Optimizing Jython using `invokedynamic` and Gradual Typing

by

Shashank Bharadwaj

B.E., Visvesvaraya Technological University, 2008

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical, Energy and Computer Engineering

2012

This thesis entitled:
Optimizing Jython using `invokedynamic` and Gradual Typing
written by Shashank Bharadwaj
has been approved for the Department of Electrical, Energy and Computer Engineering

Jeremy Siek

Bor-Yuh Evan Chang

Sriram Sankaranarayanan

Date

Bharadwaj, Shashank (M.S.)

Optimizing Jython using `invokedynamic` and Gradual Typing

Thesis directed by Assistant Professor Jeremy Siek

Compilers for statically typed languages such as C/C++ and Java use the types from the program to generate high performance code. Although the runtimes of dynamically typed languages such as Python, Ruby and Javascript have evolved significantly over the past few years; they have not yet reached the performance of their statically typed counterparts. A new type system called *gradual typing* has been proposed in which the program can optionally be annotated with type information. In this thesis, I implement gradual typing in Jython (an implementation of Python language for the Java virtual machine) and generate type-specialized code. Also, I use `invokedynamic`, a new byte code introduced in Java 7, to implement optimizations in Jython; both in an effort to close the gap between Java and Jython in particular and statically type and dynamically typed language in general.

Acknowledgements

I thank my advisor, Jeremy Siek for his guidance, constant encouragement and support. Under his guidance I have learnt everything I know about compilers and programming languages; I could not have completed this thesis without him. I am grateful to my mentor Jim Baker, who was always there to catch glitches in my thinking and who has always been a constant source of encouragement. I thank my committee: Bor-Yuh Evan Chang and Sriram Sankaranarayanan for their for their helpful advice and encouragement. I thank Michael Vituosek, my fellow graduate student, for interesting discussions around the design choices and begin a superb work partner. I thank Christopher Poulton for helping with the implementation. I thank Carl Friedrich Bolz for reviewing and helping me improve this thesis. A big thank you to Kaushik, Karthik, Srinivas, Deepak and especially Arul and Bharath for all their support and motivation. Last, but by no means the least, I thank my parents and my sister for being a constant source of support, inspiration and motivation.

Contents

Figures

Chapter 1

Introduction

Based on the type system they use, modern programming languages can broadly be classified as either being dynamically-typed or statically-typed. Both systems have well known advantages and disadvantages. Statically-typed languages (such as C, C++, or Java) perform type-checking at compile-time and require the program to have type information on all the variables. This not only ensures that the compiler generates meaningful type errors, but also facilitates high performance code generation, because the compiler has to generate code for only one type. This approach makes a program rigid, making it harder handle situations where the type of a value depends on runtime information and also to adapt to changing requirements.

Dynamically-typed languages (such as Perl, Python, or Javascript) are popular for their ease of use and rapid development cycles [13]. These languages do not perform type checking during compilation so that the programmer does not have to provide any type information on the program. Although this feature can be helpful when developing a prototype, in this type system the compiler generates inefficient code since the compiler must assume the most generic case for all the expressions.

There has been a wealth of research into improving performance of dynamic languages. Chambers and Ungar used static profiling techniques to bring some of the traditional optimizations using data-flow analysis, such as sub-expression elimination, dead code elimination, constant folding and copy propagation, into dynamic languages [3]. Hölzle and Ungar used **type feedback** (where they extract type information from runtime and feed that back into the compiler), to decide which calls should be inlined [5]. Brunthaler cached the calls so as to avoid multiple lookup for calls [2]. Even with all these optimizations, dynamically-typed languages have not reached the speed of statically-typed languages. A new approach to improving the performance of dynamically-typed languages is necessary.

In this thesis, I explore two main ways to improve the performance of Jython, an implementation of the Python programming language that runs on the Java virtual machine (JVM). First, I use the `invokedynamic` byte-code introduced along with Java 7 to implement various optimizations. Second, I implement *gradual typing* [11] in Jython and enable type-specialized code generation. I show that these approaches help close the gap between Jython and Java in particular and the techniques can be used to close the performance gap between dynamically-typed and statically-typed languages in general.

The paper is organized as follows: Chapter 2 provides an introduction to *gradual typing* and `invokedynamic`; Chapter 3 presents the various optimizations implemented in Jython by using `invokedynamic`; in Chapter 4 I present the details of an implementation of gradual typing in Jython; I end the thesis with some concluding remarks in Chapter 5.

Chapter 2

Background

In this chapter I present background reading material required for the thesis. In Section 2.1 I present a brief overview of gradual typing and in Section 2.2 I present an overview on `invokedynamic` framework in Java 7.

2.1 Introduction to Gradual Typing

Traditionally statically typed programming languages (such as Java, C and C++) have dominated industry usage for decades [14], their popularity is largely attributed to earlier detection of programming mistakes, better documentation in the form of type signatures, more opportunities for compiler optimizations and increased runtime efficiency. Dynamic languages (such as Python, Ruby and Javascript) are heavily used for prototyping since the code is much more compact. These languages allow programmers to get results quickly since these languages are generally not as verbose as their statically typed counterparts and are more flexible which helps reduce overheads and increase productivity [7].

In 2007, Siek and Taha introduced **gradual typing** to bridge the gap between the dynamically-typed languages and statically-typed languages by developing a framework that allowed a language to have some parts to be completely dynamically-typed and other parts to be statically typed [11] [9]. In this section I will provide a brief introduction to gradual typing.

The essence of gradual typing is that it allows input program to have optional types, where some parts of the program can have type annotations and other parts type annotations can be left

Variables	x	$\in$	$\mathbb{V}$
Ground Types	γ	$\in$	$\mathbb{G}$
Constants	c	$\in$	$\mathbb{C}$
Types	τ	$::=$	$\gamma \mid ? \mid \tau \leftarrow \tau$
Expressions	e	$::=$	$c \mid x \mid \lambda x : \tau.e \mid ee \mid$
			$\lambda x.e \equiv \lambda x : ?.e$

Figure 2.1: Syntax for gradually-typed lambda calculus.

out. Parts of the program without type annotations are assumed to be dynamic by default. The gradual typing system then converts this input program to an intermediate language with explicit casts, inserting casts where the program transitions from dynamic typed part into a statically typed part. These casts are enforced at runtime so that, if the value flowing in from the dynamically typed part of the program does not match the static types, an exception is raised. An important property of gradual typing systems is that if a program is completely annotated, then the type-checker would catch *all* the type errors at compile time. In the remainder of this section I will explain the type-checking and the cast insertion phase of a gradual typing system on a slight variation on the simply-typed lambda calculus language.

2.1.1 Static semantics

The syntax for the language gradually-typed lambda calculus is shown in Figure 2.1; where the simply-typed lambda calculus is extended with the type ? used to represent dynamic types. The lambda function without annotation on the parameter implies that the parameter x is dynamic and is a short-hand for writing $\lambda x : ?.e$. With the language defined, let us look at the type-checking rules to understand the type system of this language.

Figure 2.2 shows the type checking rules for this language. The rules look very similar to the rules for the simply-typed lambda calculus except for the function application case ($\Gamma \vdash e_1\, e_2$). Here, instead of requiring that the types of the argument to the function and the type of the function's parameter to be equal, the authors require that the types be **consistent** instead. **Consistency** written as $\sim$ is the basis of gradual typing, and allows for dynamic types to be used in place of any

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{x = \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\text{typeof}(c) = \tau}{\Gamma \vdash c : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 : ? \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \, e_2 : ?} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \rightarrow \tau' \\ \Gamma \vdash e_2 : \tau_2 \qquad \tau_2 \sim \tau\end{array}}{\Gamma \vdash e_1 \, e_2 : \tau'}$$

Figure 2.2: Type system for the gradually-typed lambda calculus.

other type and vice versa. Rules for consistency are defined below:

$$\frac{}{\tau \sim \tau} \qquad \frac{}{? \sim \tau} \qquad \frac{}{\tau \sim ?}$$

$$\frac{\tau_1 \sim \tau_3 \qquad \tau_2 \sim \tau_4}{\tau_1 \rightarrow \tau_2 \sim \tau_3 \rightarrow \tau_4}$$

Note that the consistency relation is reflexive and symmetric but not transitive. The consistency relation places a less restrictive constraint on the types, than type equality. Since all types are consistent with the type ?, the static type checker would allow an argument of any type to be passed into a parameter of type ?, making the function far more flexible. To ensure that the gradual type system does not produce type errors in statically typed parts at runtime, runtime checks are performed at appropriate places. These checks throw runtime exceptions if the checks fail. The checks can be thought of as guards making sure that the potentially malformed programs that were allowed at compile time, actually adhere to the typing rules. For example, if at compile time a expression of type dynamic was applied, then the cast makes sure that at runtime the value at that point is a function. We shall now look at the pass which inserts these checks, neatly called cast insertion phase.

The Figure 2.3 shows the rules for cast insertion phase. Cast insertion takes as input the expression and produces back a casted expression and is similar in many rules to the type-checker with the addition of an output term. The rules for function application $(e_1 \, e_2)$ are the most

$\boxed{\Gamma \vdash e \rightsquigarrow e' : \tau}$

$$\frac{x = \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x : \tau} \qquad \frac{\text{typeof}(c) = \tau}{\Gamma \vdash c \rightsquigarrow c : \tau}$$

$$\frac{\Gamma, x : \tau_1 \vdash e \rightsquigarrow e' : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e \rightsquigarrow \lambda x : \tau_1 . e' : \tau_1 \to \tau_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau \to \tau' \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau\end{array}}{\Gamma \vdash e_1\, e_2 \rightsquigarrow e_1'\, e_2' : \tau'} \qquad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : ? \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2}{\Gamma \vdash e_1\, e_2 \rightsquigarrow (\langle \tau_2 \to ? \rangle\, e_1')\, e_2' : ?}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau \to \tau' \\ \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2 \qquad \tau_2 \neq \tau \qquad \tau_2 \sim \tau\end{array}}{\Gamma \vdash e_1\, e_2 : e_1'\, (\langle \tau \rangle\, e_2')\, \tau'}$$

Figure 2.3: Cast insertion for the gradually-typed lambda calculus.

interesting ones. In the first rule for application, the function e_1 goes through cast-insertion and is converted to e_1' and expects type τ for the argument and the argument e_2' indeed has type τ, and no casts are needed in this case as the application can never fail at runtime. In the second rule for application, the function e_1' has type dynamic ?, and the argument e_2' has type τ_2. Here we insert a cast $(\langle \tau_2 \to ? \rangle\, e_1')$ on the function to make sure that the term e_1' is indeed a function that can take a term of type τ_2 as input. In the third case of function application, the function e_1' has type $\tau \to \tau'$ but the argument has type τ_2. This will not be a type error only if $\tau \sim \tau_2$, and at runtime we need to make sure to cast the term e_2' to τ and therefore this cast is inserted before the function is applied.

After the cast insertion phase, we are left with a program very similar to the original input program but with the addition of casts. Therefore we can extend the original language by with casts:

$$\text{Expressions} \quad e \quad ::= \quad ... \mid (\langle \tau \rangle\, e)$$

And the additional typing rule for casts would be:

$$\frac{\Gamma \vdash e : \tau' \qquad \tau' \sim \tau}{\Gamma \vdash (\langle \tau \rangle\, e)\ : \tau}$$

2.1.2 Dynamic semantics

This new intermediate language can then either be compiled to run or interpreted. In this subsection we will review the runtime semantics of the new language. Before diving into the details, we first define a grammar to describe the different parts of the evaluation.

$$
\begin{array}{llll}
\text{Simple values} & s \in \mathbb{S} & ::= & x \mid c \mid \lambda x : \tau . e \\
\text{Values} & v \in \mathbb{V} & ::= & s \mid (\langle\ ?\rangle\, s) \\
\text{Errors} & \varepsilon & ::= & \text{CastError} \mid \text{TypeError} \mid \text{KillError} \\
\text{Results} & r & ::= & v \mid \varepsilon
\end{array}
$$

The type result r is a super set of errors e and values v, where values can either be one of the simple values s or a simple value with a cast. Of the three types of errors defined, KillError was added by the original authors to prove type safety. The CastError symbolizes errors that the runtime system is able to catch and thus raise exceptions. The TypeError represents the class of errors which if uncaught would cause undefined behavior such as segmentation faults.

The evaluation rules are defined using big-step style with substitution, in the form $e \hookrightarrow_n r$, where e evaluates to the result r with a derivation depth of n. The evaluation rules for terms resulting in values are presented in Figure 2.4 and the rules resulting in errors are presented in Figure 2.5.

It is interesting to look at the rules of evaluation of casted expressions. Here, unbox is a helper function which will get the underlying value by removing the cast around it. The rule for function casts (2^{nd} rule in Figure 2.4) handles the case of casting to a function type. If the run-time type $\tau \to \tau'$ is consistent with the target type $\sigma \to \sigma'$, then the cast is removed. Then, a new function is created which wraps the inner value v and a cast is inserted to produce a well-typed value of the appropriate type.

If the runtime type does not match the expected type, then a CastError is raised as shown in the first rule in Figure 2.5. If a simple value is applied (function call) then the cast raises a TypeError. The authors prove that this approach is type-safe [11]. Building on their initial

$$\frac{e \mid \mu \hookrightarrow_n v \mid \mu' \qquad \emptyset \mid \Sigma \vdash \text{unbox } v : \gamma}{(\langle \gamma \rangle\, e) \mid \mu \hookrightarrow_{n+1} \text{unbox } v \mid \mu'}$$

$$\frac{\begin{array}{cc} e \mid \mu \hookrightarrow_n v \mid \mu' & \emptyset \mid \Sigma \vdash \text{unbox } v : \tau \to \tau' \\ \tau \to \tau' \sim \sigma \to \sigma' & z = \text{maxv } v + 1 \end{array}}{(\langle \sigma \to \sigma' \rangle\, e) \mid \mu \hookrightarrow_{n+1} \lambda z : \sigma.\, (\langle \sigma' \rangle\, \text{unbox } v\, (\langle \tau \rangle\, z)\,) \mid \mu'}$$

$$\frac{e \mid \mu \hookrightarrow_n v \mid \mu'}{(\langle\ ? \rangle\, e) \mid \ \hookrightarrow_{n+1} (\langle\ ? \rangle\, v) \mid \mu'} \qquad \frac{0 < n}{\lambda x : \tau . e \mid \mu \hookrightarrow_n \lambda x : \tau . e \mid \mu}$$

$$\frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n \lambda x : \tau e_3 \mid \mu_3 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x := v_2]e_3 \mid \mu_3 \hookrightarrow_n v_3 \mid \mu_4 \end{array}}{e_1\, e_2 \mid \mu_1 \hookrightarrow_{n+1} v_3 \mid \mu_4}$$

$$\frac{0 < n}{c \mid \mu \hookrightarrow_n c \mid \mu} \qquad \frac{e_1 \mid \mu_1 \hookrightarrow_n c_1 \mid \mu_2 \qquad e_2 \mid \mu_2 \hookrightarrow_n c_2 \mid \mu_3}{e_1\, e_2 \mid \mu_1 \hookrightarrow_{n+1} \delta c_1\, c_2 \mid \mu_3}$$

Figure 2.4: Evaluation rules going to values for the gradual typing intermediate language.

$$\frac{\begin{array}{c} e \mid \mu \hookrightarrow_n v \mid \mu' \\ \emptyset \mid \Sigma \vdash \text{unbox } v : \sigma \qquad \sigma \nsim \tau \end{array}}{(\langle \tau \rangle\, e) \mid \mu \hookrightarrow_{n+1} \text{CastError} \mid \mu'}$$

$$\frac{e \mid \mu \hookrightarrow_0 \text{KillError} \mid \mu}{} \qquad \frac{0 < n}{x \mid \mu \hookrightarrow_n \text{TypeError} \mid \mu}$$

$$\frac{e_1 \mid \mu \hookrightarrow_n v_1 \mid \mu' \qquad v_1 \notin \text{FunVal}}{e_1\, e_2 \mid \mu_1 \hookrightarrow_{n+1} \text{TypeError} \mid \mu'}$$

$$\frac{e \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{(\langle \tau \rangle\, e) \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'} \qquad \frac{e_1 \mid \mu \hookrightarrow_n \varepsilon \mid \mu'}{e_1\, e_2 \mid \mu \hookrightarrow_{n+1} \varepsilon \mid \mu'}$$

$$\frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n v_1 \mid \mu_2 \\ v_1 \in \text{FunVal} \qquad e_2 \mid \mu_2 \hookrightarrow_n \varepsilon \mid \mu_3 \end{array}}{e_1\, e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_3} \qquad \frac{\begin{array}{c} e_1 \mid \mu_1 \hookrightarrow_n v_1 \mid \mu_2 \\ e_2 \mid \mu_2 \hookrightarrow_n v_2 \mid \mu_3 \\ [x := v_2]e_3 \mid \mu_3 \hookrightarrow_n \varepsilon \mid \mu_4 \end{array}}{e_1\, e_2 \mid \mu_1 \hookrightarrow_{n+1} \varepsilon \mid \mu_4}$$

Figure 2.5: Evaluation rules going to errors for the gradual typing intermediate language.

work, Siek et. al. extend gradual type system to handle object oriented languages [9]. They also introduced an efficient way of storing the runtime cast information [12] [10].

2.2 Introduction to `invokedynamic`

In 2010, Java-7 introduced `invokedynamic` [8], a new byte-code in the Java virtual machine (JVM) to give more control to developers of dynamic languages on method dispatch. Specifically `invokedynamic` provides user-defined call site semantics, so that a language runtime can link (and re-link) the call site based on runtime information. In this section I present an introduction on this new bytecode.

The JVM was initially designed to support only the Java programming language, but it quickly outgrew it's initial purpose. Today many languages are implemented on top of the JVM owing to the JVM's type-safe byte code, optimizing just in compilers (JIT), scalable concurrency, wide distribution, and the rich ecosystem of libraries and debugging tools surrounding it. Not only do existing languages such as Python, Javascript, and Ruby have JVM implementations; but there are also plenty of languages developed solely on the JVM such as Scala, Fortress, and Groovy. The total number of languages with JVM implementations is more than 200 [15].

A large number of these languages have dramatic semantic differences when compared to Java, making the translation from a non-Java language into JVM byte code difficult. For example the language may support strong updates to an object, where the programmer can change a bound method; or support mutable classes; or support arithmetic operations on dynamically typed objects or multiple return values. Compiling such features require the implementation to keep additional Java data structures, and possibly use of Java core reflection API which increase the execution overhead and complexity. To address this issue the Da Vinci Machine project was set up and they introduced the `invokedynamic` framework in Java 7.

Traditionally, before Java 7, the Java virtual machine only supported 4 instructions to invoke a method:

- `invokevirtual` invokes the method on an instance of an object, dispatching on the virtual type of the object.
- `invokestatic` invokes a static method on a named class.
- `invokeinterface` invokes a method on an interface; the virtual machine searches the methods implemented by the runtime object and dispatches to the appropriate method.
- `invokespecial` invokes a method on an instance requiring special handling such as the constructor method.

With Java 7, a new byte code for method invocation was added called `invokedynamic` [6]. `invokedynamic` invokes the method which is the target of the call site object bound to the `invokedynamic` instruction. At the first execution of the `invokedynamic` instruction, the Java virtual machine calls a bootstrap method which returns the call site object, and this is bound to the specific `invokedynamic` instruction. Unlike other invoke methods mentioned above, each `invokedynamic` instruction in the byte code has a unique linkage state. A key point to emphasize here is that `invokedynamic` is only a new instruction at in the JVM bytecode and does not have an equivalent Java representation. It is unique because this is the first time that the JVM has added a feature which does not have a direct equivalent in Java.

During the first call of an `invokedynamic` instruction, a bootstrap method is called. The bootstrap method is provided by the custom language implementer, such as the sample bootstrap method shown below:

```java
public CallSite bootstrap(Lookup l, String name, MethodType type) {
    return new CallSite(...);
}
```

Listing 2.1:

The contract of the bootstrap method is as follows:

- The JVM passes the `Lookup` object (explained in the next paragraph), the method name and the method type as parameters to the bootstrap method.

- The bootstrap method has to return an object of class `java.invoke.CallSite` (or it's subclass), where the language runtime is free to point this `CallSite` to any method satisfying the received method type.

When the bootstrap method call returns, the CallSite object is bound to the specific `invokedynamic` instruction by the JVM and the association is unchanged unless changed explicitly by the language runtime. After the `CallSite` object is bound, the JVM extracts the target method reference from the `CallSite` object (stored in the private field `CallSite.target`) and dispatches a call to that method. Every other invocation of this particular `invokedynamic` instruction will use the bound `CallSite` object to dispatch.

To be able to point to arbitrary methods, Java 7 introduces the concept of method handles. Every method handle has a specific type (an object of `MethodType`) and behaves as a typed function pointer. The `CallSite` object stores the target of the call site in a `MethodHandle` object. Method handles are created by looking up the method or field on a `Lookup` object. The `Lookup` object is a factory for creating method handles. Method handles do not perform access checks when they are called, but rather when they are created. Therefore, method handle access restrictions are enforced when a method handle is created.

The simplest form of method handle is a direct method handle, Listing 2.2 shows an example which creates a method handle to `System.out.println` and calls it. Direct method handles can emulate all the invoke bytecode instructions (other than `invokedynamic`) via corresponding `Lookup` methods, such as, `invokevirtual` via `Lookup.findVirtual`; `invokestatic` via `Lookup.findStatic` and so on.

Method handles are not limited to just direct method handles, there are other kinds of method handles such as adapter method handles, bound method handles and control flow method handles.

Adapter method handles are a way to create a wrapped method handle of a new type, which

```java
MethodHandle println = MethodHandles.lookup().findVirtual(
    PrintStream.class, "println",
    MethodHandles.methodType(void.class, String.class));
println.invokeExact(System.out, "hello, world");
```

Listing 2.2: Example of creating a direct method handle and invoking it.

first calls the wrapper method which then delegates to the original target. These adapter method handles provide functionality that is frequently desired such as `convertArguments` : which can be used to cast an argument to a different type, `dropArguments`: which can be used to ignore one or more parameters and `permuteArguments`: which can re-order the parameters before calling the underlying function.

Bound method handles provide a way to bind arbitrary parameters to an object (provided that the types match) and this returns a wrapped method handle which has the same type as the previous method handle with the bound parameter dropped.

Control flow method handles were introduced so that language runtimes can encode complex logic into the method dispatch mechanism. The `MethodHandles.guardWithTest` provides the following semantics:

$$\forall S \quad \lambda(\mathit{test}, \mathit{target}, \mathit{fallback}). \quad \lambda\ \mathsf{a} : S. \quad \texttt{if} \ \ (\mathit{test}\ \mathsf{a}) \ \ \texttt{then} \ \ \mathit{target}\ \mathsf{a}\ \texttt{else} \ \ \mathit{fallback}\ \mathsf{a}$$

This can be used to implement inline caches or guarded calls. Another control flow mechanism is `SwitchPoint.guardWithTest` where one can register many *target* and *fallback* method handles to an instance of `SwitchPoint` but there is no *test* method handle. All the calls are always dispatched to the *target* method handle, until `SwitchPoint.invalidateAll` method is called, at which point all the method handles are switched to use the *fallback* method handle.

Constant method handles always return a (constant) value bound to the method handle without dispatching to any method.

```java
public static CallSite bootstrap(Lookup lookup, String name,
        MethodType type, String base64Array) {
    BASE64Decoder decoder = new BASE64Decoder();
    byte[] bytes = decoder.decodeBuffer(base64Array);

    Class<?> returnType = type.returnType();
    Object value = convertAsArray(bytes, returnType);
    return new ConstantCallSite(MethodHandles.constant(returnType,
                                                        value));
}
```

Listing 2.3: Lazy decoding of a string into bytes.

After looking at the different types of method handles, we can now look at a couple more examples of bootstrap methods to better understand the possibilities enabled by `invokedynamic`. Listing 2.3 shows how the bootstrap argument can be used to create a string to byte decoder. Here, the string in `base64Array` is only decoded on the first `invokedynamic` call, which calls the bootstrap method. Also, here we use a constant method handle to cache the value of the byte array, therefore every `invokedynamic` call would just return this cached value [4].

The Listing 2.4 shows an example which install a generic target method `install` in the bootstrap method. The `install` method takes all the runtime arguments and can re-bind the call site if as appropriate.

Although `invokedynamic` is not exposed to the Java language itself, it is being used to implement upcoming language features in the language such as lambda expressions and virtual extension methods. It is also used in various other projects such as: Nashorn project which is an implementation of JavaScript on the JVM; JRuby which is an implementation of Ruby on the JVM and has proved useful in efficiently compiling complex language features.

```java
public static CallSite bootstrap(Lookup lookup, String name, MethodType
    type) {
    MyCallSite callSite = new MyCallSite(lookup, name, type);
    MethodHandle install = INSTALL.bindTo(callSite);
    callSite.setTarget(install);
    return callSite;
}

public static Object install(MyCallSite callSite, Object[] args) {
    MethodType type = callSite.type();
    Class<?> receiverClass = args[0].getClass();
    MethodHandle target = callSite.lookup.findVirtual(receiverClass,
        callSite.name, type.dropParameterTypes(0, 1));
    target = target.asType(type);
    callSite.setTarget(target);
    return target.invokeWithArguments(args);
}
```

Listing 2.4: Binding a call site based on runtime arguments instead of bootstrap arguments.

Chapter 3

Jython Optimizations

Compiling a dynamic language with powerful features such as Python [16] to Java byte code is nontrivial, and generating high performance Java bytecode is challenging. There are a few challenges in the efficient compilation of Python to Java byte code. Firstly, the JVM is tuned to optimize naïve Java code, but code generated by Jython is usually complex and does not equate to naïve Java code. Secondly, Python has a number of dynamic features limiting the ability to apply any optimizations at compile-time. Since Python has late-binding of names, it prevents Jython from loading built-in functions or names directly; they can only be loaded at runtime. Python provides the ability to change an object's method resolution order (MRO) which implies that method dispatch would require heavy use of runtime data structures and creating nested call graphs for even simple method calls, limiting the JVM's ability to inline them.

In this chapter I tackle some of these challenges and show how optimizations can be implemented with the use of `invokedynamic`. In Section 3.1 I describe the general framework used for some of the optimizations and then use this framework to for two different optimizations described in Subsection 3.1.1 and Subsection 3.1.2. In Section 3.2 I present a way to optimize the function dispatch mechanism in Jython. Section 3.3 presents a way of utilizing programmer input to improve the runtime performance. Finally, in Section 3.4 I evalute the optimizations implemented.

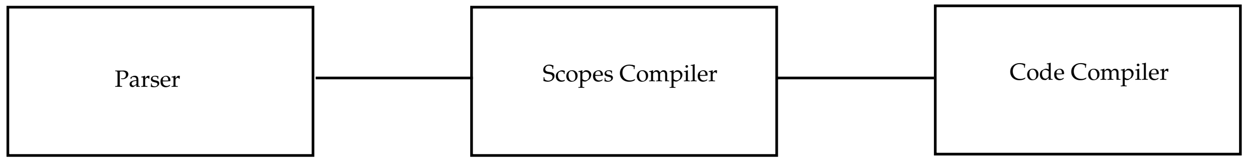

Figure 3.1: The current Jython framework.

3.1 A framework for AST transformations

Jython currently does not perform any optimization passes, therefore before diving into performing optimizations it was important to first develop an extendable framework that would support optimization transformations. In this section I will discuss the framework I developed in Jython.

A simplified overview of current Jython compiler framework is shown in Figure 3.1. Although this representation an oversimplification, it suffices for our discussion here. As with most compilers, the parser produces an abstract syntax tree (AST) – a tree containing nodes where each node corresponds to a Python statement or expression. In Jython this AST is then passed to the Scopes Compiler which creates another data-structure containing information regarding scopes. Then the AST and the scopes data-structures are passed to the Code Compiler which compiles each node to Java byte code creating a Java class. The AST nodes are tightly coupled with the language and traditionally new nodes were only added when the language changed. This limits the number of optimizations possible with the current framework.

To make the framework more extensible, I extended the AST by adding custom nodes that were not part of the language definition. These nodes represent the optimized nodes and could be compiled differently. The AST from the parser would still generate the same nodes as before, but then one or more optimization passes would then operate over the AST adding custom nodes as required. An overview of this approach is shown in Figure 3.2. The AST transformation passes would generate a new AST as output which would include the custom added nodes representing parts of the program for which optimized code can be generated. This AST

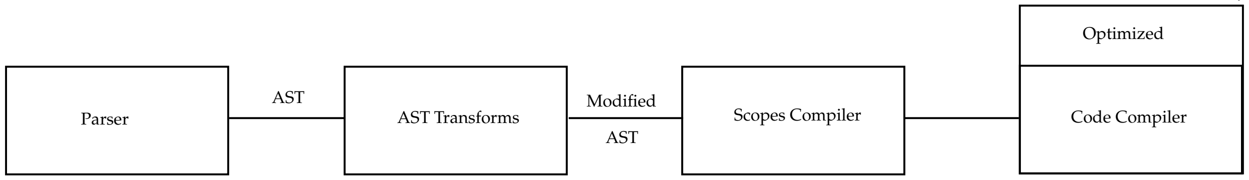

Figure 3.2: Overview of the AST transformation framework.

would then pass through the Scopes Compiler which remains unchanged for the most part. A new class called `OptimizedCodeCompiler` is created and this extends the previously existing `CodeCompiler`, thus all the old nodes would still be handled by the `CodeCompiler` class and the `OptimizedCodeCompiler` would only handle the newly added custom nodes that require optimized byte code generation. The output from this stage is Java byte code which is potentially optimized for performance.

In the remainder of this section I will dive into how `for` loops in Python can be optimized in Jython (Subsection 3.1.1) and how global name lookups for built-in functions can be optimized (Subsection 3.1.2).

3.1.1 Optimizing loops in Jython

Consider the Python program shown in Listing 3.1, it iterates over the variable `i` ranging from 1 to 10,000,000 and performs some mathematical operations in each iteration. Jython uses a generic compilation strategy for all loops, and the output, de-compiled to Java, is presented in Listing 3.2.

In the generic loop compilation strategy presented in Listing 3.2, we see that in lines 1-4 initializes `PyObjects` corresponding to 1, 10M, `xrange` and the first item from the `xrange` object. Lines 5-13 show the actual loop and the arithmetic operations take place in lines 8-10.

```
for i in xrange(1, 10000000):
    sum += (sum ^ i) / i
```

Listing 3.1: A Python program with `for` loop.

```java
PyObject _2 = Py.newInteger(1);
PyObject _3 = Py.newInteger(10000000);
xrangeObj = frame.getglobal("xrange").__call__(ts, _2, _3).__iter__();
iterTmp = ((PyObject) xrangeObj).__iternext__();
while (iterTmp != null) {
    frame.setlocal(2, iterTmp);
    PyObject sumPyObj = frame.getlocal(1);
    localPyObj = sumPyObj._iadd(frame.getlocal(1)
                                ._xor(frame.getlocal(2))
                                ._div(frame.getlocal(2)));
    frame.setlocal(1, localPyObj);
    iterTmp = ((PyObject) xrangeObj).__iternext__();
}
```

Listing 3.2: The de-compiled output of current compilation strategy for the Python program in Listing 3.1 in Jython.

In line 6 the current iteration item is stored onto the frame (variable `i`), line 7 loads the variable `sum`); line 11 stores the result of the operation back into the variable `sum` and line 12 loads the next object of the iteration. From the generated output we observe that the Jython compilation strategy is to use a generic solution by using iterators and a while loop iterating over them. Although this maintains all Python semantics; the generated code is not highly optimized since it provides very little opportunity for optimization by the JVM. Since the JVM was primarily designed for the Java language, most just in time (JIT) compilers developed for the JVM such as HotSpot (from Oracle), J9 (from IBM) or Zing (from Azul Systems) are primarily tuned to recognize Java code and then optimize them. Therefore generating idiomatic Java byte code would help the JVM optimize code, improving the performance of Jython. If we could generate code that resembles a Java **`for`** loop, such as the code presented in Listing 3.3 then the JVM could use readily recognize it which would enable JIT optimizations.

```
for(int i = 0; i < 10000000; i++) {
    sum += (sum ^ i)/ i;
}
```

Listing 3.3: Ideal compilation.

We could potentially use the information that the built-in functions such as `range` and `xrange` always iterate over integers, and then implement a compilation strategy that would generate a Java **`for`** loop for such Python loops iterating over these built-in functions, but we have to be careful to not break Python semantics. At runtime the name `"xrange"` can be bound to any arbitrary method, for example it maybe bound to an iterator that returns floating point numbers instead of integers; or possibly a function that returns `None`. So the compilation strategy must first check if the name `"xrange"` is indeed bound to the Python's built-in `xrange`.

Our approach to handle such situations, is to use an `invokedynamic` check as a guard and generated optimized byte code, in the fallback case we could fallback to Jython's interpreter or to a generated generic loop. Such an approach with fallback to Jython's interpreter is shown in Listing 3.4. The main point here is that we make an `invokedynamic` call and pass the object bound to "xrange". This call would return normally if the object is indeed bound to Python's built-in xrange function, else it throws an IndyOptimizerException. If the call returns normally, then we proceed to line 3 where we declare integers as loop iterators and in lines 5-8 we perform the loop operation. If the call throws an exception then we fallback to using Jython interpreter to execute that part of the code instead.

To implement this optimization, I created a new (custom) AST node called OptimizedFor which extends the already existing node For. In the AST transform phase, instances of For loop using "xrange" or "range" to iterate over the loop were replaced with this new node OptimizedFor. Then in the OptimizedCodeCompiler class, this new node was compiled to generate code corresponding to the one presented in Listing 3.4.

I compared the running times of the Python program in Listing 3.1 under 3 Python platforms: CPython (version 2.7.3); Jython (version 2.7.0a2) and the custom Jython version with this loop optimization enabled. The result is presented in Figure 3.3. The figure shows a bar graph, where the height of the bar determines how much faster the program ran when compared to Jython 2.7.0a2. We see that CPython runs the program about 7.75 times faster than Jython; and we see that our optimization has enabled this program to run *15* times faster than stock Jython. Therefore, by generating idiomatic Java code and using the knowledge of the return types of key builtin functions, we have enabled the JVM to optimize the loop and improve the performance by a factor of 15x in simple cases.

3.1.2 Optimizing global name lookups

Global names in Python are resolved dynamically, at runtime and the Jython compiler cannot rely on the names of the variables at compile-time but has to always perform a lookup on the variable string at runtime. Functions and classes are often defined in the global scope of a module and in most Python code they are rarely re-bound to other objects. With this insight, we can implement an optimization which will cache the global objects such that the overhead of lookup is reduced. This cache has to be invalidated when the global object is rebound. This would make the

```
try {
        indyDispatch();
        int start = 0, stop, step = 1, downCountFlag = 0;
        if(pyObjStart)
            start = pyObjStart.asInt();
        if(pyObjStop)
            stop = pyObjStop.asInt();
        if(pyObjStep){
            step = pyObjStep.asInt();
            if(step == 0)
                Py.ValueError("step argument must not be zero");
            else if(step < 0):
                downCountFlag = 1;
        }

        PyObject iterTmp = null;
        for(int i = start; (downCountFlag == 0) ? i < stop : i > stop;
           i += step){
            iterTmp = Py.newInteger(i);
            // Do the loop stuff here
        }
    } catch (IndyOptimizerException e) {
    // fallback to Python byte code virtual machine
}
```

Listing 3.4: The de-compiled output of a fast implementation of the Python program in Listing 3.1 using `invokedynamic`.

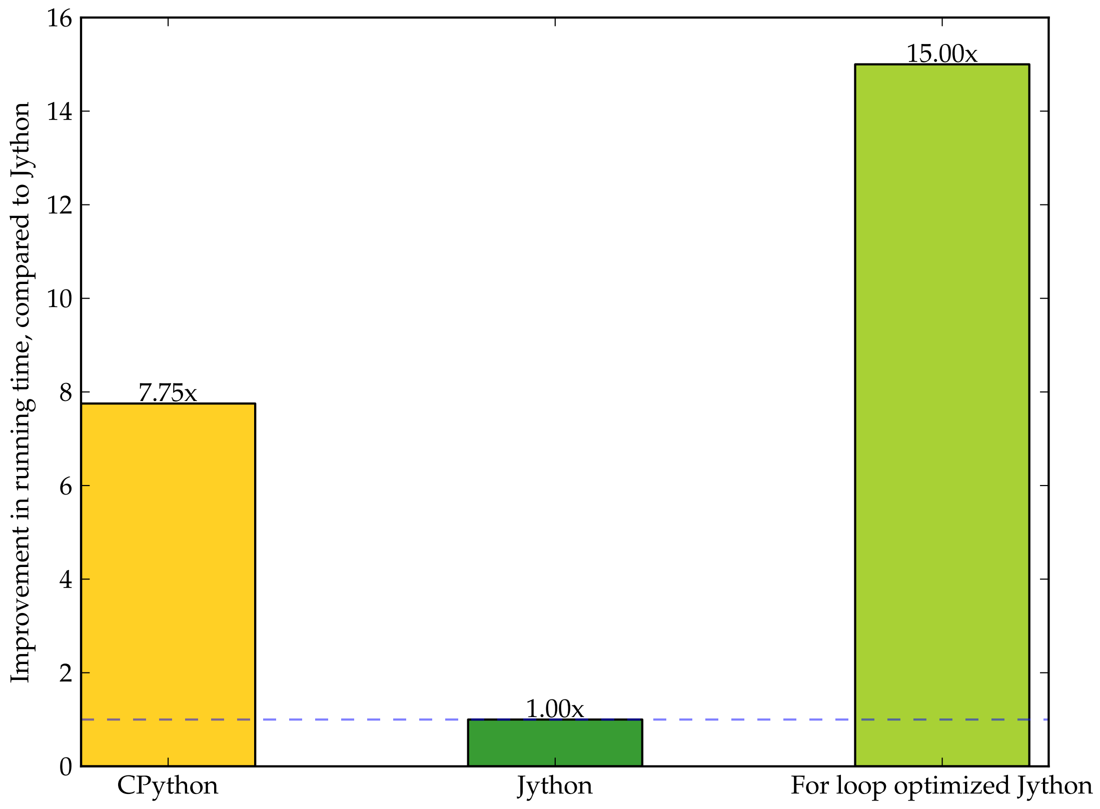

Figure 3.3: Comparing the performance of the loop optimization.

```python
s = "hello world"

def foo():
    print s
```

Listing 3.5: Example Python program to show a global call.

global accesses fast (since the lookup is avoided), while making the re-binding of global variables slower than it currently is. I use the `SwitchPoint` mechanism provided with `invokedynamic` to implement this optimization.

Consider the example presented in Listing 3.5, here the string `s` is defined in the global scope of a module and is used in the function `foo`. Listing 3.6 shows the Java equivalent of the Java byte code produced when compiling this example with current version of Jython. At compile-time the compiler knows that the variable `s` does not exist in the local scope of the function `foo` since the variable `s` does not appear as the parameters of the function nor in the body of the function. So the compiler can infer that the variable `s` was defined in the global scope. As seen in line 2 of the Java code, that Jython generates a `getGlobal` call on the frame.

Even though this seems harmless, each load of a global variable results in a frame lookup which can be costly when trying to optimize hot loops. Consider for example a simple variation on the previous example show in Listing 3.7, here the variables `a` and `b` are both global variables and are accessed 5,000,000 times in the loop in line 6. Here we can see that loading global variables from the frame would be costly.

Each `getGlobal` call on the frame is internally dispatched to a lookup on a `PyDict` object called `globals` in the frame object. `PyDict` contains a HashMap mapping `PyStrings` to

```java
1 public PyObject foo$3(ThreadState ts, PyFrame f) {
2     PyObject local = f.getGlobal("s");
3     Py.Println(local);
4     return Py.None;
5 }
```

Listing 3.6: Java equivalent of Jython generated Java byte code for Python example in Listing 3.5.

```python
a = 1
b = 2

def foo():
    result = 0
    for _ in xrange(5000000):
        result += a + b       # 2 global accesses
    print(result)

foo()
```

Listing 3.7: Another Python program stressing the global lookup.

`PyObjects` where a given string such as `"s"` will be looked up. Going back to the first example in Listing 3.5, let us look at how to improve the performance. The goal here is to cache the value of the `PyObject` bound to the string `"s"` in line 2. The cache has to be invalidated if the global variable `"s"` changed. SwitchPoint and `invokedynamic` provide the right tools to for this optimization. We can make the `getGlobal` call an `invokedynamic` (in line 2 in Listing 3.6) call which first performs a lookup to get the global object and caches it at the call-site. The object is actually cached in the `MethodHandle` associated with the call-site and the method handle has a switch point guard around it. The global switch point is turned off whenever a global variable is updated. This ensures that all `getGlobal` call-sites would re-load the value of the cached object maintaining Python semantics.

To implement this optimization, we first need to move the globals out of the frame and associate it with the module instead. Associating the globals `PyDict` would mean all the `setGlobal` and `getGlobal` operations would happen on this dictionary and it would make sure that we invalidate the switch points consistently. Each module in Python is implemented as a class extending `PyFunctionTable` in Jython. I created a new field of in the base class `PyFunctionTable` to hold a reference to the globals and all the `getGlobal` and `setGlobal` are now redirected from operations on the frame to operations on this new field. Listing 3.8 shows the optimized code with `invokedynamic` call dispatch for `getGlobal`. The only difference in the generated byte code is in line 2 where the `getGlobal` lookup on the frame is replaced by an `invokedynamic` bytecode,

```java
public PyObject foo$3(ThreadState ts, PyFrame f) {
    PyObject local = invokedynamic<getGlobalBSM>(this, "s");
    Py.Println(local);
    return Py.None;
}
```

Listing 3.8: Java equivalent of the `invokedynamic` optimized `getGlobal` call for the Python example in Listing 3.5.

bound to the `getGlobalBSM` bootstrap method. The `invokedynamic` call is given the current object (**`this`**) and the string of the variable (`"s"`) as arguments.

The bootstrap method is shown in Listing 3.9, and this method create a new `CallSite` bound to the method handle point to the method `initGetGlobal` in the `InvokeDynamicSupport` class. The method `initGetGlobal` is shown in Listing 3.10. In this method, in line 3, we first get the value associated with the string from the `PyFunctionTable` which is module object. In lines 4-6 we create a new constant method handle. A constant method handle is a method handles which always returns the value it is bound to, irrespective of the inputs. We wrap the constant method handle with `dropArguments` to make sure the type of the `target` method handle matches the type of the call-site. In lines 7-9 we wrap the method handle with a switch point, and in line 10 we add this switch point to the HashMap `global_sps` against the name of the global variable (`index`). Then in line 11, the call site is updated with this new constant method handle. Lines 12-16 implement a re-linking logic, which triggers the re-caching mechanism.

With this logic implemented, we can look back at the stress-test example in Listing 3.7. Each of the global variable reads of variables `a` and `b` in line 7 will be a `invokedynamic` instruction which would cache the value of the global variable in a constant method handle. Since the global variables are not modified, the switch point would never be invalidated; thus making global lookups considerably faster.

The Figure 3.4 shows a bar graph of the performance improvement of running the program in Listing 3.7 on Jython versus running the same program on CPython and the custom build of Jython with the global lookup mechanism via method handles enabled. We see that caching the

```
public static CallSite getGlobalBootstrap(Lookup lookup, String name,
                                          MethodType type) {
    GetGlobalCallSite site = new GetGlobalCallSite(type);
    MethodHandle init = lookup.findStatic(
        InvokeDynamicSupport.class, "initGetGlobal",
        methodType(PyObject.class, GetGlobalCallSite.class,
                   PyFunctionTable.class, String.class));
    init = insertArguments(init, 0, site);
    site.setTarget(init);
    return site;
}
```

Listing 3.9: The init method called by the bootstrap method on the first call (or a switch point invalidate) of the `getGlobal` `invokedynamic` call.

```
public static PyObject initGetGlobal(GetGlobalCallSite site,
  PyFunctionTable self, String index) throws Throwable {
    PyObject value = self.getglobal(index);
    MethodHandle target = dropArguments(
        constant(PyObject.class, value), 0, GetGlobalCallSite.class,
        PyFunctionTable.class, String.class);
    SwitchPoint sp = new SwitchPoint();
    target = sp.guardWithTest(target, MH_GET_GLOBAL);
    target = insertArguments(target, 0, site);
    self.global_sps.put(index, sp);
    site.setTarget(target);
    if(site.changedCount == GetGlobalCallSite.MAX_CHANGE_COUNT) {
        self.global_sps.remove(index);
        site.setTarget(MH_GET_GLOBAL_FALLBACK);
    }
    return value;
}
```

Listing 3.10: The init method called by the bootstrap method on the first call (or a switch point invalidate) of the `getGlobal` `invokedynamic` call.

global value in a method handle increases the performance by a factor of *2.5x*. We can also see that this performance is about 2 times better than CPython.

3.2 Changing the dispatch mechanism

Chapter 2 explains the `invokedynamic` instruction. This section will mainly deal with using the instruction to change the dispatch mechanism of Jython. Subsection 3.2.1 shows how the call dispatch mechanism work before invokedynamic and Subsection 3.2.2 presents the current implementation using `invokedynamic`.

3.2.1 Current dispatch mechanism in Jython

In this section I describe how how function dispatch currently works in Jython, with an example. Consider the example Python code presented in Listing 3.11. The function `foo` calls the function `bar` with an argument 10.

Jython compiles this Python program into Java byte code and Listing 3.12 shows the Java equivalent of the generated Java byte code. lines 3-5 show the equivalent of the function `bar`. All Jython generated Java methods have the same signature, which is: `PyObject` as the return type and `ThreadState, PyFrame` as arguments. The `ThreadState` stores the state of the current thread and a reference to previous frames and the `PyFrame` object stores references to parameters and local variables among other things.

Having a uniform signature for all generated methods makes the compiler design a little bit easier, all the arguments passed into a function is stored in the frame; but, this also means that the generated code cannot be optimized nearly as well if we had used the internal stack of the JVM for passing arguments.

In Listing 3.11, at line 5, there is a call to the function bar; but in general when the compiler encounters a call, it does not know the function, method or class being called. Therefore the compiler must first get the object bound to the name and then call it. Lines 8-9 of Listing 3.12 show this exact procedure, where, in order to respect Python semantics, first the object bound to

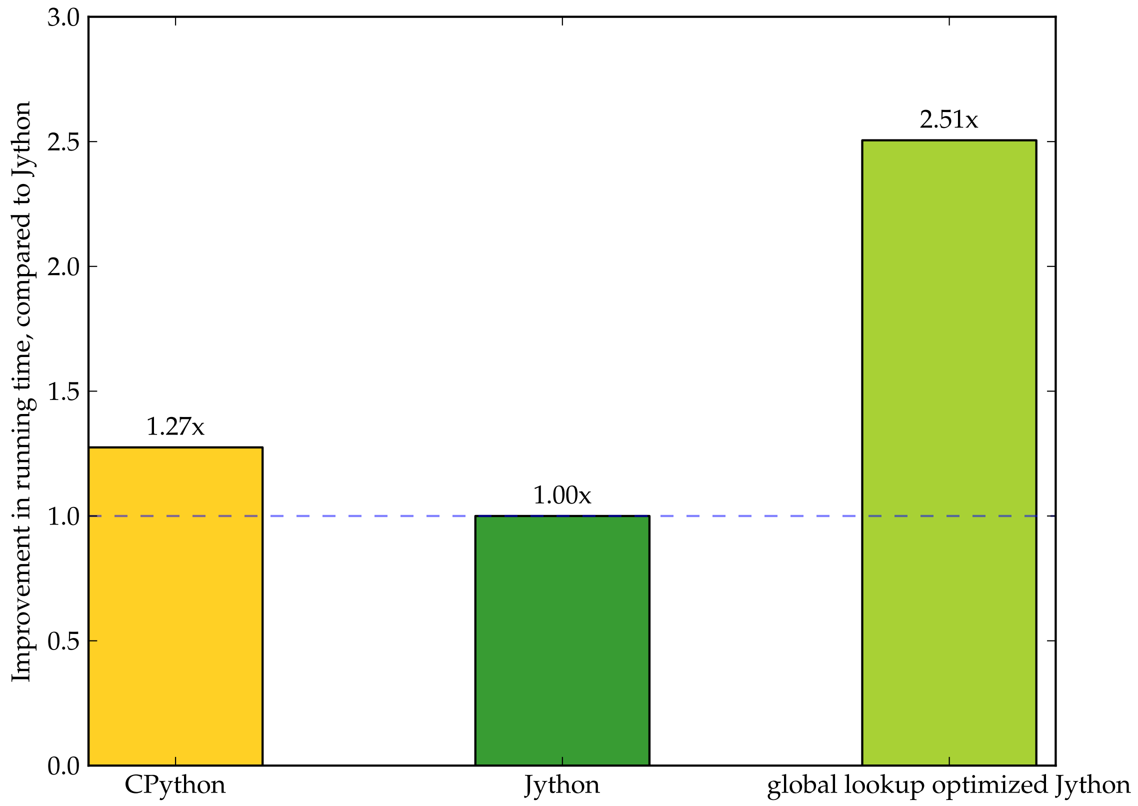

Figure 3.4: Comparing the relative performance the `getGlobal` stress test from Listing 3.7.

```python
def bar(n):
    pass

def foo():
    bar(10)
```

Listing 3.11: Example Python program.

```
PyInteger _10 = new PyInteger(10);

public PyObject bar$2(ThreadState ts, PyFrame f) {
    return Py.None;
}

public PyObject foo$3(ThreadState ts, PyFrame f) {
    PyObject local = f.getName("bar");
    local.__call__(ts, _10);
    return Py.None;
}

public PyObject call_function(int func_id, PyFrame f, ThreadState ts) {
    switch(func_id) {
      case 2: return bar$2(ts, f);
      case 3: return foo$3(ts, f);
      ...
    }
}
```

Listing 3.12: Java equivalent of Jython generated Java byte code for Python example in Listing 3.11.

```
 1 public class PyFunction extends PyObject {
 2
 3     public PyCode func_code;
 4
 5     @Override
 6     public PyObject __call__(ThreadState ts, PyObject arg0) {
 7         return func_code.call(ts, arg0, ...);
 8     }
 9
10 }
```

Listing 3.13: Showing the relevant portions of PyFunction.

the name `"bar"` is loaded (line 8) and then the `__call__` method is called on this object with the current `ThreadState` object (`ts`) and the argument `_10` (line 9). The dispatch mechanism then depends on the type of object loaded.

In this case the object bound to the name `"bar"` is a `PyFunction`, Listing 3.13 shows the relevant parts of `PyFunction`. All run-time objects in Jython extend `PyObject` and `PyFunction` is no different. The `__call__` method call on line 9 in Listing 3.12 calls into line 6 of Listing 3.13. Each `PyFunction` object is associated with one `PyCode` object which is the actual implementation of the function definition. This extra level of indirection is added to support mutable function objects.

Line 7 calls the `call` method on the `func_code` object with the arguments and a few other parameters (omitted here since they are not important to this discussion). This call re-directs to `PyBaseCode` which creates a new `PyFrame` object using the passed arguments and then redirects to line 7 in Listing 3.14. At this point the arguments passed in at the call-site have been put on the newly created `PyFrame` object. `PyTableCode` extends `PyBaseCode` which in-turn extends `PyCode` and contains a reference `funcs` to the object in which the method `bar$2` is defined in. It also has the `func_id` (2 for function `bar`) of the method. This method calls the `call_function` method on the `funcs` object which then takes us back to Listing 3.12 line 13. The `call_function` defines a switch-case on `func_id` and calls the methods with their corresponding `ids` and since the function `bar` has `id` 2, we call `bar$2` in line 15.

```java
public class PyTableCode extends PyBaseCode {

    PyFunctionTable funcs;
    int func_id;

    @Override
    public PyObject call(ThreadState ts, PyFrame frame, PyObject
      closure) {
        PyObject ret;
        ...
        ret = funcs.call_function(func_id, frame, ts);
        ...
        return ret;
    }
}
```

Listing 3.14: showing the relevant portions of PyTableCode.

So, we see that for even a simple method call, there is a lot of back-and-forth between the generated Java byte code and the Jython runtime. In the next sub-section we will device a way of avoiding this by using `invokedynamic`.

3.2.2 Dispatch mechanism in Jython with **`invokedynamic`**

In this sub-section we will see how to use `invokedynamic` machinery to dispatch to functions so that the JVM can readily optimize these calls. We would like to perform two optimizations, first we want use `invokedynamic` calls to dispatch function calls instead of the `__call__` mechanism seen earlier. Next, we want to make use of parameter passing in the JVM instead of using frames so that accessing parameters in a function is easier.

3.2.2.1 Dispatching to a method using **`invokedynamic`**

Since the function resolution happens at runtime in Python, we do not know which function will be called at compile time or even at link time. Since the bootstrap method of a `invokedynamic` call is called at link-time, at this time we still do not have all the information we need to dispatch the call. So at bootstrap time we bind the call-site to a generic dispatch method. We make sure to pass the `CallSite` object to this generic dispatch method which will be called at runtime on the first call to the `invokedynamic` bytecode. At this point we can find out the target method of a given function call and re-bind the call site appropriately. This would ensure that future calls to the same function at that call-site would be directly dispatched to the actual target method.

In order for this optimization to work, every runtime function should be associated with a method handled. The runtime function object in Jython is `PyFunction`, but the actual code object is stored in the `func_code` attribute of `PyFunction` which is of type `PyCode`. So a new field is added to `PyCode` called `mHandle` which stores the method handle associated with that code object, and this function can then be called via this method handle.

Listing 3.15 shows the de-compiled Java byte code generated by Jython for the example in Listing 3.11 using the `invokedynamic` machinery. We see that line 9 is now replaced by an

```java
PyInteger _10 = new PyInteger(10);

public PyObject bar$2(ThreadState ts, PyFrame f) {
    return Py.None;
}

public PyObject foo$3(ThreadState ts, PyFrame f) {
    PyObject local = f.getName("bar");
    invokedynamic<bootstrap>(ts, local, _10);
    return Py.None;
}
```

Listing 3.15: Dispatch using `invokedynamic` for code shown in Listing 3.11.

`invokedynamic` call instead of the previous `__call__` method call. We have already seen that at compile time we do not know what method we are dispatching to, therefore we must wait till runtime to make the decision of where to dispatch a call. The `invokedynamic` byte code is provided with the thread state, the `local` variable corresponding to the function object and `_10` as arguments. We also see that this `invokedynamic` byte code is bound to `bootstrap` as the bootstrap method.

Listing 3.16 shows the bootstrap method, that the `invokedynamic` call is bound to. This method this returns a generic `JythonCallSite` object (a sub-class of `CallSite`) which binds the call-site to a the `installCS` method. After calling the bootstrap method on the first call of `invokedynamic`, the JVM proceeds to call the `installCS` method. This method is shown in Listing 3.17.

The `installCS` is given the call site object from before and the rest of the arguments passed in from the actual `invokedynamic` call-site is received in the `args` Object array. Since we are dealing with specializing function dispatch, we check to see if the 1^{st} argument is actually

```java
public CallSite BSM(Lookup lookup, String name, MethodType type) {
    return new JythonCallSite(lookup, name, type);
}
```

Listing 3.16: Bootstrap method for function dispatch.

```
public static Object installCS(JythonCallsite callSite,
                               Object[] args) throws Throwable {
    if (args[1] instanceof PyFunction) {
        PyFunction pyFunc = (PyFunction) args[1];
        int noOfArgs = args.length - 2;
        if (pyFunc.func_code.mHandle != null) {
            if (pyFunc.func_code instanceof PyBaseCode) {
                PyBaseCode fcode = (PyBaseCode) pyFunc.func_code;
                if (!fcode.varkwargs && fcode.varargs &&
                    noOfArgs == fcode.co_argcount) {
                    MethodHandle target = getCachedMH(callSite,pyFunc);
                    target = target.bindTo(callSite);
                    target = target.asType(callSite.type());
                    callSite.setTarget(target);
                    return target.invokeWithArguments(args);
                }
            }
        }
    }
    return fallback(callSite, args);
}
```

Listing 3.17: Installing a function to the call site.

```
1  public static Object fallback(JythonCallsite callSite, Object[] args) {
2      MethodType fallbackType = callSite.type().dropParameterTypes(1, 2);
3      MethodHandle fb = callSite.lookup.findVirtual(
4          PyObject.class, "__call__", fallbackType);
5      MethodType desiredType = fb.type().
6          changeParameterType(0, ThreadState.class).
7          changeParameterType(1, PyObject.class);
8      MethodHandle fbWithPermute = MethodHandles.permuteArguments(
9          fb, desiredType,
10         getPermuteArray(desiredType.parameterCount()));
11     callSite.setTarget(fbWithPermute);
12     return fbWithPermute.invokeWithArguments(args);
13 }
```

Listing 3.18: Fallback code for function dispatch.

a `PyFunction` (numbering starts from 0 and the 0^{th} element is the thread state object) in line 3 of Listing 3.17. Then in line 6 we make sure that a method handle exists in the code object of the current `PyFunction`, and in lines 10-11 we make sure that the current call does not contain star arguments and double star arguments, and that the number of arguments passed in matches the expected number of arguments. In line 11, we fetch the method handle `target` (the caching mechanism is explained a bit later) associated with the current function object and in lines 12-14 re-bind the call site to point to the `target` method handle. In line 15 we invoke the `target` method handle with the appropriate arguments. Subsequent `invokedynamic` calls at this call-site will be directly dispatched to the method pointed by the method handle `target`, removing a lot of overhead.

If any of the conditions in checks from lines 3-10 fail in Listing 3.17, the `fallback` method is invoked. This method looks up the `__call__` method in `PyObject` and then calls it, mimicking the behavior of the previous approach. Listing 3.18 shows the outline for this method, in lines 3-5 we lookup the `__call__` method of the appropriate type in `PyObject`. The arguments are ordered so that thread state object is the 0^{th} parameter and the function object is the 1^{st} parameter, but we wish to dispatch this method handle with the function object in the receiver position (in the 0^{th} position). Lines 5-10 accomplish exactly this goal by permuting the arguments on the method

```
1  public static MethodHandle setupMHForCall(MethodHandle target) {
2      MethodType targetType = target.type();
3      MethodHandle before = MH_CALL_BEFORE.asCollector(PyObject[].class,
4          targetType.parameterCount() - LEADING_NON_ARGS);
5      MethodHandle targetWithBefore = foldArguments(target, before);
6      MethodHandle targetWithBandA = setupAfter(
7          targetWithBefore, MH_CALL_AFTER);
8      MethodHandle gwt = setupGuardWithTest(targetWithBandA,
9          target.type().parameterCount() - LEADING_NON_ARGS);
10     return gwt;
11 }
```

Listing 3.19: Code for setting up the method handle for call.

handle. This new method handle `fbWithPermute` is updated as the call-site's target and in line 12 this method handle is invoked with the appropriate arguments.

The method handle stored on a the code object points to the Java method which contains the implementation of the code. When the method handle is created, the `setupMHForCall` method is called to install these wrappers around the target method handle. The Listing 3.19 shows the outline of this method. In lines 3-4 the target method handle is wrapped to first call the `before` method handle using the `foldArguments` method. The `before` method handle points to a method that first creates a new frame object (of type `PyFrame`) and adds all the arguments to the frame. Line 6 creates a wrapper around the `targetWithBefore` method handle setting up a method handle to call a method to call after the previous method handle has been executed. This `after` method is responsible for updating the frame in case an exception was raised in the previous method. Finally, in line 8, the `setupMHForCall` creates a wrapper around the previous `targetWithBandA` guarding the call with a test method. The test method checks to see if the `PyFunction` object that was bound at the particular call site has changed. If it has changed, the fallback method is called which will re-bind the call site to the correct target. If the test returns true, then the target bound call site is invoked.

3.2.2.2 Changing the calling convention

Now, we will look at changing the calling convention so that instead of passing the arguments via the frame object, the arguments are also passed using the JVM operand stack. Jython maintained a uniform calling convention since the previous dispatch mechanism required it. Since we are now using method handles, which are typed and more powerful, it would be straight-forward to implement this change.

The Listing 3.21 shows the Java equivalent of the byte code generated by Jython after implementing this change. We see that the function `bar$2` now accepts a new parameter `n` of type `PyObject`. The code compiler is updated to generate Java methods with the first three arguments always being `ThreadState`, `PyFrame` and `PyObject`. The third argument is the function object itself. The return type remains same at `PyObject`. After the 3 standard parameters, the number of parameters equal to the argument count of the function (in Python) is added to the generated method. In our example the function `bar` had 1 argument (`n`), which is added to the method signature as seen in line 4 in Listing 3.21.

Recall that from Listing 3.14, previously `PyTableCode` was used to dispatch a given call to the `call_function` method, which invoked the appropriate method. We cannot use this approach directly anymore, since the expected function would have a different signature. Instead we make use of method handles to invoke the target directly instead of delegating to the `call_function` method. To do this, I created a new code object called `PyMethodCode`, the relevant parts of this class are shown in Listing 3.20.

The method handle `mHandle` is created along with the code object creation. The switch case in lines 8-18 dispatch to the target method handle depending on the number of arguments passed in. The third argument is always set to null, since the function object is not needed for this dispatch to work. Line 14 shows an example where if the number of arguments was one, as in the case in our example (Listing 3.11), the target is invoked using the method handle passing the appropriate parameters. The rest of the function dispatch stack remains largely unchanged.

```java
public class PyMethodCode extends PyBaseCode {

    @Override
    public PyObject call(ThreadState ts, PyFrame frame, PyObject
             closure, PyObject[] args) {
        PyObject ret;
        ...
        switch(args.length) {
      case 0:
        ret = (PyObject) mHandle.invokeExact(frame, ts,
                      (PyObject) null);
        break;
      case 1:
        ret = (PyObject) mHandle.invokeExact(frame, ts,
                      (PyObject) null, args[0]);
            break;
           ...
    }
        ...
        return ret;
    }
}
```

Listing 3.20: The new `PyMethodCode` replacing the old `PyTableCode`

```
PyInteger _10 = new PyInteger(10);

public PyObject bar$2(ThreadState ts, PyFrame f, PyObject func,
                      PyObject n) {
    return Py.None;
}

public PyObject foo$3(ThreadState ts, PyFrame f, PyObject func) {
    PyObject local = f.getName("bar");
    invokedynamic<bootstrap>(ts, local, _10);
    return Py.None;
}
```

Listing 3.21: Using the new dispatch mechanism coupled with the calling convention change for the example in Listing 3.11.

The bootstrap method the call to a generic install call site, which then binds the correct target loaded from the code object. The fallback method is also unchanged, but the fallback will invoke the `__call__` on `PyMethodCode` instead of the previous `PyTableCode`.

3.2.3 Performance evaluation

To evaluate the performance gains by this approach, we will use the classical fibonacci calculated as implemented in Listing 3.22. Figure 3.5 shows a bar chart where the performance of the fibonacci function from Listing 3.22 is compared by measuring the running time on three Python platforms: Python, Jython and a custom build of Jython with the dispatch mechanism implemented. The running time of Jython is used as the baseline, and we see that both stock Jython and CPython have similar running times. The new dispatch mechanism which uses `invokedynamic` enables the JIT in the JVM to better optimize the code at runtime giving a *2.32*x performance improvement.

3.3 Reducing frame overhead

Even though we see that in the previous performance experiment, the fibonacci function used `invokedynamic` to dispatch to the target call, the method handle was wrapped to call another

```
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Listing 3.22: An implementation of calculating the n^{th} number in the fibonacci series.

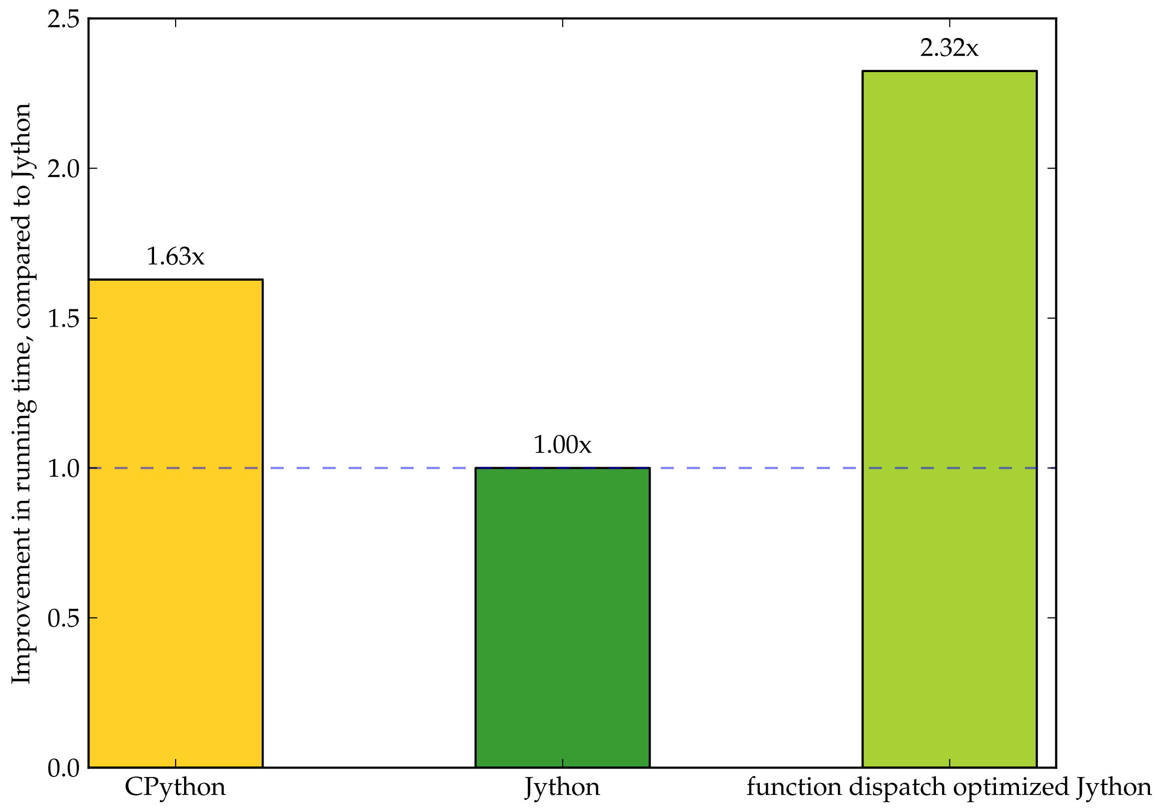

Figure 3.5: Comparing the relative performance running the fibonacci function from Listing 3.22.

```
>>> import sys
>>> def print_locals(a, b):
...     pi = 3.14
...     print(sys._getframe().f_locals)
...
>>> print_locals(4, 6)
{'a': 4, 'pi': 3.14, 'b': 6}
```

Listing 3.23: Accessing the local variables from the frame.

method created a frame and added all the arguments into the frame. The Python frame is essential for debugging purposes and allows for easy introspection of Python execution. A Python program can easily get to the current Python frame, as shown in Listing 3.23.

Creating a new Python frame object before each call hampers performance on the JVM. We have seen that, since we have changed the calling convention, we can use JVM's operand stack to pass arguments to functions instead of passing them via the Python frame. The globals are also moved out of the frame, therefore reducing its dependence on the frame. We can further reduce our dependency on the frame by using JVM's locals array to store local variables, instead of doing the same on a Python frame object. Although this is possible, we would still have to support Python introspection abilities such as the program in Listing 3.23 as we cannot break Python semantics.

To get around this issue, we have introduced a new function decorator `@frameless` to indicate that frames are not required for that particular function. This way the programmer is explicitly telling the compiler to not create frames since they know that, in the current scope, the frame need not be exposed to Python code.

To be able to mark a given function as frameless, the following are required:

- The function cannot rely on using the frame for introspection.
- The function cannot have inner (nested) functions, since the nested function would require us to construct a frame for accessing variables in the bounded scope.

If a Python function is annotated with `@frameless`, and if the scope analysis determines the function is a closure, it cannot be frameless, regardless of the decorator; a compilation error is raised in such a situation. The programmer should either remove the annotation or move the nested function.

We have already seen how we move global variables and parameters out of the frame. Now, we look into the details of the implementation of the not relying on Python frames local variables, by using JVM's locals array instead. We have All AST nodes are updated to now have a new boolean field `frameless`. Once a function is annotated with `@frameless`, the AST node for the corresponding function definition is updated setting `frameless` to **`true`**.

The scope analysis is updated to ensure that a frameless function definition node, does not contain another function (or class) definition in it's body. If it finds a nested definition, a compilation error is raised. The `OptimizedCodeCompiler` is updated to not use frames for loading and storing variables if the current scope is marked frameless. Variables are accessed via the `Name` AST node. If the current scope is frameless, and the variable is local, then the variable is either loaded form or stored in the JVM's array.

A new boolean field `frameless` is also created in the class `PyMethodCode`, the runtime code object. This boolean object is set to **`true`** if the function corresponding to code object is marked frameless. During the call to a function, if the `PyMethodCode` object is marked frameless, then the method handle wrapper is updated to bind a **`null`** object to the frame, instead of call the additional method which constructs the frame. This removes the overhead of an additional method call when dispatching to a frameless function.

Consider the fibonacci function annotated with the `@frameless` decorator on the function in Listing 3.24. In this case, there is only one parameter, `n`, and because of the frameless annotation, Jython does not use the Python frame object to load this parameter, instead uses the JVM operand stack, by which the argument was passed. Like the previous case, I performed a performance evaluation, comparing the running times for the fibonacci function in CPython, stock Jython and the frameless-supported Jython. In the first two cases I used the original function from Listing 3.22 where as in the third case, I used the function in Listing 3.24. Figure 3.6 charts the performance across the three implementations. We see that using frameless (and the previous `invokedynamic`

```
@frameless
def fibonacci(n):
    if n < 2:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)
```

Listing 3.24: A fibonacci function marked frameless.

optimizations) gives a *6*x performance speed-up. Compared to the previous function dispatch optimizations, we get a *2.6*x speed-up just by removing the frame access and frame creation.

3.4 Performance evaluation

In the previous sections we used micro-benchmarks to evaluate the performance of the implemented optimizations. Although micro-benchmarks are good way to assess improvements, they are not representative of the applications that run with Jython (or Python in general). In this section I will measure the overall impact of the optimizations using macro-benchmarks which are more representatives of typical Python applications. To measure the performance, I've selected benchmarks from the Unladen Swallow project [18] and the PyPy project[1] . I have selected 10 programs to test the performance, first I will briefly explain each benchmark program and then discuss the results.

call-simple is a small program that contains a large number of calls to a small number functions and stresses the function call overhead.

chaos is a small Python program simulating fractals like the chaos game.

crypto-pyaes is a program that uses the `pyaes` library – a cryptographic Python module implementing the AES algorithms – and tests encryption and decryption using 128-bit keys.

django uses the popular Django templating library to create a 150x150-cell HTML table.

fannkuch is a Python program, inspried from the famous LISP benchmark [1].

[1] The benchmark programs and the runner can be found at `http://bitbucket.org/shashank/benchmarks`

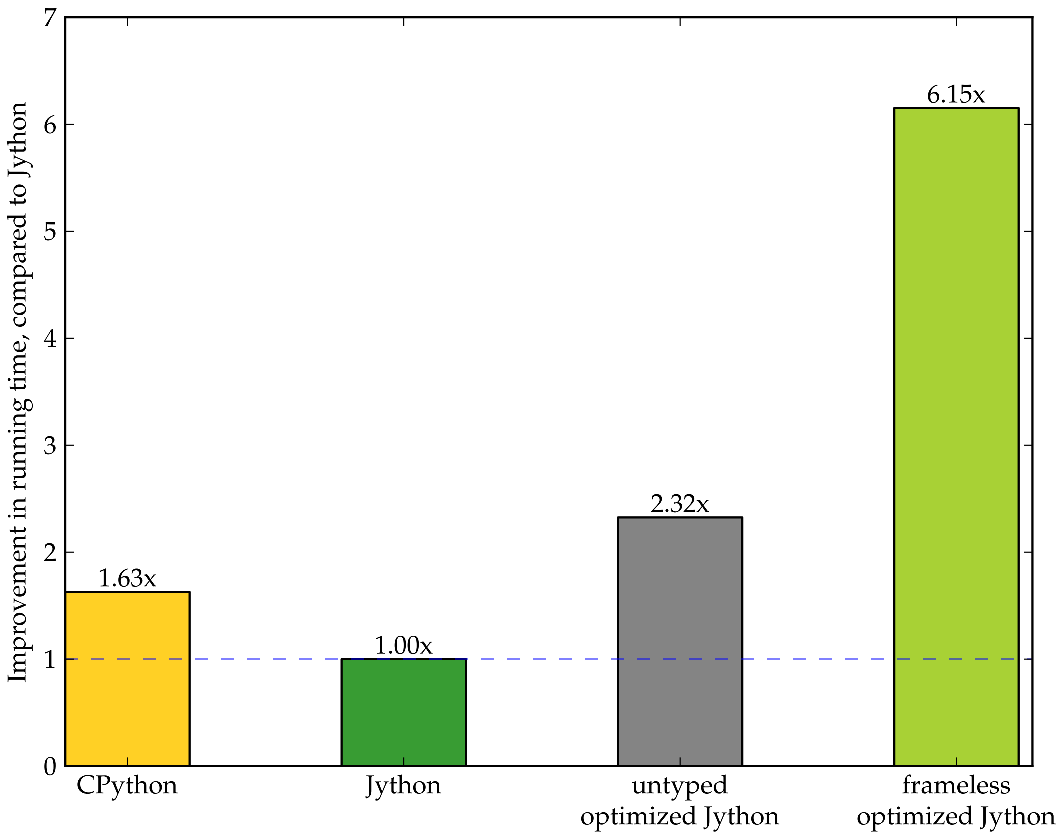

Figure 3.6: Comparing the performance running a frameless version from Listing 3.24.

pyflate-fast is a pure-Python gzip and bzip2 decoder/decompressor.

raytrace-simple – a Python raytracing renderer.

regex-effbot is a stress test for the Python's regular expression (regex) engine.

richards is a Python program that simulates the task dispatcher in the kernel of an operating system.

telco program captures the essence of a telephone company billing application including application calculations.

The performance measurements were run on the JVM from the 64-bit Oracle JDK SE 7 update 10 (b18). Each benchmark was run 150 times and the average of the result is reported here. Before collecting the results, the first few runs of the benchmark were ignored since this typically represents JVM warm-up time. Since the benchmarks are pure-Python and do not contain any programmer annotations, we do not measure the effect of the `@frameless` annotation from Section 3.3.

The figures in this section show the performance speedup achieved with the optimizations enabled and is expressed as a percentage improvement (or degradation) in running time over the original Jython code base (Jython without any of the above optimizations). First we will look at the impact of implementing the optimizations in isolation.

- The loop optimization introduced in Subsection 3.1.1, produces a 0.42% overall *decrease* in performance. The breakdown is shown in Figure 3.7. As we can see from the figure, some benchmarks improve in performance while other degrade, and in case of 2 benchmarks (django and telco) there is no difference in the performance.

- The `getglobal` optimization introduced in Subsection 3.1.2, increases the performance by 4.04%. The breakdown is in Figure 3.8. We observe that the benchmark call-simple increases in performance by 55% and the richard benchmark improves by 14%.

- The new function dispatch mechanism introduced in Section 3.2, *decreases* the performance of the overall test suite by 2.46%. The break down is shown in Figure 3.9, most applications performance is marginally decreased (0-3%) and the benchmark django decreases by 9%.

Now, we will look at the performance of these benchmarks with the `getGlobal` optimization and the new function dispatch combined. This produces an overall improvement in performance of 3.2%. The breakdown is shown in Figure 3.10. Although the overall number is lesser than just the `getGlobal` optimization (which is +4.04%), the results are a bit skewed because of the huge performance improvement provided by call-simple. If we just consider the mean performance of the rest 9 benchmarks without call-simple, then we get the following numbers:

- only `getGlobal`: -0.51%
- only function dispatch change: -2.32%
- both combined: *+0.12%*

This test shows that although the dispatch mechanism change degraded the performance of the overall suite, when combined with the `getGlobal` optimization we get a *slight* increase in the overall suite. Also, we suspect that some of the performance degradation is due to the overhead of using wrapped method handles, and we expect this performance to get better as the `invokedynamic` framework grows mature.

Finally, we investigate the performance of all 3 optimization enabled together. The 3 optimizations enabled together produce an overall performance increase of *5.7%*. The breakdown is presented in Figure 3.11. We see that these optimizations together produce a 31% speedup in the case of call-simple and on the other extreme makes the plyflate-fast benchmark run 5% slower. The overall increase in performance of 5.7% is more than any of the optimization individually.

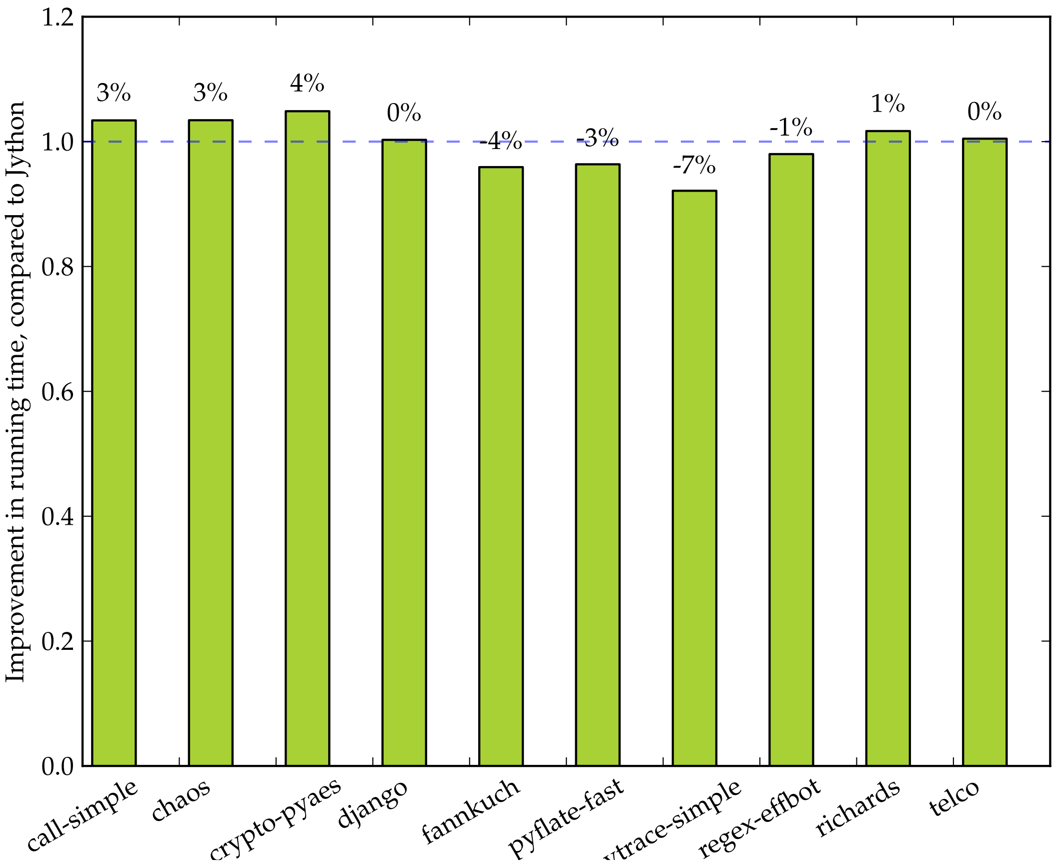

Figure 3.7: Comparing the performance of the `for` loop optimization implemented in Subsection 3.1.1. The geometric mean of the benchmarks is -0.42%

.

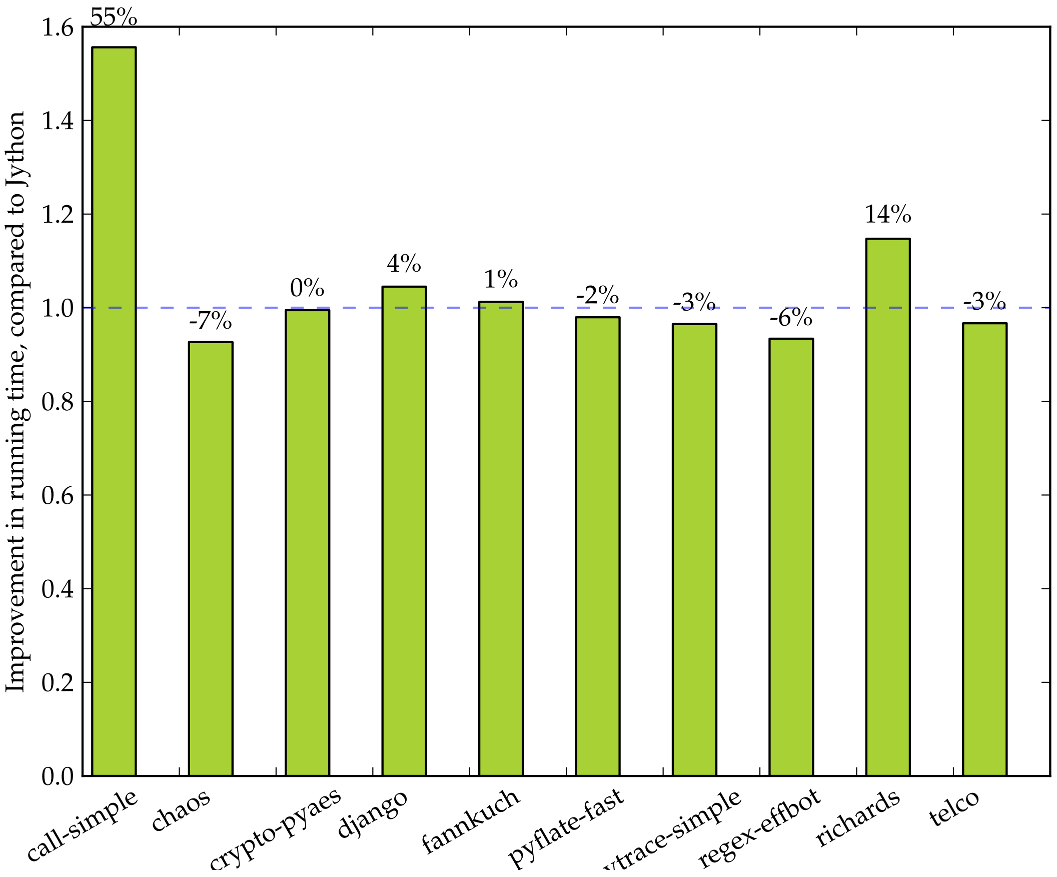

Figure 3.8: Comparing the performance of the `getGlobal` optimization implemented in Subsection 3.1.2. The geometric mean of the benchmarks is 4.04%

.

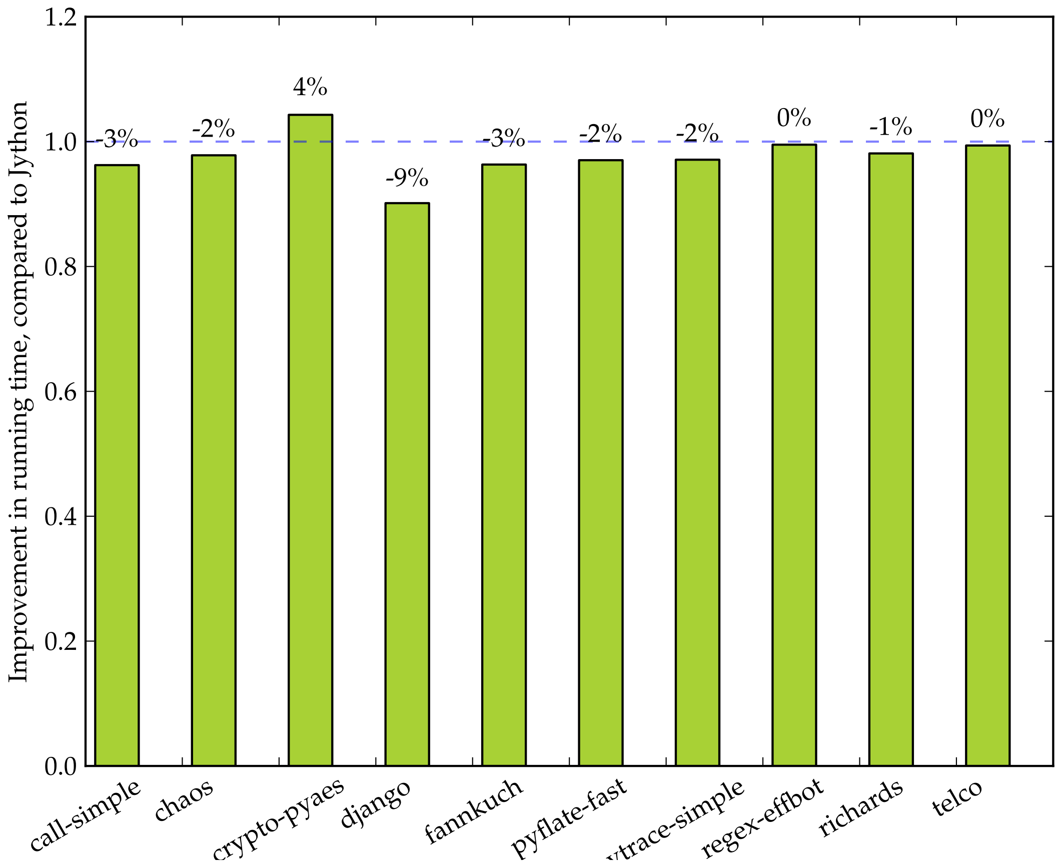

Figure 3.9: Comparing the performance of the change in dispatch mechanism implemented in Section 3.2. The geometric mean of the benchmarks is -2.46%

.

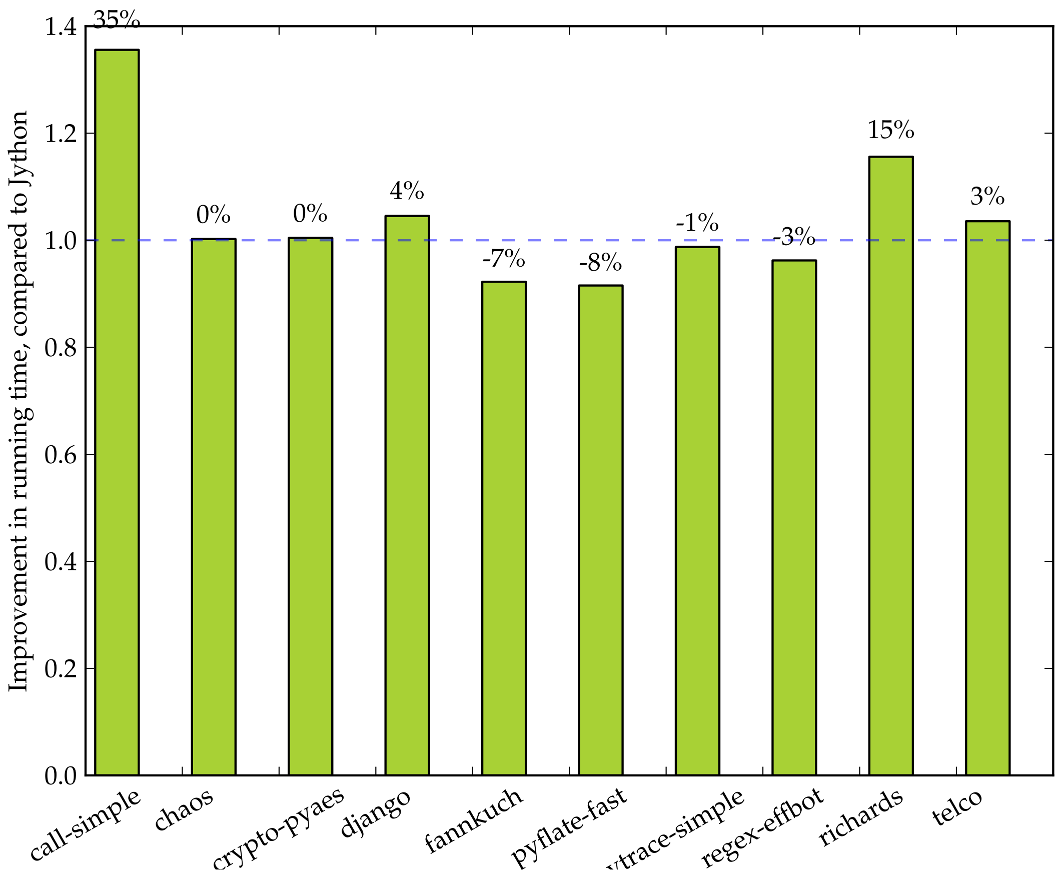

Figure 3.10: Comparing the performance of the dispatch mechanism change (from Section 3.2) and the `getGlobal` optimization (from Subsection 3.1.2) together. The geometric mean of the benchmarks is 3.2%

.

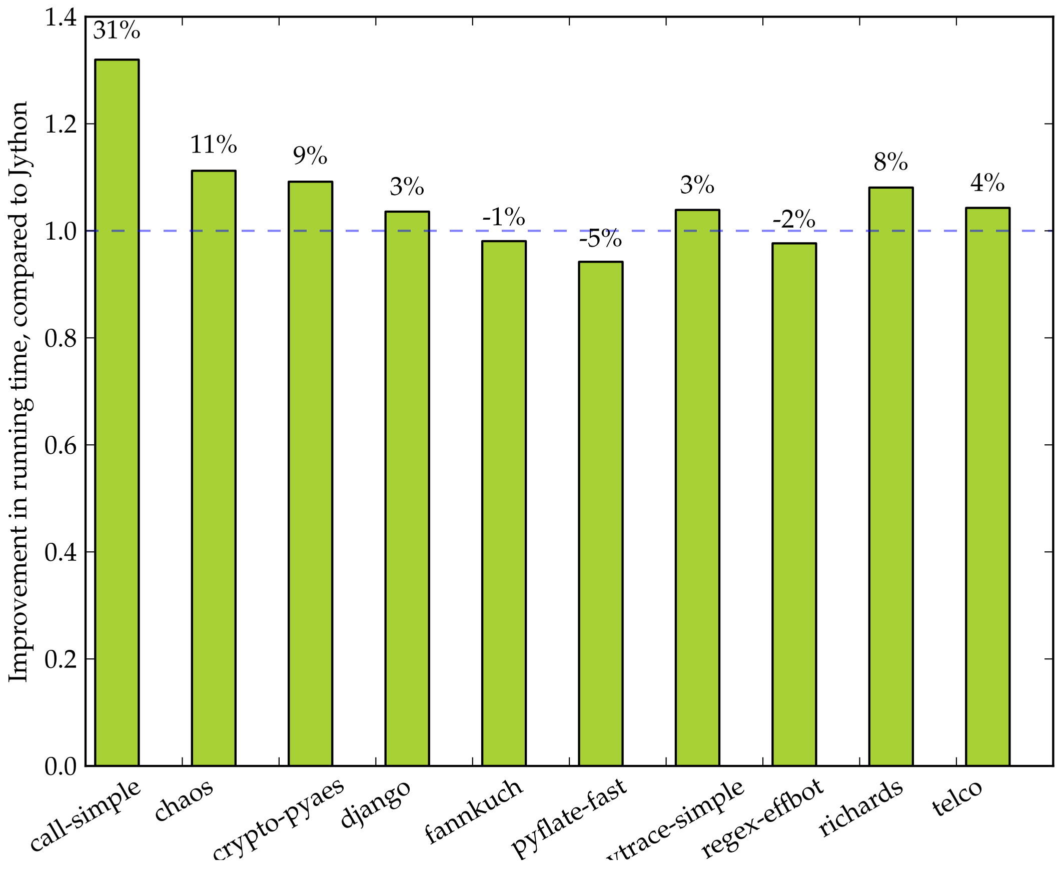

Figure 3.11: Comparing the performance of the optimizations implemented in this chapter using a few (macro) application benchmarks. The geometric mean of the benchmarks is 5.7%

.

Chapter 4

Gradual Typing for Jython

In the preceding chapter we improved the performance of Jython by generating optimized code while making sure that Python semantics are maintained, using the `invokedynamic` framework. But the runtime performance of Jython is severely limited since Jython does not use the primitive types provided by the JVM, but uses wrapped objects instead. For example a Python integer is represented as an object of class `PyInteger` in Jython. Simple operations such as addition of two Python integers is then implemented as a method in the `PyInteger` class to fully support Python semantics. The performance would greatly improve if we used the `iadd` operand in the JVM. In this chapter we will implement *gradual typing* in Jython not only to generate type-specialized code, but also provide clear error messages at API boundaries.

In this chapter I will discuss about how gradual typing can be implemented in Jython for a subset of the features in Python. I will show that using gradual typing and type-specialization, we can achieve performance comparable to Java. Before diving in too deep, I will present the design choices we made in implementing the gradual typing system in Section 4.1.

4.1 Design Choices

We expect a programmer to write completely un-typed (dynamic) code during the prototype phase. Once the prototype shows some promise, the programmer would then go back and add type-annotations both as documentation in the exposed APIs and also on the performance intensive functions of the code. One design principle we have followed is that Gradual Jython should always be backwards compatible with Jython maintaining all Python semantics. This implies that all un-typed code in Gradual Jython should work the same way as it does in Jython.

The restriction that Gradual Jython has to be backwards compatible implies that the Gradual Jython typechecker cannot reject any Python programs. Therefore something like: `'hi'+ 42` should not be rejected, since Python throws a runtime error in this case, and so should Gradual Jython. The Gradual Jython system can only reject programs if the module has type annotations in it.

In Gradual Jython, we have introduced types in the compiler. This means that names of Python built-in types such as `int`, `bool`, `float` and `str` always correspond to the type they represent. Re-assigning these builtin names is allowed, but does not affect the compilation, since the compiler only looks at the string representation of the type variable. Therefore, if the builtin type `float` is assigned to point to a user defined class, if the name is used in the type field in Gradual Jython; the typed-expression would still behave as the Python builtin float. This restriction is an important one when trying to generate type-specialized code. If the compiler cannot make assumptions about the types, then it would make it very hard to generate type-specialized code. In this thesis I have assumed that the built-in types cannot be re-defined. This does not break backwards compatibility, since the types can be re-defined in normal Python code.

In addition to supporting the following standard Python builtin types: `bool`, `int`, `float` and `str`; we also propose a new type `jint`. The new type – `jint` is specific to Gradual Jython. The reason for introducing `jint` is that standard integers in Python are allowed to dynamically grow into longs which is only limited by the amount of memory in the virtual machine. When performing performance optimizations, we would like to represent the standard integer as a native Java integer, which has a fixed length. If done this way, it would have an adverse affect, as every operation on an integer must be checked for overflow. To alleviate this problem we decided to provide a new type “`jint`” which represents the Java integer. A programmer can choose to use this type if they are trying to optimize their Python code for performance, and explicitly do not care about overflows (possible because they know it will never happen).

Another design choice is making typed-functions frameless automatically. This removes another burden on the programmer to add the `@frameless` annotation. One of the reasons we expect a programmer to add type annotations is for performance improvement. We have seen before, in Section 3.3, using the Java locals array instead of Python frames significantly increases the performance. Therefore, if a typed-function does not have nested function, the said typed-function is marked frameless by the Gradual Jython system. If the programmer decides that they need frames for this function; for debugging purposes for example, then they can annotate the function with `@frameful` annotation.

After having discussed the design choices made in our implementation of gradual typing in Jython, we can now dive into the details of how Gradual Jython was implemented. To implement gradual typing we need to accomplish the following:

- Extend the language syntax to provide type annotations. Jython currently only supports Python 2.5 syntax, that does not have function annotations. We also need to add type annotations for assignments to variables.

- Implement a type checker to make sure malformed programs are caught and descriptive type errors are thrown to help the programmer. The above two points are further elaborated in Section 4.2.

- Implement a cast-insertion phase which would insert casts in places where dynamic stuff is begin passed into statically typed portions of the program and vice-versa. This phase is presented in Section 4.3.

- Generate code to enforce these casts at runtime. Also, use the type-annotations to generate type-specialized-code to improve performance. This part is detailed in Section 4.4.

4.2 Parsing and type checking

The work presented in this thesis is based off Jython 2.5.2 which only supports Python 2.5 syntax. Although PEP-3107 [17] provides a syntax for function annotation, it was only introduced

```
1 def fib(n:jint) -> jint:
2     one: jint = 1
3     two: jint = 2
4     if n < two:
5         return n
6     return fib(n - one) + fib(n - two)
```

Listing 4.1: A function to calculate fibonacci with types.

in Python 3.x releases. To support gradual typing we need a way to specify types on functions as well as assign types to variables. Therefore we extended the parser to support specifying type variables in assignment statements. The syntax we use for specifying types is:

```
answer: int = 42
```

where the variable `answer` is assigned the constant `42` of type `int`. The important thing to note here is that this is an experimental syntax, only used here to show how gradual typing can be introduced to a dynamic language. The discussion of a concrete syntax for annotating variables is controversial and is out of the scope of this thesis. We also implemented the function annotation syntax of PEP-3017, so that we can specify types for parameters and return values of functions. The parser and the type checker extended and implemented by Michael Vitousek.

To easily illustrate each step in the introduction of gradual typing in Jython, I will use a running example. We will use a modified version of the fibonacci function we examined in the Chapter 3, as presented in Listing 4.1. Here the first line annotates the function, with marking the parameter n with `jint` and the return type of this function as `jint`. In lines 2-3 two `jint` variables are created and lines 4-6 is performs the fibonacci operation just as before.

The AST is first traversed and checked with the type checking rules. Each node is also assigned a type. The root of all AST nodes in Jython is a class named `PythonTree` and we added a field `internalType` to it. In the type checking phase, this type is updated after first processing the node. This means that after the type checking phase we get back an AST in which each node is associated with a type. For un-typed nodes, this would be dynamic. The type checker rejects malformed programs by throwing a `TypeError`. Since we made a choice to allow all completely un-typed Python programs to pass through the type-checker in order to be backwards compatible, we do *not* perform any type-inference. Even constants such as `42, 'hi'` are treated as dynamic. This ensures that all plain-Python (un-typed) programs type check without raising type errors.

We have implemented gradual typing for a subset of Python, without dealing with objects. All objects are treated as dynamic. The only types recognized are the basic types and function types. The type checker implementation is closely matched by the rules in Figure 2.2.

4.3 Cast Insertion and type specialization

After the type checking phase we pass the AST to the "Explicate" pass which inserts casts and marks code for type-specialization. The cast insertion implementation closely follows the theory and inserting casts as directed. The current AST is given as input to the cast insertion phase, which returns a modified AST inserting casts and type-specialized nodes in the appropriate places.

The cast insertion phase is guided by the gradual typing rules in Subsection 2.1.1, more specifically in Figure 2.3. The important idea is that casts must to be inserted when dynamically-typed part of the code transitions to statically-typed part. In our implementation of gradual typing, where we want to generate type-specialized code, we also need to insert casts if the code transitions from statically-typed to dynamically-typed. This is because the statically-typed portion could potentially be type-specialized, implying that we could use primitive types of the JVM. These casts would allow us to convert from such type specialized code back to a subclass of `PyObject` which represents the dynamic type.

Casts are represented by the new `InsertCast` node. The node is implemented as a Python expression containing the source (`from`) and target (`to`) types of the cast and the expression undergoing the cast (`rest`). The internal type of the node itself is set to the target (`to`) type. If a given Python expression and it's children do no have the same internal type, then a cast is inserted between them, casting the child node into the type of the parent expression.

It is possible that sometimes identity casts are introduced, casting a `int` to another `int`. To remove such casts, the AST is passed through another passed called "Constantify" pass, which removes redundant identity casts. Currently, we do not perform any type-specialization for the Python types `str` and `int`. The Jython runtime objects `PyString` and `PyInteger` are used to implement them. Thus casts from and to these types are removed in the “Constantify” pass as well.

Now we will look at the type-specialization part of the explicate pass. When we identify a node with a type other than dynamic, we replace the current node with a typed node. The code compiler is then updated to generate type specialized code for these nodes. We have created the following typed nodes: TypedBinOp, TypedCompare, TypedName, TypedReturn, TypedCall, TypedUnaryOp. Each of them extend their corresponding untyped nodes. If we identify that the child nodes of these nodes are typed, then we replace them with a Typed variant.

For example, if both the operands of a BinOp (left and right), are both typed of the same type, say `jint` then we generate a TypedBinOp node of type `jint` replacing the previous node. If only one of the nodes (left or right in case of binop) has a type other than dynamic and the other is dynamic, then we insert a InsertCast node marking the point where a cast has to be inserted, but we do not generate a Typed node.

4.4 Code generation

In this section I will describe how we generate specialized code for typed AST nodes and also look at how we handled typed and casted functions, leveraging the `invokedynamic` framework. The code generation is detailed in two parts, first we show how to generate code for the Gradual Jython types and second we show how to generate type-specialized code for functions.

The AST is passed to the `TypedCodeCompiler` which handles the type specialized code generation. The `TypedCodeCompiler` extends the previous `OptimizedCodeCompiler` and handles all the newly introduced Typed nodes and the `InsertCast` node. The basic outline for code generation is unchanged from the previous code compiler. At each node, we first visit the children of the current node, generating code for each of them and in the end the code for the current node is generated.

```python
import math

def std(x: jint, y: jint, z: jint) -> jint:
    three: jint = 3
    avg: jint = (x + y + z)/three
    e1: jint = x - avg
    e2: jint = y - avg
    e3: jint = z - avg
    return math.sqrt((e1*e1 + e2*e2 + e3*e3)/three)

print std(1, 2, 5)
```

Listing 4.2: A typed function computing the standard deviation among 3 numbers.

4.4.1 Standard Gradual Jython types

As mentioned previously, we support the following types: `bool`, `int`, `jint`, `float` and `str`. The nodes "Constantify" pass will remove casts containing `str` and `int` and these types fallback to using the previous implementation. The code generation phase currently only specializes `jint`, `bool` and `float`. We use the JVM primitive type **`int`** to represent a Gradual Jython type `jint`; types `bool` and `float` are represented using **`boolean`** and **`double`** in the JVM respectively.

Let us look at an example of generating code for a TypedName node. If a TypedName node has the type `jint`, then generate code such as `iload`, `istore` for the load and store operations respectively. For a TypedBinOp node, and for the add operation on type `jint`, we generate an JVM `iadd` instruction.

Consider the a function to compute the standard deviation of 3 numbers such as the one in Listing 4.2. Here all the variables are marked `jint`. Type specialized Gradual Jython would generate Java byte code similar to the Java code presented in Listing 4.3.

The InsertCast node for the standard types use helper methods in Jython runtime. If there exists a cast from a dynamic object to a `jint`, the virtual method on the object with signature **`int`** `PyObject.asInt()` is invoked. This is a method defined on `PyObject` throws a runtime error if invoked on a non-number. For a cast from an object of type `jint` to a dynamic or even a `int`, we make use of the helper function: `PyObject Py.newInteger(`**`int`** `i)`, defined in the

```
public PyObject std(PyFrame f, ThreadState ts, PyObject func,
                    int x, int y, int z) {
    int three = 3;
    int avg = (x + y + z)/three;
    int e1 = x - avg;
    int e2 = y - avg;
    int e3 = z - avg;
    PyObject t = Py.newInteger((e1*e1 + e2*e2 + e3*e3)/three);
    PyObject sqrt = invokedynamic<getGlobal>(this, "math")
        .__getattr__("sqrt");
    return invokedynamic<call>(ts,  sqrt, t);
}
```

Listing 4.3: The type specialized code generated by Jython for the typed-Python program in Listing 4.2.

```
1  public PyObject fibonacci$1(PyFrame frame, ThreadState ts, int n) {
2      int one = 1;
3      int two = 2;
4      if (n < two) {
5          return n;
6      }
7      PyObject res =
8          invokedynamic<call>(
9              invokedynamic<getGlobalBSM>("fibonacci"), ts, n - one) +
10         invokedynamic<call>(
11             invokedynamic<getGlobalBSM>("fibonacci"), ts, n - two);
12     return res;
13 }
```

Listing 4.4: The java code corresponding to typed-byte code generated by Jython.

class `Py.java`. This method never fails since the input is statically guaranteed to be an integer. We implement similar casts on the other types `bool` and `float` which are represented using JVM's **`boolean`** and **`double`** primitive types.

4.4.2 Functions

The function type determines the corresponding method signature in Gradual Jython. Consider the example shown in Listing 4.1, since the parameter type of `n` is `jint`, the function signature in Java byte code for the corresponding parameter should be integer. A Java equivalent of the output from the Gradual Jython compiler is shown in Listing 4.4. We see that in line 1, the parameter is indeed of type **`int`**. The compiling of such typed functions follows the new compilation introduced in Section 3.2. We generate the `invokedynamic` instruction to dispatch to the correct callsite. If the target method uses Java primitives types, the same is reflected in the method handle that points there.

After reviewing the compilation of typed-function calls, let us now examine how function casts are implemented. The InsertCast node on a function indicates that a function needs to be cast into a new type. To support casting, we have to ensure that the arguments and return values match the expected type. To help with casting, each runtime function code object (`PyCode` in

```
1 def identity(n):
2     return n
3
4 typed_id: (jint->jint) = identity
```

Listing 4.5: An example of a function cast.

Jython) now has a new field `type` which stores the threesome type of the function.

The concept of threesomes was introduced by Siek and Wadler [12] as an efficient way of storing runtime cast information. Each threesome has three attributes: source type, target type and middle type. The source type represents the type of the original object, the target type represents the type of the cast to which the object has to be casted to. The middle type represents the most restrictive type that the object has been cast to. The authors show that all the information in a series of casts can be condensed into threesome with these three types. The middle type is computed by merging the middle types of the incoming types. The merge operator computes the greatest lower bound on the type.

The threesome object is implemented as a simple class with source, middle and target types. Consider the example Gradual Jython program shown in Listing 4.5. Here, the function `identity` is of type dynamic, and in line 3 `typed_id` has cast the function into taking and returning `jint`s.

In the Gradual Jython implementation, the function code object corresponding to `identity` , will have `type` as dynamic. When the function is cast (line 3) into `jint->jint`, a new copy of the original object is created. In this copy, the `type` threesome is updated: source type would be the source of the original; target type would be the type of the cast and the middle type of threesome would be the merged type of the original middle type (dynamic) and the cast. Then the method handle on the copy object is updated to ensure the correct parameter and return types. This copy is then stored as the variable `typed_id`. Creating a copy is essential, since the original function `identity` could still be called with any object.

To update the method handle on the cast object, we make use of the method handle framework, specifically `MethodHandles.filterReturnValue` and `MethodHandles.filterArguments` to convert the return value and the arguments to the expected type. As before, we use the helper functions such as `int PyObject.asInt()` and `Py.newInteger(int i)` on individual arguments to covert them. Each code object now stores the original method handle, in a new field `originalHandle` which points to the method handle without any casts applied. The method handle used for dispatching (`mHandle`) would be the same as `originalHandle` if there are no casts on the function. If casts exist, at the point of the cast, we update the `mHandle` with the conversion and this method handle will be used to dispatch to the corresponding code. By storing the original method handle, we avoid multiple conversions even if multiple casts exist on a function.

4.4.3 Evaluation

We will now look at the performance impact of generating type specialized code. We will use the example from Listing 4.1. Figure 4.1 shows the performance comparison chart. We see that the type specialized Jython code is *35*x faster than Jython without type annotation. This version is only about 60% slower than a similar fibonacci method implemented in pure Java. The gray bar (Untyped optimized Jython) shows the performance of Jython with all the optimizations developed in the previous chapter applied to the untyped fibonacci function. We observe that adding types and generating type-specialized code results in approximately a 15x speedup.

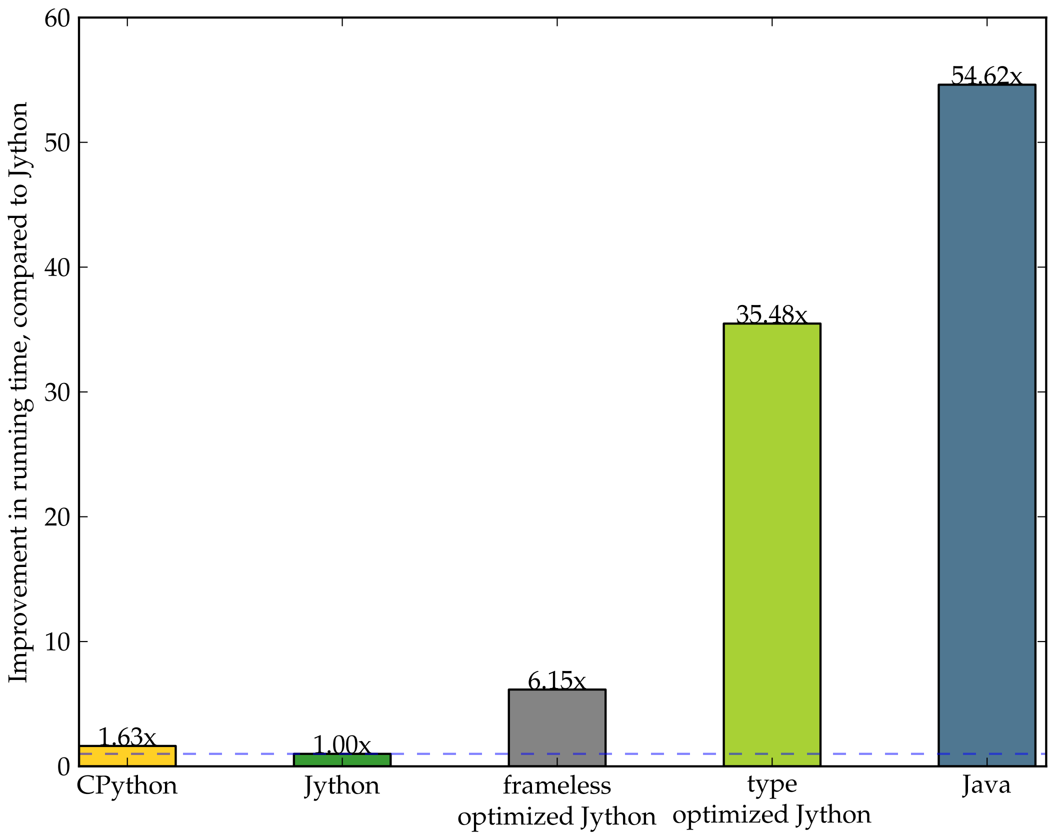

Figure 4.1: Comparing the relative performance running the typed fibonacci function.

Chapter 5

Conclusions and Future Work

Python programs are hard to optimize given their dynamic nature. In this thesis we have used the `invokedynamic` byte code introduced in Java 7 to generate optimized code in Jython and saw that it significantly improved performance, providing upto *15*x speed ups in some examples. We proposed that using a type system where the programmer can choose to provide optional type annotations on the program, can provide significant performance improvement. We implemented gradual typing in Jython and saw that the optimized code generated performed an order of magnitude better than before, and only *60%* slower than Java. The code for this project is available on bitbucket at `https://bitbucket.org/shashank/jython-gradual`. All the optimizations implemented in this thesis will be pushed upstream to Jython.

In the future, we can add support for object types in Gradual Jython which have been ignored in this thesis. Generating type specialized objects could not only lead to better performance but also provide better Java integration in Jython. Adding blame tracking to Gradual Jython would help improve the error messages, providing developers clear error messages when casts fail.

Bibliography

[1] KR Anderson and Duane Rettig. Performing Lisp analysis of the FANNKUCH benchmark. ACM SIGPLAN Lisp Pointers, 1994.

[2] Stefan Brunthaler. Efficient inline caching without dynamic translation. In Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10, page 2155, New York, New York, USA, March 2010. ACM Press.

[3] C. Chambers and D. Ungar. Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. ACM SIGPLAN Notices, 24(7):146–160, July 1989.

[4] Rémi Forax. JSR-292 Cookbook, 2012.

[5] 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, volume 29, pages 326–336, New York, New York, USA, August 1994. ACM Press.

[6] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java virtual machine specification (Java SE 7 Edition). 2012.

[7] Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. 2004.

[8] JR Rose. Bytecodes meet combinators: invokedynamic on the JVM. Proceedings of the Third Workshop on Virtual ..., 2009.

[9] J Siek and Walid Taha. Gradual typing for objects. ECOOP 2007Object-Oriented Programming, 2007.

[10] Jeremy Siek, Ronald Garcia, and Walid Taha. Exploring the design space of higher-order casts. Programming Languages and Systems, 2009.

[11] Jeremy G Siek and Walid Taha. Gradual Typing for Functional Languages. IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP, pages 81—-92, 2006.

[12] Jeremy G. Siek and Philip Wadler. Threesomes, with and without blame. Proceedings for the 1st workshop on Script to Program Evolution - STOP '09, pages 34–46, 2009.

[13] Andreas Stuchlik and Stefan Hanenberg. Static vs. dynamic type systems: an empirical study about the relationship between type casts and development time. . . . of the 7th symposium on Dynamic languages, pages 97–106, 2011.

[14] TIOBE Software BV. TIOBE Programming Community Index for October 2012, 2012.

[15] Robert Tolksdorf. Programming languages for the Java Virtual Machine JVM, 2009.

[16] Guido van Rossum. The Python Language Reference. Python Software Foundation, 2012.

[17] Collin Winter and Tony Lownds. Function Annotations. Python Enhancement Proposal - 3107, 2006.

[18] Collin Winter, Jeffrey Yasskin, and Et Al. Unladen Swallow. http://code.google.com/p/unladen-swallow/.