is given on how the new features of the JVM, as described earlier, were used instead of relying on commonly used implementation approaches. Lila's support of dynamic method dispatch is discussed separately in the next chapter.

8.1 Introduction

For the purpose of demonstrating the nature and implementation of different method dispatch approaches and algorithms, a new high-level, general purpose, dynamic programming language was designed and implemented. The feature set is therefore intentionally kept small. Lila is dynamically typed, supports first-class and higher-order functions, as well as object-oriented programming using an object system based on classes featuring multiple inheritance. The syntax is inspired from languages such as Common Lisp, Dylan, JavaScript, and ML. The implementation of Lila features an interpreter which interactively compiles and evaluates expressions, as well as an ahead-of-time compiler. It primarily makes use of the new features of the JVM, demonstrating their usage and suitability.

Designing and implementing a small language is easier than extending or modifying an existing full-featured language implementation running on the JVM, as it not necessary to deal with the integration of new features into the language design and extending the language's semantics. Another problem of existing implementations of full-featured languages is their often high complexity and the involved effort required to understand details. Also, a small language is easier to explain and demonstrate to, as well as easier to understand for the reader.

Multiple existing languages were initially considered as testbeds for this thesis, such as Java (and in particular MultiJava), JRuby and Jython. The latter are already partially based on the features introduced with JSR 292, but the effort required for their extension would exceed the scope of this thesis. The previously mentioned new dynamic language Golo was also considered, as it has a simple language design and its implementation made use of the JVM's new features from the beginning. Unfortunately it is not developed openly yet and will only be released as open-source software when a first stable version is completed.

8.2 Overview

8.2.1 Features

For the language design most features not relevant for the discussion of method dispatch approaches were intentionally omitted and the most extensive variants were chosen for features that are. For example, the class-based object system allows multiple inheritance instead of limiting it to single inheritance.

Lila is dynamically typed. Instead of variables, values have types, that are run-time values, allowing programs to use type introspection. Every value is a first-class object. Programs consist of one or more statements, each being either a definition or an expression. A definition may introduce a new variable, class or function. Lila is also dynamic: new definitions (e.g., new classes) can be added at runtime, and existing definitions may be redefined. Expressions may either be bindings (through `let`), method invocations (arguments are passed by value), or one of the control flow primitives: the boolean operators && and || are expressions performing short-circuit evaluation; `if` is the conditional expression; `while` is the pre-test loop.

Functions are first-class values and lexical scoping rules apply. If expressions inside functions access non-local variables, closures are created, so access is still possible even when the functions are invoked outside of their direct lexical scope. Functions are expected to always return exactly one value and may have variadic parameter lists, i.e., additional, non-required arguments can be passed. These additional arguments are collected into an array. Function definitions are dynamic, i.e., new functions can be created at runtime.

Lila has support for polymorphic operations in the form of dynamic, symmetric multiple dispatch using the concept of multi-methods, and dynamic predicate dispatch with arbitrary expressions. The syntax, usage and implementation of these two features will be presented separately in the next chapter.

As noted before, Lila has a reduced feature set and thus some limitations. Assignments are not supported, but could be added by using mutable data structures like references in SML or Clojure. The language has no pointers, multiple return values or support for logical grouping of definitions, such as through namespaces or a module system.

The design and feature set was mainly inspired by Dylan, which derives from Common Lisp and Scheme, instead of Java, to demonstrate which advanced language features can be easily implemented on the JVM, but are not supported by its main target. For comparison, Ruby has support for neither multiple dispatch nor multiple inheritance. Python supports multiple inheritance, but not multiple dispatch. MultiJava features multiple dispatch, but is static and does not support multiple inheritance.

8.2.2 Syntax

The grammar of Lila is given in Figure 8.1. The syntax is ALGOL-like and mainly inspired by Dylan (definitions and naming), Java and JavaScript (braces for blocks), and ML (bindings). Function calls are always prefix and there is no special case for infix operators. Identifiers may contain characters that are considered special and reserved for operators in most popular languages, but are common in the Lisp family of programming languages.

They are suitable for use in arithmetic (–, +, ∗, /) and comparison functions (<, >, =), used to indicate class names (name wrapped with < and >, e.g. `<object>`), suggest a function is a predicate (suffix ?), or might have side-effects (suffix !). Superclasses are expression, as the interpreter evaluates them.

⟨*program*⟩ ::= (⟨*definition*⟩ | ⟨*expression*⟩)+

⟨*definition*⟩ ::= "**def**" ⟨*identifier*⟩ "**=**" ⟨*expression*⟩
| "**defn**" ⟨*identifier*⟩ ⟨*parameter-list*⟩ ⟨*body*⟩
| "**defclass**" ⟨*identifier*⟩ ⟨*superclasses*⟩ ⟨*properties*⟩

⟨*superclasses*⟩ ::= "**(**" [⟨*expression*⟩ ("**,**" ⟨*expression*⟩)*] "**)**"

⟨*properties*⟩ ::= "**{**" ⟨*identifier*⟩ ("**;**" ⟨*identifier*⟩)* ["**;**"] "**}**"

⟨*expression*⟩ ::= ⟨*expression*⟩ "**||**" ⟨*expression*⟩
| ⟨*expression*⟩ "**&&**" ⟨*expression*⟩
| ⟨*call-expression*⟩
| "**let**" ⟨*identifier*⟩ "**=**" ⟨*expression*⟩ ⟨*body*⟩
| "**if**" ⟨*expression*⟩ ⟨*body*⟩ "**else**" ⟨*body*⟩
| "**while**" ⟨*expression*⟩ ⟨*body*⟩
| "**fn**" ⟨*parameter-list*⟩ ⟨*body*⟩
| ⟨*integer*⟩ | ⟨*string*⟩ | ⟨*boolean*⟩ | ⟨*identifier*⟩
| ⟨*sequence*⟩

⟨*call-expression*⟩ ::= ⟨*expression*⟩ ⟨*arguments*⟩

⟨*arguments*⟩ ::= "**(**" [⟨*expression*⟩ ("**,**" ⟨*expression*⟩)*] "**)**"

⟨*parameter-list*⟩ ::= "**(**" [⟨*identifier*⟩ ("**,**" ⟨*identifier*⟩)*
["**...**" ⟨*identifier*⟩]] "**)**"

⟨*body*⟩ ::= "**{**" ⟨*expression*⟩ ("**;**" ⟨*expression*⟩)* ["**;**"] "**}**"

⟨*sequence*⟩ ::= "**(**" ⟨*expression*⟩ ("**,**" ⟨*expression*⟩)* "**)**"

⟨*boolean*⟩ ::= "**true**" | "**false**"

⟨*identifier*⟩ ::= ⟨*char*⟩ (⟨*char*⟩ | ⟨*digit*⟩)*

⟨*char*⟩ ::= "**a**"-"**z**" | "**A**"-"**Z**" | "**_**" | "**-**" | "**+**"
| "*****" | "**/**" | "**?**" | "**!**" | "**<**" | "**>**" | "**=**"

Figure 8.1: Grammar of Lila

8.3 Implementation

Like the design, also the implementation focuses on simplicity. Due to Lila's dynamic nature, it is primarily implemented as an interactive interpreter, that compiles and evaluates expressions, and interprets definitions by calling into the runtime. The implementation also features an ahead-of-time compiler that generates code for such definitions, so the program can be executed soley with the runtime. Most features of Lila are implemented using the new mechanisms introduced with JSR 292. Instead of implementing optimizations inside the compiler, the implementation relies on the optimizations performed by the VM.

8.3.1 Architecture

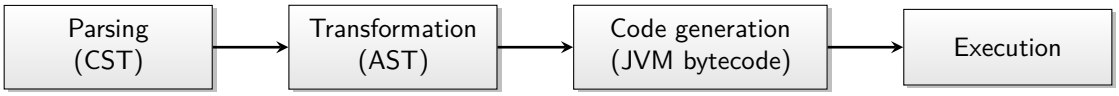


Figure 8.2: Overview of evaluation stages

The parser, compiler and interpreter are implemented in Ruby and make use of JRuby's interoperability with Java. Ruby was chosen because it allows quick prototyping and has easy to use libraries offering DSLs for both writing parsers (parslet) and emitting JVM bytecode (bitescript). The parser is implemented as a parsing expression grammar (PEG). The runtime system of the language is implemented in Java, as it is portable in the sense that its components could be reused for the creation or extension of other languages running on the JVM. Both the compiler and the runtime make use of the new features introduced with JSR 292. The specific implementation details are described in the next sections.

Evaluation of statements is performed as shown in the overview of the evaluation stages shown in Figure 8.2. First, the statement is parsed into a concrete syntax tree (CST), then transformed into an abstract syntax tree (AST). If the parsed statement is a definition, such as a class definition, parts of the AST node required for the instantiation may be evaluated, e.g., the superclass expressions of a class definition, and the interpreter calls into the runtime to instantiate the defined object. The interpretation code for class definitions is shown in Listing 8.1 and consists of evaluating the superclass expressions and calling into the runtime to dynamically create a new instance of a class.

```
1  class ClassDefinition < Struct.new :name, :superclasses, :properties
2    def interpret(interpreter)
3      superclasses = self.superclasses.map { |superclass|
4        interpreter.eval superclass
5      }.to_java LilaClass
6      properties = (self.properties || []).to_java :string
7      lilaClass = LilaClass.make self.name, superclasses, properties
8      RT.setValue self.name, lilaClass
9      lilaClass
10   end
11   # ...
12 end
```

Listing 8.1: Interpretation of class definitions

If the parsed statement is an expression, JVM bytecode is generated for it inside a static method of a new class. If the expression is a simple value, it is boxed, as Lila is using a custom class hierarchy. An example for strings is shown in Listing 8.2. The string value, contained in the instance variable `@value`, is loaded as a constant and boxed.

```
class StringValue < Value
  # ...
  def compile(context, builder)
    builder.new LilaString
    builder.dup
    builder.ldc @value
    builder.invokespecial LilaString, '<init>',
      [Java::void, Java::java.lang.String]
  end
  # ...
end
```

Listing 8.2: Code generation for strings

Listing 8.3 contains another example, showing how `while` loops are compiled. Notable is the call to the method `isTrue` before the branch, which determines if a Lila object is true: all values except the boolean value `false`. The loop also discards results of the body, and itself results in `false`.

```
class Loop < Expression
   # ...
   def compile(context, builder)
      test_label = gensym
      builder.label test_label
      @test.compile context, builder
      builder.invokevirtual LilaObject, 'isTrue', [Java::boolean]
      end_label = gensym
      builder.ifeq end_label
      @body.compile context, builder
      builder.pop
      builder.goto test_label
      builder.label end_label
      BooleanValue.new(false).compile context, builder
    end
    # ...
end
```

Listing 8.3: Code generation for `while` loops

After the code generation, the resulting bytecode of the class is dynamically loaded inside a custom Java class loader, that is shown in Listing 8.4. Once loaded, the code is executed by acquiring a method handle and invoking it (`run`).

```java
public class DynamicClassLoader extends ClassLoader {
    public Class<?> define(String name, byte[] classBytes) {
        return super.defineClass(name, classBytes, 0, classBytes.length);
    }

    public MethodHandle findMethod(Class<?> clazz, String name, MethodType type)
        throws Throwable
    {
        return MethodHandles.lookup().findStatic(clazz, name, type);
    }

    public LilaObject run(Class<?> clazz) throws Throwable {
        MethodHandle run = findMethod(clazz, "run", LilaObject.class);
        return (LilaObject)run.invokeExact();
    }
}
```

Listing 8.4: Lila's class loader

8.3.2 Variables

Variables are either local, introduced as parameter names through functions, or global, introduced through variable definitions, which extend the interpreter's environment (`RT.ENV`, a hashmap associating variable names with Lila objects). Bindings are translated to applications of an anonymous function containing the body with the bound value, e.g., the following two expressions are equivalent.

```
let x = 1 { x + 1 }
```

```
(fn (x) { x + 1 })(1)
```

Variables referencing parameters are compiled to the instruction `aload`, which loads a reference onto the operand stack from a the array of local variables at the given index. Identifiers referencing global variables are compiled into `invokedynamic` instructions which make use of the runtime's bootstrap method `bootstrapValue`, as shown in Listing 8.5. It gets the value of the referenced variable from the environment, creates a method handle constantly returning this value, and returns a mutable call site targeting this handle. Only one method handle is created for each variable.

```java
static Map<String, List<MutableCallSite>> valueCallSites = new HashMap<>();

static CallSite bootstrapValue(Lookup lookup, String name, MethodType type) {
    name = StringNames.toSourceName(name);
    MethodHandle methodHandle = getOrCreateValueMethodHandle(name);
    MutableCallSite callSite = new MutableCallSite(methodHandle);
    List<MutableCallSite> callSites = valueCallSites.get(name);
    if (callSites == null)
        callSites = new ArrayList<>();
    callSites.add(callSite);
    valueCallSites.put(name, callSites);
    return callSite;
}
```

```
14
15 static Map<String, MethodHandle> valueMethodHandles = new HashMap<>();
16
17 static MethodHandle getOrCreateValueMethodHandle(String name) {
18     MethodHandle methodHandle = valueMethodHandles.get(name);
19     if (methodHandle == null)
20         methodHandle = createValueMethodHandle(name);
21     return methodHandle;
22 }
23
24 static MethodHandle createValueMethodHandle(String name) {
25     LilaObject value = ENV.get(name);
26     MethodHandle methodHandle = MethodHandles.constant(LilaObject.class, value);
27     valueMethodHandles.put(name, methodHandle);
28     return methodHandle;
29 }
```

Listing 8.5: Bootstrap method for variable access

The bootstrap method is also recording every call site. When the variable is redefined, through the runtime's method `setValue` (shown in Listing 8.6), the targets of all recorded call sites are updated. The advantage of using `invokedynamic` and a constant method handle is the reduced overhead for lookups in the environment and the possibility that the value will be inlined.

```
1  static void setValue(String name, LilaObject value) {
2      ENV.put(name, value);
3      List<MutableCallSite> callSites = valueCallSites.get(name);
4      if (callSites != null) {
5          MethodHandle methodHandle = createValueMethodHandle(name);
6          for (MutableCallSite callSite : callSites)
7              callSite.setTarget(methodHandle);
8          MutableCallSite.syncAll(callSites.toArray(new MutableCallSite[]{}));
9      }
10 }
```

Listing 8.6: Redefining a variable updates all call sites

An alternative, more memory efficient implementation could make use of switch points (guards that behave like a volatile boolean and can be invalidated, see 7.3.1). The bootstrap method would create one switch point for each variable, record it instead of each call site, and use it to guard the constant method handle. When the variable is redefined only the switch point needs to be invalidated. Further executions will now result in the invocation of the fallback method updating the target of the particular call site only with a further guarded method handle constantly returning the value of the variable. The notable difference is the lazyiness.

8.3.3 Object system

Lila's object system is based on classes and supports multiple inheritance. The root of the type hierarchy is the class `<object>`. Classes may have state in the form of properties, but do not contain behavior (methods). This is similar to the object systems found in Common Lisp and Dylan. The Java class hierarchy only supports single inheritance, which is also reflected in the JVM. Thus, Java classes can not be directly used for the implementation, but a custom class hierarchy is implemented using Java classes.

```
class LilaObject {
    static LilaClass lilaClass;
    static {
        lilaClass = new LilaClass(true, "<object>", LilaObject.class);
        LilaClass.lilaClass =
          new LilaClass(true, "<class>", LilaClass.class, LilaObject.lilaClass);
    }
    LilaClass type;
    HashMap<String,LilaObject> properties;
    // ...
}
```

Listing 8.7: `LilaObject`, implementation of Lila class `<object>` in Java

The Lila class `<object>` is implemented as the Java class `LilaObject`. Instances of the Lila class `<class>` represent classes (cf. `Class` in Java). It is implemented as the Java class `LilaClass`. Each instance references its direct superclasses in an array. All built-in Lila classes (e.g. `<string>`, `<array>`, `<function>`) are represented like this (i.e., as `LilaString`, `LilaArray` and `LilaFunction`). Each Java class backing a built-in Lila class has a static field `lilaClass` that specifies the corresponding Lila type.

For example, the Lila instance `<object>` is stored in `LilaObject.lilaClass`. The implementation of these aspects are shown in Listing 8.7 and Listing 8.8 for `LilaObject` and `LilaClass` respectively.

```
class LilaClass extends LilaObject implements Comparable<LilaClass> {
    static LilaClass lilaClass;
    String name;
    Class<?> javaClass;
    boolean builtin;
    LilaClass[] superclasses;
    String[] classProperties;
    List<LilaClass> allSuperclasses;
    Set<LilaClass> allSubclasses;
    // ...
}
```

Listing 8.8: `LilaClass`, implementation of Lila class `<class>` in Java

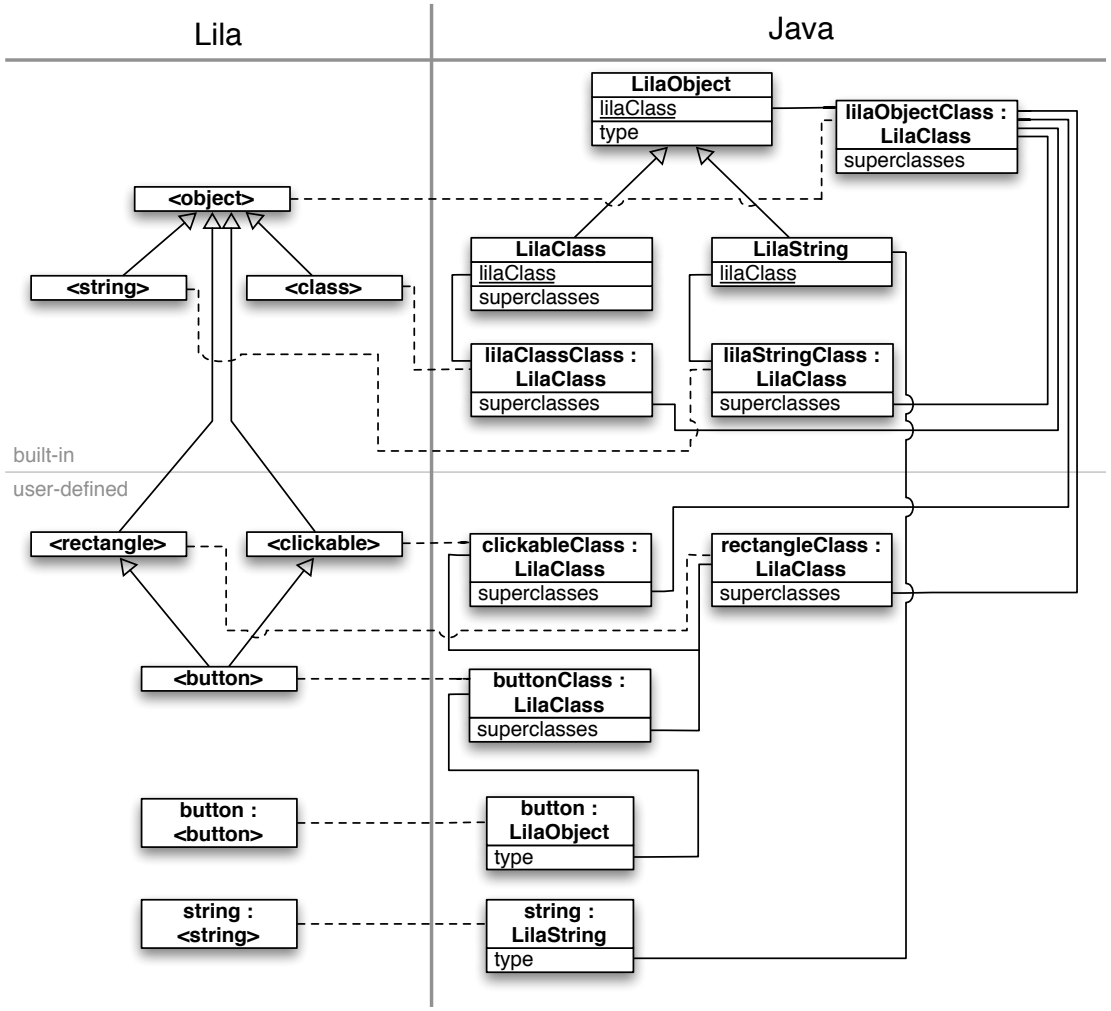


Figure 8.3: Class and object hierarchy in Lila and representation in Java

Instances of user-defined classes are not backed by actual Java class, i.e., no additional Java class inheriting from `LilaObject` are generated. Instead, they are all instances of `LilaObject`, which holds the instance's type in the field `type`. The example in Figure 8.3 shows a combined class and object hierarchy making use of multiple inheritance in Lila, and its internal representation in Java.

This approach is similar to the one used in other language implementations. For example, Jython uses instances of the Java class `PyObject` to represent Python object instances and the Java class `PyClass` to represent class instances. JRuby uses the Java classes `RubyBasicObject` and `RubyClass` internally. The field `metaClass` of the `RubyBasicObject` is used to reference the object's dynamic type. Another reason for other languages to use a similar custom class hierarchy even though they may only support single inheritance is their dynamic nature: it is not possible to add new methods to existing Java classes.

Lila uses an instantiation protocol similar to that of Dylan. The built-in function `make` takes a class and additional arguments as its parameters, instantiates an object of the type and calls the method `initialize` to perform initialization. If the Lila class is built-in, the Java method `make` is invoked with the additional arguments. Listing 8.9 shows the implementation of `make`.

```
static LilaObject make(LilaClass lilaClass, LilaArray rest)
    throws Throwable
{
    Class<?> javaClass = lilaClass.getJavaClass();
    LilaObject object = null;
    if (lilaClass.isBuiltin()) {
        MethodHandle mh = lookup.findStatic(javaClass, "make", builtinMakeType);
        object = (LilaObject)mh.invokeExact(rest.array);
    } else {
        object = new LilaObject(lilaClass);
        // ...
    }
    return object;
}
```

Listing 8.9: Built-in Lila function make implemented in Java and invoked with Lila values

For example, class definitions are expanded into variable definitions of class instantiations, i.e., the following two definitions are equivalent.

```
defclass <foo> (<object>) { bar };
```

```
def <foo> = make(<class>, "<foo>", make-array(<object>), make-array("bar"));
```

As the Lila class `<class>` is built-in, its internal `make` method is looked up and invoked with the additional arguments, the superclasses and properties. The Lila values are extracted from the array and unboxed beforehand. The method is shown in Listing 8.10.

```
static LilaClass make(String name, LilaClass[] superclasses, String[] properties) {
    if (superclasses == null || superclasses.length == 0)
        superclasses = new LilaClass[] { LilaObject.lilaClass };
    LilaClass result = new LilaClass(false, name, null, superclasses);
    result.classProperties = properties;
    return result;
}
```

Listing 8.10: The internal Java method `LilaClass.make` is invoked with unboxed values

8.3.4 Functions

Definition

A function expression is compiled into a static Java method with equivalent signature, i.e., Java's native calling convention is used. Nested functions are hoisted to the top-level through a lambda lifting (closure conversion) pass. At the same time also free variables are eliminated and closures are created. This is commonly implemented by prepending an environment parameter to functions and replacing free variables with a lookup into this environment. Another common approach, e.g. used in Clojure, is passing the values by using instance fields.

The Lila compiler keeps using the native calling convention instead. First, free variables are prepended to the parameter lists of outer functions in an analysis pass before generating code, by descending into nested functions and recursively adding closed parameters to parent functions. Partial function application is used to create the closure.

A function definition is, like a class definition, only syntactic sugar for a variable definition with an anonymous function value. The example shown in Listing 8.11 generates an adder function for a given value, expecting another argument and performing the summation of both arguments. Note that the inner function references a parameter introduced in the outer function. Listing 8.12 shows the bytecode generated by the compiler.

```
defn adder (a) {
  fn (b) {
    +(a, b)
  }
};
```

Listing 8.11: Adder function

```
public static LilaObject run();
  Code:
     0: invokedynamic #43,  0
        // InvokeDynamic #2:__fun1:()LLilaFunction;
     5: areturn

public static LilaObject __fun1(LilaObject);
  Code:
     0: invokedynamic #35,  0
        // InvokeDynamic #2:__fun2:()LLilaFunction;
     5: aload_0
     6: invokevirtual #41
        // Method LilaFunction.close:(LLilaObject;)LLilaFunction;
     9: areturn

public static LilaObject __fun2(LilaObject, LilaObject);
  Code:
     0: invokedynamic #20,  0
        // InvokeDynamic #0:"+":()LLilaObject;
     5: aload_0
     6: aload_1
     7: invokedynamic #28,  0
         // InvokeDynamic #1:call:(LLilaObject;LLilaObject;LLilaObject;)LLilaObject;
    12: areturn
```

Listing 8.12: Bytecode generated for the adder function

During the compilation all generated methods are recorded and at load time registered in the runtime, which creates method handle for the method and boxes it as a `LilaFunction`, as shown in Listing 8.13.

```
static Map<String,LilaFunction> FUNCTIONS = new HashMap<>();

static void registerInternalFunction
    (Class<?> clazz, String internalName, String name,
     boolean isVar, Class<?> rtype, Class<?>... ptypes)
    throws Throwable
{
    MethodType type = methodType(rtype, ptypes);
    MethodHandle handle = lookup.findStatic(clazz, internalName, type);
    LilaFunction function = new LilaFunction(name, handle);
    function.setVariadic(isVar);
    FUNCTIONS.put(internalName, function);
}
```

Listing 8.13: Generated methods are registered in the runtime

The actual function expression is only compiled into an `invokedynamic` instruction, similar to variable references. The instruction refers to the name of the generated method, takes no arguments and and will return the function object. It makes use of the runtime's bootstrap method `bootstrapFunction`, shown in Listing 8.14, which looks up the registered method by name and returns a constant call site targeting a method handle constantly returning the function that was created when the method was registered. An analysis has shown that the function value is directly loaded onto the stack and no method call is performed.

```
static CallSite bootstrapFunction (Lookup lookup, String name, MethodType type) {
    name = StringNames.toSourceName(name);
    LilaFunction function = FUNCTIONS.get(name);
    MethodHandle valueHandle =
        MethodHandles.constant(LilaFunction.class, function);
    return new ConstantCallSite(valueHandle);
}
```

Listing 8.14: Bootstrap method for functions

For each parameter that was added during the closure conversion pass, the parameter value is loaded onto the operand stack and closed over through the `close` method of `LilaFunction` (shown in Listing 8.15), which partially applies the underlying method handle using the previously presented `bindTo` method and creates a new function object for it.

```
public LilaFunction close(LilaObject value) {
    MethodHandle boundMethodHandle =
        this.methodHandle.bindTo(value);
    LilaFunction function =
        new LilaFunction(this.getName(), boundMethodHandle);
    function.setVariadic(this.isVariadic());
    return function;
}
```

Listing 8.15: Closure creation by partially applying method handle

Invocation

A function call is compiled into an `invokedynamic` instruction calling the runtime's bootstrap method `bootstrapCall`. For example, the call of the function + in the previously shown adder example generates the following bytecode:

```
InvokeDynamic #1:call:(LLilaObject;LLilaObject;LLilaObject;)LLilaObject;
```

Both the expected result type and the types of all arguments are `LilaObject`, as Lila is dynamically typed and the concrete types of the arguments at the given call site are unknown. Static analysis could infer more specific types, but this was not implemented. The signature includes both the function object and the arguments. The name passed to the bootstrap method is ignored, as the function object is invoked by value and not by name. The constant `"call"` is used in all cases and only for debugging purposes.

The bootstrap method has no access to the operands on the stack and therefore it is not able to perform the dispatch and construct an appropriate call site. Instead, it first delegates this task to a fallback method. This indirection is only performed for the first invocation and is similar to the approach previously shown for the implementation of inline caches. This is done by creating a mutable call site and initializes it with an adapter collecting all but the first argument (invoked function) into an array. The adapter in turn will call another adapter invoking the fallback method with the first argument bound to the call site. The procedure is shown in Listing 8.16.

```
1 static CallSite bootstrapCall(Lookup lookup, String name, MethodType type) {
2     CallSite callSite = new LilaCallSite(type);
3     MethodHandle target = fallback.bindTo(callSite)
4         .asCollector(LilaObject[].class, type.parameterCount() - 1);
5     callSite.setTarget(target);
6     return callSite;
7 }
```

Listing 8.16: Bootstrap method for function calls delegates dispatch to fallback method

The fallback method can perform the actual dispatch and linking, i.e., the target of the call site is updated with the actual target method handle, a polymorphic inline cache similar to the one presented in the example of the previous section (see Section 7.3.2). The major difference is the guard, which instead of checking a single type, checks if the call site's target method handle matches the method handle that was determined, as functions are first-class values and the call site may refer to several different functions. As the invoked function is passed as an argument, the call site's signature is incompatible with the target method handle. Therefore, an adapter method handle is created. Listing 8.17 contains the relevant section of the fallback method. The adapter drops this first argument, adapts the types of the arguments and calls the actual target method handle determined by `methodHandleForArguments`, which is explained in detail in the next section.

```
LilaFunction function = (LilaFunction)callable;
MethodType callSiteType = callSite.type();
int argumentCount = callSiteType.parameterCount() - 1;
MethodHandle mh = methodHandleForArguments(function, argumentCount);
MethodHandle target = MethodHandles
    .dropArguments(mh, 0, LilaObject.class)
    .asType(callSiteType);
// ...
```

Listing 8.17: The fallback method for function invocations adapts the call site to the target method

Variable parameter lists

Lila functions can have variable parameter lists accepting additional arguments after any required parameters. The ellipsis keyword (`...`) separates the required parameters from the rest parameter, which is passed all additional arguments as an array (cf. ellipsis in Java, `&rest` in Common Lisp, `#rest` in Dylan, etc.). For example, the function `make-array` returns an array containing all passed arguments and could be implemented as follows:

```
defn make-array (...rest) {
  rest
};
```

In Java, the calling convention for variadic methods differs from normal method calls. The compiler statically determines the arity of the method. If the method is variadic, an array is created and the additional arguments are stored into it directly at the call site. Compilers of dynamically typed languages or languages performing advanced method dispatch might be unable to determine the arity of the target method. Other languages perform this adaption in the simulator method. Using the `invokedynamic` instruction this step can be moved into the bootstrap method and the compiler needs no special case for variadic calls, i.e., can remain using the normal calling convention.

```
MethodHandle methodHandleForArguments(LilaFunction function, int argumentCount) {
    MethodHandle handle = function.methodHandle;
    int requiredParameterCount = handle.type().parameterCount();
    if (function.isVariadic())
        requiredParameterCount--;
    if (function.isVariadic() && argumentCount >= requiredParameterCount) {
        int pos = requiredParameterCount;
        handle = MethodHandles.filterArguments(handle, pos, boxAsArray);
        int count = (argumentCount - requiredParameterCount);
        handle = handle.asCollector(LilaObject[].class, count);
    }
    return handle;
}
```

Listing 8.18: Adaption if the target method handle to the call site

In Lila, this automatic adaption of the call site to the possible variadic function is performed in the method `methodHandleForArguments`, which returns the target method handle for a given function and call site argument count. It is shown in Listing 8.18.

The method checks if the called method is variadic and additional arguments were passed. If this is the case, the function's underlying method handle is wrapped with one adapter collecting the additional arguments into a Java array, which invoked another adapter filtering this last argument, boxing the Java array into a Lila array. The method handle `boxAsArray` is not referencing a method of the runtime, but is simply a method handle for the constructor of `LilaArray`:

```
lookup.findConstructor(LilaArray.class,
                       methodType(void.class, LilaObject[].class));
```

The built-in function `apply` allows calling a function with arguments given in an array. Like a normal method call, it simply uses `methodHandleForArguments` to build an appropriate adapter method handle for the given arguments and invokes it with `invokeWithArguments`.

```
LilaObject apply(LilaObject[] arguments) {
  return (LilaObject)methodHandleForArguments(this, arguments.length)
      .invokeWithArguments((Object[])arguments);
}
```

Chapter 9

Dynamic Dispatch in Lila

> *There are four general techniques for speeding up an algorithm: caching, compiling, delaying computation, and indexing.*
>
> — Peter Norvig,
> *A Retrospective on PAIP [50], lesson 21.*

Lila supports both symmetric multiple dispatch using the concept of multi-methods, and predicate dispatch with arbitrary expressions. This chapter describes the usage and implementation details of both features.

9.1 Multiple dispatch

9.1.1 Usage

Like CLOS and Dylan, Lila supports dynamic multiple dispatch. Method implementations are defined using the keyword `defmm`, which adds a new method implementation to the multi-method of the given name, implicitly creating a multi-method if does not exist yet. The grammar for multi-method definitions is shown in Figure 9.1. Each parameter of a method implementation can be optionally specialized for a particular class by using the the double colon `::` keyword. If no parameter specializer is given, the class `<object>` is used by default. The keyword should not be confused with the scope resolution operator used in languages like C++ and Ruby.

⟨*definition*⟩ ::= . . .
| "**defmm**" ⟨*identifier*⟩ ⟨*typed-parameter-list*⟩ ⟨*body*⟩

⟨*typed-parameter-list*⟩ ::= "**(**" [⟨*typed-parameter*⟩ ("**,**" ⟨*typed-parameter*⟩)*
["**...**" ⟨*identifier*⟩]] "**)**"

⟨*typed-parameter*⟩ ::= ⟨*identifier*⟩ ["**::**" ⟨*expression*⟩]

Figure 9.1: Grammar for multi-method definitions

Listing 9.1 shows an example demonstrating the definition of multi-methods and the usage of multiple dispatch in combination with multiple inheritance.

```
defclass <a> (<object>);
defclass <b> (<a>);
defclass <c> (<object>);
defclass <d> (<b>, <c>);

defmm foo (x :: <a>, y :: <c>) { 1 };
defmm foo (x :: <b>, y :: <d>) { 2 };

def b = make(<b>);
def c = make(<c>);
def d = make(<d>);

foo(b, c); // ⇒ 1
foo(d, d); // ⇒ 2
```

Listing 9.1: Multiple dispatch using multi-methods in Lila

When new classes and new method implementations are added at run-time, the multiple dispatch mechanism takes these into account. Listing 9.2 demonstrates this behavior.

```
defclass <e> (<d>);
def a = make(<a>);
def e = make(<e>);
foo(a, e); // ⇒ 1

defmm foo (x :: <a>, y :: <e>) { 3 };
foo(a, e); // ⇒ 3
```

Listing 9.2: Multi-methods consider dynamically added classes and method implementations

Similar to normal functions, multi-methods support variable argument lists, and applying argument lists using the built-in function `apply`. An example is shown in Listing 9.3.

```
defmm bar (x :: <a> ... rest) {
    concatenate(make-array(1, 2), rest)
};

defmm bar (x :: <b> ... rest) {
    concatenate(make-array(3, 4), rest)
};

bar(a, 3, 4); // ⇒ [1, 2, 3, 4]
apply(bar, make-array(b, 5, 6)); // ⇒ [3, 4, 5, 6]
```

Listing 9.3: Multi-methods support variable argument lists

Similar to Dylan and CLOS, the method implementation is passed an implicit `next-method` function that invokes the next most applicable method implementation. The function can be called with the current arguments, new arguments or no arguments. In the

latter case, the arguments of the current method invocation are automatically passed. The function solely invokes the next most specific method implementation without performing any dispatch. If no further applicable method is available, the variable `next-method` is bound to `false`. The method implementations shown in Listing 9.4 make use of the `next-method` function. The program generates the following output:

```
baz(<c>)
#[Object #[Class <c>]]
#[Function baz]
baz(<b>)
#[Object #[Class <c>]]
#[Function baz]
baz(<a>)
#[Object #[Class <b>]]
```

```
defmm baz (x :: <a>) {
  print("baz(<a>)");
  print(as-string(x));
  print(as-string(next-method));
};

defmm baz (x :: <b>) {
  print("baz(<b>)");
  print(as-string(x));
  print(as-string(next-method));
  next-method(b);
};

defmm baz (x :: <c>) {
  print("baz(<c>)");
  print(as-string(x));
  print(as-string(next-method));
  next-method();
};

baz(c);
```

Listing 9.4: Multi-methods are passed an implicit `next-method` function that invokes the next most specific method implementation

9.1.2 Implementation

Lila's implementation of multiple dispatch is based on the Single-Receiver Projection [38] implementation technique, that was previously described in Section 5.3.1. The technique was chosen in particular because it supports dynamic dispatch, has good dispatch time performance, is space efficient, supports calling the next most specific method implementation, and is easy to implement. Most importantly, it also supports incremental updates of the dispatch structures efficiently.

Each method implementation is compiled into static Java method with a leading additional argument that is used for the `next-method` function. For each multi-method, a SRP dispatcher is created, which stores references to all method implementations, ordered by specificity. The definition is shown in Listing 9.5. For the purpose of ordering the method implementations, the C3 superclass linearization algorithm was adopted, that was previously presented in Section 5.2. The dispatch tables are lists containing bitsets, so the dispatch structure is memory efficient, dispatch is fast, and incremental updates can be easily performed.

```
public class SRPDispatcher {
    ArrayList<ArrayList<BitSet>> tables = new ArrayList<>();
    ArrayList<Method> methods = new ArrayList<>();

    public SRPDispatcher(int arity) {
        for (int i = 0; i < arity; i++)
            tables.add(new ArrayList<BitSet>());
    }
    // ...
```

Listing 9.5: The SRP dispatcher contains the set of methods and one single dispatch table for each argument position

As Lila is dynamic and supports updates to the sets of methods and classes, the SRP dispatch tables need to be incrementally updated. When a new class is added to the environment, all multi-methods are updated by adding a new entry to each of the single dispatch tables containing the applicable method implementations. When a new method implementation is added to the multi-method, the position of the method implementation is searched in the bitset according to the specificity, and the bitset is shifted to make space for the new implementation. These operations are performed by the methods `addNewClass` and `addNewMethod` respectively, as shown in Listing 9.6.

```
void addMethod(List<BitSet> table, LilaClass specializer, int pos) {
    for (LilaClass subtype : specializer.getAllSubclasses()) {
        int identifier = subtype.getIdentifier();
        BitSet bitset = table.get(identifier);
        if (bitset == null) {
            bitset = new BitSet();
            table.set(identifier, bitset);
        }
        bitset.set(pos);
    }
}

void addNewClass(LilaClass type) {
    for (int i = 0; i < tables.size(); i++) {
        ArrayList<BitSet> table = tables.get(i);
        table.add(type.getIdentifier(), null);
        int j = 0;
        for (Method method : methods)
            if (type.isSubtypeOf(method.getSpecializer(i)))
                addMethod(table, type, j++);
    }
}
```

```
23
24 void addNewMethod(Method method) {
25     int index = Collections.binarySearch(methods, method);
26     if (index < 0)
27         index = ~index;
28     methods.add(index, method);
29     for (int i = 0; i < tables.size(); i++) {
30         ArrayList<BitSet> table = tables.get(i);
31         for (BitSet bitset : table) {
32             if (bitset == null)
33                 continue;
34             for (int j = bitset.length(); j > index; j--)
35                 bitset.set(j, bitset.get(j - 1));
36             bitset.set(index, false);
37         }
38         addMethod(table, method.getSpecializer(i), index);
39     }
40 }
```

Listing 9.6: Adding new classes and methods to the dispatch tables

Dispatch is performed by using the unique class identifiers of all argument types to index into the dispatch tables, logically anding the bitsets of all entries, and using each position of the bitset that results to index into the list of methods. The result is a list of all applicable methods, sorted by specificity. The operation is shown in Listing 9.7.

```
1  static MethodHandle[] noMethods = new MethodHandle[0];
2
3  BitSet getSet(LilaClass[] types, int pos) {
4      int identifier = types[pos].getIdentifier();
5      return tables.get(pos).get(identifier);
6  }
7
8  MethodHandle[] dispatch(LilaClass[] types) {
9      BitSet set = getSet(types, 0);
10     if (set == null)
11         return noMethods;
12     set = (BitSet)set.clone();
13     for (int i = 1; i < types.length; i++) {
14         BitSet otherSet = getSet(types, i);
15         if (otherSet == null)
16             return noMethods;
17         set.and(otherSet);
18     }
19     MethodHandle[] result = new MethodHandle[set.cardinality()];
20     for (int i = 0, j = set.nextSetBit(0);
21          j >= 0;
22          j = set.nextSetBit(j + 1), i++)
23     {
24         result[i] = this.methods.get(j).getMethodHandle();
25     }
26     return result;
27 }
```

Listing 9.7: SRP dispatch for a given set of argument types

Exisiting implementations of multiple dispatch on the JVM compile to static and non-extensible type cascades or the visitor pattern, or are dynamic and implement custom simulator objects or use the Java reflection API [32]. Lila's compiler and runtime rely on the newly introduced `invokedynamic` instruction, which is more flexible, allowing the efficient implementation of dynamic and extensible multi-methods.

Multi-method calls are implemented similarly to functions. The method `bootstrap–Call` is creating a call site and binding it to a fallback method, so arguments on the stack can be examined, in particular the called object. In this case, the fallback method handles multi-method calls and uses the method `targetHandle`, as shown in Listing 9.8, to construct the target method handle for the specific call. First, all argument types are extracted and the SRP dispatcher is used to determine all applicable method implementations for the particular call. After that, the `next–method` function is created for all further applicable method implementations. Next, the first argument of the method handle of the most specific method implementation is bound to the `next–method` function. Finally, the resulting method handle is adapted to the particular call site, i.e., additional arguments are collected into an array, if the multi-method is variadic.

```
1  static MethodHandle targetHandle
2      (LilaMultiMethod mm, LilaObject[] args, LilaClass[] types, int argumentCount)
3  {
4      MethodHandle[] methods = mm.dispatcher.dispatch(types);
5      LinkedList<MethodHandle> nextMethods = new LinkedList<>();
6      for (int i = 1; i < methods.length; i++)
7          nextMethods.add(methods[i]);
8      LilaObject nextMethod = makeNextMethod(mm, nextMethods, args);
9      MethodHandle mh = methods[0].bindTo(nextMethod);
10     return methodHandleForArguments(mm, mh, argumentCount);
11 }
```

Listing 9.8: Construction of the target method handle for a multi-method invocation

The construction of the `next–method` function consists of binding the `callNextMethod` method, which performs the call of the next most specific method implementation, to the multi-method, the remaining applicable method implementations, and argument count. The invocation method determines which arguments should be used for the call, and creates a new adapting method handle. The construction method is shown in Listing 9.9, and the invocation method is shown in Listing 9.10.

```
1  static LilaObject makeNextMethod
2      (LilaMultiMethod mm, LinkedList<MethodHandle> nextMethods, LilaObject[] args)
3  {
4      if (nextMethods.size() == 0)
5          return LilaBoolean.FALSE;
6      MethodHandle nextMethod = callNextMethod
7          .bindTo(mm).bindTo(nextMethods).bindTo(args);
8      LilaFunction fn = new LilaFunction(mm.name, nextMethod);
9      fn.setVariadic(true);
10     return fn;
11 }
```

Listing 9.9: Construction of the `next–method` function

```
public static LilaObject callNextMethod
    (LilaMultiMethod mm, LinkedList<MethodHandle> nextMethods,
     LilaObject[] oldArgs, LilaArray args)
    throws Throwable
{
   MethodHandle mh = nextMethods.remove();
   LilaObject[] newArgs = args.array;
   LilaObject[] passedArgs = newArgs.length > 0 ? newArgs : oldArgs;
   LilaObject nextMethod = makeNextMethod(mm, nextMethods, passedArgs);
   mh = mh.bindTo(nextMethod);
   mh = methodHandleForArguments(mm, mh, passedArgs.length);
   return (LilaObject)mh.invokeWithArguments((Object[])passedArgs);
}
```

Listing 9.10: Calling the next most specific method implementation

Binding the call site directly to the resulting dispatched target method handle is invalid, as other multi-methods, or even other functions, as well as other argument types could occur. Similar to functions, the handle needs to be guarded by a check to ensure the target method handle is still valid. Building on this idea, a a polymorphic inline cache could be constructed, but the test would not only check the called object, but also all argument types. The hypothetical `check` method could be defined like in Listing 9.11, and would be partially applied with the multi-method and types that for which the target method handle is valid.

```
public static boolean check
    (LilaMultiMethod siteMM, LilaClass[] types, LilaMultiMethod mm, LilaObject... args)
{
    if (mm != siteMM)
        return false;
    for (int i = 0; i < types.length; i++) {
        if (types[i] != args[i].getType())
            return false;
    }
    return true;
}
```

Listing 9.11: Hypothetical definition of a polymorphic inline cache test for multi-methods

However, this solution is inefficient, as each test in the linear search checks the multi-method and all types. A better solution is the construction of a polymorphic inline cache for each level (multi-method, first argument type, second argument type, etc.), essentially resulting in a dispatch tree similar to Partial Dispatch. On each fallback the resulting target method handle is cached for the given multi-method and argument types (see Listing 9.12). This separate cache is necessary, as the whole inline cache needs to be recompiled, because the method handle API does not provide any means to update existing guard method handles.

This solution is faster, as each guard performs only one test. In addition, each cached target method handle is already bound to the appropriate `next-method` function, so it does not need to be recreated.

```
void cache(LilaMultiMethod mm, LilaClass[] types, MethodHandle mh) {
    Map entry = multiMethodCache.get(mm);
    if (entry == null) {
        entry = new HashMap<>();
        this.multiMethodCache.put(mm, entry);
    }
    int last = types.length - 1;
    for (int i = 0; i < last; i++) {
        LilaClass type = types[i];
        Map currentEntry = (Map)entry.get(type);
        if (currentEntry == null) {
            currentEntry = new HashMap<>();
            entry.put(type, currentEntry);
        }
        entry = currentEntry;
    }
    entry.put(types[last], mh);
}
```

Listing 9.12: Secondary call site cache used for the construction of the primary polymorphic inline cache

The actual polymorphic inline cache is constructed recursively, as shown in Listing 9.13. On the first level the multi-method is checked using the method `checkMethod`, and on the remaining levels the argument types are checked using the method `checkType`. The resulting cache method handle is wrapped by an adapter method handle collecting all arguments into an array, so each test has access to all arguments. As a result, it is also necessary to wrap the target method handle with and adapter spreading out all arguments again.

9.1.3 Evaluation

The multiple dispatch implementatin was evaluated by conducting two experiments that focus on measuring the elapsed time for method invocations. Each experiment performed 10 iterations of 10 million double dispatch method invocations. In the first experiment the call site was monomorphic, i.e., always the same arguments were passed to the method. In the second experiment the call site was polymorphic, i.e., for each argument position an object was passed to the method that was chosen randomly from a given set of applicable objects. The source code of these experiments can be found in Appendix A.

The experiments were conducted in Java, Common Lisp, Dylan, Groovy, and Lila. The experiments written in Java made use of `instanceof` cascades and the Visitor pattern. The experiments written in Common Lisp were evaluated in two implementations: SBCL generates native code, and ABCL runs on the JVM. The experiments written in Dylan were evaluated using the OpenDylan compiler, which generates native code. Except Java, all languages are dynamic. The languages not running on the JVM were added for reference purposes.

The results of the monomorphic experiment are shown in Figure 9.2. Both versions written in Java performed comparable and ahead of all others. It is likely that the code of the method implementations were inlined. The versions written in Dylan and Common Lisp, using compilers generating native code, were an order of magnitude slower. The ver-

```java
static MethodHandle compile(LilaCallSite callSite, int argumentCount) {
    LinkedList<LilaMultiMethod> methods = new LinkedList<>();
    methods.addAll(callSite.multiMethodCache.keySet());
    return compileLevel0(callSite, argumentCount, methods)
        .asCollector(LilaObject[].class, callSite.type().parameterCount());
}

static MethodHandle compileLevel0
    (LilaCallSite callSite, int count, LinkedList<LilaMultiMethod> remaining)
{
    if (remaining.size() == 0)
        return getFallback(callSite, count);
    LilaMultiMethod mm = remaining.remove();
    Map entry = callSite.multiMethodCache.get(mm);
    LinkedList<LilaClass> types = new LinkedList<>();
    types.addAll(entry.keySet());
    return MethodHandles.guardWithTest(checkMethod.bindTo(mm),
                                       compileLevelN(1, callSite, count, entry, types),
                                       compileLevel0(callSite, count, remaining));
}

static MethodHandle getFallback(LilaCallSite callSite, int argumentCount) {
    return RT.fallback.bindTo(callSite)
        .asCollector(LilaObject[].class, argumentCount - 1)
        .asSpreader(LilaObject[].class, argumentCount);
}

static boolean checkMethod(LilaMultiMethod mm, LilaObject[] args) {
    return args[0] == mm;
}

static MethodHandle compileLevelN
    (int n, LilaCallSite cs, int count, Map entry, LinkedList<LilaClass> remaining)
{
    if (remaining.size() == 0)
        return getFallback(cs, count);
    LilaClass type = remaining.remove();
    MethodHandle target;
    if (n + 1 == count) {
        MethodHandle mh = (MethodHandle)entry.get(type);
        target = mh.asSpreader(LilaObject[].class, count);
    } else {
        Map nextEntry = (Map)entry.get(type);
        LinkedList<LilaClass> types = new LinkedList<>();
        types.addAll(nextEntry.keySet());
        target = compileLevelN(n + 1, cs, count, nextEntry, types);
    }
    return MethodHandles.guardWithTest(checkType.bindTo(type).bindTo(n),
                                       target,
                                       compileLevelN(n, cs, count, entry, remaining));
}

static boolean checkType(LilaClass type, Integer pos, LilaObject[] args) {
    return args[pos].getType() == type;
}
```

Listing 9.13: Recursive compilation of the secondary call site cache into the primary polymorphic inline cache consisting of guard method handles

sion written in Lila had an initial overhead, which is caused by the construction and update of the cache, and accompanying just-in-time compilation and optimization. Performance improved significantly with each iteration after the initialization phase. Each iteration in Lila took only twice as long as in Dylan. It is likely that the remaining overhead is caused by the adaption of the call site. Finally, the slowest implementation was Groovy, which had an initial overhead similar to Lila and its performance imoroved, but each iteration still took three times as long as an iteration in Lila.

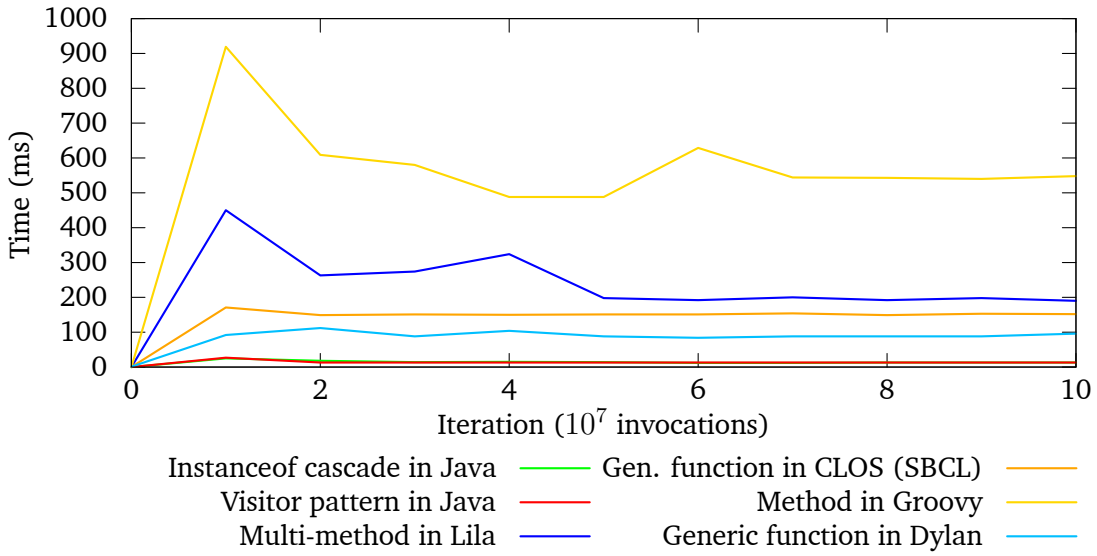


Figure 9.2: Results of the monomorphic call site experiment for multiple dispatch. Lower running times are better.

The results of the polymorphic experiment are shown in Figure 9.3. The running time of each iteration was adjusted by the amount of time required for obtaining the two random argument values. Also note that the time scale is logarithmic. Again, both versions written in Java performed comparable and ahead of all others. Their running times are considered the baseline for the following comparisons. The version written in Common Lisp (using ABCL), and the version written in Groovy performed worst, spending almost 400 times as long for each iteration as the versions in Java. Dylan performed better, but still took 64 times as long. Initially, the version written in Lila and Common Lisp (using SBCL) performed similarly. After the initialization and optimization phase, the performance of the version written in Lila improved significantly. On average the version using SBCL took 12 times as long, the version in Lila only 3 times as long.

In conclusion, the implementation of multiple dispatch in Lila is very efficient, especially for highly polymorphic call sites, outperforming all other dynamic languages. Even tough the implementation is not as performant as the manual double dispatch written in Java, it can be considered practical and provides many advantages, such as invoking the next most specific method implementation, extensibility, i.e., new method implementations can be provided independently, and support for dynamism in the form of changes to the sets of methods and classes.

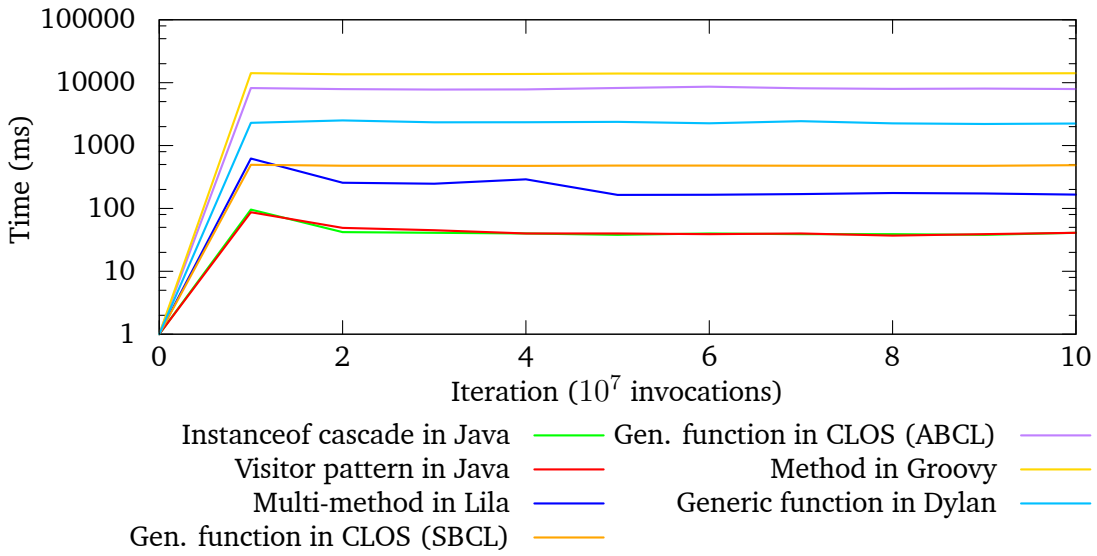


Figure 9.3: Results of the polymorphic call site experiment for multiple dispatch. Lower running times are better.

9.2 Predicate dispatch

9.2.1 Usage

Lila supports dynamic predicate dispatch. Similar to multi-methods, predicate methods have several method implementations associated to them, each guarded by a predicate. Predicates may consist of conjunctions, disjunctions, class tests and tests of arbitrary expressions. Predicate methods support variable argument lists, and consider dynamically added classes and method implementations. However, calling the next most specific method implementation is not possible.

Predicate method implementations are defined using the keyword `defpm`, which adds a new method implementation to the predicate method of the given name, implicitly creating one if does not exist yet. The grammar for predicate method definitions is shown in Figure 9.4.

⟨*definition*⟩ ::= ...
| "**defpm**" ⟨*identifier*⟩ ⟨*parameter-list*⟩
"**when**" ⟨*predicate*⟩ ⟨*body*⟩

⟨*predicate*⟩ ::= ⟨*predicate*⟩ "**||**" ⟨*predicate*⟩
| ⟨*predicate*⟩ "**&&**" ⟨*predicate*⟩
| "**test**" ⟨*expression*⟩
| "**not**" ⟨*predicate*⟩
| ⟨*type-predicate*⟩
| ⟨*expression-binding*⟩
| "**(**" ⟨*predicate*⟩ "**)**"

$$\langle \textit{type-predicate} \rangle \quad ::= \quad \langle \textit{expression} \rangle \text{ "::" } \langle \textit{expression} \rangle$$

$$\langle \textit{expression-binding} \rangle \quad ::= \quad \langle \textit{identifier} \rangle \text{ ":=" } \langle \textit{expression} \rangle$$

Figure 9.4: Grammar for predicate method definitions

Listing 9.14 contains the running example demonstrating the definition and usage of predicate methods. The method `isomorphic?` determines if two binary trees are isomorphic.

```
1  defclass <tree-node> (<object>);
2  defclass <data-node> (<tree-node>) { left; right };
3  defclass <empty-node> (<tree-node>);
4
5  defpm isomorphic? (t1, t2)
6    when (t1 :: <empty-node>)
7      && (t2 :: <empty-node>)
8  { true };
9
10 defpm isomorphic? (t1, t2)
11   when (t1 :: <empty-node>)
12     || (t2 :: <empty-node>)
13 { false };
14
15 defpm isomorphic? (t1, t2)
16   when (t1 :: <data-node>)
17     && (t2 :: <data-node>)
18 {
19   isomorphic?(get(t1, "left"), get(t2, "left"))
20     && isomorphic?(get(t1, "right"), get(t2, "right"));
21 };
22
23 def empty = make(<empty-node>);
24 def empty-tree = make(<data-node>, empty, empty);
25 def tree1 = make(<data-node>, empty, empty-tree);
26 def tree2 = make(<data-node>, empty-tree, empty);
27
28 isomorphic?(empty, empty); // ⇒ true
29 isomorphic?(empty, tree1); // ⇒ false
30 isomorphic?(tree1, tree2); // ⇒ false
31 isomorphic?(tree1, tree1); // ⇒ true
32 apply(isomorphic?, make-array(empty, empty)); // ⇒ true
```

Listing 9.14: Determining binary tree isomorphism using predicate dispatch

9.2.2 Implementation

The implementation of the predicate dispatch is based on the dispatch tree implementation technique [11], described previously in Section 6.3, that canonlicalizes the predicates into a simpler form, constructs a lookup DAG, and create a decision tree (single dispatch tree) for each node of the DAG.

Each method implementation that is added to the predicate method is compiled into a Java method with the same signature, as no implicit `next-method` argument is needed. Similarly, each expression that appears inside the implementation's predicate is compiled into a Java method. A cache stores the method handle for each expression and ensures that each expression is only compiled once. For example, the expression `t1` that appears in the predicates of all method implementations of the running example is only compiled once into the following Java method:

```
public static LilaObject __exp1(LilaObject, LilaObject);
  Code:
     0: aload_0
     1: areturn
```

The predicate of the new method implementation is first canonicalized into a disjunctive normal form that only consists of class tests. Each predicate method contains a set of cases, a mapping of a conjunction to a set of applicable method implementations. The method implementation is added to the set of each conjunction of the resulting disjunction, and also to a list of all target method implementations. This procedure is shown in Listing 9.15.

```
public void addMethod(Predicate predicate, Method method) {
    for (Predicate conjunction : predicate.canonicalize()) {
        Case c = this.cases.get(conjunction);
        if (c == null) {
            c = new Case(conjunction);
            this.cases.put(conjunction, c);
        }
        c.methods.add(method);
    }
    if (!this.methods.contains(method))
        this.methods.add(method);
}
```

Listing 9.15: Adding a method implementation to a predicate method

Following that, the lookup DAG and the single dispatch tree for each node is constructed. There are three possible evaluation strategies of the resulting dispatch tree: Interpretation of the dispatch tree, invoking the target method handle when reaching a leaf node; compilation into a tree of guard method handles, similar to how multi-methods are compiled; and compilation into a dispatch function, directly generating bytecode.

All three evaluation strategies were implemented and evaluated. The last strategy, compiling the dispatch tree into executable code, was chosen because it performed best. For each single dispatch tree, first all arguments are loaded on the stack, the expression method for the tested expression is invoked, and the class identifier of the resulting object's type is determined. Then, the dispatch tree's nodes are compiled into less-than and equality branches. For each leaf node, the index of the resulting method implementation in the list of all target method implementations is stored in a local variable. At the end of the dispatch, the list of all method implementations is loaded, and the stored method index is used to get the actual target method implementation. An excerpt of the dispatch function that is generated for the runnting example is shown in Listing 9.16.

The fallback method for predicate methods is similar to that of functions. It constructs a polymorphic inline cache, with one entry for each predicate method. Each guard targets the method `invoke`, shown in Listing 9.17, which is partially applied with the predicate method. It calls the dispatch function and invokes the resulting method handle.

```
1 static LilaObject invoke (LilaPredicateMethod pm, LilaObject[] args) {
2     Method m = (Method)pm.dispatcher.invokeWithArguments((Object[])args);
3     return (LilaObject)m.getHandle().invokeWithArguments((Object[])args);
4 }
```

Listing 9.17: Dispatch and invocation of target method implementation

9.2.3 Evaluation

The predicate dispatch implementation was evaluated by conducting an experiment similar to the polymorphic call site experiment for multiple dispatch. The running example, determining tree isomorphism, was implemented in Lila and in JPred, the extension of Java with support for predicate dispatch. Each version performed 10 iterations of 10 million method invocations, i.e., the call site was highly polymorphic. The source code of these experiments can be found in Appendix B.

The results of the experiments are shown in Figure 9.5. The predicate method in Lila is an order of magnitude slower than the predicated method in JPred, and in both cases the dispatch time is constant. The experiment was also conducted for a monomorphic call site, but as the results are very similar to that of the polymorphic version they are not shown here.

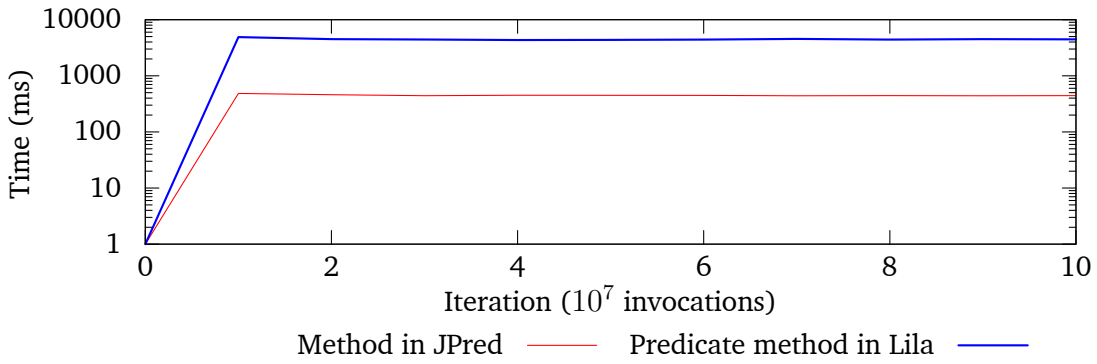


Figure 9.5: Results of the call site experiment for predicate dispatch

It is notable that an earlier implementation, based on compiling the dispatch tree to guard method handles instead of bytecode, was two orders of magnitude slower than the version in JPred. This suggests that the performance problem is likely caused by the remaining use of method handles. In the current implementation, the method `invoke` of the multi-method calls the dispatch function to obtain the specific method implementation, which is than subsequently called. It is likely that this indirection is probably not optimized by the JVM. It is possible that a direct invocation of the final method implementation using the `invokestatic` instruction in the leaf node of the dispatch tree performs better and needs further investigation.

Furthermore, this suggests that the implementation of multiple dispatch can be improved by compiling the polymophic inline cache to bytecode instead of method handles. The drawback of such an approach is the increased difficulty and complexity of the implementation.

```
public static Method dispatch(LilaPredicateMethod, LilaObject, LilaObject);
    Code:
       0: aload_1
       1: aload_2
       2: invokestatic  #12
          // Method __exp1:(LLilaObject;LLilaObject;)LLilaObject;
       5: invokevirtual #18
          // Method LilaObject.getType:()LLilaClass;
       8: invokevirtual #24
          // Method LilaClass.getIdentifier:()I
      11: istore_3
      12: iload_3
      13: ldc           #25     // int 2
      15: if_icmpge     101
      18: iload_3
      19: ldc           #26     // int 0
      21: if_icmpne     56
      24: aload_1
      25: aload_2
      26: invokestatic  #29
          // Method __exp2:(LLilaObject;LLilaObject;)LLilaObject;
      29: invokevirtual #18
          // Method LilaObject.getType:()LLilaClass;
      32: invokevirtual #24
          // Method LilaClass.getIdentifier:()I
      35: istore_3
      36: iload_3
      37: ldc           #25     // int 2
      39: if_icmpge     49
      42: ldc           #26     // int 0
      44: istore        4
      46: goto          53
      49: ldc           #30     // int 3
      51: istore        4
      53: goto          98
       // ...
     130: aload_0
     131: invokevirtual #37
          // Method LilaPredicateMethod.getMethods:()Ljava/util/List;
     134: iload         4
     136: invokeinterface #43,  2
          // InterfaceMethod java/util/List.get:(I)Ljava/lang/Object;
     141: checkcast     #45
          // class Method
     144: areturn
```

Listing 9.16: Excerpt of the dispatch function for the running example

Chapter 10

Related Work

This chapter describes and compares concepts, features and approaches of other virtual machines and programming environments with support for dynamic dispatch to those provided by the JVM.

The most notable and widely used virtual machine beside the JVM is Microsoft's Common Language Runtime (CLR). The Dynamic Language Runtime (DLR) is an extension to the CLR and includes means for implementing dynamic method dispatch efficiently on top of the CLR. Another virtual machine supporting advanced dynamic dispatch is Parrot, which is designed to run dynamic languages efficiently.

10.1 Dynamic Language Runtime

Like the JVM, the Common Language Runtime (CLR) is a virtual machine featuring garbage collection and just-in-time (JIT) compilation. It is part of the .NET framework, which also includes the Common Language Infrastructure (CLI), consisting of the Common Type System (CTS) and Common Language Specification (CLS). These specifications define the possible datatypes and language elements, how programs are executed and interactions between them. Thus, the CLR is designed to not only support one primary, but various different programming languages. All languages targeting a CLI-compatible runtime compile to Common Intermediate Language (CIL), which is an object-oriented, high-level and platform-independent instruction set, similar to that of the JVM. A proprietary implementation of the .NET framework is developed by Microsoft and the Mono project aims at providing a binary-compatible open-source implementation.

The CIL supports non-virtual static single dispatch through the `call` instruction, and dynamic single dispatch through the `callvirt` instruction (cf. JVM `invokevirtual`). Like in the JVM, dynamically dispatched method invocation are still statically checked at compile-time, and the generated instructions contain references to static types. There is no support for dynamic types and it is not possible to invoke methods and refer to types that are unknown at compile-time.

Unlike the JVM (through the `invokedynamic` instruction), the CIL does not provide any instruction to perform custom dispatch. Instead, the .NET Framework was extended with the Dynamic Language Runtime (DLR), a framework that provides a shared dynamic type system and a high-level API to perform dynamic dispatch. Call site simulator objects are instances of the class `CallSite`, parameterized by the return and parameter types. Associated to the call site object is a call site binder (instance of `CallSiteBinder`), which performs the method linking step and may contain further static information (e.g., the target name).

When the call site is invoked, the binder's `bind` method is used to link the call site to an actual target. It is conceptually similar to the JVM bootstrap method, but is also passed the arguments on the stack, so they can be used for the linking process directly, without first binding the target to an intermediate method performing the actual dispatch and binding the final target. Also, the binding method is not passed the call site object and does not bind the call site to a particular method directly, but returns an expression tree, that is then compiled and linked by the DLR. For this purpose, the bind method is also passed expressions representing the parameters and the return label. For example, the bind method could simply return arbitrary code, represented as an expression, or indirectly bind the call site by returning a call expression invoking a method reference that was obtained through reflection.

There is also no distinction between mutable and constant call sites like in the JVM. Instead, the expression returned from the bind method may contain a restriction expression. If the expression is not guarded by a restriction, the call site is constantly bound to the expression. If it contains a restriction, it is evaluated by the framework to determine if the target expression is valid for the current invocation. In case it is invalid, the bind method is invoked again to obtain a new, valid target expression. Furthermore, the expressions and restrictions are cached, i.e., it is not necessary to manually build an inline cache. As a result, the bind method is only used if no cached expression can be found, for which the restriction holds. Listing 10.1 shows an example of this behavior. The resulting expression is guarded (through `Expression.IfThen`) by a type check of the first argument. As a result, the second invocation is still valid. However, it fails for the third invocation and the bind method is called to obtain a new target expression. Note that the last invocation does not result in a rebind, as a valid expression is available in the cache.

```
1  public class TestBinder : CallSiteBinder {
2      public override Expression Bind
3          (object[] args, ReadOnlyCollection<ParameterExpression> parameters,
4           LabelTarget returnLabel)
5      {
6          Console.WriteLine("bind");
7          Type type = args[0].GetType();
8          ParameterExpression param = parameters[0];
9          Expression typeExpression =
10             Expression.Call(param, typeof(object).GetMethod("GetType"));
11         Expression typeTest = Expression.Equal(typeExpression,
12                                                Expression.Constant(type));
13         MethodInfo method = typeof(Test).GetMethod("answer" + type.Name);
14         Expression call = Expression.Call(null, method);
15         return Expression.IfThen(typeTest,
16                                  Expression.Return(returnLabel, call));
17     }
18 }
19
20 class Life {}
21
22 public class Test {
23     public static int answerLife() { return 42; }
24     public static int answerTest() { return 23; }
25
26
```

```
27      static void call(CallSite<Func<CallSite, object, int>> site, Object value) {
28          int result = site.Target(site, value);
29          Console.WriteLine(result);
30      }
31
32      public static void Main(string[] args) {
33          CallSiteBinder binder = new TestBinder();
34          var site = CallSite<Func<CallSite, object, int>>.Create(binder);
35          Life life = new Life();
36          Test test = new Test();
37          call(site, life); // bind, 42
38          call(site, life); // 42
39          call(site, test); // bind, 23
40          call(site, life); // 42
41      }
42  }
```

Listing 10.1: Example of a dynamically bound call site implemented in C# using the DLR demonstrating the caching behavior

In addition to the late-binding mechanism, the DLR also simplifies the implementation of programming languages in general. Compilers can be implemented by generating an AST based on expression trees, which are generic and language-agnostic, instead of emitting intermediate code directly or through a third-party library, like on the JVM. The DLR then lazily compiles portions of the AST into CIL when necessary. This is especially useful for implementors of dynamic languages. Using the DLR, many languages have been implemented to run on the CLR, such as IronRuby, IronPython, IronScheme, or IronLua. For example, IronPython first generates an AST specific to Python and then transforms it into a DLR AST. The DLR also provides a hosting API, which allows interoperability with other languages running on the CLR, by representing objects and operations using a shared MOP. For instance, objects may extend `DynamicObject` and implement the method `TryInvokeMember`, which is called when a certain method is invoked on the particular object.

Unlike Java, the majority of the languages for the CLR were extended to support this interoperability mechanism. For example, C# was extended with the `dynamic` type, which allows users to perform dynamic operations directly and without resorting to reflection. The compiler produces the appropriate dynamic call site infrastructure as shown in the previous example automatically. Thus, APIs implemented in foreign languages can be seamlessly integrated using standard syntax.

In summary, the DLR's late-binding mechanism is similar to that of the JVM, but operates on a higher level. In addition, the DLR provides a MOP enabling interoperability and improves code generation by providing a code generation facility based on expression trees. However, because IL is lazily generated, this has an impact on performance.

10.2 Parrot

The Parrot Virtual Machine is designed to execute dynamic programming languages and provide interoperability between them. It implements most features of these languages directly in the VM, such as closures, coroutines, multiple return values and continuations, eliminating the unnecessary reimplementation in each language.

It supports primitive types and an aggregate data type called polymorphic container (PMC), which may be built in structures (e.g., objects, exceptions, coroutine, continuation, hash-table, etc.) or may be language-specific, dynamically loaded at runtime. As such it provides a kind of MOP, e.g., allowing the implementation of a custom type system. For instance, one such user-defined object system is P6object, which is class-based, supports multiple inheritance and uses the C3 class linearization algorithm. Both PMCs and the object PMC may have methods associated to it. Invocation of these methods performs dynamic multiple dispatch. The dispatch is implemented using virtual method tables. The virtual machine also features a multisub PMC, which is similar to a multi-method and performs multiple dispatch. Its built-in dispatch algorithm is calculating specificity based on the Manhattan distance, but is exchangeable.

Unlike the instruction sets of virtual machines such as the JVM and CLR which are stack-based, the Parrot Intermediate Language (PIL) and VM are register-based. The instruction set is very extensive and the VM allows dynamically loading additional opcodes (hence called “dynops”) as needed. Opcodes are also polymorphic and multiple dispatch is performed statically at compile-time of the opcode library. Therefore, the Parrot VM can be extended independently with new instructions and new meta types. For example, instructions for performing complex custom dispatch can be added on a VM level and can be used by different client languages. However, due to it’s highly extensible and featureful nature, the implementation of the VM is suffering from performance problems.

Chapter II

Conclusion

This thesis investigated how advanced variants of dynamic method dispatch can be efficiently implemented on the JVM. It presented the design and implementation of efficient dynamic multiple dispatch and dynamic predicate dispatch, based on the latest features added to the JVM. An evaluation showed that the result is promising, demonstrating the feasibility and practicality of efficiently implementing advanced dynamic dispatch variants even for dynamic languages.

The solution is implemented in the high-level language Java, and suitable for the integration into other JVM-based languages. It does not require any modifications to the JVM specification or implementations. Both the dynamic multiple and predicate dispatch implementation support dispatch on arbitrary arity and support variable argument lists. Method implementations of multi-methods are able to invoke the next most specific method implementation. Both multi-methods and predicate methods are declarative and expressive solutions to the expression problem, requiring no boilerplate code. Furthermore, the solution is suitable for languages with an object system that supports multiple inheritance. Both method dispatch variants support dynamism, i.e., changes to the sets of methods and classes are considered on subsequent dispatch operations.

The performance of multi-methods is not as fast as existing static variants, but faster than implementations of multiple dispatch in other dynamic languages, especially for polymorphic call sites. The implementation of predicate dispatch demonstrated that, even though performance is considerably slower than that of static implementations, it is practical to adopt predicate dispatch also for dynamic languages.

For the purpose of this thesis the dynamic language Lila was developed. It demonstrates the use of the new features added to the JVM with Java 7, namely late-binding through the `invokedynamic` instruction, and support for function pointers in the form of method handles. Both were crucial for implementation of the dispatching solutions. Their introduction has significantly improved the support for features of dynamic languages. The overhead of method invocation using `invokedynamic` is negligible and a linked call site performs comparably to those of existing method invocation instructions.

However, there are still some shortcomings. The use of combined method handles is not yet as fast as generating bytecode with the same behavior. Method handles are opaque and cannot be updated after creation. This is especially inconvenient in the case of guard method handles. The late-binding instruction is also just a basic building block, which improves performance and flexibility of language runtimes, but not the interoperability between languages. An important goal should be the removal of the permanent generation of the JVM, which will make dynamic code generation more memory efficient.

11.1 Future Work

There are several areas where the current implementation of dynamic multiple dispatch and predicate dispatch can be extended and improved. Future work will include investigating if compiling the nested polymorphic inline caches of multi-method call sites to bytecode improves performance. Both multiple and predicate dispatch can be improved by using static and run-time information to optimize the call sites. The dispatch tree algorithm of the predicate dispatch implementation already constructs very efficient dispatch trees, but is also able to use compile-time and run-time information to produce even more specialized and optimized dispatch trees. The addition of efficient profiling nodes to the dispatch trees should lead to highly specialized call site specific dispatch trees and needs further investigation. Adopting partial dispatch for the multiple and predicate dispatch implementations will likely improve their performance.

Predicate methods could also be extended to support binding, and better implication rules could be added to the implementation to make it more powerful. The multiple dispatch implementation also has no support for an "inapplicable" method yet, but it can be easily added.

Even though Lila was developed for demonstration purposes and the design focused on simplicity, the implementation could be optimized in several areas to improve its performance, and it could be developed into a full-featured, general-purpose language with little effort.

Future work also includes integrating the developed solution into existing languages. For example, the replacement of Groovy's multiple dispatch implementation is a potential candidate. Furthermore, it also makes adopting the JVM as a target platform more desirable. Similar to ABCL, an implementation of Dylan on the JVM has potential. The thesis demonstrated how the implementation of a major feature performs more efficiently on a virtual machine than an exisiting implementation generating native code.

Bibliography

[1] Eric Allen et al. “Modular multiple dispatch with multiple inheritance”. In: *Proceedings of the 2007 ACM symposium on Applied computing*. SAC ’07. Seoul, Korea: ACM, 2007, pp. 1117–1121. ISBN: 1-59593-480-4.

[2] Bowen Alpern et al. “Efficient Dispatch of Java Interface Methods”. In: *Proceedings of the 9th International Conference on High-Performance Computing and Networking*. HPCN Europe 2001. London, UK, UK: Springer-Verlag, 2001, pp. 621–628. ISBN: 3-540-42293-5.

[3] Bowen Alpern et al. “Efficient Implementation of Java Interfaces: Invokeinterface Considered Harmless”. In: *In Proc. 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. 2001, pp. 108–124.

[4] Eric Amiel, Olivier Gruber, and Eric Simon. “Optimizing multi-method dispatch using compressed dispatch tables”. In: *Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*. OOPSLA ’94. Portland, Oregon, United States: ACM, 1994, pp. 244–258. ISBN: 0-89791-688-3.

[5] Jonathan Bachrach. *Partial Dispatch: Optimizing Dynamically–Dispatched Multimethod Calls with Compile–Time Types and Runtime Feedback*. 1999.

[6] Kim Barrett et al. “A monotonic superclass linearization for Dylan”. In: *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’96. San Jose, California, United States: ACM, 1996, pp. 69–82. ISBN: 0-89791-788-X.

[7] Gerald Baumgartner, Martin Jansche, and Konstantin Läufer. *Half & Half: Multiple Dispatch and Retroactive Abstraction for Java*. Tech. rep. 2002.

[8] Daniel G. Bobrow et al. “Common Lisp Object System specification”. In: *SIGPLAN Not.* 23.SI (Sept. 1988), pp. 1–142. ISSN: 0362-1340.

[9] Luca Cardelli and Peter Wegner. “On understanding types, data abstraction, and polymorphism”. In: *ACM Comput. Surv.* 17.4 (Dec. 1985), pp. 471–523. ISSN: 0360-0300.

[10] Craig Chambers. “Object-Oriented Multi-Methods in Cecil”. In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP ’92. London, UK, UK: Springer-Verlag, 1992, pp. 33–56. ISBN: 3-540-55668-0.

[11] Craig Chambers and Weimin Chen. “Efficient multiple and predicated dispatching”. In: *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’99. Denver, Colorado, United States: ACM, 1999, pp. 238–255. ISBN: 1-58113-238-7.

[12] Weimin Chen and Karl Aberer. *Efficient Multiple Dispatching Using Nested Transition-Arrays*. Tech. rep. 2001.

[13] Weimin Chen and Volker Turau. “Multiple-Dispatching Based on Automata”. In: (1995).

[14] Curtis Clifton et al. “MultiJava: Design rationale, compiler implementation, and applications”. In: *ACM Trans. Program. Lang. Syst.* 28.3 (May 2006), pp. 517–575. ISSN: 0164-0925.

[15] Curtis Clifton et al. “MultiJava: modular open classes and symmetric multiple dispatch for Java”. In: *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’00. Minneapolis, Minnesota, United States: ACM, 2000, pp. 130–145. ISBN: 1-58113-200-X.

[16] Pascal Costanza et al. “Filtered dispatch”. In: *Proceedings of the 2008 symposium on Dynamic languages*. DLS ’08. Paphos, Cyprus: ACM, 2008, 4:1–4:10. ISBN: 978-1-60558-270-2.

[17] L. Peter Deutsch and Allan M. Schiffman. “Efficient implementation of the smalltalk-80 system”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL ’84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302. ISBN: 0-89791-125-3.

[18] Howard Dierking. *Bjarne Stroustrup on the Evolution of Languages*. Apr. 2008. URL: `http://msdn.microsoft.com/en-us/magazine/cc500572.aspx`.

[19] Karel Driesen. “Selector table indexing & sparse arrays”. In: *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*. OOPSLA ’93. Washington, D.C., USA: ACM, 1993, pp. 259–270. ISBN: 0-89791-587-9.

[20] Karel Driesen and Urs Hölzle. “Minimizing row displacement dispatch tables”. In: *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*. OOPSLA ’95. Austin, Texas, USA: ACM, 1995, pp. 141–155. ISBN: 0-89791-703-0.

[21] Karel Driesen, Urs Hölzle, and Jan Vitek. “Message Dispatch on Pipelined Processors”. In: *Proceedings of the 9th European Conference on Object-Oriented Programming*. ECOOP ’95. London, UK, UK: Springer-Verlag, 1995, pp. 253–282. ISBN: 3-540-60160-0.

[22] Roland Ducournau. “Coloring, a versatile technique for implementing object-oriented languages”. In: *Softw. Pract. Exper.* 41.6 (May 2011), pp. 627–659. ISSN: 0038-0644.

[23] Éric Dujardin. *Efficient Dispatch of Multimethods in Constant Time Using Dispatch Trees*. Tech. rep. INRIA, 1996.

[24] Eric Dujardin, Eric Amiel, and Eric Simon. “Fast algorithms for compressed multimethod dispatch table generation”. In: *ACM Trans. Program. Lang. Syst.* 20.1 (Sept. 1996), pp. 116–165. ISSN: 0164-0925.

[25] Christopher Dutchyn and Paul Lu. “Multi-dispatch in the java virtual machine: Design and implementation”. In: *In Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS-01*. 2001, pp. 77–92.

[26] Margaret A. Ellis and Bjarne Stroustrup. *The annotated C++ reference manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1990. ISBN: 0-201-51459-1.

[27] Michael Ernst, Craig Kaplan, and Craig Chambers. “Predicate Dispatching: A Unified Theory of Dispatch”. In: Springer-Verlag, 1998, pp. 186–211.

[28] Michael Ernst, Craig Kaplan, and Craig Chambers. “Predicate Dispatching: A Unified Theory of Dispatch”. In: *Proceedings of the 12th European Conference on Object-Oriented Programming*. ECCOP ’98. London, UK, UK: Springer-Verlag, 1998, pp. 186–211. ISBN: 3-540-64737-6.

[29] Paolo Ferragina, S. Muthukrishnan, and Mark de Berg. “Multi-method dispatching: a geometric approach with applications to string matching problems”. In: *Proceedings of the thirty-first annual ACM symposium on Theory of computing*. STOC ’99. Atlanta, Georgia, United States: ACM, 1999, pp. 483–491. ISBN: 1-58113-067-8.

[30] Rémi Forax. *droid292*. URL: `http://code.google.com/p/droid292/`.

[31] Rémi Forax. *JSR 292 Goodness: named parameters*. Jan. 2011. URL: `http://weblogs.java.net/blog/forax/archive/2011/01/21/jsr-292-goodness-named-parameters`.

[32] Remi Forax, Etienne Duris, and Gilles Roussel. “A Reflective Implementation of Java Multi-Methods”. In: *IEEE Trans. Softw. Eng.* 30.12 (Dec. 2004), pp. 1055–1071. ISSN: 0098-5589.

[33] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[34] Joseph (Yossi) Gil and Yoav Zibin. “Efficient dynamic dispatching with type slicing”. In: *ACM Trans. Program. Lang. Syst.* 30.1 (Nov. 2007). ISSN: 0164-0925.

[35] Google. *Design Elements - Chrome V8*. URL: `http://developers.google.com/v8/design`.

[36] Friedrich Gräter, Sebastian Götz, and Julian Stecklina. “Predicate-C: an efficient and generic runtime system for predicate dispatch”. In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS ’11. Lancaster, United Kingdom: ACM, 2011, 2:1–2:8. ISBN: 978-1-4503-0894-6.

[37] Wade Holst and Duane Szafron. “Incremental Table-Based Method Dispatch for Reflective Object-Oriented Languages”. In: *TOOLS (23)*. 1997, pp. 63–.

[38] Wade Holst et al. *Multi-Method Dispatch Using Single-Receiver Projections*. Tech. rep. University of Alberta, Canada, 1998.

[39] Urs Hölzle, Craig Chambers, and David Ungar. “Optimizing Dynamically-Typed Object-Oriented Languages with Polymorphic Inline Caches”. In: *Proceedings ECOOP ’91, LNCS 512*. Springer-Verlag, 1991, pp. 21–38.

[40] Urs Hölzle and David Ungar. “Optimizing dynamically-dispatched calls with run-time type feedback”. In: *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. PLDI ’94. Orlando, Florida, USA: ACM, 1994, pp. 326–336. ISBN: 0-89791-662-X.

[41] Daniel H. H. Ingalls. “A simple technique for handling multiple polymorphism”. In: *Conference proceedings on Object-oriented programming systems, languages and applications*. OOPLSA ’86. Portland, Oregon, USA: ACM, 1986, pp. 347–349. ISBN: 0-89791-204-7.

[42] INSA-Lyon. URL: http://golo-lang.org/.

[43] Chanwit Kaewkasi. "Towards performance measurements for the Java Virtual Machine's invokedynamic". In: *Virtual Machines and Intermediate Languages*. VMIL '10. Reno, Nevada: ACM, 2010, 3:1–3:6. ISBN: 978-1-4503-0545-7.

[44] Eric Kidd. *Efficient Compression of Generic Function Dispatch Tables*. Tech. rep. Hanover, NH, USA, 2001.

[45] Tim Lindholm et al. *The Java Virtual Machine Specification: Java SE 7 Edition*. Java Series. Prentice Hall, July 2012. ISBN: 9780133260441.

[46] Jon Masamitsu. *JEP 122: Remove the Permanent Generation*. Dec. 2012. URL: http://openjdk.java.net/jeps/122.

[47] Todd Millstein. "Practical predicate dispatch". In: *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA '04. Vancouver, BC, Canada: ACM, 2004, pp. 345–364. ISBN: 1-58113-831-8.

[48] Todd Millstein et al. "Expressive and modular predicate dispatch for Java". In: *ACM Trans. Program. Lang. Syst.* 31.2 (Feb. 2009), 7:1–7:54. ISSN: 0164-0925.

[49] Radu Muschevici et al. "Multiple dispatch in practice". In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 563–582. ISBN: 978-1-60558-215-3.

[50] Peter Norvig. *A Retrospective on Paradigms of AI Programming*. Oct. 1997. URL: http://norvig.com/Lisp-retro.html.

[51] Tamiya Onodera and Hiroaki Nakamura. "Optimizing Smalltalk by selector code indexing can be practical". In: *ECOOP'97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Vol. 1241. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1997, pp. 302–323. ISBN: 978-3-540-63089-0.

[52] Oracle. *MethodHandle (Java Platform SE 7)*. 2013. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandle.html.

[53] Oracle. *MethodHandles.Lookup (Java Platform SE 7)*. 2013. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/MethodHandles.Lookup.html.

[54] Oracle. *Package java.lang.invoke (Java Platform SE 7)*. 2013. URL: http://docs.oracle.com/javase/7/docs/api/java/lang/invoke/package-summary.html.

[55] Candy Pang et al. "Multi-Method Dispatch Using Multiple Row Displacement". In: *LNCS*. 1999, pp. 304–328.

[56] Peter Pirkelbauer, Yuriy Solodkyy, and Bjarne Stroustrup. "Design and evaluation of C++ open multi-methods". In: *Sci. Comput. Program.* 75.7 (July 2010), pp. 638–667. ISSN: 0167-6423.

[57] 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*. GPCE '07. Salzburg, Austria: ACM, 2007, pp. 123–134. ISBN: 978-1-59593-855-8.

[58] J. R. Quinlan. “Induction of Decision Trees”. In: *Mach. Learn* (Mar. 1986), pp. 81–106. ISSN: 0885-6125.

[59] John Rose. *Anatomy of a Call Site*. Oct. 2007. URL: `http://blogs.oracle.com/jrose/entry/anatomy_of_a_call_site`.

[60] John Rose. *Method handles and invokedynamic*. Apr. 2013. URL: `http://wikis.oracle.com/display/HotSpotInternals/Method+handles+and+invokedynamic`.

[61] John Rose. *Symbolic freedom in the VM*. Jan. 2008. URL: `http://blogs.oracle.com/jrose/entry/symbolic_freedom_in_the_vm`.

[62] John Rose. *Tail calls in the VM*. July 2007. URL: `http://blogs.oracle.com/jrose/entry/tail_calls_in_the_vm`.

[63] John R. Rose. “Bytecodes meet combinators: invokedynamic on the JVM”. In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*. VMIL ’09. Orlando, Florida: ACM, 2009, 2:1–2:11. ISBN: 978-1-60558-874-2.

[64] Andrew Shalit, David Moon, and Orca Starbuck. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison Wesley Publishing Company, Sept. 11, 1996. ISBN: 0201442116.

[65] Guy Steele. *Fortress Wrapping Up*. July 2012. URL: `https://blogs.oracle.com/projectfortress/entry/fortress_wrapping_up`.

[66] Sun Microsystems Inc. *JSR 292: Supporting Dynamically Typed Languages on the Java Platform*. Aug. 2008.

[67] Attila Szegedi. *Dynalink*. URL: `http://github.com/szegedi/dynalink`.

[68] Jan Vitek and R. Nigel Horspool. “Compact Dispatch Tables for Dynamically Typed Object Oriented Languages”. In: *In Gyimothy, T., Ed., Compiler Construction. 6th International Conference, CC’96. Proceedings Proceedings of CC: International Conference on Compiler Construction, (Linkoping*. Springer-Verlag, 1996, pp. 309–325.

[69] Matthias Zenger and Martin Odersky. “Independently extensible solutions to the expression problem”. In: *In Proc. FOOL 12*. 2005.

[70] Yoav Zibin and Joseph Yossi Gil. “Fast algorithm for creating space efficient dispatching tables with application to multi-dispatching”. In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. OOPSLA ’02. Seattle, Washington, USA: ACM, 2002, pp. 142–160. ISBN: 1-58113-471-1.

[71] Yoav Zibin and Joseph (Yossi) Gil. “Incremental algorithms for dispatching in dynamically typed languages”. In: *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’03. New Orleans, Louisiana, USA: ACM, 2003, pp. 126–138. ISBN: 1-58113-628-5.

Appendices

Appendix A

Multiple dispatch experiments

```
defclass <surface> (<object>);
defclass <display> (<surface>);
defclass <etch-a-sketch> (<surface>);

defclass <shape> (<object>);
defclass <polygon> (<shape>);
defclass <rectangle> (<polygon>);

defmm draw (display :: <display>, shape :: <shape>) {
  true // RENDER a SHAPE
};

defmm draw (display :: <display>, rectangle :: <rectangle>) {
  true // RENDER a RECT
};

defmm draw (display :: <display>, polygon :: <polygon>) {
  true // RENDER a POLY
};

defmm draw (etch-a-sketch :: <etch-a-sketch>, shape :: <shape>) {
  true // SKETCH a SHAPE
};

defmm draw (etch-a-sketch :: <etch-a-sketch>, rectangle :: <rectangle>) {
  true // SKETCH a RECT
};

defmm draw (etch-a-sketch :: <etch-a-sketch>, polygon :: <polygon>) {
  true // SKETCH a POLY
};

let objects1 = make-array(make(<etch-a-sketch>),
                          make(<display>)),
    objects2 = make-array(make(<rectangle>),
                          make(<polygon>)),
    object1 = nth(0, objects1),
    object2 = nth(0, objects2)
{
  benchmark(fn () {
    dotimes (10000000) {
      random-element(objects1);
      random-element(objects2);
    }
  }, 5);
```

```
  benchmark(fn () {
    dotimes (10000000) {
       draw(random-element(objects1),
            random-element(objects2));
    }
  }, 10);
  benchmark(fn () {
    dotimes (10000000) {
       draw(object1, object2);
    }
  }, 10);
};
```

Listing A.1: Lila

```
(defclass surface () ())

(defclass display (surface) ())

(defclass etch-a-sketch (surface) ())

(defclass shape () ())

(defclass polygon (shape) ())

(defclass rectangle (polygon) ())

(defmethod draw ((display display) (shape shape))
  t #| "RENDER a SHAPE" |#)

(defmethod draw ((display display) (rectangle rectangle))
  t #| "RENDER a RECT" |#)

(defmethod draw ((display display) (polygon polygon))
  t #| "RENDER a POLY" |#)

(defmethod draw ((etch-a-sketch etch-a-sketch) (shape shape))
  t #| "SKETCH a SHAPE" |#)

(defmethod draw ((etch-a-sketch etch-a-sketch) (rectangle rectangle))
  t #| "SKETCH a RECT" |#)

(defmethod draw ((etch-a-sketch etch-a-sketch) (polygon polygon))
  t #| "SKETCH a POLY" |#)

(let* ((objs1 (list (make-instance 'etch-a-sketch)
                    (make-instance 'display)))
       (objs2 (list (make-instance 'rectangle)
                    (make-instance 'polygon)))
       (obj1 (car objs1))
       (obj2 (car objs2)))

  (dotimes (j 5)
    (time (dotimes (i 10000000)
            (nth (random 2) objs1)
            (nth (random 2) objs2))))
```

```
42
43  (dotimes (j 10)
44    (time (dotimes (i 10000000)
45            (draw (nth (random 2) objs1)
46                  (nth (random 2) objs2)))))
47
48  (dotimes (j 10)
49    (time (dotimes (i 10000000)
50            (draw obj1 obj2)))))
```

Listing A.2: Common Lisp

```
1  module: test
2
3  define class <surface> (<object>) end;
4
5  define class <display> (<surface>) end;
6
7  define class <etch-a-sketch> (<surface>) end;
8
9  define class <shape> (<object>) end;
10
11 define class <polygon> (<shape>) end;
12
13 define class <rectangle> (<polygon>) end;
14
15 define method draw (display :: <display>, shape :: <shape>)
16   #t; // "RENDER a SHAPE"
17 end;
18
19 define method draw (display :: <display>, rectangle :: <rectangle>)
20   #t; // "RENDER a RECT"
21 end;
22
23 define method draw (display :: <display>, polygon :: <polygon>)
24   #t; // "RENDER a POLY"
25 end;
26
27 define method draw (etch-a-sketch :: <etch-a-sketch>, shape :: <shape>)
28   #t; // "SKETCH a SHAPE"
29 end;
30
31 define method draw (etch-a-sketch :: <etch-a-sketch>, rectangle :: <rectangle>)
32   #t; // "SKETCH a RECT"
33 end;
34
35 define method draw (etch-a-sketch :: <etch-a-sketch>, polygon :: <polygon>)
36   #t; // "SKETCH a POLY"
37 end;
38
39 define function benchmark (fn, n)
40   for (i from 0 below n)
41     let (elapsed-seconds, elapsed-microseconds)
42       = timing ()
43           fn()
44         end timing;
```

```
      format-out("%d.%s sec\n",
                 elapsed-seconds,
                 integer-to-string(elapsed-microseconds, size: 6));
  end for;
end;

let objs1 = list(make(<etch-a-sketch>),
                 make(<display>));
let objs2 = list(make(<rectangle>),
                 make(<polygon>));

benchmark(method ()
  for (i from 0 below 10000000)
    objs1[random(2)];
    objs2[random(2)];
  end for;
end, 5);

benchmark(method ()
  for (i from 0 below 10000000)
    draw(objs1[random(2)],
         objs2[random(2)]);
  end for;
end, 10);

let obj1 = objs1[0];
let obj2 = objs2[0];
benchmark(method ()
  for (i from 0 below 10000000)
    draw(obj1, obj2);
  end for;
end, 10);
```

Listing A.3: Dylan

```
class Shape {}

class Polygon extends Shape {}

class Rectangle extends Polygon {}

class Surface {}

class Display extends Surface {}

class EtchASketch extends Surface {}

public class Test {
    static void draw(Display display, Shape shape) {
        true // "RENDER a SHAPE"
    }

    static void draw(Display display, Polygon polygon) {
        true // "RENDER a POLY"
    }

```

```
22      static void draw(Display display, Rectangle rectangle) {
23          true // "RENDER a RECT"
24      }
25
26      static void draw(EtchASketch e, Shape shape) {
27          true // "SKETCH a SHAPE"
28      }
29
30      static void draw(EtchASketch e, Polygon polygon) {
31          true // "SKETCH a POLY"
32      }
33
34      static void draw(EtchASketch e, Rectangle rectangle) {
35          true // "SKETCH a RECT"
36      }
37  }
38
39  def Random random = new Random()
40
41  def benchmark(fn, n) {
42      for (int i = 0; i < n; i++) {
43          long startTime = System.nanoTime()
44          fn()
45          long stopTime = System.nanoTime()
46          long runTime = stopTime - startTime
47          System.out.println(runTime / 1000000)
48      }
49  }
50
51  Surface display = new Display()
52  Shape rectangle = new Rectangle()
53  Surface etchASketch = new EtchASketch()
54  Shape polygon = new Polygon()
55
56  def objs1 = [ display, etchASketch ]
57  def objs2 = [ rectangle, polygon ]
58
59  benchmark({
60      for (int i = 0; i < 10000000; i++) {
61          Object obj1 = objs1[random.nextInt(2)]
62          Object obj2 = objs2[random.nextInt(2)]
63      }
64  }, 5)
65
66  benchmark({
67      for (int i = 0; i < 10000000; i++) {
68          Object obj1 = objs1[random.nextInt(2)]
69          Object obj2 = objs2[random.nextInt(2)]
70          Test.draw(obj1, obj2)
71      }
72  }, 10)
73
74  Object obj1 = objs1[0]
75  Object obj2 = objs2[0]
76  benchmark({
77      for (int i = 0; i < 10000000; i++)
78          Test.draw(obj1, obj2)
```

```
}, 10)
```

Listing A.4: Groovy

```
package common;

import java.util.Random;

public class Utilities {

        public static void benchmark(Code code, int count) {
                for (int j = 0; j < count; j++) {
                        long startTime = System.nanoTime();
                        code.execute();
                        long stopTime = System.nanoTime();
                        long runTime = stopTime - startTime;
                        System.out.println(runTime / 1000000);
                }
        }

        public static void work(String ignored) {
                new Boolean(true);
        }

        static Random random = new Random();

        public static <T> T randomElement(T[] elements) {
                return elements[random.nextInt(elements.length)];
        }

}
```

Listing A.5: Common utility code of the Java experiments

```
package test2;

import common.Code;
import common.Utilities;

class Shape {}

class Polygon extends Shape {}

class Rectangle extends Polygon {}

abstract class Surface {
        abstract void _draw(Shape shape);
        abstract void _draw(Polygon polygon);
        abstract void _draw(Rectangle rectangle);

        void draw(Shape shape) {
                if (shape instanceof Rectangle) {
                        Rectangle rectangle = (Rectangle)shape;
                        _draw(rectangle);
```

```
                } else if (shape instanceof Polygon) {
                        Polygon polygon = (Polygon)shape;
                        _draw(polygon);
                } else
                        _draw(shape);
        }
}

class Display extends Surface {

        public void _draw(Shape shape) {
                Utilities.work("RENDER a SHAPE");
        }

        public void _draw(Polygon polygon) {
                Utilities.work("RENDER a POLY");
        }

        public void _draw(Rectangle rectangle) {
                Utilities.work("RENDER a RECT");
        }
}

class EtchASketch extends Surface {
        public void _draw(Shape shape) {
                Utilities.work("SKETCH a SHAPE");
        }

        public void _draw(Polygon polygon) {
                Utilities.work("SKETCH a POLY");
        }

        public void _draw(Rectangle rectangle) {
                Utilities.work("SKETCH a RECT");
        }
}

public class InstanceofExample {
        public static void main(String[] args) {
                final Surface display = new Display();
                final Shape rectangle = new Rectangle();
                final Surface etchASketch = new EtchASketch();
                final Shape polygon = new Polygon();

                final Surface[] objs1 = new Surface[] {
                        (Surface) display, (Surface) etchASketch };
                final Shape[] objs2 = new Shape[] {
                        (Shape) rectangle, (Shape) polygon };

                Utilities.benchmark(new Code() {
                        public void execute() {
                                for (int i = 0; i < 10000000; i++) {
                                        Surface obj1 = Utilities.randomElement(objs1);
                                        Shape obj2 = Utilities.randomElement(objs2);
                                }
                        }
                }, 5);
```

```
78
79              Utilities.benchmark(new Code() {
80                      public void execute() {
81                              for (int i = 0; i < 10000000; i++) {
82                                      Surface obj1 = Utilities.randomElement(objs1);
83                                      Shape obj2 = Utilities.randomElement(objs2);
84                                      obj1.draw(obj2);
85                              }
86                      }
87              }, 10);
88
89              final Surface obj1 = objs1[0];
90              final Shape obj2 = objs2[0];
91              Utilities.benchmark(new Code() {
92                      public void execute() {
93                              for (int i = 0; i < 10000000; i++) {
94                                      obj1.draw(obj2);
95                              }
96                      }
97              }, 10);
98      }
99  }
```

Listing A.6: Java (instanceof cascade)

```
1   package test1;
2
3   import common.Code;
4   import common.Utilities;
5
6   abstract class Shape {
7           abstract void drawOn(Drawable drawable);
8   }
9
10  class Polygon extends Shape {
11          void drawOn(Drawable drawable) {
12                  drawable.draw(this);
13          }
14  }
15
16  class Rectangle extends Polygon {
17          void drawOn(Drawable drawable) {
18                  drawable.draw(this);
19          }
20  }
21
22  interface Drawable {
23          void draw(Shape shape);
24          void draw(Polygon polygon);
25          void draw(Rectangle rectangle);
26  }
27
28  abstract class Surface implements Drawable {}
29
30  class Display extends Surface {
31
```

```
32          public void draw(Shape shape) {
33                  Utilities.work("RENDER a SHAPE");
34          }
35
36          public void draw(Polygon polygon) {
37                  Utilities.work("RENDER a POLY");
38          }
39
40          public void draw(Rectangle rectangle) {
41                  Utilities.work("RENDER a RECT");
42
43          }
44  }
45
46  class EtchASketch extends Surface {
47          public void draw(Shape shape) {
48                  Utilities.work("SKETCH a SHAPE");
49          }
50
51          public void draw(Polygon polygon) {
52                  Utilities.work("SKETCH a POLY");
53          }
54
55          public void draw(Rectangle rectangle) {
56                  Utilities.work("SKETCH a RECT");
57          }
58  }
59
60  public class VisitorExample {
61
62          public static void main(String[] args) {
63                  final Surface display = new Display();
64                  final Shape rectangle = new Rectangle();
65                  final Surface etchASketch = new EtchASketch();
66                  final Shape polygon = new Polygon();
67
68                  final Surface[] objs1 = new Surface[] {
69                          (Surface) display, (Surface) etchASketch };
70                  final Shape[] objs2 = new Shape[] {
71                          (Shape) rectangle, (Shape) polygon };
72
73                  Utilities.benchmark(new Code() {
74                          public void execute() {
75                                  for (int i = 0; i < 10000000; i++) {
76                                          Surface obj1 = Utilities.randomElement(objs1);
77                                          Shape obj2 = Utilities.randomElement(objs2);
78                                  }
79                          }
80                  }, 5);
81
82                  Utilities.benchmark(new Code() {
83                          public void execute() {
84                                  for (int i = 0; i < 10000000; i++) {
85                                          Surface obj1 = Utilities.randomElement(objs1);
86                                          Shape obj2 = Utilities.randomElement(objs2);
87                                          obj2.drawOn(obj1);
88                                  }
```

```
89                      }
90              }, 10);
91
92              final Surface obj1 = objs1[0];
93              final Shape obj2 = objs2[0];
94              Utilities.benchmark(new Code() {
95                      public void execute() {
96                              for (int i = 0; i < 10000000; i++) {
97                                      obj2.drawOn(obj1);
98                              }
99                      }
100             }, 10);
101     }
102 }
```

Listing A.7: Java (visitor pattern)

Appendix B

Predicate dispatch experiments

```
defclass <tree-node> (<object>);
defclass <data-node> (<tree-node>) { left; right };
defclass <empty-node> (<tree-node>);

defpm isomorphic? (t1, t2)
  when (t1 :: <empty-node>)
    && (t2 :: <empty-node>)
{ true };

defpm isomorphic? (t1, t2)
  when (t1 :: <empty-node>)
    || (t2 :: <empty-node>)
{ false };

defpm isomorphic? (t1, t2)
  when (t1 :: <data-node>)
    && (t2 :: <data-node>)
{
  isomorphic?(get(t1, "left"), get(t2, "left"))
    && isomorphic?(get(t1, "right"), get(t2, "right"));
};

def empty = make(<empty-node>);
def empty-tree = make(<data-node>, empty, empty);
def tree1 = make(<data-node>, empty, empty-tree);
def tree2 = make(<data-node>, empty-tree, empty);

let objs = make-array(empty, tree1, tree2) {
  benchmark(fn () {
    dotimes (10000000) {
       random-element(objs);
       random-element(objs);
    }
  }, 5);

  benchmark(fn () {
    dotimes (10000000) {
       isomorphic?(random-element(objs),
                   random-element(objs));
    }
  }, 10);

  benchmark(fn () {
    dotimes (10000000) {
       isomorphic?(tree1, tree2);
```

```
    }
  }, 10);
};
```

Listing B.1: Lila

```
abstract class TreeNode {}

class DataNode extends TreeNode {
    public TreeNode left;
    public TreeNode right;
}

class EmptyNode extends TreeNode {}

public class TreeIsomorphism {

    public boolean isomorphic(TreeNode t1, TreeNode t2)
        when t1@EmptyNode && t2@EmptyNode
    { return true; }

    public boolean isomorphic(TreeNode t1, TreeNode t2)
        when t1@EmptyNode || t2@EmptyNode
    {return false; }

    public boolean isomorphic(TreeNode t1, TreeNode t2)
        when t1@DataNode && t2@DataNode
    {
      return isomorphic(t1.left, t2.left)
        && isomorphic(t1.right, t2.right);
    }

    public boolean isomorphic(TreeNode t1, TreeNode t2) {
        return false;
    }

    public static void main(String[] args) {
        TreeNode empty = new EmptyNode();
        DataNode emptyTree = new DataNode();
        emptyTree.left = empty;
        emptyTree.right = empty;
        DataNode tree1 = new DataNode();
        tree1.left = empty;
        tree1.right = emptyTree;
        DataNode tree2 = new DataNode();
        tree2.left = emptyTree;
        tree2.right = empty;
        TreeNode[] objs = new TreeNode[] { empty, tree1, tree2 };
        TreeIsomorphism iso = new TreeIsomorphism();

        Utilities.benchmark(new Code() {
            public void execute() {
                for (int i = 0; i < 10000000; i++) {
                    Utilities.randomElement(objs);
                    Utilities.randomElement(objs);
                }
```

```
51              }
52          }, 5);
53
54          Utilities.benchmark(new Code() {
55              public void execute() {
56                  for (int i = 0; i < 10000000; i++) {
57                      iso.isomorphic(Utilities.randomElement(objs),
58                                     Utilities.randomElement(objs));
59                  }
60              }
61          }, 10);
62
63          Utilities.benchmark(new Code() {
64              public void execute() {
65                  for (int i = 0; i < 10000000; i++) {
66                      iso.isomorphic(tree1, tree2);
67                  }
68              }
69          }, 10);
70      }
71  }
```

Listing B.2: JPred