A JAVA-BASED LANGUAGE
WITH TYPE-SAFE DYNAMIC CODE GENERATION
型安全な動的コード生成をサポートする
Java の言語拡張

by

Yutaka Oiwa
大岩 寛

A Master Thesis
修士論文

Submitted to
the Graduate School of Science
the University of Tokyo
on 6–7 February 2001
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Information Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

In this thesis I describe an extension to the Java language that supports type-safe dynamic code generation.

Dynamic code generation is a powerful technique for reducing execution time of programs. It composes a portion of the execution code dynamically at runtime, generally mixing with some runtime-information. Since generated code holds runtime values such as number of an iteration or a condition of a branch as constants, it may run faster than statically-generated code.

However, current dynamic code generators have several disadvantages. First, as most dynamic code generator are designed to generate native machine code, programs using dynamic code generation are not portable. Second, most systems have weakness on high-level language support for describing dynamic code fragment. Especially, safety of dynamic composition of code fragments is not sufficiently supported. Though it is partly solved in functional languages, it is not directly applicable to imperative settings. Many Java-based systems provide functions which generate code dynamically by directly manipulating bytecode. On those systems, safety of the generated code is the user's responsibility. Some systems allow the user to write a code fragment using high-level language constructs such as C and thus ensure type-safety within the fragment of code, but type-safety of more than one code fragments that are composed at runtime are not automatically guaranteed.

To solve these problems, I define a strongly-typed language for dynamic code as an extension to Java language. This language gives precise types to dynamic code fragments. Our notion of types includes not only the types of code fragments of the base language, but also information of conditions that guarantee type safety of their composition. Type-checking these information confirms statically that result of composition of dynamic code fragments is still type-safe.

I also present an implementation of this extended language in this thesis. The runtime system is implemented on Java virtual machine. As most Java virtual machines equip Just-in-time (JIT) compilers that translate machine-independent intermediate bytecode to native machine code, the system achieves both execution speed and portability at the same time.

論文要旨

本論文では型安全な動的コード生成をオブジェクト指向に基づいてサポートする Java 言語の拡張について述べる。

動的コード生成はプログラムの実行時間を短縮するための有用な方法であり、プログラムの実行時に動的にプログラムの一部を構築し、通常はその際に実行時に得られた情報をその中に含める。生成されたコードはループの実行回数や分岐の条件などの実行時に得られる情報を定数として埋め込んだものとなり、従って静的に生成されたコードよりも高速に実行することができる可能性が高い。

しかしながら、現存する動的コード生成器には数々の問題がある。まず初めに、ほとんどの動的コード生成器は実マシンコードを生成するため、動的コード生成器を用いたプログラムは移植性が悪くなることが多い。次に、動的コード片に対する高レベル言語によるサポートは一般に弱い。とりわけ、実行時のコード片の合成に関する安全性は十分に考慮されていない。この問題は関数型言語ではある程度解決されているが、その解決法を逐次的言語にそのまま適用するのは困難である。Java 言語上で実行時のコード生成をサポートするライブラリのほとんどは、低レベルで複雑なバイトコードの操作しかサポートしない。これらのライブラリを用いる際、生成されたコードが安全に実行可能であることはライブラリ利用者の責任である。また、C 言語のような高級言語を用いてコード片を記述することで、1 つのコード片の中での型の安全性を保証することのできるシステムも存在するが、その場合でもコード片を組み合わせた際のコード片相互間での型安全性の保証は自動的にはなされない。

これらの問題点を解決するため、本論文では動的コードの記述のための強く型付けされた言語を、Java 言語の拡張として定義する。この言語は動的コード片に対して詳細な型付けを与える。この動的コード片の型は動的コード片を評価した時の結果の型だけでなく、コード片の結合の際に満たしているべき制約の情報を含む。これらの詳細な情報に対する型検証を行なうことで、動的コード片を結合した結果もまた正しいコードになっていることを静的に保証することが可能となる。

また、この拡張された言語に対する実装も本論文中で示す。本言語の実行時システムは Java 仮想マシンのバイトコード言語上で実装されている。多くの Java 仮想マシンはマシンに依存しない中間バイトコードを実マシンコードに変換する効率の良い実行時コンパイラ (JIT コンパイラ) を備えているので、本実行システムは実行効率と移植性の双方を両立させることができる。

Acknowledgments

I express my gratitude to my research supervisor Prof. Akinori Yonezawa. He always takes a kindly interest on me and provides me a good research environment.

I am very thankful to Dr. Hidehiko Masuhara. He gave me many pieces of invaluable advise on this study. Without his help, this thesis could not exist here.

I thank Dr. Kenjiro Taura very much. His stern but tender attitude to me reproaches my laziness and makes me encouraged toward studies.

I also thank Dr. Naoki Kobayashi and members of “Principle of Programming Language” research group in Yonezawa and Kobayashi laboratories. Discussion in the meetings made my knowledge about the semantics and type systems of computer languages deeper and made inspiration of the type system of this research.

Finally I express my appreciation to all members of Yonezawa Laboratory, delightful members of Italk and all my families. They all support me on research, daily life and refreshment. Their kindness made it possible to complete this research. Thanks!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Dynamic code generation (DCG) is a technique that generates and executes fragments of executable code during the run-time of a program. Usually, programs handles three types of data: compile-time constants, runtime constants, and dynamic data. Compile-time constants are the values which are already known at compile-time. Usually, values of this kinds are written directly into program code and are subject to optimization performed by compilers. Values of dynamic data changes during program runs, and usually not subject to optimization. Runtime constant are the middle of those two values: their value is not known at the compile time, but once its value is determined at the very early stage of computation, it does not varies during rest of computation.

Runtime constants are usually not targets of optimization: it cannot be optimized at compilation time, obviously. However, if the program repeatedly makes some decisions (ex. target of branches) depending on those constants, it uses lot of execution time wastfully to make such predictable decisions. If dynamic code generation is used, a program which is specialized to that constant values can be generated. The generated program does not require any time to make a predictable decision. There is plenty of values which are subject to runtime code optimizations: number of iterations, user-given instructions to manipulate data, automata decision table for regular expressions, and more.

There is three ways to implement DCG. The first is to directly write down a program which generates machine instructions on a memory. However, writing a correct program in this approach is extremely difficult and error prone. The second is to automatically generate a program that generates an optimized version of a given generic program–so called *(run-time) partial evaluation*. As this approach relies on static analysis to determine where to optimize, resulted optimization would be too conservative. The third is to define a language in which the programmer can write a fragment of dynamic code as an expression in a high-level language. This approach can solve the problems of former two approaches. DynJava is based on this approach.

The DynJava is an extended language to Java in which the user can write dynamic code fragments in the syntax that is similar to the Java's. In addition, dynamic code fragments in DynJava are statically typed, where type safety of dynamically composed code is statically guaranteed.

Another generic problem with DCG is portability and efficiency. Usually, runtime code generators have to generate native machine code which are to execute. This means that programmers must write a machine-specific code for each platform on which the program needs to run. Use of intermediate language solves the portability problem, but it requires lot of additional computation time to interpret them.From other point of view, the intermediate code of program is a typical example of runtime constant.

DynJava solves this problem by using Java virtual machin (JVM) Java virtual machine is an interpreter of an intermediate language called bytecode, but many implementations of JVM equips JIT compiler which translates bytecode to efficient native language at the first time the code is called. As JVM and JIT compiler becomes ubiquitous recently, the DynJava system enjoys both portability and efficiency.

I devised a type system for DynJava, and implemented a system that translates DynJava program to a Java program.

1.2 Related work

There are many tools which manipulates bytecode, for example JavaClass API [4], and `gnu.bytecode` package included in Kawa Scheme [1], and they can be directly used to generate execution code dynamically on Java Virtual Machine (JVM). However, as those tools treat a dynamically generated program as a stream of untyped instructions, the user could generate type-unsafe code. Keeping safety of the generated code completely owe to the user's responsibility, and generally it is very hard to maintain.

'C [13] is an extension to C language which supports writing dynamic code in the syntax of C language. In 'C, user can write a fragments of dynamic code and combine them to generate a function in C language at runtime. 'C has become a basic idea of the this research. Especially `-syntax and code-generator approach comes from 'C. However, as 'C does not support any context, type safety of the generated code is still not guaranteed enough. For example, two code fragments `‘{return 5;}` and `‘{return "string";}` cannot co-exist in one function obviously, but as these two specifications are both typed “`void cspec`” in ‘C, the inconsistency is not detected. Our DynJava can detects those inconsistency at compilation time using context information. In addition to this, in ‘C users must write a special construct explicitly to use variables or labels across two or more code fragments, and must maintain the consistency of them carefully. DynJava provides easier way to share one variable between two specifications than ‘C does.

‘C maintains some extent of portability using VCODE and ICODE libraries, which supports dynamic generation of machine code for several RISC platforms [6]. Our implementation using Java virtual machine which runs on more numbers of platform. Code generators using VCODE runs very fast, but generated codes are not efficient and not fast. Generators using ICODE generates better code, but generator itself is slower than one using VCODE. Current implementations of JIT compiler are still slower than ICODE, but it may improve further.

Kawa Scheme [1] is an interpreter and compiler of Scheme language implemented on Java virtual machine. Kawa Scheme supports not only interpretive execution of the program and static compilation but also runtime compilation of lambda closures to reduce runtime costs for interpretation. It can be seen as another example which shows the effectiveness of dynamic code generation. However, as it compiles only special forms (syntactic structres like `if` and `case`), and as it uses boxed representation for all values, code generated by Kawa compiler is still slower than Java codes and dynamic codes generated by DynJava.

Another approach to generate code dynamically in a type-safe way is a runtime specialization technique, that uses program analysis. For example, [11] describes about runtime program specialization on Java bytecode, and many related publications on this topic exist. Since this approach extracts dynamic code fragments from a given single-level program, specialized program is always type-safe. However, degree of optimizations (specialization) depends on the preciseness of the analysis. If the target program is simple, full-automatic analysis is sufficient produce a good result. However, if the program gets complicated, the program author will have better knowledge about program’s property and where to optimize than the automatic analyzer. In this case, our DynJava will become a powerful tool to implement program-dependent runtime optimization easily by hand.

There are studies on of type safety of dynamic program composition. Modal-ML [14] is one of such studies on the functional language ML. Due to simple syntax and semantics, restriction on a correct context for a dynamic code fragments can be easily checked by matching types of all free variables. However, in imperative languages such as Java, context should have more precise information to determine the correctness, because a program fragment depends not only on the types of free variables, but also various information such as labels and exceptions. Our type system is extended to handle these properties and ensures the correctness of composition in an imperative language.

MobileML [9] defines type system for dynamically-bound code fragments on the ML language, using a notion of context. The base idea of context type checking in DynJava is inspired by Mobile-ML. However, its context notion binds only variables, because of the same reason as Modal-ML.

1.3 Outline of this thesis

This thesis is organized as follows. In Chapter 2, I present a overview of DynJava Language. In Chapter 3 the type-system of DynJava, which is the principal part of the language, is described. Chapter 4 explains the implementation of DynJava compiler, and Chapter 5 evaluates the performance of it through experiments. The whole thesis is concluded in Chapter 6.

Chapter 2

Language Overview

The DynJava language adds, in order to generate new classes, a few constructs for combining several parts of dynamic code fragments together. In this section, I present a brief overview of DynJava.

2.1 Code fragments and contexts

In DynJava, user can write dynamic code fragments as an *code specifications*, which has the syntax which is similar to `cspec` in ‘C [13]. They are combined to generate an anonymous subclass of some abstract class. To check the type-safety of dynamic code compositions described in Chapter 3, the code specification’s type includes its context information.

A context information of a code specification is very precise in DynJava. It includes following information:

Name of the base class (superclass) As in Java language all program code is enclosed in methods which is members of some classes, there is always a “current class”, and its superclass. In the instance method, methods and fields of the current class is accessible through simple name, and `this` can be used as an value of those classes. We call this “context (or environment) is an *instance context*.” In the class method, only class methods and class fields are accessible through simple name, and `this` cannot be used. We call this “*class context*”.

Because the current class of dynamically-generated code is anonymous in DynJava, an instance of dynamically-generated class is used as those of superclass, the definition of which is statically available. We call this "base class". Context holds the name of the superclass, with a flag whether the context is class context.

The base class of an instance context is notated as "`extends` *classname*",[1] and those of an class context as "`static extends` *classname*".

If no base class is specified by user, it is assumed to "`static extends java.lang.Object`", because code fragment which only depends on an static field of the `Object` class can be appear in any method of any class.

Variables and fields of current class *Variable context* holds types of all variables which is visible in the current scope, like usual environment. In this type system, variable context also holds all fields which are accessible by simple name, and by `this.`*name* notation.

The entry of the simple name in the variable context is notated as "*type name*" (ex. "`int i`" or "`Vector v`"). The entry of the field of `this` is notated as "*type* `this.`*name*".

Note that if some name is available through `this`, the name is always accessible through simple name. However, type of the *name* and `this.`*name* may be different, as local variable may hide instance fields and class fields.

In a program does not use any dynamic code specification, the information of fields is simply derivable from the name of current class. The type-checker automatically consults the definition of current class and expands the fields to the variable context (see "canonical context expansion" described later). The reason that the language defines explicit notation of the field in the variable context is to allow programmer to add an fields to the current class without defining some dummy class.

Methods and Constructors *Method context* holds the list of methods and constructors which are defined in or inherited by the current class. For each method a context holds 1) the name of the method, 2) type of the return value, 3) types of the formal parameters, and 4) list of exception types which may be thrown by the method. For each constructors, context holds the $1'$) where the constructor is defined, either in current class or in direct superclass, and 3) and 4).

Method entries in method context are notated as "$type_r$ *name*`(`$type_1$`,` $type_2$`,` ...`)` `throws` exn_1`,` exn_2`,` ...", where $type_r$ is the return type, $type_1, type_2, \ldots$ are the

[1]Keywords used in notations of contexts are chosen from the reserved keywords of the Java language, to prevent name conflicts.

types of the formal parameters, and exn_1, exn_2, ... are the types of the exceptions which may be thrown. For example, `clone` method defined in the `Object` class in the JDK is notated as "`Object clone() throws CloneNotSupportedException`". For constructor entries, return types are omitted and either `super` or `this` are used in place of *name*.

As same as variable context entry, entry of methods and constructors which are defined in the superclass are expanded automatically from the baseclass information by the type checker.

Type of return value The type of the return value obviously depends on the declared type of surrounding method. In one context, only one type can be used for the parameter of `return` statement. This information is notated as "`return` *t*". The context in which only `return` statements without argument can be used is notated "`return void`".

Context is allowed to contain no return type specification. Code specification with such contexts can not contain any "`return`" statement in it, but can be used in the places whatever return type is required.

`break`**-able Labels** *Label context* holds all labels which is available for use with `break` or `continue` statement. `continue`-able label is provided by `while` and `for` statements, and `break`-able label is provided by those and `switch` statements. In addition to this, those statements without label is treated to be providing null-label. `continue`-able label with tag *t* is notated as `continue` *t*, and `break`-able label of those as `break` *t*. Null-labels are notated simply as `continue` and `break`.

Throwable exceptions At each point in the programs, *exception context* holds a set of exception types which are handled by either method caller or try-catch clause. For example, at the point of star (⋆) in the following program flagment, exceptions of the class `java.io.IOException`, `ClassNotFoundException` and of any subclass of those class can be thrown.

```
void method_1(int x) throws IOException {
    try { ⋆ }
    catch (ClassNotFoundException e) { ... }
}
```

In this type system, this information is notated as "`throws IOException`", "`throws ClassNotFoundException`".

Other information of context There are two miscellaneous states which context holds. First, "constructor state" means that the current block is used as a body of a constructor, and that some constructor of the current class or of direct superclass must be explicitly invoked at the top of current block. This state is notated as "`super`". Second, "switch state", notated as "`switch`", means that the current block is used as a body of a `switch` statement, and the special labels `case` *i*`:` and `default` can be used.

For example, the context specification `<int x>` specifies contexts where `x` is bound to a `int` value, and `<return int>` specifies that contexts are methods that return `int` values.

2.2 Code specifications

DynJava have two kinds of code specifications, *statement specifications* and *expression specifications*. Each of them corresponds to Java's statement or expression, respectively. The type of a statement specification is written as `code_spec`<Γ>, where Γ is the context on which the specification depends. The type of an expression specification is written as t `exp_spec`<Γ>, where t is the type of values generated by evaluating a generated code fragment from the specification. The type t is called a target type of the expression statement type.

A code specification begins with a backquote (`` ` ``), followed by a context specification and either a statement or an expression. Statement specifications have the form "`` ` ``<*context*>`{`*body*`}`", and expression specifications have the form "`` ` ``*type*<*context*>`(`*body*`)`".

Code specifications can have free (unbound) variables, if they are declared in the context specification. The following is the examples for code specifications.

```
`<String x>{ System.out.println("hello, " + x + "!"); }
`double<int x>(x + 1.5)
```

When a code specification is used in a program where a value of some specific code specification type is required, its context specification and target type can be omitted. For example, in the program below, the type of the statement specification in line 1 should matches to the type of the left-hand side of the assignment, which is `code_spec<int x;`

`return int>`. Similarly, the type of the expression specification in line 2 is deduced to `double exp_spec<int x>`.

```
1  code_spec<int x; return int> c1 = ‘{ return x; };
2  double exp_spec<int x> c2 = ‘(x + 1.5);
```

The places where (and how) the type of the specifications can be deduced are following:

1. Right-hand side of assignments (type deduced from left-hand side)
2. Initializers in variable declarations (from the declared type)
3. An argument of `return` (from the return type of enclosing method)
4. Inside `? :` expressions, where above rules applies for the type of `? :` expressions (from the deduced type of the expressions)

2.3 Embedding another context

In the body of a code specifications, another code specification can be embedded by writing `@` followed by an identifier. The code generated from inner code specification is inlined into the code from outer code specification. For example, in the code below, compiling `c2` will generate the almost same code as one generated from `c3`.

```
code_spec<String g, x> c1 =
     ‘{ System.out.println(g + x + "!"); };

code_spec<String g; return void> c2 =
     ‘{ String x = "Michael"; @c1; return; };

code_spec<String g; return void> c3 =
     ‘{
        String x = "Michael";
        System.out.println(g + x + "!");
        return;
      };
```

When a code specification is embedded by `@`, the context specification of inner specification is always checked against surrounding code and the context of outer specification, to ensure that the composition is correct. In the above example, c1 requires that variables g and x must be bound to type String. The code surrounding @c1 in c2 provides binding of x. g is not bound by c2 itself, but it requires outer context to bind g. Therefore, type-checking above code succeeds.

In addition to variables, labels (or break points) can also be "free". In DynJava, anonymous break point, which is provided by loop constructs without label, is treated as "null label". Break and continue statements may point to labels which are bound outside current code specifications. In the following program, break in c1 escape the for loop in c2.

```
code_spec<break; int x> c1 =
     ‘{ if (x == 5) break; }

code_spec<> c2 =
     ‘{ for(int x = 0; x < 10; x++) {
             System.out.print(" " + x);
             @c1;
        }
      };
```

2.4 Embedding constant primitive values

Primitive values can also be embedded (or "lift"ed), by using $-prefix.

```
String message = "hello";
code_spec<String x> c1 =
     ‘{ System.out.println($message + ", " + x + "!"); };
code_spec<return void> c2 =
     ‘{ String x = "Michael"; @c1; return; };
      // prints "hello, Michael!" and escape current method
```

The expression $message in above program embeds runtime value of the variable message in to the code specification c1. The values which can be embedded by the $-expression are limited to primitive values and strings. This reflects a limitation of Java language and Java virtual machine.

2.5 Class specifications

In DynJava, code specifications must be compiled into class to use.

In order to generate a class, DynJava provides class specification constructs which begins with keyword `class_spec`. construct. A class specification looks like a the class definition that lacks bodies of methods, but its instance acts as a “generator of new classes”. The code below is a small example of class specifications.

```
 1  // "interface"
 2  abstract class Method { abstract void invoke(); }
 3
 4  class_spec MethodGen extends Method {
 5    void invoke(); // this class overrides invoke() in class Method
 6  }
 7
 8  public class Test {
 9    public static void main(String[] args) {
10      MethodGen cs = new MethodGen();
11      cs.<void invoke()> = ‘{ System.out.println("Hello"); };
12
13      cs.compile(); // generates class
14
15      Method m = new cs(); // generates instance
16      m.invoke();
17    }
18  }
```

In the class specifications there is a field for each declared methods. These fields has appropriate types assigned by the type checker. For example, `cs.<void invoke()>` has given the type `code_spec<extends Method; return void; void invoke()>`.

To define the actual body of the methods, user assigns code specifications to the fields of class generator i.e. an instance of a class defined by `class_spec`). The fields of the class generators are indicated by an extended syntax, *e*`.`<*method signature*>, which appears in line 11 above.

To generate an instance of dynamically-generated class, extended form of `new` expression, which takes a class generator rather than class name as an argument is used (see line 16). It returns a reference to new instance, which is typed to the base type declared in the class specification declaration. If the code specification is not compiled explicitly, it is automatically compiled at the first call to `new`.

In addition to above syntax, DynJava allows to create an anonymous class generator inside method. If the keyword `class_spec` is used without class name, and with the variable declaration after declaration body, It generates an instance of an anonymous class specification directly. Above example can be rewritten using anonymous class specification as follows:

```
// "interface"
abstract class Method { abstract void invoke(); }

public class Test {
  public static void main(String[] args) {
    class_spec extends Method { void invoke(); } cs;

    cs.<void invoke()> = ‘{ System.out.println("Hello"); };

    // cs.compile(); // can be omitted
    Method m = new cs(); // generates class and instance
    m.invoke();
  }
}
```

2.6 Syntactic sugar

As a notation of context specification is usually long, and same context specification appear many time in programs, DynJava provides syntactic sugar for it. If the context specification contains term like `@e`, the context specification contained in the type of `e` is "quoted". For example, in the program shown in the previous section, the type notation `code_spec<@(cs.<void invoke();>)>` represents the type of the code specification in the example. Context specifications can also be extended after quotation, like `< @e; int x, y; >`.

Chapter 3

Type System

In this chapter, I describe the typing system of the DynJava. As described in the previous chapter, DynJava type-checks each code specification using its context information so that the dynamically composed codes preserves type safety.

A type judgment $\Gamma; \Delta \vdash e : t$ determine an expression or statements e has type t under current environment Γ and *outer* environment Δ.

3.1 Definitions

Firstly, I define some operators which handles various entities appears in both original Java language and DynJava.

Definition 3.1 (Subtype and type compatibility relation) *subtype relation* $\prec$ *and type compatibility relation* $\rightsquigarrow$ *on primitive and reference types are defined as follows:*

$$\frac{t' \textit{ is a subclass of } t}{t' \prec t}$$

$$\frac{}{\texttt{byte} \rightsquigarrow \texttt{int}} \qquad \frac{}{\texttt{short} \rightsquigarrow \texttt{int}} \cdots$$

$$\frac{t' \prec t}{t' \rightsquigarrow t} \qquad \frac{t' \textit{ implements interface } t}{t' \rightsquigarrow t}$$

See the specification of Java language [8] for other rules on primitive types.

Definition 3.2 (Class Description) $\mathrm{defs}(c)$, *called an description of class c, is the set containing following entries which are defined in the class c in Java language.*

1. *an instance field "$t\ \ x$", where t is the type and x is the name of field.*

2. *a class (static) field "*`static` $t\ \ x$*".*

3. *instance method "$t_r\ \ n$(${\tilde{t}_p}\ \ {\tilde{n}_p}$) `throws` $\tilde{e}$", where t_r is a return type, n is a name of method, $\tilde{t}_p$ is a list of argument types, $\tilde{n}_p$ is a list of argument names, and $\{\tilde{e}\}$ is a set of thrown exception. Actually, $\tilde{t}_p$ and $\tilde{t}_n$ are interleaved in the notation.*

4. *static method "*`static` $t_r\ \ n$(${\tilde{t}_p}\ \ {\tilde{n}_p}$) `throws` $\tilde{e}$*", where t_r, n, $\tilde{t}_p$, $\tilde{t}_n$, $\{\tilde{e}\}$ is as same as instance methods.*

5. *constructors "*`this(`$\tilde{t}_p$`) throws` $\tilde{e}$*", where t_r, $\tilde{t}_p$, $\tilde{t}_n$, and $\tilde{e}$ is as same as methods.*

The inheritance description of class c, notated $\mathrm{defs}_I(c)$ *is the same but the entries declared* `private` *is not contained.*

Definition 3.3 (Context Type) *Context Γ is a list of the following elements:*

1. *base-class and class context specification "*`extends` t*" or "*`static extends` t*", where t is a class type.*

2. *variable binding "$t\ \ x$" or "t* `this.`x*", where t is a type and x is an identifier.*

3. *instance method binding "$t_r\ \ n$(${\tilde{t}_p}$) `throws` $\tilde{e}$", where t_r is a type, n is a identifier, $\tilde{t}_p$ is a list of types, and $\{\tilde{e}\}$ is a set of types which are subtype of* `java.lang.Exception`.

4. *static method binding "*`static` $t_r\ \ n$(${\tilde{t}_p}$) `throws` $\tilde{e}$*", where t_r is a type, n is a identifier, $\tilde{t}_p$ is a list of types, and $\{\tilde{e}\}$ is a set of types which are subtype of* `java.lang.Exception`.

5. *constructor binding "n(${\tilde{t}_p}$) `throws` $\tilde{e}$", where n is either a* `this` *or* `super`*, and t_r, $\tilde{t}_p$, and $\tilde{e}$ meet same constraints as those of method binding.*

6. *label binding, one of "*`break`*", "*`continue`*", "*`break` l*" or "*`continue` l*", where l is an identifier.*

7. *exception binding "*`throws` e*", where $e \prec$* `java.lang.Throwable`.

8. *constructor context specification "*`super`*".*

9. *switch context specification "*`switch`*".*

It is an error if a context contains:

1. *two base-class specification is in it.*
2. *two methods or constructors with same name and same argument types, but the return type differs.*
3. *two variable bindings of form* `this.`*x with same name, but the type does not match.*
4. *both* `super` *and* `switch` *in it.*
5. `super` *and no base-class specification of the form "*`extends` *t".*

In addition to this, a special context $\circ$*, which means a context is unavailable, is defined. All operations on contexts defined below are not defined on* $\circ$*.*

A context with error does appear only in user-given context or the result of canonical context expansion from user-given context. These context with error is rejected by the compiler.

Definition 3.4 (Exception context and label context) *Exception context* exns(Γ) *is the set of all exceptions whose corresponding exception binding is in* Γ*. Label context lbls*(Γ) *is the set of all label binding contained in* Γ*.*

$$\mathrm{exns}(\Gamma) = \{e \mid \texttt{throws}\ e \in \Gamma\}$$
$$\mathrm{lbls}(\Gamma) = \{b \in \Gamma \mid b \textit{ is a label binding}\}$$

Definition 3.5 (Base class, staticness of the context) *The base class and staticness of the context is defined as follows:*

$$(\text{is-instance}(\Gamma), \text{base}(\Gamma)) = \begin{cases} (\text{true}, t) & \textit{if } \text{“}\texttt{extends}\ t\text{”} \in \Gamma \\ (\text{false}, t) & \textit{if } \text{“}\texttt{static extends}\ t\text{”} \in \Gamma \\ (\text{false}, \texttt{java.lang.Object}) & \textit{otherwise} \end{cases}$$

The context Γ with is-instance(Γ) = true is called an instance context, and others are called an class context.

Definition 3.6 (Switch state removing *) *Switch state removing operator is defined as follows:*

$$\Gamma^* := \Gamma \setminus \{\texttt{switch}\}$$

In Java language, case labels `case i:` and `default:` must be appear directly inside `switch` statement. For example, label `case 1:` in the following program

```
switch(x) {
 while(true) {
   case 1: ...
 }
}
```

is invalid. This operator removes the declaration `switch`, which means these labels are allowed, from the context and forces the correct placement of the case labels. Labels used for `break` and `continue` has no limitation like case labels, and therefore not removed by this operator.

Definition 3.7 (Variable reference $\Gamma(x)$) *The type of the variable x in context Γ is defined as follows:*

$$\frac{\Gamma = \Gamma', t\ \ x}{\Gamma(x) = t}$$

$$\frac{\Gamma = \Gamma', t\ \ \texttt{this.}x}{\Gamma(\texttt{this.}x) = t}$$

$$\frac{\Gamma = \Gamma', s \qquad s \text{ does not match rules above} \qquad \Gamma'(x) = t}{\Gamma(x) = t}$$

In other word, $\Gamma(x)$ is the type in the variable binding of x which appears last in Γ.

Definition 3.8 (Exception handle relation $\in_E$) *An exception class t is handled by the set of exception classes T if $t \in_E T$ defined below is satisfied:*

$$\frac{t \prec \texttt{java.lang.Error}}{t \in_E T} \qquad \frac{t \prec \texttt{java.lang.RuntimeException}}{t \in_E T}$$

$$\frac{\exists u \in T.\ t \prec u}{t \in_E T}$$

In Java Language, exceptions of type t can be thrown if 1) t is a subclass of class `Error` or `RuntimeException`, or 2) some supertype of t is handled by the outer syntactic block or method caller. The operator cares about those rules and makes the typing rules simpler.

Definition 3.9 (Exception subset relation $\subset_E$) *Set of exceptions T' is a subset of T if the condition*

$$T' \subset_E T \iff \forall t' \in T'.\ t' \in_E T$$

is met.

Definition 3.10 (Label provision $\in_L$) *Label request r is provided by label context L if $l \in_L L$ defined below is met.*

$$\frac{r \in L}{r \in_L L}$$

$$\frac{\texttt{continue}\ \ l \in_L L}{\texttt{break}\ \ l \in_L L}$$

$$\frac{\texttt{break}\ \ l \in_L L}{\texttt{break} \in_L L} \qquad \frac{\texttt{continue}\ \ l \in_L L}{\texttt{continue} \in_L L}$$

When `continue` can be used at some point of program, `break` can also be used at that point. And also, when `break` with some label is allowed, `break` without label is also allowed (although branch target of these two `break`s may be different). The definition above reflects those properties.

Definition 3.11 (Canonical context expansion) *Canonical context of context Γ is computed from Γ by the following procedure.*

1. *Assume $D = \mathrm{defs}_I(\mathrm{base}(\Gamma))$.*
2. *For all instance field specification "t $\texttt{this.}x$" in Γ, add variable binding "$t\ \ x$" if Γ does not contains any variable binding of x.*
3. *For all static field "$\texttt{static}\ \ t\ \ x$" $\in D$, add variable binding "$t\ \ x$" if Γ does not contains any variable binding of x.*
4. *For all static methods "$\texttt{static}\ \ t_r\ \ n(\tilde{t}_p\ \ \tilde{n}_p)\ \ \texttt{throws}\ \ \tilde{e}$" $\in D$, add "$\texttt{static}\ \ t_r\ \ n(\tilde{t}_p)\ \ \texttt{throws}\ \ \tilde{e}$" to Γ.*
5. *If Γ is an instance context,*
   (a) *For all instance field "$t\ \ x$" $\in D$, add variable binding "$t\ \ x$" if Γ does not contains any variable binding of x.*
   (b) *For all instance field "$t\ \ x$" $\in D$, add variable binding "t $\texttt{this.}x$".*
   (c) *For all instance methods "$t_r\ \ n(\tilde{t}_p\ \ \tilde{t}_n)\ \ \texttt{throws}\ \ \tilde{e} \in D$, add "$t_r\ \ n(\tilde{t}_p)\ \ \texttt{throws}\ \ \tilde{e}$" to Γ.*
6. *If $\texttt{super} \in \Gamma$, for all instance constructors "$\texttt{this}(\tilde{t}_p\ \ \tilde{n}_p)\ \ \texttt{throws}\ \ \tilde{e}$" $\in D$, add "$\texttt{super}(\tilde{t}_p)\ \ \texttt{throws}\ \ \tilde{e}$" to Γ.*

7. *Duplicated entries are removed as it is useless.*

Whenever the user specifies the context, it is expanded into canonical context.

Canonical context is the context which contains all information about fields and methods which are to be inherited from the base class. A user-given context is converted into canonical context by above rule. In all following rules and definition, canonicalness of the context is assumed. Variable bindings without `this` is only added when other binding is not exist, to allow hiding field by other variables.

Definition 3.12 (Context subtype relation $\prec_C$) *A context Γ' is sub-context of a context Γ iff all of following conditions are met:*

1. *(Base class)* $\text{base}(\Gamma') \succ \text{base}(\Gamma)$
2. *(Type of context)* $\text{is-instance}(\Gamma') \Rightarrow \text{is-instance}(\Gamma)$
3. *(Variables)* "$t\ x$" $\in \Gamma' \Rightarrow \Gamma'(x) = \Gamma(x)$
4. *(Instance fields)* "$t\ \texttt{this.}x$" $\in \Gamma' \Rightarrow \Gamma'(\texttt{this.}x) = \Gamma(\texttt{this.}x)$
5. *(Return Types)* $\texttt{return}\ t \in \Gamma' \Rightarrow \texttt{return}\ t \in \Gamma$
6. *(Methods)* "$t_r\ n(\tilde{t}_p)\ \texttt{throws}\ \tilde{e}'$" $\in \Gamma' \Rightarrow \exists\{\tilde{e}\}.$ ("$t_r\ n(\tilde{t}_p)\ \texttt{throws}\ \tilde{e}$" $\in \Gamma \wedge \{\tilde{e}\} \subset_E \{\tilde{e}'\}$)
7. *(Constructors)* "$n(\tilde{t}_p)\ \texttt{throws}\ \tilde{e}'$" $\in \Gamma' \Rightarrow \exists\{\tilde{e}\}.$ ("$n(\tilde{t}_p)\ \texttt{throws}\ \tilde{e}$" $\in \Gamma \wedge \{\tilde{e}\} \subset_E \{\tilde{e}'\}$)
8. *(Exceptions)* $\text{exns}(\Gamma') \subset_E \text{exns}(\Gamma)$
9. *(Labels)* $\forall e \in \text{lbls}(\Gamma').\ e \in_L \Gamma$
10. *(Constructor context)* $\texttt{super} \in \Gamma' \Leftrightarrow \texttt{super} \in \Gamma$
11. *(Switch context)* $\texttt{switch} \in \Gamma' \Rightarrow \texttt{switch} \in \Gamma$

This relation is written as $\Gamma' \prec_C \Gamma$.

This rule defines the compatibility relation between contexts and is the more core part of the type system. $\Gamma' \prec_C \Gamma$ means that a fragment of code depending on context Γ' can be used in place where context Γ is provided.

First, I describe sub-rules 1 and 2. If a fragment uses `this` as a value of type t, it must be available and its type must actually be a subtype of t. Else, if a fragment does not use `this` at all (i.e. Γ' is a class context), and depends on static method which are defined in t, then its actual base-class must actually be a subtype of t, too. The rules implements those restrictions.

Second, sub-rules 3 and 4 are almost obvious. However, in 3, I must notice that t and $\Gamma'(x)$ may not be equal, as some binding in Γ' may be hidden by another binding.

Next, sub-rule 5 constraints the type of return values if Γ' depends on a type of return values. If Γ' does not contain specifications of return type, Γ may contain any return type, because the fragment depends on Γ' may not contain any `return` statement.

If type-checker compares two method specifications with sub-rules 6 and 7, the type-checker does not require that exceptions thrown by those methods are strictly equal. Instead, if the actual implementation (depends on Γ) throws only a subset of exceptions of the declared set (in Γ'), it is acceptable.

Finally, two context must match strictly on whether it is constructor context or not, as the constructors are permitted and required to be called only in the top of the constructor body. Other sub-rules seems obvious.

3.2 Variables, expressions and simple statements

$$\frac{\Gamma(x) = t}{\Gamma; \Delta \vdash x : t} \qquad \text{(TR-VarRef)}$$

$$\frac{\Gamma; \Delta \vdash x : t \qquad t \leadsto t'}{\Gamma; \Delta \vdash x : t'} \qquad \text{(TR-Coerce)}$$

$$\frac{\Gamma; \Delta \vdash x : t \qquad t \leadsto t' \lor t' \leadsto t}{\Gamma; \Delta \vdash (t')x : t'} \quad \frac{\Gamma; \Delta \vdash x : t \quad t \text{ is not final} \quad t' \text{ is interface}}{\Gamma; \Delta \vdash (t')x : t'} \qquad \text{(TR-Cast)}$$

These rules are a part of typing rules of original Java language, expressed by DynJava's typing notation. Both DynJava and Java have many typing rules for operators such as `+`, `-`, `? :`, etc., but they are omitted here.

$$\frac{\Gamma^*; \Delta \vdash e : \textit{expression}}{\Gamma; \Delta \vdash e\texttt{;} : \textit{statement}} \quad \text{(TR-ExpStmt)}$$

$$\frac{\Gamma^*; \Delta \vdash s : \textit{statement} \qquad \Gamma; \Delta \vdash e : \texttt{boolean}}{\Gamma; \Delta \vdash \texttt{if (}e\texttt{)}\ s : \textit{statement}} \quad \text{(TR-If-1)}$$

$$\frac{\Gamma^*; \Delta \vdash s_1, s_2 : \textit{statement} \qquad \Gamma; \Delta \vdash e : \texttt{boolean}}{\Gamma; \Delta \vdash \texttt{if (}e\texttt{)}\ s_1\ \texttt{else}\ s_2 : \textit{statement}} \quad \text{(TR-If-2)}$$

These rules are also a part of typing rules of original Java language,

$$\frac{\texttt{super} \notin \Gamma}{\Gamma; \Delta \vdash \varepsilon : \textit{statement list}} \quad \text{(TR-StmtListNil)}$$

$$\frac{\texttt{super} \notin \Gamma \qquad \Gamma; \Delta \vdash l : \textit{statement list} \qquad \Gamma; \Delta \vdash s : \textit{statement}}{\Gamma; \Delta \vdash s\texttt{;}\ \ l \vdash \textit{statement list}} \quad \text{(TR-StmtList)}$$

$$\frac{\texttt{super} \notin \Gamma \qquad \Gamma^*, t\ x; \Delta \vdash l : \textit{statement list}}{\Gamma; \Delta \vdash t\ x; l : \textit{statement list}} \quad \text{(TR-VarBind)}$$

$$\frac{\begin{array}{c}\texttt{super} \in \Gamma \qquad \Gamma; \Delta \vdash s : \textit{constructor call} \\ \Gamma \setminus \{\texttt{super}\}; \Delta \vdash l : \textit{statement list}\end{array}}{\Gamma; \Delta \vdash s\texttt{;}\ \ l : \textit{statement list}} \quad \text{(TR-ConstrBody-1)}$$

$$\frac{\begin{array}{c}\texttt{super} \in \Gamma \qquad \texttt{super()} \in \Gamma \\ \Gamma \setminus \{\texttt{super}\}; \Delta \vdash l : \textit{statement list}\end{array}}{\Gamma; \Delta \vdash l : \textit{statement list}} \quad \text{(TR-ConstrBody-2)}$$

$$\frac{\Gamma^*; \Delta \vdash l : \textit{statement list}}{\Gamma; \Delta \vdash \{l\} : \textit{statement}} \quad \text{(TR-Block)}$$

These rules are typing rule for blocks. The rule (TR-VarBind) introduces new variable binding into current context. If a current context is a constructor context, it must call one of constructors at the top of method body (TR-ConstrBody-1), or superclass of the current class must have a constructor without argument (TR-ConstrBody-2). The conditions `super` $\notin \Gamma$ and `super` $\in \Gamma$ forces this restriction.

3.3 Exceptions

$$\frac{\Gamma; \Delta \vdash e : t \qquad t \in_E \text{exns}(\Gamma)}{\Gamma; \Delta \vdash \texttt{throw}\ e\texttt{;} : \textit{statement}} \quad \text{(TR-Throw)}$$

An exception can only be thrown, if some superclass of the exception is included in the current exception context.

$$\frac{\Gamma, t\ e; \Delta \vdash b : \textit{statement list}}{\Gamma; \Delta \vdash \texttt{catch(}t\ \ e\texttt{)}\ \ \texttt{\{}b\texttt{\}} : \textit{catch\_clause}\langle t\rangle} \qquad \text{(TR-Catch)}$$

$$\frac{\Gamma; \vdash b : \textit{statement list}}{\Gamma; \Delta \vdash \texttt{finally}\ \ \texttt{\{}b\texttt{\}} : \textit{finally\_clause}} \qquad \text{(TR-Finally)}$$

$$\frac{\Gamma, \texttt{throws}\ \ t_1, \ldots, \texttt{throws}\ \ t_n; \Delta \vdash b : \textit{statement list} \qquad \Gamma; \Delta \vdash c_i : \textit{catch\_clause}\langle t_i\rangle}{\Gamma; \Delta \vdash \texttt{try}\ \ \texttt{\{}b\texttt{\}}\ \ c_1 \cdots c_n : \textit{statement}} \qquad \text{(TR-Try-C)}$$

$$\frac{\Gamma; \Delta \vdash b : \textit{statement list} \qquad \Gamma; \Delta \vdash f : \textit{finally\_clause}}{\Gamma; \Delta \vdash \texttt{try}\ \ \texttt{\{}b\texttt{\}}\ \ f : \textit{statement}} \qquad \text{(TR-Try-F)}$$

$$\frac{\begin{array}{c}\Gamma, \texttt{throws}\ \ t_1, \ldots, \texttt{throws}\ \ t_n; \Delta \vdash b : \textit{statement list} \\ \forall i \in \{1 \ldots n\}.\ \Gamma; \Delta \vdash c_i : \textit{catch\_clause}\langle t_i\rangle \qquad \Gamma; \Delta \vdash f : \textit{finally\_clause}\end{array}}{\Gamma; \Delta \vdash \texttt{try}\ \ \texttt{\{}b\texttt{\}}\ \ c_1 \cdots c_n\ \ f : \textit{statement}} \qquad \text{(TR-Try-CF)}$$

Exception handling is performed using `try-catch` statement in Java. Body of the try-clause is typechecked on an extended context which all exception types appeared in catch-clause is added. `finally`-clause is not treated specially, since it only evaluates some statements before re-throwing the thrown exception.

3.4 Methods

Invocation of the method in current class

$$\frac{\begin{array}{c}\text{is-instance}(\Gamma) \\ t_r\ \ n\texttt{(}t_1\texttt{,}\ \ t_2\texttt{,}\ \ \ldots\texttt{,}\ \ t_n\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in \Gamma \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash \texttt{this.}n\texttt{(}e_1\texttt{,}\ \ e_2\texttt{,}\ \ \ldots\texttt{,}\ \ e_n\texttt{)} : t_r} \qquad \text{(TR-MethInvk-This)}$$

$$\frac{\Gamma; \Delta \vdash \texttt{this.}n\texttt{(}e_1\texttt{,}\ \ e_2\texttt{,}\ \ \ldots\texttt{,}\ \ e_n\texttt{)} : t_r}{\Gamma; \Delta \vdash n\texttt{(}e_1\texttt{,}\ \ e_2\texttt{,}\ \ \ldots\texttt{,}\ \ e_n\texttt{)} : t_r} \qquad \text{(TR-MethInvk-This2)}$$

$$\frac{\begin{array}{c}\texttt{static}\ \ t_r\ \ n\texttt{(}t_1\texttt{,}\ \ t_2\texttt{,}\ \ \ldots\texttt{,}\ \ t_n\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in \Gamma \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash n\texttt{(}e_1\texttt{,}\ \ e_2\texttt{,}\ \ \ldots\texttt{,}\ \ e_n\texttt{)} : t_r} \qquad \text{(TR-MethInvk-Static)}$$

This rule type-checks the invocation of a method which is defined in the current class. When a method is invoked, all of following conditions must be met:

1. For all exceptions which may be thrown by the method, it must be properly handled.

2. All actual argument must be coerce-compatible to the type appeared in the formal parameter list.

3. If the method to be called is an instance method, context must also be instance context (`this` must be available to use).

A method invocation of form $\texttt{this.}n\texttt{(...)}$ is treated specially in DynJava, as `this` may contain addition methods which is not defined in the base-class if used in code specifications.

$$\frac{\begin{array}{c}\texttt{super(}t_1\texttt{, } t_2\texttt{, ... , } t_n\texttt{) throws } \tilde{e} \in \Gamma \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash \texttt{super(}e_1\texttt{, } e_2\texttt{, ... , } e_n\texttt{)} : \textit{constructor call}} \quad \text{(TR-Constr-Super)}$$

$$\frac{\begin{array}{c}\texttt{this(}t_1\texttt{, } t_2\texttt{, ... , } t_n\texttt{) throws } \tilde{e} \in \Gamma \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash \texttt{this(}e_1\texttt{, } e_2\texttt{, ... , } e_n\texttt{)} : \textit{constructor call}} \quad \text{(TR-Constr-This)}$$

These rules are for constructor invocation. These are used with rule (TR-ConstrBody-1). As these rules and (TR-StmtList) does not allow constructor invocation inside `try-catch` statement, it is implied that the exceptions thrown by the superclass constructor must be caught by the caller of current constructor.

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e_0 : t \qquad t_r \;\; n\texttt{(}t_1 \;\; n_1\texttt{, } t_2 \;\; n_2\texttt{, ... , } t_n \;\; n_n\texttt{) throws } \tilde{e} \in \text{defs}(t) \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash e_0\texttt{.}n\texttt{(}e_1\texttt{, } e_2\texttt{, ... , } e_n\texttt{)} : t_r}$$

(TR-MethInvk-Instance)

$$\frac{\begin{array}{c}\texttt{static } t_r \;\; n\texttt{(}t_1 \;\; n_1\texttt{, } t_2 \;\; n_2\texttt{, ... , } t_n \;\; n_n\texttt{) throws } \tilde{e} \in \text{defs}(t) \\ \Gamma; \Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma; \Delta \vdash e_n : t_n \\ \{\tilde{e}\} \subset_E \text{exns}(\Gamma)\end{array}}{\Gamma; \Delta \vdash t\texttt{.}n\texttt{(}e_1\texttt{, } e_2\texttt{, ... , } e_n\texttt{)} : t_r} \quad \text{(TR-MethInvk-Class)}$$

These rules are for invocation of method through instance or class name. it must obviously check the consistency of thrown exceptions.

Return from method

$$\frac{\texttt{return void} \in \Gamma}{\texttt{return;} : \mathit{statement}} \qquad \text{(Ret-V)}$$

$$\frac{\texttt{return } t \in \Gamma \qquad \Gamma; \Delta \vdash e : t}{\texttt{return } e\texttt{;} : \mathit{statement}} \qquad \text{(Ret-NV)}$$

On each `return` statement, the type of the return value is checked against those contained in context Γ. if Γ does not contain any `return` *t* specifications, `return` statement is not accepted at all.

3.5 Labels

In Java language, `break` and `continue` can be used only inside loop constructs or `switch`. Also, labeled `break` can only appear inside the loop construct with the same label. Typing rule below describes these properties.

Label-binding Constructs

$$\frac{\Gamma; \Delta \vdash e : \texttt{boolean} \qquad \Gamma^*, \texttt{continue}; \Delta \vdash s : \mathit{statement}}{\Gamma; \Delta \vdash \texttt{while (}e\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-While-NL)}$$

$$\frac{\Gamma; \Delta \vdash e : \texttt{boolean} \qquad \Gamma^*, \texttt{continue } l; \Delta \vdash s : \mathit{statement}}{\Gamma; \Delta \vdash l\texttt{: while (}e\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-While-L)}$$

These two rules are defined for while statement. Both rules extend context Γ with `continue` declaration, and then check the statement *s* under the extended context.

$$\frac{\begin{array}{c}\Gamma^*; \Delta \vdash e_1 : \mathit{expression} \vee e_1 = \varepsilon \qquad \Gamma; \Delta \vdash e_2 : \texttt{boolean} \\ \Gamma^*; \Delta \vdash e_3 : \mathit{expression} \vee e_3 = \varepsilon \\ \Gamma^*, \texttt{continue}; \Delta \vdash s : \mathit{statement}\end{array}}{\Gamma; \Delta \vdash \texttt{for(}e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-For)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e_1 : t \qquad \Gamma, t\ x; \Delta \vdash e_2 : \texttt{boolean} \\ \Gamma^*, t\ x; \Delta \vdash e_3 : \mathit{expression} \vee e_3 = \varepsilon \\ \Gamma^*, t\ x, \texttt{continue}; \Delta \vdash s : \mathit{statement}\end{array}}{\Gamma; \Delta \vdash \texttt{for(}t\ x\ \texttt{=}\ e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-For-B)}$$

$$\frac{\begin{array}{c}\Gamma^*; \Delta \vdash e_1 : \mathit{expression} \vee e_1 = \varepsilon \qquad \Gamma^*; \Delta \vdash e_2 : \texttt{boolean} \\ \Gamma^*; \Delta \vdash e_3 : \mathit{expression} \vee e_3 = \varepsilon \\ \Gamma^*, \texttt{continue } l; \Delta \vdash s : \mathit{statement}\end{array}}{\Gamma; \Delta \vdash l\texttt{: for(}s_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-For-L)}$$

$$\frac{\begin{array}{c}\Gamma; \Delta \vdash e_1 : t \qquad \Gamma^*, t\ x; \Delta \vdash e_2 : \texttt{boolean} \\ \Gamma^*, t\ x; \Delta \vdash e_3 : \mathit{expression} \vee e_3 = \varepsilon \\ \Gamma^*, t\ x, \texttt{continue } l; \Delta \vdash s : \mathit{statement}\end{array}}{\Gamma; \Delta \vdash l\texttt{: for(}t\ x\ \texttt{=}\ e_1\texttt{;}e_2\texttt{;}e_3\texttt{)}\ s : \mathit{statement}} \qquad \text{(TR-For-LB)}$$

These rules are for `for` statement. Rules with postfix “B” correspond to `for` statement with local variable binding, and one with postfix “L” correspond to one with label. The rules appends `continue` label specifications and local variable binding of x, if any, for type-checking the body s. The condition (e_2) must be boolean, and the inclement expression e_3 may be an expression of any type, or may be empty. If no variable binding is done, e_1 is also any expression or empty. If it is with variable binding, e_1 must be compatible to t.

Label Uses

$$\frac{\texttt{break} \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{break;} : \textit{statement}} \qquad \frac{\texttt{break}\ l \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{break}\ l\texttt{;} : \textit{statement}} \qquad \text{(TR-Break)}$$

$$\frac{\texttt{continue} \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{continue;} : \textit{statement}} \qquad \frac{\texttt{continue}\ l \in_L \Gamma}{\Gamma; \Delta \vdash \texttt{continue}\ l\texttt{;} : \textit{statement}} \qquad \text{(TR-Continue)}$$

These rules are for `break` and `continue` statements. These are allowed only when corresponding binding in the label context is exist. See definition 3.10 for more details about $\in_L$.

Switch

$$\frac{\Gamma; \Delta \vdash e : \texttt{int} \qquad \Gamma^*, \texttt{switch}, \texttt{break}; \Delta \vdash s : \textit{statement list}}{\texttt{switch(}e\texttt{)\{}s\texttt{\}} : \textit{statement}} \qquad \text{(TR-Switch)}$$

$$\frac{\texttt{switch} \in \Gamma \qquad i : \textit{integer constant} \qquad \Gamma; \Delta \vdash s : \textit{statement}}{\texttt{case}\ i\texttt{:}\ \ s : \textit{statement}} \qquad \text{(TR-SwitchLbl-C)}$$

$$\frac{\texttt{switch} \in \Gamma \qquad \Gamma; \Delta \vdash s : \textit{statement}}{\texttt{default:}\ \ s : \textit{statement}} \qquad \text{(TR-SwitchLbl-D)}$$

If the switch statement is appear in the program, its body is type-checked under special context with flag `switch`. Case labels and default label are only allowed inside this context. As switch statement allows `break`, it is added to the context.

3.6 Dynamic codes

Statement and expression specifications

$$\frac{R; \Gamma \vdash l : \textit{statement list}}{\Gamma; \circ \vdash \texttt{`<}R\texttt{>\{}l\texttt{\}} : \texttt{code\_spec<}R\texttt{>}} \qquad \text{(TR-Tick-Cspec)}$$

$$\frac{R; \Gamma \vdash e : t}{\Gamma; \circ \vdash \texttt{`}t\texttt{<}R\texttt{>(}e\texttt{)} : t\ \texttt{exp\_spec<}R\texttt{>}} \qquad \text{(TR-Tick-Espec)}$$

Values of code specification types are introduced by this rule. These rule requires that outer context is $\circ$, to prevent nested code specification from appearing in the program.

As these rules show, a context of a code specification is used as a current environment of the body of the specification, which is supposed to supply all name bindings, method bindings, label bindings, etc. Γ is saved to outer context, and only used in rules below.

Value lifting and code specification embedding

$$\frac{\Delta(x) = t \qquad t \text{ is primitive type}}{\Gamma; \Delta \vdash \texttt{\$}x : t} \quad \frac{\Delta(x) = \texttt{java.lang.String}}{\Gamma; \Delta \vdash \texttt{\$}x : \texttt{java.lang.String}} \qquad \text{(TR-Tick-Lift)}$$

$$\frac{\Delta(x) = t\ \texttt{exp\_spec<}R\texttt{>} \qquad R \prec_C \Gamma}{\Gamma; \Delta \vdash \texttt{@}x : t} \qquad \text{(TR-Tick-EmbedE)}$$

$$\frac{\Delta(x) = \texttt{code\_spec<}R\texttt{>} \qquad R \prec_C \Gamma}{\Gamma; \Delta \vdash \texttt{@}x\texttt{;} : \textit{statement}} \qquad \text{(TR-Tick-EmbedS)}$$

If a `$`-expression is appear inside a code specification, the name must be bound in the current *outer context* Δ, not Γ. In addition to this, type of the x is limited to either a primitive type or string, which is the limitation of Java virtual machine.

The rules (TR-Tick-EmbedE/S), which corresponds to `@`-expression also refers to outer context Δ. The type of x must be either statement specification or expression specification. In addition to this, the type-checker checks whether the context required by the specification is actually provided by current context Γ, using context subtype relation. The `@`-expression is treated as of component type of the specification, only when the context check is succeed. This rule and next coerce rules assure the consistency of the composition of the code specifications.

Type compatibilities between specification types

$$\frac{\Gamma; \Delta \vdash x : \texttt{code\_spec<}R\texttt{>} \qquad R \prec_C R'}{\Gamma; \Delta \vdash x : \texttt{code\_spec<}R'\texttt{>}} \qquad \text{(TR-Coerce-CS)}$$

$$\frac{\Gamma; \Delta \vdash x : t\ \texttt{exp\_spec<}R\texttt{>} \qquad R \prec_C R'}{\Gamma; \Delta \vdash x : t\ \texttt{exp\_spec<}R'\texttt{>}} \qquad \text{(TR-Coerce-ES)}$$

This rules correspond to rule (TR-Coerce) in Section 3.2. If a code specification is a subject to implicit type coercion, the component type of the specification must match exactly, and its context specification must be sub-context of that of coercion target.

3.7 Classes

The typing rule of the class definition is partly recursive. The typechecker firstly corrects all member of the class before typecheck the method body.

Firstly, some new operators are defined.

Definition 3.13 (Method context of class) *Instance method context of a class is a context which is visible from the instance methods and constructors. It is defined, using class description* $D = \mathrm{defs}(c)$*, and assuming* c' *as a direct superclass of* c*, as follows:*

$$\begin{aligned}
\mathrm{ctx}_I(D, c') = \mathrm{ctx}_I(c) \coloneqq\ & \texttt{extends}\ \ c' \\
& \cup \{t\ \ x,\ \ t\ \ \texttt{this.}x \mid t\ \ x \in D\} \cup \{t\ \ x \mid \texttt{static}\ \ t\ \ x \in D\} \\
& \cup \{t_r\ \ n\texttt{(}\tilde{t}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \mid t_r\ \ n\texttt{(}\tilde{t}_p\ \ \tilde{n}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in D\} \\
& \cup \{\texttt{static}\ \ t_r\ \ n\texttt{(}\tilde{t}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \mid \texttt{static}\ \ t_r\ \ n\texttt{(}\tilde{t}_p\ \ \tilde{n}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in D\} \\
& \cup \{\texttt{this(}\tilde{t}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \mid \texttt{this(}\tilde{t}_p\ \ \tilde{n}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in D\} \\
& \cup \{\texttt{super(}\tilde{t}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e}\ \ \mid \texttt{this(}\tilde{t}_p\ \ \tilde{n}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in \mathrm{defs}(c')\}
\end{aligned}$$

Class method context of a class is a context which is visible from class methods in the class. It is defined, using class description $D = \mathrm{defs}(c)$*, as follows:*

$$\begin{aligned}
\mathrm{ctx}_C(D, c') = \mathrm{ctx}_C(c) \coloneqq\ & \texttt{static extends}\ \ c' \\
& \cup \{t\ \ x \mid \texttt{static}\ \ t\ \ x \in D\} \\
& \cup \{\texttt{static}\ \ t_r\ \ n\texttt{(}\tilde{t}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \mid \texttt{static}\ \ t_r\ \ n\texttt{(}\tilde{t}_p\ \ \tilde{n}_p\texttt{)}\ \ \texttt{throws}\ \ \tilde{e} \in D\}
\end{aligned}$$

Method declaration

$$\frac{\mathrm{ctx}_I(c), \tilde{t}\ \ \tilde{n}, \texttt{return}\ \ t_r, \texttt{throws}\ \ \tilde{e}; \circ \vdash l : \textit{statement list}}{\circ, \circ \vdash t_r\ \ x_m\texttt{(}\tilde{t}\ \ \tilde{x}\texttt{)}\ \ \texttt{throws}\ \ \tilde{e}\ \ \{l\} : \textit{method}\langle c\rangle} \qquad \text{(TR-Method-I)}$$

$$\frac{\mathrm{ctx}_C(c), \tilde{t}\ \ \tilde{n}, \texttt{return}\ \ t_r, \texttt{throws}\ \ \tilde{e}; \circ \vdash l : \textit{statement list}}{\circ, \circ \vdash \texttt{static}\ \ t_r\ \ x_m\texttt{(}\tilde{t}\ \ \tilde{n}\texttt{)}\ \ \texttt{throws}\ \ \tilde{e}\ \ \{l\}\ \ \{l\} : \textit{method}\langle c\rangle} \qquad \text{(TR-Method-C)}$$

$$\frac{\mathrm{ctx}_I(c), \tilde{t}\ \ \tilde{n}, \texttt{throws}\ \ \tilde{e}, \texttt{super}; \circ \vdash l : \textit{statement list}}{\circ, \circ \vdash c\texttt{(}\tilde{t}\ \ \tilde{x}\texttt{)}\ \ \texttt{throws}\ \ \tilde{e}\ \ \{l\} : \textit{method}\langle c\rangle} \qquad \text{(TR-Method-Constr)}$$

Typing rule is defined for the method declarations. It creates initial context from the members of current class, arguments, return type, and throwable exceptions to type-check the body.

$$\frac{\begin{array}{c}\texttt{this(}t_1\ n_1\texttt{,}\ t_2\ n_2\texttt{,}\ \texttt{...,}\ t_n\ n_n\texttt{)}\ \texttt{throws}\ \tilde{e} \in \mathrm{defs}(t)\\ \Gamma;\Delta \vdash e_1 : t_1,\quad \ldots,\quad \Gamma;\Delta \vdash e_n : t_n \qquad \{\tilde{e}\} \subset_E \mathrm{exns}(\Gamma)\end{array}}{\Gamma;\Delta \vdash \texttt{new}\ t\texttt{(}e_1\texttt{,}\ e_2\texttt{,}\ \texttt{...,}\ e_n\texttt{)} : t} \quad \text{(TR-New)}$$

The typing rule for instance creation is similar to that of method invocation.

3.8 Class specifications

Class specification type is introduced in the form of class specification definition. The declaration "`class_spec` n `extends` c' $\{D\}$" introduces name of the type n as a type $\texttt{class\_spec}\langle D, c'\rangle$.

$$\frac{n \text{ is a class specification type}}{\Gamma;\Delta \vdash \texttt{new}\ n\texttt{()} : n} \quad \text{(TR-New-CLS)}$$

$$\frac{\begin{array}{c}\texttt{super} \notin \Gamma\\ \Gamma^*, \texttt{class\_spec}\langle D, c'\rangle\ x;\Delta \vdash l : \textit{statement list}\end{array}}{\Gamma;\circ \vdash \texttt{class\_spec extends}\ c'\ \{D\}\ x;\ l} \quad \text{(TR-New-AnonCLS)}$$

To create an instance of class specification, use normal `new` statement. Extended `class_spec` syntax can be used to generate class specification anonymously.

$$\frac{\Gamma;\Delta \vdash e : \texttt{class\_spec}\langle D, c'\rangle \qquad t_r\ n\texttt{(}\tilde{t}\ \tilde{n}\texttt{)}\ \texttt{throws}\ \tilde{e} \in D}{e\texttt{.<}t_r\ n\texttt{(}\tilde{t}\ \tilde{n}\texttt{)>} : \texttt{code\_spec<}\,\mathrm{ctx}_I(D, c'), \tilde{t}\ \tilde{n}, \texttt{return}\ t_r, \texttt{throws}\ \tilde{e}\texttt{>}} \quad \text{(TR-CLS-Field-I)}$$

$$\frac{\Gamma;\Delta \vdash e : \texttt{class\_spec}\langle D, c'\rangle \qquad \texttt{static}\ t_r\ n\texttt{(}\tilde{t}\ \tilde{n}\texttt{)}\ \texttt{throws}\ \tilde{e} \in D}{e\texttt{.<}t_r\ n\texttt{(}\tilde{t}\ \tilde{n}\texttt{)>} : \texttt{code\_spec<}\,\mathrm{ctx}_C(D, c'), \tilde{t}\ \tilde{n}, \texttt{return}\ t_r, \texttt{throws}\ \tilde{e}\texttt{>}} \quad \text{(TR-CLS-Field-I)}$$

$$\frac{\Gamma;\Delta \vdash e : \texttt{class\_spec}\langle D, c'\rangle \qquad \texttt{this(}\tilde{t}\ \tilde{n}\texttt{)}\ \texttt{throws}\ \tilde{e} \in D}{e\texttt{.<this(}\tilde{t}\ \tilde{n}\texttt{)>} : \texttt{code\_spec<}\,\mathrm{ctx}(D, c'), \tilde{t}\ \tilde{n}, \texttt{throws}\ \tilde{e}, \texttt{super>}} \quad \text{(TR-CLS-Field-Constr)}$$

Reference to the field of class specifications have special syntax. Note that the context specifications given for the fields matches the one appeared in the rules in the previous section. These field references can be used as a left-hand-side of the assignment.

$$\frac{\Gamma;\Delta \vdash e : \texttt{class\_spec}\langle D, c'\rangle}{\Gamma;\Delta \vdash e\texttt{.compile()} : \texttt{void}} \qquad \text{(TR-CLS-Compile)}$$

$$\frac{\begin{array}{c}\Gamma;\Delta \vdash e : \texttt{class\_spec}\langle D, c'\rangle \\ \texttt{this(}t_1\ n_1\texttt{,}\ t_2\ n_2\texttt{,}\ \ldots\texttt{,}\ t_n\ n_n\texttt{)}\ \texttt{throws}\ \tilde{e} \in D \\ \Gamma;\Delta \vdash e_1 : t_1, \quad \ldots, \quad \Gamma;\Delta \vdash e_n : t_n \qquad \{\tilde{e}\} \subset_E \mathrm{exns}(\Gamma)\end{array}}{\Gamma;\Delta \vdash \texttt{new}\ e\texttt{(}e_1\texttt{,}\ e_2\texttt{,}\ \ldots\texttt{,}\ e_n\texttt{)} : c'} \qquad \text{(TR-CLS-New)}$$

The rule (TR-CLS-New) typechecks the instance generation of dynamically-generated class. As the generated class is anonymous, the value is treated as an instance of its superclass c'.

Chapter 4

Implementation

4.1 Overview

In this chapter, I present the current implementation of DynJava. The implementation of DynJava is based on runtime code generator approach. For each code specifications, our compiler system generates one runtime code generator. The code generators are combined by the static part of the program and generate bytecode at runtime. Generated bytecode are loaded to Java virtual machine using custom classloader and linked to the running program.

Our compiler system consists of two parts: the language preprocessor and the code postprocessor, which utilizes Java's original compiler `javac` as a back-end code generator. This approach is similar to those of Tempo [2, 12].

First, the language preprocessor reads the source code and type-checks the whole code. For each code specification, the preprocessor generates a code generator class, with a *template* of dynamic code in the Java language. Next, The template is then processed by the `javac` and compiled into bytecode. Finally, the code postprocessor reads the result of `javac` and translates it to Java program which generates the bytecode on runtime code generation. Compiling postprocessor-generated code will produce code generators (or runtime compilers) of code specifications.

Using `javac` as a prototype-generator for dynamic code generators have two merits and one demerit. First, it makes whole system simpler and easy to implement, as preprocessor and postprocessor does only need to handle constructs which are closely related to dynamic code generation. Second, generated code are partly optimized by `javac`. However, it makes postprocessor slightly depends on the specific version of `javac`. Actually,

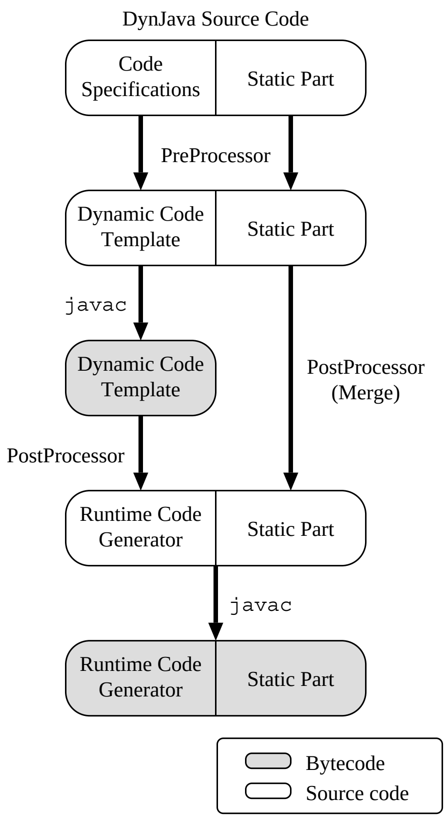


Figure 4.1: Global structure of DynJava compiler

as many JVM instructions heavily correspond to the specific constructs of Java language, the problem is not so serious. Current postprocessor implementation is tuned to work with `javac` contained in Linux version of JDK 1.3.0 (IBM build).

Currently, the preprocessor is implemented as a plug-in for EPP [10], a extensible Java preprocessor package. The postprocessor and runtime code generator is implemented using JavaClass [4], a library for bytecode manipulation. The process for all DynJava constructs except for `switch` is currently implemented. As runtime code generator should run very fast to minimize code generation cost, I'm planning to re-implement the code generator using more light-weight methods (see Appendix A).

Types in DynJava	Representation on Java
`code_spec<...>`	`yl.dyncode.cspecV`
`void exp_spec<...>`	`yl.dyncode.cspecV`
`boolean exp_spec<...>`	`yl.dyncode.cspecZ`
`char exp_spec<...>`	`yl.dyncode.cspecC`
`byte exp_spec<...>`	`yl.dyncode.cspecB`
`short exp_spec<...>`	`yl.dyncode.cspecS`
`int exp_spec<...>`	`yl.dyncode.cspecI`
`long exp_spec<...>`	`yl.dyncode.cspecL`
`float exp_spec<...>`	`yl.dyncode.cspecF`
`double exp_spec<...>`	`yl.dyncode.cspecD`
c `exp_spec<...>` (*c* is not primitive type)	`yl.dyncode.cspecA`

Table 4.1: Type mapping between DynJava and Java representation

4.2 Translations by the preprocessor

Firstly, I describe the translation performed by the preprocessor with the description of current implementation.

4.2.1 Code specification types

As the subtype relation between code specification types is complex, it cannot be directly mapped into the subtype relation between Java's classes or interfaces. In the implementation, the preprocessor maps all code specification types into ten predefined abstract classes, according to Table 4.1. This mapping is obviously projective: two or more types are mapped into one representation. This is generally not a problem because:

1. Type checking is done globally by the preprocessor before mapping.
2. If the user intentionally bypass the preprocessor's type check by mixing two classes separately compiled with a different set of related classes, the runtime system or the verifier in JVM detects unsafe code and refuse to loaded it. A similar behavior is observed when the user bypass the `javac`'s type check by the same way.

One limitation imposed by this mapping is that a class can not have two methods of same name, if the types of the arguments are equivalent except for code specification types.

4.2.2 Class specifications

If class specifications are appeared in the program, it is converted into normal class definitions. The generated class is a subclass of class `yl.dyncode.classspec`, which is a part of the runtime system, and have following fields and methods:

1. Fields holding statement specifications for method body, one per each method and constructor declarations
2. Fields holding reflection object for constructors, one per each constructor declarations
3. A method `compile`, which generates requested class.
4. Methods named `__new`, one per each constructor declarations

First, For each method declarations in the class specification, the preprocessor emits one field declarations, which has appropriate statement specification types, described in 3.8. The name of the field is determined from the name of the method, type of a return value, and type of arguments. In current implementation, it is a concatenation of the name and type signature used in the class file, replacing characters invalid for identifiers by "`_`". For example, the field name representing `int invoke(double x, Object y)` is "`invoke_DLjava_lang_Object__I`", because its type signature is "`(DLjava/lang/Object;)I`".[1] The names of the arguments are not included in the field name, because having two methods only whose argument names differ is invalid.

Second, for each constructor declarations, a method `__new` and a field is generated. When the method is called, it calls the constructor of the generated class using Java's reflection API, to instantiate an object of the generated class. A reflection object for the constructor is generated during compilation (see below) and stored into a field in class specification object. The name of those fields are `__new_` concatenated with type signature.

In addition to those, the preprocessor generates method `void compile()`, which generates class from code specifications. Most of the work which should done for class generation is already implemented in the superclass `yl.dyncode.classspec`. The method `compile` passes the list of methods and statement specifications to runtime via interfaces defined in `classspec` and asks `classspec` to generate a class. If the compile succeeds, it

[1]This naming rule is decided to make debug easier. If name collision become real problem, it may be changed to use hash of type signature, for example `__invoke_9d19612617d1ca76`.

acquires reflection objects for every constructors using reflection API, which are required by `__new`'s. An example of translation is shown in Figure 4.2.

In actual implementation of the preprocessor using EPP, there is small difficulty to use type signature for the name of field. In EPP, type checker is only available after all class declarations are parsed, but it is not possible to expand short form of class name like `Object` to its full name like `java.lang.Object` without type checker. This problem is solved using some tricks to delay expansion of field name until all names of defined classes are collected and type checker is become available.

4.2.3 `new` expression

When the `new` expression with the argument of class specification instances instead of class name appears, it is translated into the invocation of method `__new`, described above. Syntactically, there is two patterns for calling `new` with class specification. Firstly, if `new` is followed by open parenthesis "`(`", it is always assumed to be extended one. If a dotted name is appeared, it is ambiguous that it may be either class name or reference to class specification instance. In this case, typechecker is used to determin whether it is normal instance creation or extended one.

4.2.4 Reference to methods in class specification

If a methods in a class specification is referenced using `.<···>` syntax, it is simply expanded to its mangled form, used in class spec translation. For example, an expression `class_A1.<void invoke()>` is translated to `class_A1.invoke__V`.

4.3 Translating body of code specifications

The bodies of the code specifications are translated into runtime code generator by close cooperation of the preprocessor and the postprocessor. In this section, I enumerates the constructs of DynJava language, and describes how the runtime code generator is generated for each constructs in those translation process.

Generally, translation tactics are carefully chosen to keep the following properties:

- If a original construct is an expression, the types of the expression and result of translation must match.
- The exceptions thrown by the expression or statement matches to those of translation result.

Original:

```
class_spec Class_A1 extends yl.dyncode.SimpleMethod {
    int x;
    this();
    void invoke();
}
```

Translated: (Reformatted and redundant parenthesis removed)

```
class Class_A1 extends yl.dyncode.classspec {
    Class_A1(){ super("yl.dyncode.SimpleMethod"); }

    yl.dyncode.cspecV this__V;
    yl.dyncode.cspecV invoke__V;

    private java.lang.reflect.Constructor
            __new__Lyl_dyncode_SimpleMethod_;
    yl.dyncode.SimpleMethod __new(){
        if (class_obj == null) compile();
        try {
            return (yl.dyncode.SimpleMethod)
                __new__Lyl_dyncode_SimpleMethod_.newInstance(new Object[]{});
        } catch (java.lang.Exception e) {
            e.printStackTrace();
            throw new java.lang.Error("while instantiating object: " + e);
        }
    }

    void compile(){
        try {
            this.add_field("x", "I", false);
            this.add_method("<init>", "()V", "", false, this__V);
            this.add_method("invoke", "()V", "", false, invoke__V);
            this.generate_class();
            __new__Lyl_dyncode_SimpleMethod_ =
                class_obj.getConstructor(new java.lang.Class[]{});
        } catch (java.lang.Exception e) {
            e.printStackTrace();
            throw new java.lang.Error("while compiling classspec: " + e);
        }
    }
}
```

Figure 4.2: Example of class_spec translation by the preprocessor

- If the control flow is altered by the statement, result of translation should also alter the control flow in the way resembles to original expression.
- If the result of translation must be processed further by postprocessor, it should have specific pattern which is easy for the postprocessor to detect. Especially, it is desirable that its structure is simple enough to prevent optimizer component in `javac` from simplify it more.

4.3.1 Arithmetics and other simple expressions

For simple arithmetics and other simple expressions, DynJava preprocessor simply generates the code as-is into the template. For example, from the expression specification `‘int<>(5 + 3)`, the preprocessor generates following template class as a inner class of the current class:

```
class _C0 extends yl.dyncode.cspecI {
    _C0() {}
    private int __template() {
        return (5 + 3);
    }
}
```

The specification itself is translated into the expression “`new _C0()`”. The method `__template` has the same return type and `throws` declaration as the specification has, so that it passes the `javac`’s type checking. From this code, `javac` generates following bytecode for the method `__template`:

```
BIPUSH 8
IRETURN
```

The postprocessor removes the return instruction appeared at the last of the template method. Note that after removing a return instruction, the generated code leaves one value on the stack for expression specification.

Almost same approach is taken for statement specifications. Statement specifications are translated into the program which generates the code which does not leaves any value on the stack. If `RETURN` instruction is appeared other than the last of the template method, it is translated to the `GOTO` instruction to the end of generated code.

Other instructions appeared in the compiled template are translated to the program which generates those instructions verbatimly at runtime. Most instructions can be simply translated to the code which appends one instructions to output list. However, like usual assembly language, forward branch instructions need some treatment, because the target position of the branch is not yet decided. In current implementation, the postprocessor generates the code which generates branch instruction with dummy (null) target address. And it appends the code which back-fills the target address to branch instructions to the last of generated code.

4.3.2 Embedded code and constant lifting

If some constant value is embedded to code specifications, it must be translated into the instructions which pushes the exact value onto the stack. if `$`-expression is appeared in the code specifications, teh preprocessor allocates an instance field in the template class, and stores the runtime constant value into the field at the constructor of the template class. The expression is then translated to a reference to the field allocated by the preprocessor. For example, for `int exp_spec<>{ $i }`, the preprocessor generates following template class.

```
class _C1 extends yl.dyncode.cspecI {
    int __c0;
    _C1(int __a0) { __c0 = __a0; }
    private int __template() {
        return __c0;
    }
}
```

The `$`-expression itself is translated to `new _C1(i)`. The name begins with `__` is reserved by the system and its naming schema is carefully managed. The name begins with `__c` is only used by value lifting and specification embedding.

`Javac` translates this code into following bytecode:

```
ALOAD_0
GETFIELD Test$_C1.__c0 I
```

In this code, `ALOAD_0` loads the reference to `this`, which is type of the template class. As the type of template class may never appear in the final result, it must be compiled from a code which is generated temporarily by the preprocessor. In the code above, following `GETFIELD` instruction refers to the field generated by constant lifting expression. The postprocessor converts this instruction sequence into the code which generates instructions which pushes the constant value stored in the field to the stack. As specification of

JVM only allows the constant of numeric or string types in the class file, DynJava also restricts the types of $-expressions to those types.

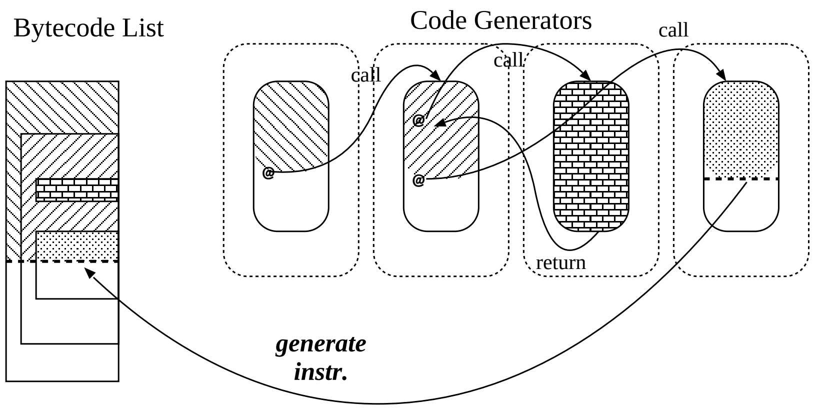
For @-expression, the preprocessor also allocates the instance field and stores the instance of template class in it. In addition to this, the preprocessor allocates dummy static method, which has same return types and `throws` declaration as the embedded specification has. Then the @-expression is translated into the invocation of that method. For example, if `x` has type `int exp_spec<throws IOException>`, "`` `int<throws Exception>(@x) ``" is translated into the following template. Note that the declaration of method `__t0` makes generated template type-safe.

```
class _C2 extends yl.dyncode.cspecV {
    yl.dyncode.cspecV __c0;
    _C1(yl.dyncode.cspecV __a0) { __c0 = __a0; }
    private static int __t0 throws IOException
        { throw new Error(""); }
    private int __template() throws Exception {
        return __t0();
    }
}
```

The postprocessor translates the invocation of method with name `__t*` into the nested invocation of the code generator (Figure 4.3). Because each code generator created the code which pushes one value onto the stack, or nothing for statement specifications, embedding one generated bytecode into another produces correct bytecode with regards to the stack depth and types. However, as each template of code uses the slots of local variables in its own way, naive merger of two codes will conflict with each other on the usage of local variables. Our postprocessor and generated code generator counts the number of used local variable slots at each embed point and shifts the slot numbers used by inner code fragment to unused ones.

4.3.3 Variables and fields

If the code refers to a variable which is bound by context specifications (not bound syntactically inside statement specification), it must be resolved to refer outer code at code generation time. First, the type-checker in the preprocessor determines whether a variable is bound syntactically or by context. If it is bound syntactically, the preprocessor simply



Patterned bytecode fragments are generated by the code generator with corresponding patterns. The code generated by inner code generator is embedded into code generated by outer one.

Figure 4.3: Nested code generator invocation by `@`-expression.

emits the variable reference into template. Else, the preprocessor allocate a static field with the same name and type, and emits the reference to the field. For example, for the code

```
‘<int i>{ int j = 0; System.out.println("r=" + (i + j)); }
```

it emits the following template:

```
class _C3 extends yl.dyncode.cspecV {
    yl.dyncode.cspecV __c0;
    static int i;
    private int __template() {
        int j = 0; System.out.println("r=" + (i + j));
    }
}
```

`Javac` compiles it to the instructions `GETFIELD` and `PUTFIELD`. The postprocessor finds those instructions referring the template class, and translates it to the code which asks the “caller” code generator to generate the variable reference.

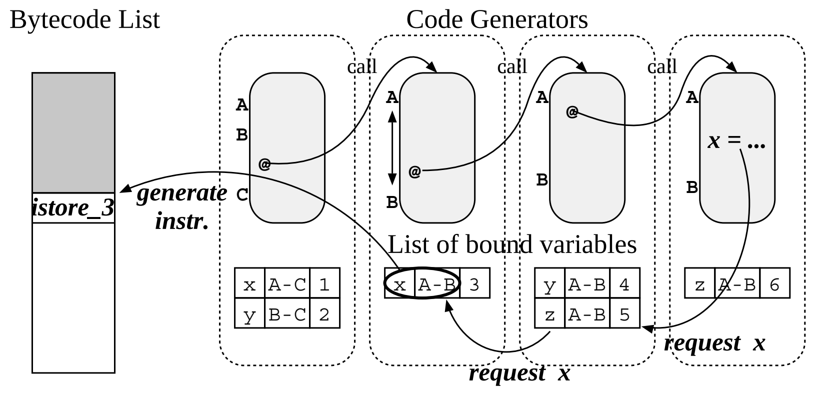


Figure 4.4: Resolving free variables

If a code generate is asked to generate the variable reference, it checks the referred variable is bound by the current position of generating code, by using debugging information generated by `javac`. If it binds the variable, it generates appropriate instructions to access a local variable slot. If it does not bind, delegates the request to its caller again (Figure 4.4). If the request does not resolved even by the topmost code specification, it is delegated to the "method generator" which controlling generation of code for the current method, and compared to its argument specification. If it also fails, the request goes to the "class generator" and it generates an appropriate field reference instruction.

If an expression refers to a field of `this` (`this.`*name*), it is translated as if it refers to a variable of special name `__this_`*name*. The request on such special variable is always delegated to its class generator.

If an expression refers to `this` alone, it is translated as if it is special variable `__this`. It is always converted into the instruction `ALOAD_0` by the postprocessor.

4.3.4 Methods

If the code invokes a static method defined in the context specification, a dummy static method with same return type, same formal parameter types, and same `throws` declaration is added to the template. Then the invocation is translated into an invocation of dummy method. If the postprocessor encounters the `INVOKESTATIC` instruction referring the template class, it is converted to the code which generates `INVOKESTATIC` instruction referring current generating class.

If the code invokes a instance method on `this`, almost same approach is taken. However, dummy instance method is generated instead. Also, because a reference to `this` is needed at the bottom of arguments on the stack to invoke an instance method, following additional translations are done:

1. An formal parameter of type `Object` is added to the top of formal parameter list.
2. An argument `__this` is added to the top of arguments.

The reference to `__this` inserted by 2. is turned to a code which pushes the reference to `this`, which is needed on the invocation.

Method invocation on other classes or instances does not need special treatment. The preprocessor inserts it into template verbatimly.

4.3.5 Return statement

If `return` statement appears in the code, a static dummy method with name `__return` is added and the statement is translated into an invocation of it (with an arguments if the return type is not `void`). The postprocessor translates it to an appropriate `*RETURN` instruction.

4.3.6 Labels and break statement

If a loop constructs (`while` and `for`) appears in the code, the type-checker checks whether there is embedded statement specifications which may refer the loop constructs by label (including *null* label). If the loop is not referred by inner statement specifications, no translations are performed by the preprocessor. If it is referred, the preprocessor inserts following code at the top of loop body, where *label* is a name of the label associated with the loop.

```
if (__providelabel("__TICK_LABEL_BREAK_ label")) break;
else if (__providelabel("__TICK_LABEL_CONTINUE_ label")) continue;
```

The special method `boolean __providelabel(String)` is defined in the superclass of the template class, `yl.dyncode.cspec`. This inserted code is compiled into following 8 bytecode instructions by `javac`.

```
ldc              "__TICK_LABEL_BREAK_ label"
```

```
    invokestatic      yl.dyncode.cspec.__providelabel
                           (Ljava/lang/String;)Z
    ifeq              L1
    goto              LB
L1: ldc               "__TICK_LABEL_CONTINUE_ label" (9)
    invokestatic      yl.dyncode.cspec.__providelabel
                           (Ljava/lang/String;)Z
    ifeq              L2
    goto              LC
L2:
```

The postprocessor removes above 8 instructions and records the targets of `LB` and `LC` in code generator.

If `break` or `continue` statement appears in the code, the type-checker checks the target loop of the statement is either in the code, or outside the code specification. In the former case, no translations is done by the preprocessor. In the latter case, it is translated to either

```
throw __break("__TICK_BREAK_ label");
```

or

```
throw __break("__TICK_CONTINUE_ label");
```

The special method `java.lang.Throwable __break(String)` is also defined in the class `cspec`. The postprocessor translates invocation of this special methodto the code which firstly generates an unconditional branch instruction, and then asks caller code generator to fill-in appropriate branch target. `ATHROW` instruction which corresponds to `throw` is removed.

4.3.7 Switch

Currently, switch statement is not supported by implementation.

Chapter 5

Experiments

In this chapter, I show some experimental result of the performance of DynJava implementation. I implemented a runtime optimizer for fast Fourier transform (FFT). FFT calculates the discrete Fourier transform of size n

$$Y[i] = \sum_{j=0}^{n-1} X[j]\,\omega_n^{-ij} \quad \text{where } \omega_n = e^{2\pi\sqrt{-1}/n}$$

in the computational order less than $O(n^2)$. There is a well-known algorithm [5] of $O(n \log n)$ when n equals to some power of 2. For general number n, there are several algorithms to achieve low computational order, described in [7], but all of them are depending on the size n. In the experiments, I implemented the Cooley-Tukey fast Fourier transform [3], along with the naive DFT routine.

The Cooley-Tukey fast Fourier transform can be applied when n can be factored into $n = n_1 n_2$, and is represented by the following formula:

$$Y[i_1 + i_2 n_1] = \sum_{j_2=0}^{n_2-1} \left[\left(\sum_{j_1=0}^{n_1-1} X[j_1 + j_2]\,\omega_{n_1}^{-i_1 j_1} \right) \omega_n^{-i_1 j_2} \right] \omega_{n_2}^{-i_2 j_2}.$$

The formula can be interpreted as (1) inner DFT of size n_1 repeated n_2 times (however input data are not continuously stored in array X), (2) n times of multiplication called "twiddler", then (3) outer DFT of size n_2 repeated n_1 times. If n is factored into more than 2 prime numbers, the lgorithm can be applied recursively for inner and outer DFTs. Assuming that the computation size of the FFT of size n is written as $F(n)$, the Cooley-Tukey algorithm reduces $F(n)$ from $O(n^2)$ to $O[n_2\,F(n_1) + n + n_1\,F(n_2)]$. When n is p^k (p is prime), and naive DFT is used for the DFT of size p, the actual F(k) is become

$O(p^2\ n\ \log n)$. In the [7], other algorithms for $n = 4n'$, for $n = n_1 n_2$ where $\gcd(n_1, n_2) = 1$, and for prime number n is described, but these are not implemented in this experiment. We also implemented a routine which implements the same algorithm using an object for representation of nested DFT calculations and compared the performance.

DynJava version of the FFT implementation is about 160 lines of DynJava code (counting only method body) and consists of followings:

1. The code generator for base case (n is prime), about 25 lines
2. The code generator for Cooley-Tukey iteration case (n is factored), about 40 lines
3. The dispatcher calling above two routines, about 10 lines
4. The interface called from other part of program, about 25 lines
5. Support routines (ex. prepare array of ω's), about 60 lines

The length of these two code is roughly equal to the length of the program which does not use dynamic code generation.

In the implementation, two methods returning statement specification type are defined. They tooks following arguments:

1. size of the input data (an immediate value of `int`)
2. from/to what array data should be read or stored (an expression specification of type `double[]`)
3. the position in the array where first datum is stored (an expression specification of type `int`)
4. read/write interval for data (an immediate of `int`).

and they return statement specifications which calculates FFT by appropriate algorithms, with constants described above as an immediate value are embedded. Those specifications are either used to produce a body of user-requested FFT routine, or embedded into the position of inner FFT of Cooley-Tukey algorithms. For example, if $n = 210$, which is factored to $16 = (2 \cdot 7) \cdot (3 \cdot 5)$, generic DFT routine of size of 2 and 7, which consists of two nested `for` loops are embedded into the for loop appeared in the Cooley-Tukey FFT routine of size 14, and the routine of size 14 is embedded into `for` loop in another Cooley-Tukey routine, which is size 210. Finally, the routine consists of 4-depth nested `for` loop are generated.

Size of data sets		# of factors	# of iters.	Dynamic Generation Time	Code Sz.	1st Time	Static Time	Over-head
960	$2^6 \cdot 3 \cdot 5$	8	200	**2.11**	2761	2652	2.15	−1.9%
1024	2^{10}	10	50	**2.16**	3483	4155	2.39	−9.7
2048	2^{11}	11	50	**4.23**	3855	5274	5.19	−18.5
3600	$2^4 \cdot 3^2 \cdot 5^2$	8	25	**7.85**	2766	2304	8.34	−5.9
6561	3^8	8	25	15.08	2770	2290	13.90	+8.4
8192	2^{12}	12	50	**24.95**	4594	7934	25.64	−2.7
10000	$2^4 \cdot 5^4$	8	25	25.7	2764	2296	25.2	+1.9
16384	2^{13}	13	50	**53.40**	4963	9419	56.96	−6.3
30030	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13$	6	25	97.0	2066	1358	92.3	+5.1
44100	$2^2 \cdot 3^2 \cdot 5^2 \cdot 7^2$	8	5	**130.1**	2789	2428	136.5	−4.7
65535	$3 \cdot 5 \cdot 17$	3	10	899.2	1318	1593	878.0	+2.4
65536	2^{16}	16	20	–	–	>2min.	272.45	–
1048575	$2 \cdot 5^2 \cdot 11 \cdot 31 \cdot 41$	6	1	6051	2062	7230	5915	+2.2

(unit: Time: [ms], Code Size: [byte])

Runtime environment is: Linux 2.2, IBM build of JDK1.3.0 with JIT enabled, PentiumIII 500MHz, 256MB of main memory

Table 5.1: Computation time of the FFT

Table 5.1 shows the execution time of the FFT computation for various data set size. Input values to the FFT are the same random complex numbers. The times shown in the row “Dynamic” is the execution time of dynamically-generated code, and times in the row “Static” is that of the routine using object representation. For each data size, the test performs 5 sets of repeated FFT calculation, with predefined number of iterations. The “Time” shown in the table are an average of the execution time of 3 sets, excluding the fastest and slowest sets, and divided by the number of iterations. Before beginning iterations, the program performs FFT calculation once, to perform JIT compilation. The time needed for the first invocation is measured separately and shown in the column “1st Time”. “1st Time” minus “Time” shows the guessed time needed for JIT compilation. For size of 65536, first call of FFT routine does not last in 2 minutes, which may be caused by bug of either library or JIT compiler. I also measured the length of bytecode of the generated calculation method. As usual length for method in Java seems to be around 200 bytes, and most method in Java does not exceed 2000 bytes, the code generated with this experiments are exceptionally long and deeply nested for JIT compiler, though this setting is intentional.

When the data set size contains many small factors, especially many 2’s, the dynamically-generated version of the routine runs faster than statically-compiled one. The time needed for JIT compilation is around 2 seconds with 8 prime factors, and increase to 9 seconds with 13 factors. However, performance gain of about 1 milliseconds overcome this overhead with about 10000 iterations, even the setting of experiments seems hard for JIT compiler.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

I presented a strongly typed language DynJava that supports dynamic code generation. The user can write dynamic code fragments using high-level language constructs easily. The presented type system statically guarantees that dynamically composed code fragments are type safe, and makes programer away from painful debugging of dynamic code, which tends to be try-and-error basis.

This thesis also presents an implementation of DynJava on Java virtual machine. The current implementation demonstrates that the system can be used to easily implement dynamic optimization of a FFT program.

6.2 Future work

I am planning to implement more efficient runtime code generator in the near future. The new runtime code generator will generate a bytecode by directly generating byte streams of instructions into array, instead of library-based, object-based approach which is currently used. Its idea is described in Appendix A. Also, implementation of `switch` statement should be completed.

Current design of DynJava is focused to allow almost all constructs of static language to be written flexibly in dynamic code specification. To make a DynJava language more generally useful, some new constructs or syntax sugars, some of which may be specific to dynamic codes, will be needed. Language design of DynJava may be refined with experiences applying DynJava to large scale of actual applications.

Appendix A

An Efficient Dynamic Code Generator

In this chapter, I describe an idea of efficient dynamic code generator for DynJava, which replaces current implementation using JavaClass library.

A.1 Background

As described in Chapter 4, the DynJava compiler generates a prototype of byte code stream for each code specifications using `javac`. A byte code stream consists of two parts: codes which are translated to appropriate processes by the postprocessor, and codes which are copied into dynamically-generated bytecode sequence. The algorithm described in this chapter handles the latter part.

A planned code generator directly emits bytecode instructions into an byte array. As both input of the preprocessor and output of the code generators are in same bytecode language, many instructions can be copied simply in byte-to-byte basis. For example, an instruction `IADD` (integer addition), whose instruction number is $\mathtt{60}_{16}$, can be translated to the program code like “`bytecode[ip++] = 0x60;`”.

However, even if the instruction is not for translation described in chapter 4, The following instructions must not be copied byte-to-byte way:

An instruction which operates on local variables As described in Section 4.3.2, a code generator which is nested inside another generator must generate a code whose accesses to local variables have shifted the indices.

A branch instruction An operand to a branch instruction is an offset to branch target from the instruction, instead of absolute address in the bytecode. If a branch jumps over the code generated from other embedded code specifications or codes referring free variables, which have variable length, the offset to the target of branch may be changed. If a range between the branch instruction and the branch target does contain only fixed-length instructions, the branch can be generated verbatimly.

An instruction which refers to a constant pool entry As each template bytecode have own constant pool, instructions from two different code specifications do not agree on the entry number of constant poll unless it is resolved globally in output class.

In addition, some of these instructions have variable length. As JVM's bytecode language is focused to reduce the size of bytecode, which may also reduces execution time in embedding environments, frequently-appearing instructions have its short form. For example, an instruction `ILOAD` *n*, which loads an `int` value in the *n*'th local variable slot to the top of operand stack, has two-byte representation, "$\texttt{15}_{16}$, *n*". However, if *n* is smaller than 3, it is represented by one byte, ($\texttt{1A}_{16} + n$). In addition, if *n* is greater than 255, although it is rare, it requires 4 bytes, as "$\texttt{C4}_{16}$, $\texttt{15}_{16}$, $\lceil n/256 \rceil$, $n \bmod 256$". If the length of an instruction is changed, all branch instructions which jumps over the instruction must be modified.

A.2 Code generator

An efficient code generator can be implemented in following algorithm under above conditions.

1. Expand all local variable reference instructions into its longest form.

2. Break down the byte code stream at each point of code embedding and free variable reference, where the byte code stream of variable length will be inserted. I each piece of code "basic block" of the bytecode.

3. Reads through the input bytecode stream. For each instruction in the instruction stream,

   (a) if it is a normal instruction which is not handled by rules below, output the code which writes the instruction opcode to the target byte array prepared for each method, and write the original operand of the instruction, if any.

(b) if it accesses a local variable, output the code which writes an instruction opcode to the target array, then writes the number (value of an original operand) + (shift amount of local variable slot).

(c) if it refers constant pool, emit the code which consults constant pool generator in the generator of current class, which is described later, and output the instruction with the constant pool index returned from the constant pool generator.

(d) if it is a branch which is local to the current basic block, output the code which emits the instruction verbatimly.

(e) if it is a non-local backward branch, output the code which writes the instruction opcode of branch, and the operand calculated as

(the address of the top of target block)
– (the address of the top of current block)
+ [(the in-block offset of branch target from the top of target block)
– (the in-block offset of branch instruction from the top of current block)].

The equation in [ ] can be calculated by postprocessor. The addresses of the top of blocks is recorded by a rule for the top of block, which is described below.

(f) if it is a non-local forward branch, output the code which writes the instruction opcode of branch, with a dummy operand. Then, record that when the process reaches the top of the target block, it should output the code which overwrites the dummy operand, whose location is determined by the in-block offset of the branch instruction and the block number of it, by the value calculated as same as those of backward branch.

4. At the top of each basic block, output the code that records the current absolute address in the output bytecode into array, and also output the codes which is recorded by the rule of forward branch.

5. Output the code which appends the entries into the exception table of current method. The addresses in the entries of original exception table is converted to a pair of basic block and in-block offset, and output is generated by the offset and the absolute address of the block top recorded by the above rule.

```
Method int __template()
  0: B8 00 06   INVOKESTATIC #6 <Method int __t1()>
  3: 3A         ISTORE_1
  4: B2 00 07   GETSTATIC #7 <Field java.io.PrintStream out>
  7: 1B         ILOAD_1
  8: 06         ICONST_3
  9: A4 00 09   if_ICMPLE 18
 12: B8 00 06   INVOKESTATIC #6 <Method int __t1()>
 15: A7 00 04   GOTO 19
 18: 04         ICONST_0
 19: B6 00 08   INVOKEVIRTUAL #8 <Method void println(int)>
 22: B1         return
```

Figure A.1: An example of the input to the postprocessor

Operations needed for each instruction at code generation time is some of emitting few bytes to array, recording current pointer to an array, calculates offset by at most 2 additions/subtracts and emit it, and consults the constant pool generator and emit the returned index. All of these operations, except for constant pool operation, is very simple.

A use of the longest form of an instruction makes the bytecode longer. At the worst case, one-byte instruction is expanded to four bytes. However, as the output bytecode is finally becomes an input of JIT compiler, the difference between short and long forms of the same-meaning instruction does not have an impact for execution speed at all, I guess.

For example, the byte code sequence of an template generated from a statement specification

```
‘<>{ int x = @c1(); System.out.println((x > 3) ? @c1 : 0); }
```

is shown in Figure A.1. Two boxed `INVOKESTATIC` instructions are the embed points. The `RETURN` instruction at address 22 is removed before processing. There is two basic blocks in the stream, ranges 3–11 and 15–21. After removing “specific pattern” for embedding, and expanding local variable instructions of short form, the code will become one shown in Figure A.2. In this code, an address of instruction is written as an offset to the block top, and constant entry indices are removed. There are two kinds of branches in the program: the `IF_ICMPLE` is inter-block branch, and `GOTO` is intra-block branch. Two variable indices are underlined in the figure.

```
Method int __template()
 Embed: @c1
Block 0:
   0: C4 36 00 01 WIDE ISTORE %1
   4: B2 ?? ??    GETSTATIC #? <Field java.io.PrintStream out>
   7: C4 22 00 01 WIDE ILOAD %1
  11: 06          ICONST_3
  12: A4 ?? ??    IF_ICMPLE [Block 1:3]
 Embed: @c1
Block 1:
   0: A7 00 04    GOTO 4
   3: 04          ICONST_0
   4: B6 ?? ??    INVOKEVIRTUAL #? <Method void println(int)>
```

Figure A.2: An example of the intermediate code sequence

From the code in Figure A.2, the code generator shown in Figure A.3 will be generated. In the figure, variables and methods which are not defined explicitly in the code is used for explanation only. For each instructions its opcode or operand bytes are shown in Figure A.2, simple assignment for the array `bytecode` is generated. local variable instructions are also directly emitted, but its operand is added with `__shift`.

For the inter-block branch in the offset 12 of block 0, two bytes are left unfilled at the first emission at line 15, and then back-patched at the top of Block 1, lines 20–22. Intra-block branch at the offset 0 of Block 1 is simply emitted verbatimly at lines 24–25.

A.3 Constant pool generator

An constant pool is shared by all methods in a class, and therefore by all class specifications in a class. In addition, two constant pools have usually common entries in it, such as "`java/lang/Object`". Therefore, simply appending constant pools in the templates, with index shifting like those of local variables, may result in too large constant tables. For this reason, I decide that the code generator should generate a constant pool in which the duplicated entries are unified. The constant pool generator will have a hash table in it, and the each entries in the hash table records a constant index for each constant types, such as UTF8-string, class name, method name, etc. If an class name entry is requested,

```
ip = __c1.__generate(bytecode, ip, ...); // Embed

blocktop[0] = ip; // Block 0

{ bytecode[ip++] = 0xC4; bytecode[ip++] = 0x36;
  short i = 1 + __shift;
  bytecode[ip++] = i / 256; bytecode[ip++] = i % 256;} // 0 ISTORE
{ bytecode[ip++] = 0xB2;
  short i = b.request_field("java/io/PrintStream", "out");
  bytecode[ip++] = i / 256; bytecode[ip++] = i % 256;} // 4 GETSTATIC
{ bytecode[ip++] = 0xC4; bytecode[ip++] = 0x22;
  short i = 1 + __shift;
  bytecode[ip++] = i / 256; bytecode[ip++] = i % 256;} // 7 ILOAD
{ bytecode[ip++] = 0x06; }                             // 11 ICONST_3
{ bytecode[ip++] = 0xA4; ip += 2; }                    // 12 IF_ICMPLE

ip = __c1.__generate(bytecode, ip, ...); // Embed

blocktop[1] = ip; // Block 1
{ short i = ip - blocktop[0] + 4 - 12;
  int t = blocktop[0] + 13;
  bytecode[t] = i / 256; bytecode[t+1] = i % 256;}     // backpatch 12

{ bytecode[ip++] = 0xA7;
  bytecode[ip++] = 0x00; bytecode[ip++] = 0x04; }      // 0 GOTO
{ bytecode[ip++] = 0x04; }                             // 3 ICONST_0
{ bytecode[ip++] = 0xB6;
  short i = b.request_method("java/io/PrintStream", "println", "(I)V");
  bytecode[ip++] = i / 256; bytecode[ip++] = i % 256;} // 4 INVOKEVIRTUAL
return ip;
```

Figure A.3: Generated code generator

for example, an hash table entry for given string is searched. If it is not yet in the table, an entry is created, and UTF8-entry is emitted to the byte stream of constant pool, and its index is recorded. Next, if the entry already has a index for class entry, it is returned. Otherwise, an class entry is emitted to the byte stream and the index of it is returned.

References

[1] Per Bothner. Kawa—compiling dynamic languages to the Java VM. In *USENIX*, New Orleans, June 1998.

[2] Charles Consel and François Noël. A general approach for run-time specialization and its application to C. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL '96)*, pages 145–156, St. Petersburg Beach, FL, USA, January 1996.

[3] James W. Cooley and John W. Tukey. An algorithm for the machine computation of the complex Fourier series. *athematics of Computation*, 19:297–301, 1965.

[4] Markus Dahm. Byte code engineering with the JavaClass API. Technical Report B-17-98, Institut für Infomatik, Freie Universität Berlin, 7 July 1998.

[5] Pierre Duhamel and Martin Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, 19(4):259–299, April 1990.

[6] Dawson R. Engler. VCODE: A retargetable, extensive, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 160–170, Philadelphia, PA, USA, May 1996.

[7] Matteo Frigo. A fast Fourier tranform compiler. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 169–180, Atlanta, GA USA, May 1999.

[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[9] Masatomo Hashimoto and Akinori Yonezawa. MobileML: A programming language for mobile computation. In *Proceedings of the 4th International Conference*

on Coordination Languages and Models (COORDINATION 2000), number 1906 in Lecture Notes in Computer Science, pages 198–215. Springer-Verlag, 2000.

[10] Yuuji Ichisugi. Epp homepage. `http://www.etl.go.jp/~epp/`.

[11] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. In Olivier Danvy and Andrzej Filinski, editors, *Second Symposium on Programs as Data Objects (PADO II)*, In Lecture Notes in Computer Science. Springer-Verlag, Aarhus, Denmark, May 2001. To appear.

[12] François Noël, Luke Hornof, Charles Consel, and Julia L. Lawall. Automatic, template-based run-time specialization: Implementation and experimental study. In *Proceedings of the International Conference on Computer Languages*, Chicago, May 1998.

[13] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):324–367, March 1999.

[14] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and modal-ML. In *the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation (PLDI)*, pages 224–235, 1998.